Chapter 1

Introduction

The goal of this thesis is to design a *scrutable* user modelling shell for supporting user-adapted interaction. User models can support user-adapted interaction in many ways. They enable the machine to know the user well enough to improve its interpretation of user actions and requests, improving the communication bandwidth *into* the machine. Equally, the user model can drive customisation of the activities *within* the machine so that the machine can operate on behalf of the individual user. Finally, the user model can control the individualisation of the presentation of information *out* of the machine at the interface with the user.

We describe a user model as *scrutable* if the user can scrutinise the model. The scrutable user modelling shell consists of a representation for user models and tools which can support the processes of user modelling.

It is our contention that scrutability of a user model is important and useful. To see why, consider the range of user-adapted systems:

- advisors;
- consultants;
- help systems;
- recommender systems that filter information on behalf of the user;
- systems that tailor the output they produce to the particular needs of the individual;
- systems that tailor the interaction and modality to match the user's preferences, goals, task, needs and knowledge and
- the intelligent teaching systems that aim to teach, as a good teacher does, matching the teaching content, style and method to the domain and the individual student.

The overarching goal of user-adapted systems is to improve the efficiency and effectiveness of the interaction. Such systems aim to make complex systems more usable, present the user with what they† want to see, as well as speed up and simplify interactions (Malinowski, Kuhme, Dieterich, and Schneider-Hufschmidt, 1992). For example, Jameson's User Modelling Conference' Reader's guide (Jameson, Paris, and Tasso, 1997) lists the purposes of the user modelling papers as: helping the user find information; tailor information presentation to the user; adapt an interface to the user; choose suitable instructional exercises or interventions; give the user feedback about their knowledge; support collaboration; and predict the user's future behaviour.

We can characterise the operation of such user-adapted systems with the example of butler-like agents described by Nicholas Negroponte

> The idea is to build computer surrogates that possess a body of knowledge both about something (a process, a field of interest, a way of doing) and about you in relation to that something (your taste, your inclinations, your acquaintances). Namely, the computer should have dual expertise, like a cook, gardener, and chauffeur using their skills *to fit your tastes and needs* in food, planting, and driving. (Negroponte, 1995:151)

He describes a future user-adapted newspaper thus:

> Imagine a future in which your interface agent can read every newswire and newspaper and catch every TV and radio broadcast on the planet, and then construct a personalised summary. This kind of newspaper is printed in an edition of one. (Negroponte, 1995:153)

---

† This thesis uses the singular 'they' as the pronoun where we do not know whether a person is female or male. Later, we again use it in referring to users in our studies: although we know their sex, we are bound to preserve their anonymity and so prefer to avoid indicating their sex. This follows the recommendations of several guides including (Miller and Swift, 1980) and has been used by such writers as Dickens (in Bleak House), Austen (in Mansfield Park) and Eliot (in Daniel Deronda).

Such visions of user-adapted interaction involve sophisticated software systems that keep a model of the user and perform complex actions controlled by that model. This thesis is based upon the premise that a user should be able to *scrutinise* such user models.

This chapter introduces our task by defining a user modelling shell in terms of definitions for user models, user modelling tools and the notion of reusability for each. We then introduce the notion of scrutability, starting with two scenarios of user-adapted systems. These are used to explain our notion of scrutability. They also illustrate our arguments for scrutability. Finally, we give an overview of the thesis.

1.1 User modelling definitions

At present, the term *user model* is used in different ways by different researchers. There are synonyms or near synonyms, like student model and learner model. In addition, there is an array of related terms, including cognitive models, conceptual models, mental models, system models, task models, user profiles and others. We follow what appears to be an emerging use of terms, with those for user-adapted systems coming from Wahlster and Kobsa (Wahlster and Kobsa, 1986) and those for human-computer interaction (HCI) from Norman (Norman, 1983):

- user model: the system's set of beliefs about the user;
- user modelling tools: tools to create and maintain user models;
- user model consumer: programs that make use of the user model;
- user modelling shell: representation for the user model so that user models can be *reused* by different consumers as well as user modelling tools which can be *reused* for the user modelling needs of different consumer systems.

The first follows (Wahlster and Kobsa, 1986). The second is a concept that goes by different names. In user modelling shells that are implemented as a single program, the tools are the primitive operations for user modelling. Where the user modelling shell is implemented as a collection of separate programs, the 'tool' is a program that supports a primitive user modelling action as in the Software Tools approach (Kernighan and Plauger, 1976). The third term is consistent with Brajnik and Tasso (Brajnik and Tasso, 1992) and the last is consistent with the use of the term in the user modelling shells we will describe in Chapter 2.

1.1.1 User model

Our definition of a user model is illustrated in Figure 1.1. The 'real world' depicted at the top includes actual entities in the world (like animals) as well as artificial and abstract notions such are studied at school (like mathematics). Within that world, we have a part which is the *context* of the modelling task at hand.

Note that both the real world and the context are shown as irregular, messy shapes to reflect their complexity. Below the real world and current context, we show two models of the context: on the left is the user's model and on the right, that of the programmer constructing the user model and user-adapted interactive system. These too are 'real' in the sense that they exist in the real minds of real people. These are also messy shapes since they represent models hidden within human minds and these may well be quite messy. Aspects of these that relate to the user's (and programmer's) knowledge and understanding are frequently described as *mental models*. Other aspects that are important for user-adapted interaction include the user's preferences, interests, goals and plans.

Our focus is on the user model shown at the bottom of the figure. This is a tidy, artificial model. We can distinguish two major forms of user models: *cognitive* and *pragmatic*. Cognitive user modelling research builds user models which are intended to match the way that people actually think and know. Indeed, this aspect of user modelling is quite properly of importance for psychologists and educators. Perhaps the exemplar of this is the ACT-R theory of skill acquisition (Anderson, 1993) which can be used to define user models for teaching systems (Corbett and Anderson, 1995). Of course, once the theory has done its job, the resultant user model is a well-defined artificial construction of the psychologist-programmer. Our definition of user model includes user models that may have some claim to cognitive validity as well as more pragmatic user models that simply serve to support user-adapted interaction.

The various arcs in the figure indicate the way the user model is formed. First, the user must build their own model of the context, by experiences and learning within that context and the rest of the real world.

Similarly, the programmer builds their model, but based on experiences and learning that are generally different from those of any particular user. Indeed, this difference is important for the design of effective user interfaces. A major concern for Human-Computer Interaction (HCI) designers is to avoid the pitfalls of assuming users are much like themselves. Rather the programmer strives to develop an improved understanding of the user. This is also the task of the programmer developing a model of the user. It is indicated by the dotted arc from the user to the programmer models.

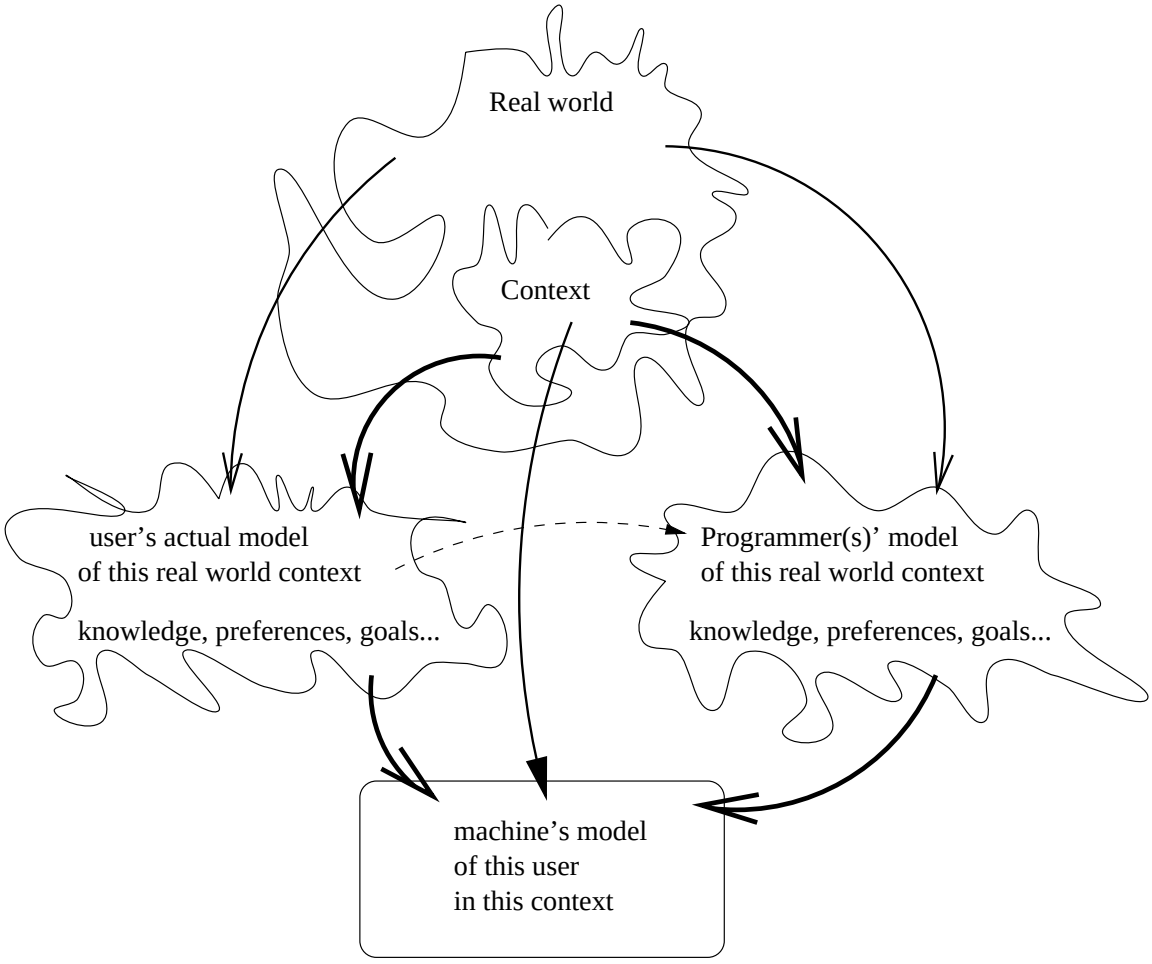


Figure 1.1 User model

The heavy lines feeding from the context to the people's models indicate the primary influences on them. The thinner lines from the rest of the real world to the people's models remind us that people's understandings are often richly connected and any single context will be modelled and interpreted in the light of broader understandings.

Once the programmer and user have models of the context, the heavy arcs from these to the user model show the two major influences on it. The arc from the user's model reflects the transfer of some of the user's beliefs, preferences, goals, interests and the like. The heavy arc from the programmer's model reflects that the model is controlled by the programmer. Their influence will be indelibly cast into the model: it is the programmer who will define the limits of what can be modelled, how it is represented, how the user can contribute to it and how it can evolve.

Finally, there is a thin arc from the context, directly to the user model. This captures the

user attributes relevant to the current context. These are real-world aspects rather than part of the user's beliefs and knowledge. For example, we may need to model the user's typing speed, an attribute of the user. This may differ from the user's beliefs about their typing speed. These aspects of the user model are the system's beliefs about user attributes.

This thesis is concerned with user models that are *individual* in the sense that the system is able to construct different models for each user. So the model at the bottom of Figure 1.1 may well be different for different users. This is not the case in much of the user modelling work by the HCI community, where the goal is to model a canonical or generic user. In that work, programmers may observe or interview users as part of the process of defining a good canonical user model. Once this has been done, the programmer can use that model to inform the creation of the interface to match the canonical user. User-adapted interaction is fundamentally concerned with tasks where a canonical user model is inadequate. For example, a customised newspaper is intended for a particular user.

We further limit our scope to *dynamic* user models which can change over time. This is necessary for user-adapted interaction, and it means that we cannot hard code the model into the machine as in much of the HCI community's user modelling. For example, a user-adapted teaching system adapts its operation to take account of the user's learning over time.

Finally, we restrict ourselves to *explicit* user models which are separable from the rest of the user-adapted interactive system. Once again, this differs from much work in the HCI community's user modelling where the programmer who creates a user model applies this to the interface design process so that the user model is implicitly built into the system.

Our definition of user model is broader than many. Certainly, it includes most uses of the term *student model* as well the near synonym that is becoming more widely used, *learner model*. Such models drive the individualisation of teaching.

In the intelligent teaching systems community, many researchers would limit the student model to a quite sophisticated model of the learner. See, for example, the collection of student modelling papers in Greer and McCalla (1994). It is also explicitly stated by Elsom-Cook (1993) when he excludes the simple model that a conventional computer-aided-instruction (CAI) program may construct. This would record the student's interactions with the system, including performance on assessment tasks. Elsom-Cook argues that this is a model of the interaction rather than the student and this is certainly so. However, if it is then used to adapt the interaction, it is effectively serving as a model of the user. For example, suppose it records that the student has answered all the assessment tasks correctly. Further, suppose that on that basis, the machine alters the

presentation of material and assessment tasks, to move more quickly and ask more challenging questions. In that case, the system has effectively modelled the user as being able to operate at a higher level and has adapted the interaction in terms of it. Moreover, the data held about the user could be used to explain the adaptation: if the user were to ask why the machine was operating differently, a good answer would include a report of the user's excellent performance on assessment tasks as a basis for making the teaching faster and the testing more challenging. One could argue that this record of performance constitutes the basis for an inferred model of the student's ability.

Our definition of user model is broad, including any individual, dynamic and explicit model the system holds about the user, including the beliefs the user holds and other user attributes. The user modelling tools are responsible for constructing and managing such user models, as well as providing a programming interface to the user models for consumer systems which can then adapt their interaction to the individual user.

1.1.2 User modelling shells

We now define our use of the term *user modelling shell* in terms of reusability, first of the user modelling information and then, the tools needed to manipulate user models.

Reusable user models

The user model itself offers the most fundamental level of reusability. Once we have a user model, it makes sense to reuse it in a range of applications. For example, consider a user model that can be used to customise a newspaper. This might well model the user's music preferences so that it can include news about music. Another consumer system might teach music appreciation and this may well need some of the same information about the user's music preferences.

This is sensible from the user's point of view. Suppose, for example, the user spends time interacting with one application, all the while helping it develop a better user model and hence, an improved ability to interact effectively. It would be very reasonable for the user to expect a second application to use relevant aspects of the same user model. This improves the consistency across applications. It also saves the user from the tedium of training new systems.

Reuse of a user model is also defensible from the point of view of implementation. It can be costly to construct and maintain user models for user-adapted interaction. This makes it appealing to amortise some of the costs over different systems, so reducing the cost of the user modelling for a particular interactive system.

This user model reuse can be achieved by maintaining a database of user models that can be accessed by various consumers. Equally, it can be achieved by storing the user model in an external form that can be accessed and interpreted by various programs. For a model to be reusable, it needs an agreed ontology and representation so that it can be understood and used by different user model consumer programs.

A reusable model is one step away from the current widespread practice in software systems where a set of flags is kept in a standard file and is used by various programs to customise the interface. For example, a system flag might indicate the name of the user's preferred text editor. The important difference is that such flags are simple directives to programs rather than a model of the user (from which these directives might be determined). The difference between such flags and a user model corresponds to the difference between *adaptable* and *adaptive* systems. This thesis is concerned with user-adapted systems, a form of adaptive system.

We also note that there may be user-adapted systems that require only modest user models. For example, it has been shown that quite small amounts of user modelling information can be used to customise a system effectively (Neal, 1987). This quantity and form of user modelling information needs only simple approaches for its maintenance. Similarly, more recent work to customise a commercial tax and investment package (Strachan, Anderson, Sneesby, and Evans, 1997) had a small and simple user model which improved user assessments of satisfaction. The first example of large scale deployment of individual user modelling has been within the Microsoft Excel user support product (Horvitz, 1997). The user models in this product track a small number of the user's recent actions and the associated inferences from a Bayesian net. The critical point here is that there is a range in the sophistication demanded of user models for different consumers. Nonetheless, we would like to support reuse of parts of a user model across different user model consumer systems.

Reusable tools for user modelling

We define a *reusable* tool for user modelling as any part of the user modelling process that can be applied in several consumer systems. These tools fulfil a number of roles.

Tools that support the creation of modelling information need to *acquire* information about the user. This may involve direct requests for the user to provide information. It can also involve use of indirect information. One of the most important sources of such indirect information is the *dialogue history*, the history of observations of the user and system as they interact. In addition to this information which comes from the user and their actions, tools need to be able to *infer* information that follows reasonably from it.

As each new aspect is added to a user model, there is the possibility of conflicting information. This may be due to the unreliability of information acquired or inferred. It may also be due to changes: for example, at one point the user model might correctly reflect that the user does not know a fact and yet later, that user may learn it. Once a conflict is detected, a tool must determine how to deal with it. For example, in the case of the user learning, the tool should alter the model.

1.2 Why scrutability and control?

With the user model as the driving force for a system's adaptation to the individual user, we characterise the issues for scrutability of user models with two scenarios of user-adapted interactions. In each case, we describe the scenario of a user interacting with a user-adapted system. Then we identify some of the questions a user may want to ask. The user should be able to find some of the answers by scrutinising their user model. These scenarios will be used to illustrate our arguments for scrutability. The goal of designing user models which can be scrutinised by the user will motivate our design of our scrutable user modelling shell.

1.2.1 Scenario 1 - a teaching system as the user model consumer

Our first scenario involves a coach which is supposed to send the user timely and useful advice:

> Jonathan uses a text editor inefficiently because he is unaware of its undo command.
> A coaching program tells him about undo, why it can be helpful and how to use it.
> Jonathan reads this information, experiments with undo and proceeds to be a somewhat more effective user of the text editor.

In this happy scenario, the coach could give helpful advice because it had developed a model of the Jonathan's editor knowledge (and, in particular, lack of knowledge of undo). This was combined with a mechanism for selecting a good teaching goal as well as the means to generate information about undo, starting with an explanation of its usefulness in order to motivate him to make the effort to learn it.

This thesis focuses on the role of the user model in this process. In particular, we are concerned with enabling a user like Jonathan to scrutinise their own user model to answer questions like these:

i. How did it know that I didn't know about undo?

ii. What else does it think I know (or don't know)?

iii. Why did the machine tell me about undo?

iv. How can I tell the machine about the things I want to know?

v. What would the machine do if it thought I knew things that I really do not know?

vi. Why did it explain undo in the way it did?

vii. Does it explain things differently depending on what it thinks it knows about me?

In these questions, Jonathan has anthropomorphised the machine. Were the advice of the scenario given by a person, it would make sense to ask such questions.

If Jonathan can scrutinise the machine's user model, he should be able to find answers to some of these questions. For example, the user model for such a system would need to represent aspects of Jonathan's knowledge of the text editor. If Jonathan is able to scrutinise the part of the user model representing his knowledge of undo, he should find it modelled as 'not known'. This should help him find part of the answer to the first question.

In addition to the actual value of the undo part of the model, the first question requires access to the *processes* which formed the model. For example, suppose the system used monitoring of Jonathan's interaction with the editor to conclude about his lack of knowledge of undo. In this case, it may be that over twelve months of monitoring, Jonathan has never used undo. Then the answer to the first question, calls for user access to information about this process.

The second question introduces the need for Jonathan to access definitions for the *ontology* used in the user model. From the user's perspective, the ontology is the meaning of the terms or categories represented in the user model. Suppose, for example, that Jonathan explored the user model for the text editor and found that it represented an aspect called xerox. If Jonathan does not know that xerox means a particular command available for the text editor, he cannot understand what the model represents. To answer the second question in our scenario, Jonathan needs to be able to find out the meanings of the elements modelled.

Some answers may not lie in the user model. For example, consider the third question, on the machine's decision to coach on undo. If part of the user model represents Jonathan's learning goals, including the goal, learn about undo, the answer to this question can come from scrutinising the user model. However, if the user model has only the user's current knowledge, the coach might have inferred this as *its* teaching goal. In this case, scrutinising the user model only gives the *foundations* for the coach's decisions. In such cases, where the answer to the user's question lies in the consumer system, the scrutable user model can only provide information for a partial answer to the question.

The fourth question raises the issue of user control. It follows from supporting user scrutiny of the model. After all, if the user is allowed to see the user model, it follows that the user should also be able to alter it.

The last three questions all relate to the action of the coach, the consumer of the user model. In these questions the scrutable user model does not have all the information needed for the answers. However, a user who can alter the user model can try *what-if* experiments, changing the user model and observing the effect on the operation of the coach. The point here is that if the user model is both scrutable and alterable by the user, it also gives an indirect mechanism for the user to explore aspects of a consumer system that is not itself scrutable.

This thesis describes the design and evaluation of our approach to modelling users in a manner that makes it possible to scrutinise parts of the model that contain the answers to questions such as those discussed in this scenario.

1.2.2 Scenario 2 - A personalised newspaper as a consumer

Now consider a scenario with a different form of modelling.

> Sarah starts Mynews, a personalised electronic ‘newspaper’. First she gets the headlines. Today, these include:
>
> - Politician caught out by wife
> - New movie release: Flipper saves the day
> - New music release: Sydney Symphony Orchestra - Carr-Boyd’s Prelude

Mynews is supposed to select just news items most likely to interest this user. It models Sarah’s interests so that it can collect reports on issues she will want to know about. Where there are several items about an issue, the user model is supposed to ensure selection of reports Sarah will prefer. For example, she may like a particular critic’s music reviews or one syndicate’s reports for European news. In this scenario, Sarah might ask:

i. How did it decide I would be interested in a Flipper movie?

ii. Or a Carr-Boyd release?

iii. What or who is Carr-Boyd?

iv. How can I let it know that I am pleased to hear about the Flipper movie?

v. Why did it give me that article about the politician?

vi. And especially at this time, during a national election campaign?

vii. Why hasn't it reported the new unemployment statistics that were due for release today?

Although this scenario would seem to need a large model of user interests and preferences, these questions are similar to those in the first scenario. The first two may relate to particular parts of the user model representing Sarah's preference for 'Flipper movies' and 'Carr-Boyd music'. As in the last scenario, the user should be able to find information to help answer the questions if they have access to the *value* of that part of the model and a record of the *processes* which contributed to that value. It may be that Sarah has rated several Flipper movies and other animal-centred children's movies, usually giving very positive ratings. She may have also ordered several such movies from an on-line service that provides such information to the user model. If such information affected the values in the user model, Sarah should be able to scrutinise the model to see that this was so.

The third question might be answered within the articles of the newspaper. In general, such questions require an explanation of part of the user model ontology, in this case, a music ontology with 'Carr-Boyd' or 'Australian contemporary art-music composer'. The fourth question relates to user control of the model.

The last three questions are likely to involve a combination of the user model and the mechanisms controlling Mynews. These questions relate to issues of media control.

At this point, we briefly explore the reasons we are concerned with answering questions such as those posed in the above scenarios. This serves two purposes. It gives the motivation for our work. At the same time, it introduces some of the notions that influenced the design of our scrutable user modelling shell for user-adapted interaction.

1.2.3 Motivations for scrutable user model

Some of our arguments for scrutability follow from the nature of the user modelling enterprise and the types of consumer systems which need user modelling to drive the adaptation of the system. Other issues are more generic, essentially being arguments in favour of making systems more comprehensible by enabling the user to scrutinise the user model. Other researchers have argued the importance of making complex systems more comprehensible. For example, Maass has argued for system transparency (Maass, 1983) and Fisher for making making complex systems comprehensible (Fisher and Ackerman, 1991).

Access to and control over personal information

Scrutability means that the user should be able to delve into the user model and modelling processes to get find answers to questions like those listed in our scenarios. More than this, we want the user to be able to control the user modelling process, as implied by some of the scenario questions where the user wants to be able to feed into the machine's processes for building the user model.

Perhaps the most compelling reason for scrutable user models is the right of the individual to know what information a system maintains about them. Both our scenarios involve personal information about the user, in one case their knowledge of a text editor and in the other their knowledge and interests relevant to creating a customised newspaper. It can be argued, as for example Kobsa does (Kobsa, 1990, Kobsa, Kuhme, and Malinowski, 1993), that long term user modelling information should be viewed in the same way as databases of personal data.

One response to this is to avoid keeping any long term modelling information at all. In many of the classes of systems we have introduced, this approach is infeasible. For example, consider a teaching system: one would think little of a private teacher who never remembered their student from one learning session to the next. As most substantial learning goals can only be achieved over a substantial time frame, and in many separate sessions, user models within an Intelligent Teaching System (ITS) need to include long term representation of the learner's developing knowledge. On this basis among others, Self (Self, 1988) argues for making the user model in teaching systems accessible to the student.

Programmer accountability

If a system's model of the user is accessible to that user, the system designer has greater accountability. The system's model must be defensible. This matter is closely linked to the user's rights to access personal information. It is, however, so important that it deserves to be stated separately.

In our scenarios, it might mean that the programmer will think about the whole user modelling task differently. Knowing that the user may scrutinise the user model, the programmer should be careful in the choice of aspects to model. For example, we would expect the programmer to avoid modelling the user as an 'incompetent user of the editor'. Similarly, in the second scenario, the programmer should be mindful of the need for the user model to capture aspects of the user they would like to see modelled. The programmer should also ensure that the news selected for the user *appears* to match this model.

Correctness and validation of the model

Scrutable user models enable the user to check and correct the model. This is an obvious but critical argument for scrutability, and it relates to the earlier one of the user's right to know what the computer stores about them. Often, it is easy to gather user modelling information from the analysis of that part of the user's behaviour which is observable by the machine.

For example, the coaching system in our first scenario might well construct its user model on the basis of monitoring use of the text editor. This process offers many possibilities for creating incorrect user models. Consider, for example, that the user allows another person to use their machine account: the inferences are no longer about the individual we intended to model. The model's accuracy can also be affected by people giving the user advice: someone might direct the user to type a series of complex commands. This could result in a user model with an inaccurate picture of the user's sophistication. If the user can easily check the user model, they can correct it.

A similar argument is central to Csinger's thesis on scrutable systems (Csinger, 1995). This applied user models to customising video presentations. The scrutability of the system was improved by presenting the relevant parts of underlying user model and allowing the user to alter the values of those parts. Csinger's system does not make the user modelling process scrutable: its focus is the scrutability of a user-model-consumer system in terms of the values of parts of the user model that it determines to be salient to the current discourse. Orwant also argues for user access to the model to ensure its accuracy (Orwant, 1994), his work being based upon a rich range of mechanisms for acquiring the modelling information.

One objection to user-adapted systems is that they infer too much from too little information (Henderson and Kyng, 1995). This concern is greatest where the base-information has low reliability as in the case of data derived from monitoring the user. Making the model scrutable ensures the user can see and control the bases of the system's inferences about them. In this way, scrutability of the user model can help address this objection to user-adapted systems. The same issue is addressed by Browne, Totterdell and Norman

> One of the difficulties is that the system has a severely restricted communication bandwidth through which to interpret the user's situation. The problem of interpreting the situated action of a user is described eloquently in Suchman (1987). (Browne, Totterdell, and Norman, 1990:156)

The authors note the need for a dependable model and observe that the longer the chain of inference about the model values, the more difficult it is to be confident of the dependability of that reasoning.

Of course, there is a potential problem in allowing users to see and alter the user model. They might just decide to lie to the system and create a model of themselves that they would like to be accurate: perhaps they would like the system to show them as experts. The playful and curious user might simply like to see what happens if they tinker with their user model. The possibility that users may corrupt their user model must be taken into account in designing the representation and support tools. It is also an important consideration in the design of experiments to assess the effectiveness of systems with user modelling. However, it is not sufficient reason for denying users the right to control the system's model of them.

Machine determinism and asymmetry of the human-machine relationship

There are inherent differences between machines and people. These introduce the possibility for user-adapted systems to be especially effective in some interactions. In particular, the processes controlling an user-adapted computer system are normally deterministic. This means that there is an accurate answer to the types of questions posed in our scenarios. By contrast, in interactions between humans, the processes behind actions may not accessible (Nisbett and Wilson, 1977). Essentially, we argue that although the machine cannot really know the user's beliefs, it should be possible for the user to know the machine's beliefs, especially the beliefs contained in a explicit model of the user.

In systems intended to be co-operative, the scrutability of the user model has the potential to play an important part in helping the user understand the system's goals and view of the interaction. This may reduce user misunderstandings, due to expectations and goals held by the system and not appreciated by the user. So, for example, in our coach of the first scenario, the aspects modelled in the user model signal the elements of the editor that the programmer of the coach considered important.

This may be useful in dealing with another of the objections to user-adapted systems: it has been argued (Browne, Totterdell, and Norman, 1990) that if the system and user both attempt to model each other, a situation described as 'hunting' could arise. This means that the user tries to model the system but as they do so, the systems changes because it is trying to model the user. Scrutability should provide a stabilising influence since it can give the user access to the user model which is the basis for system's adaptation.

In general, we can expect to enhance the quality of the collaboration if both the user and system can be aware of the beliefs each holds about the other: an accessible user model can help the user be aware of the machine's beliefs.

Aid to reflective learning

In the case of teaching systems, there are additional, compelling reasons for scrutability. Indeed, there is a case for arguing that we can *repurpose* the user model (usually called a student model in this context) to become a valuable basis for nurturing deeper learning. One argument is based upon the importance of metacognition, thinking and learning about learning. This involves knowledge and skills such as those needed to judge when one knows something and to know how to go about learning something. There is evidence that metacognition is important (Lawson, 1984, Yussen, 1985, Volet and Lawrence, 1990). Moreover, there are several ways that metacognition can be supported by giving learners access to the machine's model of their knowledge. Several researchers have argued for making the student model available to the learner on the grounds that this will support learning (Bull and Smith, 1995, Bull and Pain, 1995, Bull, Brna, and Pain, 1995, Bull, 1997, Corbett and Anderson, 1995, Crawford and Kay, 1993, Paiva, Self, and Hartley, 1995, Self, 1988, Dillenbourg and Self, 1990).

Teaching systems have explicit and clearly defined teaching processes. In particular, a critical role of the user model is to represent what the learner knows and does not know at any stage. If the model is externalised by the system, presented in a way that enables the student to reflect on and evaluate their own knowledge, the student is more likely to develop deeper understanding. So, for example in our first scenario, the user looking at the user model might observe they are modelled as knowing only the bare rudiments of the editor. They might realise that this denies them the power of the editor. Then the user may be motivated to learn more.

The discussion to this point has emphasised the possibility of a teaching system making an externalised user model part of its teaching. There is also a learner-initiated view. For example, consider the report of users interacting with one of the earliest coaching systems (Burton and Brown, 1979). Some students were curious about the coaching associated with this arithmetic drill game called West. They stopped playing the main game and began making moves designed to see the effect on the coach. Had such users been able to scrutinise the user model, they might have been able to satisfy at least part of that curiosity. Note that in an inscrutable system, such as West, the user's exploration and curiosity corrupts the user model. This means that scrutability might enable the user to explore the system without compromising the accuracy of the user model.

1.3 Summary and thesis overview

Our goal is to design a scrutable user modelling shell for supporting user-adapted interaction. This chapter introduced our task, defining the user modelling shell which supports reuse of user models and user modelling tools. We have defined scrutability in terms of the types of questions a user might be able to answer after scrutinising the user model to determine the:

- *values* of parts of the user model;
- *processes* that contributed to these values;
- *meaning* of the user modelling ontology.

We have argued for scrutability of user models on several grounds:

- the user's right to see and appreciate the meaning of personal information the computer holds about them;
- programmer accountability;
- the possibility of users correcting errors in the model;
- enabling users to have a sense of control over the adaptation of systems by controlling the user model;
- confirming the role of the machine as the servant or aid of the user;
- and, in the case of teaching systems, the potential to encourage metacognition and deeper learning.

This thesis describes the design and evaluation of our scrutable user modelling shell, in terms of our:

- definition of the properties of a user modelling shell and descriptions of the major user modelling shells which influenced our design of a scrutable user modelling shell (Chapter 2);
- definition of the requirements for a scrutable user modelling shell (Chapter 3);
- interaction model for describing user-adapted interactions and which serves as a basis for aspects of the design of our scrutable user model representation (Chapter 4);
- accretion representation which is the basis for our long term user models (Chapter 5);

- um toolkit which supports one form of accretion representation and has tools for the major elements of our user modelling shell (Chapter 6);
- design for tools that support the user in scrutinising their user model and the user modelling processes that formed it (Chapter 7);
- implementation of two consumer systems and their associated um tools - one consumer is a coach which relies largely on *epistemological* user models and the other is a movies advisor which operates in terms of user *preferences* (Chapter 8);
- evaluation of the scrutability of the user models in one formative, small-scale evaluation and a summative large-scale field test (Chapter 9);
- conclusions and directions for further work in scrutable user modelling for supporting user-adapted interaction (Chapter 10).

Chapter 2

Background : user modelling shells

This chapter provides background for our goal of creating a scrutable user modelling shell. First, we identify *essential properties* of a user modelling shell. Then we describe major user modelling shells.

2.1 Essential properties for a user modelling shell

A user modelling shell has five essential properties:

1. representation for the user model components, the atomic elements of the user model;
2. support for the use of evidence about the user from sources external to the user modelling system, especially those derived from user interactions with a system;

3. support for inference within the model, so that one set of beliefs about the user can be used to deduce additional beliefs;
4. mechanisms for managing inconsistency, uncertainty, noise and change in the user model;
5. and a degree of broad applicability across a variety of domains and user model consumer systems.

We now discuss each of these as high level requirements for a user modelling shell.

2.1.1 Representation of user model components

The atomic elements of the user model represent the system's beliefs about the user. We call these the *components* of the model. These can be usefully classified into various types of beliefs. Table 2.1 illustrates some of the range of components that might be modelled. The last column gives the classification for each. The horizontal lines separate the different classes of examples. We now discuss these examples to illustrate the diversity of components a user model might need to represent. This is an important preliminary for our understanding of the task of representation of user models.

Table 2.1. Examples of some classes of components

Description of meaning of a component	**Classification**
The system believes the user ...	
likes large fonts in screen displays	preference
dislikes Carr-Boyd music	preference
prefers to learn from examples rather than abstract descriptions of ideas	preference
knows the undo command of the text editor	knowledge
believes that the sun rotates around the earth	knowledge
knows that if (x implies y) and (y implies z) then (x implies z)	knowledge
believes text editors have a facility for undo-ing the last action	knowledge
wants to listen to new music	goals
wants to learn about the sam text editor	goals
was born in 1990	attribute
has low vision	attribute

Preferences

The first group of examples are *preferences*. These are important in systems like Mynews, the customised newspaper introduced in the second scenario of Chapter 1.

The first preference example is for large fonts. This type of user preference is handled in many existing software systems, but not as user modelling information. The important

difference is that existing software typically requires flags or preferences to be set up for each application program: it is a part of the data for the particular *application* rather that a user model associated with the *user* and available to several applications. We include it here as an important example of *minimalist* user modelling that is clearly valuable for a broad range of software.

The second example of a preference, for Carr-Boyd music, is typical of the class of information needed to support browsing, filtering and retrieval activities in large collections of objects. This is an area of growing importance as indicated, for example, by the collection of papers in the recent special issue of the Communications of the ACM (Resnick and Varian, 1997).

The third preference example might be useful for teaching systems that can use different teaching styles. This is a deeper form of customisation than the simple font preference in the first example. It is also more difficult to establish the value of components like this.

Knowledge

The second group illustrates some of the variety of components modelling a user's *knowledge*. The first pair of examples show quite simple concepts, with the first expressed as the user *knowing* and the second as the user *believing*. The critical distinction is that in the first case, the person who created the system shares the same 'knowledge' as the user. This is often described as the system and user having *shared* or *mutual* beliefs. In the second case, the person who created the system does not hold this belief and might describe it as a misconception or alternate knowledge framework.

The third example is modus ponens. This is a piece of knowledge that supports inference. Such components can be important since they permit a user model to represent a small number of user beliefs and from these to infer what the user might infer.

The final knowledge component is an example of a belief that is extremely useful even though it is not always true. We have expressed it as a generalisation that normally holds for text editors. In this sense, it is similar to the third example which permits inference: if the user encounters a new text editor, we can assume they will expect an undo facility.

Goals

The next pair of examples model the user's goals. These are important in many systems: for example a new music recommender is useful only if the user has the goal of listening to new music. Just such a goal component is given in the first of these examples.

The other example of a goal component indicates the user wants to learn about the sam

text editor. The system might respond by assisting the user in learning those aspects they are ready to learn and that will be useful, using a form that suits this user's preferences in learning and presentation style.

Attributes

The two final examples are described as user *attributes*. The user's age might be important for customising the interface. For example, most young children cannot read well. So a system might present information with spoken output. Similarly, a system creating a music programme might select children's songs. The last component indicates the user has low vision. This might be used to infer that large, clear fonts should be used for interfaces, even if this means that less information can be presented on one screen.

This review of component classes has not been exhaustive. Indeed, other classes of component are represented in some of the user modelling shells we describe in this chapter. However, this section has introduced a range of examples of components in order to indicate their diversity. This has important implications for their representation and management.

2.1.2 Support for use of external user modelling information

Once we have determined the components to be modelled, we need to establish the values of those components. The most obvious and natural basis for doing this is by taking information about the user from the environment. In early user modelling research, the user was a participant in natural language dialogue (Kobsa and Wahlster, 1989) and the user's interaction provided user modelling information. In general, user modelling might be based upon information provided by arbitrary sensors that observe the user. And, of course, the use might volunteer information directly. A user modelling shell should be able to accept information about the user from a range of external sources.

2.1.3 Support for internal inference

This is the other important mechanism for establishing the values of components in the user model. As we will see, two main forms of such reasoning have become established. *Stereotypes* enable a system to use a small amount of information about the user to infer large amounts. For example, if the user claims to be a Unix-guru we can infer the values for the many components representing aspects of Unix they should know.

We call the other major inference mechanism *knowledge-based* inference. Knowledge of the domain makes it possible to infer more about the user. For example, if we know a

user likes three of Beethoven's symphonies, and we know other symphonies are musically very similar, we can infer that the user will probably like those too. Similarly, if the user knows certain advanced commands of the text editor, we can infer they know prerequisite concepts.

2.1.4 Management of consistency

The nature of user modelling demands support for management of conflicting information about the user. We can discuss these demands in terms of the last three requirements: the nature of component modelled, information from external sources and from inference.

Inconsistency anticipated as part of the modelling task: component values that change

The whole point of modelling some components is to track changes. For example, the user's knowledge should change as they learn from a teaching system. If the model is to remain accurate, it will have to change. Equally, if the user holds misconceptions at one stage, we hope this will change. It is also in the nature of knowledge that users will forget things over time and we may need to model that.

In the case of user preferences, we would expect user's tastes change over time. Similarly, their interests may change, often on a temporary basis. For example, a system intended to customise a newspaper may need to adjust the model for the user's interest in Italy while their parents are travelling there: when they return, the model needs to change. It may even be that the user's preferences change in some systematic way. For example, at certain times of the day, they may prefer different music or want to read different sorts of news.

Another highly volatile class of components is the user's goals. By their very nature, we would expect them to be true for a period but at other times to give way to other goals.

User attributes can also change. For example, the user in Table 2.1 had low vision; at a later time, their sight may improve and this attribute of the user model should change correspondingly.

Finally, the existence of components may change. For example, a music recommendation system will need to model the user's music preferences. Since new music is being created all the time, a music preferences model must represent preferences for new music as it is created.

Inconsistency due to the source of external user modelling information

There are several different ways to deal with conflicting information about the user. Firstly, the sensors or monitors collecting the information may be unreliable or subject to noise. For example, if the user's identity is not certain, the user modelling system may actually collect information about another person. More subtle forms of a similar phenomenon occur when the user is indeed being observed but it happens that another person is telling them what to do.

Another class of difficulty occurs when the user makes slips, perhaps because they have a bad day or feel unwell, or have a temporary memory lapse. This could be regarded as noise in the user!

Yet another cause of conflict may be due to the user who is playful or curious about how the system. Such a user may try bizarre or random actions. Just this was reported in the case of the coaching system West (Burton and Brown, 1979) where some students made poor moves in the game to see how this affected the coaching actions of the system.

Even asking the user can be problematical. The user may not understand a question as it was intended. They may make a slip in their answer. They may be embarrassed to admit they do not know things.

Inconsistency due to internal inference

This is a particularly interesting source of conflict and has been the focus of considerable research in the systems described later in this chapter.

For example, suppose a user states they like Beethoven's Fourth Symphony. A music preference system may then infer that the user will like Beethoven's Fifth and Sixth Symphonies. Suppose that the user then says they loathe the Fifth Symphony. At that point, the system inference is in conflict with user's claim.

Another, somewhat different problem occurs if the user now states that they loathe the Fourth Symphony: we used this a *ground* assumption for our inferences about the other two symphonies. Intuitively, we would think the system should revise its beliefs about the Fourth Symphony and the inferred beliefs about the other two. At this point, a foundational reasoning system (Doyle, 1979, Martins and Shapiro, 1988) will revise all the beliefs where coherence reasoning (Harman, 1986) will not revise either the ground belief about the Fourth Symphony or the other inferred belief about the Sixth Symphony. Either way, some beliefs need to change. And such change may impact other components of the user model.

2.1.5 Broad applicability

This aspect defines a shell. The goal of building a shell is that we can reuse it over different domains with diverse user modelling requirements. Although this might appear to be a simple requirement, it is unlikely a single shell can support the full range of user modelling. To qualify as a user modelling shell, a system should be able to operate in different domains and manage at least some of the different classes of components we have described above.

In the user modelling work to date, there has been a strong divide. One important part comes from the student modelling community where the focus is on representing the learner's knowledge as well as other attributes and preferences that might support the teaching and learning. Another important part comes from broader user modelling research, much of this coming from the natural language dialogue work. In this, there is a greater need to model a variety of classes of components including for example, the user's preferences. A broadly applicable user modelling shell should be able to cross this divide. It should support user modelling in teaching as well as other user-adapted interaction. Sections 2.3 and 2.4 describe user modelling shells from both these communities.

2.2 Early generalised user modelling

This section describes two of the early pieces of seminal user modelling work. On our criteria above, these may not qualify as user modelling shells. However, they provide foundations for the user modelling that followed and are important for appreciating it.

2.2.1 Grundy: stereotype-based inference

One of the pervasive ideas in user modelling tools is the *stereotype*, generally acknowledged as having originated in Grundy (Rich, 1979, 1983, 1989) where people's descriptions of themselves were used to build a user model and then predict characteristics of books that they would enjoy. Grundy also operated as an advisor for the Scribe text formatter.

Rich defined stereotypes thus:

> A stereotype represents a collection of attributes that often co-occur in people. ... they enable the system to make a large number of plausible inferences on the basis of a substantially smaller number of observations. These inferences must, however, be treated as defaults, which can be overridden by specific observations. (Rich, 1989:35)

In Grundy, the user would give several words of self-description with information like

sex, age, occupation. For example, a user might say they are *athletic*. Grundy used this as a *trigger* for a large number of stereotypic inferences about the user. For the example of the athletic person, Grundy might infer they were likely to be motivated by excitement, have personal attributes like strength and perseverance, and are interested in sports. Each of these inferences had a rating indicating its strength. From this collection of inferences about the user, Grundy recommended books that matched these motivations and attributes. After making recommendations and allowing the user to respond to them, Grundy refined the user model by adjusting the rating on each component of the model.

Hierarchies of inherited stereotype components mean that, in the worst case, the system tends to assume a canonical average user (the typical built-in) when it has no information.

Grundy learnt its stereotypes as it met more users. The initial structure of the stereotype was defined by the programmer, but the rating for each inferred component evolved as the system met more users. Such learning has not been a part of later uses of stereotypes.

Rich introduced a three part taxonomy for user models and this has continued to be used. Grundy was *individualised* (rather than generic) since each user had an individual model. It was *long term* because the model persisted between interactions. Thirdly, it used *implicit* modelling of the user, rather than relying on explicit information from the user.

Grundy can be seen as introducing a general stereotype technique and exploring its application in modelling user preferences. Stereotypes of various types have been important in much user modelling research for user's knowledge and other attributes. For example, the Unix consultant (D N Chin, 1986, D Chin, 1989) used double stereotypes to reason from the user's actions to a classification of their expertise and from expertise level to a collection of assumptions about those aspects of Unix they are likely to know. A strength of the stereotype is its ability to reason on the basis of little input to infer the values of many components. It is also intuitively appealing.

2.2.2 UMFE: User Modelling Front End

UMFE (Sleeman, 1985) was the first user modelling system to use a series of implementor-defined inference rules to infer the concepts the user is likely to know. It also investigated some approaches to dealing with conflicts in the user model. The primary goal of UMFE was to support presentation of information in a form the learner should be able to understand. It expressed information in terms of concepts the user was assessed as knowing and offered simpler explanations to naive users. UMFE needed to model the difficulty of a set of concepts and the user's knowledge of them.

It operated in three main stages. It asked the user questions to establish their level of

sophistication. From this it updated the user model. Then it dealt with inconsistencies in the user model by asking the user further questions.

It had two main forms of inference, an explicit set of rules and implicit rules of the form

> if concept is known/unknown
> then conclude that another concept is known/unknown
> (with strength 1-100)

It also had implicit rules based on the concept's difficulty and importance. For example, if the user did not know a concept with concept of difficulty x, concepts of greater difficulty were inferred to be unknown. Similarly, if the user knew a concept of difficulty y, concepts of difficulty less than y were implicitly inferred as known. There was also a rating of the importance of a concept.

If a conflict arose, UMFE used the partial ordering:

$$\text{user} > \text{explicit rules} > \text{implicit inferences}$$

to attempt to resolve the conflict. When too many conflicts occurred, it did a 'witch-hunt' by asking the user all concepts relevant to their next query. To assess if a user knew a concept, it asked ''if you understand the significance of'' the concept. Parameters enabled or disabled implicit inference and set limits on allowable implicit inference errors and explicit inference errors. They also limited the number of levels of propagation of explicit inference rules.

UMFE's inference mechanisms differ from stereotypes in that they are based upon knowledge-based reasoning which captures the structure of the domain. This form of reasoning is especially important for epistemological reasoning as we will see in the systems described in the next two sections.

2.3 User modelling shells

We now describe major user modelling shells. Each shows its roots so we describe these as well as the approaches and techniques explored. This section is restricted to those shells that have come from the user modelling community (rather than the artificial intelligence and education community). These shells are important for this thesis in several ways. Firstly, they represent a range of approaches for user modelling shells. Their design was driven by various goals that differ from our concern for scrutability. We will argue that this has meant they cannot support scrutability where the user can delve into the user model to find answers to questions like those posed in the scenarios in Chapter 1. Nonetheless, they give valuable foundations for our understanding of the

scope and challenges in creating a user modelling shell. We will refer back to these systems in later chapters, when we explain the design for our scrutable user modelling shell.

2.3.1 GUMS

The Generalised User Modelling System, GUMS (Finin, 1989, Kass, 1991) was based on stereotypes containing sets of facts and rules. It maintained a collection of databases, one for each application it serviced. GUMS could accept new modelling information from an application program or a request for modelling information. In this sense it acted as a database of modelling information. It was applied in a model of the user's knowledge of an operating system and associated editing and programming tools.

GUMS stereotypes were represented in an inheritance tree hierarchy. Higher level stereotypes were more general. This representation was simpler than Grundy's in that it did not allow multiple inheritance. Each user's model was attached to one stereotype within the hierarchy.

Unlike Grundy, GUMS distinguished two types of facts in stereotypes, *definite* and *default*. The former are the essential elements that must apply for all users in the class. For example, a programmer stereotype could only apply for a person who programs. By contrast, default facts act as initial beliefs.

GUMS used a four valued logic: true, false, true-by-default and false-by-default. When GUMS received new evidence that was inconsistent with the existing user model, it resolved the conflict using a partial ordering:

true, false > true-by-default > false-by-default

(the last reflecting that inability to establish a fact is weak evidence for its falsehood).

A stereotype contained: definite facts and rules, default facts and rules; and meta-level knowledge. A definite rule of the form (P if Q) meant that if Q was true, so was P. If Q was true-by-default, so was P. By contrast, a default rule (P if Q) meant that if Q was either true or true-by-default, then P was true-by-default (meaning that P could at best be true-by-default, rather than true).

This means that GUMS had three default reasoning methods: stereotypes for generalisations about large classes of users, default rules to express stereotypic norms that vary for individuals and, finally, negation as failure when nothing else applied.

Since stereotypes had definite parts, new information about a user could require their reclassification. For example, if at one point in time, a user was classified as a

programmer and later evidence indicated the person could not program, the system recognised that a definite fact for the programmer stereotype had been violated. When such a conflict indicated that the current stereotype was incorrect, GUMS moved up its stereotype hierarchy to successively more general stereotypes until it reached one that was correct. This corresponds to giving up default stereotype information as more information about the person becomes available.

In summary, GUMS acted as a database of user modelling information. It introduced the distinction between those parts of the stereotype that define it and the default beliefs about a user. It also provided a means for reasoning about the models, based on stereotypes and rules for reasoning about them. For resolving conflicts it used a partial ordering on its four-valued logic. The individual user was represented as a node in the stereotype hierarchy. GUMS represents a refinement and exploration of the stereotypes for user modelling.

2.3.2 GUMAC

The Generalised User Modelling Architecture, GUMAC (Kass, 1991), was developed in the context of a natural-language-based system for giving financial advice. Where GUMS focused on representation and maintenance of user models, GUMAC was concerned with model acquisition based on implicit rules. A primary motivation for this work was the considerable cost of building user models like those in GUMS. The aim was to deduce useful modelling information from what the user said in a natural language interaction with the advisor.

An example of a simple rule is *relevancy*:

- if a user says P,
  the user believes P (in its entirety) is relevant to reasoning about the current goal(s) of interaction.

A more complex rule concerned *omissions*. This operated within the context of a dialog when the system expected the user's response to include a particular piece of information or response. If the user omitted it, GUMAC considered several explanations - the user:

- was unaware of the system's goals and the dialogue had run off the rails;
- failed to appreciate that this expected response was relevant to the system's goals;
- did not realise this response was necessary to achieve the system's goals;

- did not know how to make this response, as for example, when they were ignorant of a piece of information.

Like UMFE, it used its knowledge base to generate additional user modelling information. For example, the *preconditions* for knowledge were used; GUMAC assumed preconditions when the consequent was known. Similarly, GUMAC used reasoning based upon a hierarchical representation of the domain to deduce the user knew a more general concept where they knew several specialisations of that concept.

GUMAC could accept requests from a consumer program about a user's model. These took the form ''Bel(U,P)'' meaning ''does the user U believe P?''. GUMAC responded with one of: yes, U believes P; no, U does not believe P; no information. The justifications for these values were also maintained within GUMAC.

GUMAC developed rules to exploit the structure of the dialogue as well as the knowledge base relevant to the dialogue. These were then used to enhance the user model.

A major contribution of this work is the exploration of domain independent rules for implicit acquisition of modelling information. This work emphasises the importance of supporting flexible forms of implicit reasoning mechanisms in a user modelling shell. Some of these have been adopted in other user modelling shells we will describe.

2.3.3 BGP-MS - Belief, Goal and Plan Maintenance System

BGP-MS (Kobsa, 1990, 1992, Kobsa and Pohl, 1995) was first developed in association with a natural-language system for interacting with expert systems. It has been used in several domains, including, for example: dietary advice in pregnancy; ergonomic kitchen layout; in a simulation of citizen action committees and city administration officials dealing with routing of a new road; and a hypertext whose content is adapted to the user (Kobsa, 1990, Kobsa, 1992, Kobsa and Pohl, 1995). It is intended to represent a range of user modelling information and to give sophisticated support for the construction of the modelling component in a particular domain.

The basic elements of the user model are *concepts* and these are represented in an inheritance hierarchy. Each concept is described by a four-tuple: a role predicate for each relation this concept participates in; value restrictions on the arguments of each relation; a number of restrictions indicating how many of the attributes are required for an instance of this concept; a modality to indicate whether an attribute is necessary or not. Such concepts are kept in *partitions* which themselves can be organised into inheritance hierarchies. These provide a representation for alternate views of knowledge.

The SB partition represents those System's Beliefs not shared by the user. Another

partition is UB which holds User's Beliefs not shared by the system and yet another is ShB, for the Shared Beliefs of the user and the system. This means that UB contains incorrect or non-normative user beliefs. It is also possible to represent nested beliefs (indicating what the system believes the user believes the system believes ...). In addition to these partitions, BGP-MS represents partitions like SBUW for System's Beliefs about what the User Wants.

BGP-MS supports a sophisticated stereotype mechanism which builds upon the stereotype work we have already described. The stereotypes can be constructed with the aid of a graphical tool. This helps the system builder see the relationships between the various stereotypes. The system also checks the consistency of the structures created. Each stereotype has *selection* conditions that control its activation. These are defined in rules, as are the conditions for *retracting* a stereotype when too many of its assumptions prove false. Rules can be written as arbitrary functions as well as in terms of tests. These can check whether the user knows selected concepts (or their attributes). There are also parameters defining the number of active stereotypes allowed and controlling the willingness of the system to activate new stereotypes or retract existing ones whose controlling conditions are only partially met.

Where there are conflicts between stereotype-based assumptions and other sources of information, the latter apply. Conflicts arising from two stereotypes are more difficult to resolve and some of this is specified by the programmer.

Kobsa has implemented eight classes of implicit inference rules: three domain independent rules that relate to transitivity of which two are from Kass (1991); domain dependent rules like those in UMFE (Sleeman, 1985); three relating to natural language and taken from work on KNOME, the user modelling for UC, the Unix Consultant, (D Chin, 1989); and another from Chin reflecting that using a command properly indicates the user knows about this command.

There is a tight coupling between BGP-MS and the consumer system with communication based on messages. Their form is illustrated in the following examples taken from Kobsa (1995):

- bgp-ms-tell enables an application to tell BGP-MS about the user as in
  bgp-ms-tell (B S (B M (B U (B S (iota x (GIVES x MARY BOOK))))))
  meaning it is mutually believed that the user believes the system believes it knows who gives a book to Mary;
- interview-next-question-block provides the consumer system with the form of questions to ask the user and interview-response which accepts the user's answers to those questions;

- d-act reports user actions as in

  d-act AGREE ((erase disk)))

  where the user action at the interface showed agreement that the disk be erased or

  d-act PRINT-COMMAND (userdoc))

  means the user printed ‘userdoc’.

- bgp-ms-ask and bgp-ms-answer allows the consumer system to ask for information about the user and receive an answer from the user model as in

  bgp-ms-ask (:id 116 (B S (W U (printed-on userdoc lw+))))

  meaning the consumer system wants to know if the user wants to print ‘userdoc’ on printer ‘lw+’ and if this is so, the answer would have the form

  bgp-ms-answer (:id 116 : answer yes)

- alert-from-bgp-ms signals important events, for example

  (alert-from-bgp-ms :retracted-assumption
  (B S (B U (printable-on userdoc lw+))))

  warns the user model no longer believes the user believes ‘userdoc’ can be printed on lw+.

BGP-MS is an on-going project with considerable activity to extend and enhance it. There has been the addition of modal logic and enhancements to the partition hierarchy (Kobsa, 1992) enabling it to represent negative beliefs (that the user does not hold a particular belief), disjunction, implication and quantification. A similar Prolog-based shell, PROTUM (Vergara, 1994), has also been developed. There has been work towards a distributed architecture (Fink, 1996) so that different consumers can access the same user modelling information at the one time, the consumers can be distributed across different systems and group models can be supported. The flexibility has been increased by allowing user model developers to define new modalities and assumption types (Fink and Hohle, 1997) so that, for example, the programmer can specify a new assumption type I for modelling user’s interests. Also, support for modelling the user’s plans has been proposed (Kupper, 1997). BGP-MS is also tailorable so that parts of it can be deactivated if the user modelling application does not require them.

In summary, BGP-MS supports sophisticated knowledge representation and reasoning. It also has a helpful graphical interface for constructing the knowledge bases. In a sense, this is a form of scrutability support for the programmer of the system. BGP-MS offers considerable power and flexibility in its support for user modelling.

2.3.4 UMT

UM-tool (Brajnik, Guida, and Tasso, 1990) was developed to support a natural-language interface to several computer science bibliographic databases. UMT is an enhanced form (Brajnik, Tasso, and Vaccher, 1990, Brajnik and Tasso, 1992, 1994) with refinements to the non-monotonic reasoning of UM-tool. UMT has been used in a tourist advisor which, in addition to the advice, displays parts of the user model used to select that advice. The user assesses advice and can also indicate disagreement with the model used to generate it. This is important for this thesis as it is an example of limited support for the user's scrutiny and control of their user model.

UMT has the following main components:

- a database of all the user models held by the system;
- a knowledge base of stereotypes in a multiple inheritance hierarchy;
- the database of possible *user models* for the current user;
- a *rule-base* defining *constraints* on values of attributes in the user model and *inference* rules for generating new user modelling information;
- a *consistency* manager which uses an ATMS-like mechanism to maintain a collection of possible user models and a knowledge base of *choice criteria* for selecting one of the possible models based on the reliability of the sources of modelling information;
- and controlling all of these is the *model manager* which also interacts with the *observers* contributing modelling information and *consumers*.

Each of the possible user models is a self-consistent view of the user. The models are mutually exclusive representations of the user based on the information available to-date. The user model contains several forms of information: individual facts supplied by *observer* programs; defaults from the active stereotypes; and inferences from modelling rules.

UMT's stereotypes represent default assumptions about users. A *generic stereotype* is associated with each application program. This captures relevant attributes for all users of that application. Like Grundy, UMT has successively more specialised *class stereotypes* where each captures the characteristics of that particular class of users, and any person can belong to several of these classes. If the trigger for a stereotype is true or its child in the hierarchy is true, the stereotype is deemed *active*. At any point in the interaction with a user, as a stereotype becomes active, the default values in that stereotype are added to the user model.

When each new value is added to the user model, the consistency manager assesses it. Where there are inconsistencies, the defaults of a stereotype are not included in the user model. This means that active stereotypes (those whose triggers are true) can have many, perhaps all, their default assumptions overridden by information from more reliable sources. Upon each modification to the possible user models, UMT runs all relevant rules, deals with all known inconsistencies, checks the triggers on stereotypes to update the list of active stereotypes. It keeps the set of all the self-consistent possible user models and selects from these the *current user model*.

UMT also supports a development interface which provides progressive displays of the user model, the operation of the consistency manager and the way that sources of information have been used. The system engineer can also override the system operation to see the effect: for example, they might select a different model from the set of possible models. UMT is a sophisticated user modelling shell which has powerful truth maintenance mechanisms and stereotype management. It provides the programmer with some scrutability support in the programmer's development interface. It also has limited scrutability support so that the user can see and alter the values in the user model. There is no user access to the processes that formed the user model values.

2.3.5 Doppelganger

Doppelganger (Orwant, 1990, 1993, 1994, 1995) represents a considerable departure from the other shells described. It grew out of work to create newspapers customised on the basis of user preferences and interests. Its core is

- a user model server with a centralised database of user models,
- many sensors collecting information about the user,
- a toolkit of learning techniques to extrapolate from sensor data and predict future values for the user model.

Sensors run asynchronously from the rest of the system, reporting information to Doppelganger via a TCP/IP connection. As information arrives, Doppelganger offers it to the machine learning tools. Each machine learning element that is able to interpret the new sensor data adds its conclusions to the user model.

Models are stored in a Lisp-like special purpose language called SPONGE. Each user model is a Unix directory containing two types of models. The main 'domain models' hold information likes the user's preferences for news. In addition, 'conditional models' can be used to override domain models at particular times: for example, the user's special preferences for news early in the morning.

As models are stored centrally, privacy is a particular concern. Doppelganger uses a Kerberos authentication system to protect privacy. Access to parts of the model are managed by the Andrew file system to make them accessible at one of the following levels: private, public, private except for listed users, public except for listed users. It is also possible to store the model on PCMCIA cards and remove the data from the server. Since centralised modelling could pose problems for scalability, Doppelganger allows for distributed models with the servers communicating as necessary.

The focus is on unobtrusive modelling, with many sensors collecting varied information. For example, Doppelganger tracks the user's location on the basis of sensors such as 'active badges' and 'smart chairs' as well as analysis of the user's activity on the computer. Doppelganger has an accuracy estimate for each sensor and adjusts this over time.

One of the advantages of a centralised database of user models is that Doppelganger is able to apply unsupervised clustering techniques over the models to learn about 'communities'. For example, it can analyse all models by the gender of the user and define the common characteristics of the 'community of female users' and the 'community of male users'. In a similar way, Doppelganger can build communities for groups like artists, members of the Media Lab, children or students.

Communities can be used in a similar role to stereotypes. For example, suppose that all aspects we know about the user match particularly well with one community. Then Doppelganger can confidently make default assumptions about unknown aspects. For example, if known aspects about the user matched well with the community for children, and also matched well with the student community, unknown values could be predicted by combining the values in both these communities.

Communities are similar to stereotypes in that they permit prediction of default values for the user model. They differ from the stereotypes in that they are defined from a collection of user models and are dynamic, being recomputed each night. Perhaps more significantly, a user has probabilistic membership, matching some communities better than others.

For our scrutability focus, a significant feature of this work are the three interfaces which help the user see parts of the user model.

- To depict the model of a person's movements between rooms, it displays the underlying Markov model as a graph with rooms as nodes and arcs connecting rooms the user moves between. It also highlights rooms where the user spends most time.

- It shows the system's record and predictions for the time between logging into the computer in terms of a bar graph. The first two thirds of the time period displayed shows actual observations and the rest gives predicted activity. The colours in the display fade for times further in the future.
- For the news preferences, there is a four part display. The first part is a series of sliders for the individual's preference for aspects modelled: for example a long slider against the Japan preference indicates being very interested in news about Japan. Another set of sliders shows abstract parameters like preference for broad news coverage. The third set shows similarity with other users, a long slider indicating very similar users. The last set of sliders show membership of communities: a person who shares much in common with members of the Media Lab with have a long slider against that community.

The user can also interact with the user model via email messages. For example, the following mail message (Orwant 1994:153):

> From: orwant@work.media.mit.edu (Jon Orwant)
> Subject: news
> I would like more sports

gives the email response:

> I was able to parse your message.
> Bolstering you preferences for sports from .2/1 to .34/1. with confidence .8.

Significant features of Doppelganger are its server-based architecture, the role of the sensors, use of machine learning, communities and its exploration of interfaces for depicting aspects of the user model.

2.4 Student modelling shells

There is much commonality between student and user modelling as has been argued (Brusilovsky, 1993, Brusilovsky and Schwarz, 1997). Indeed, our five essential characteristics for user modelling shells apply in both. The systems in this section could well represent many of the same forms of user models as we saw in the user modelling shells.

At the same time, much of the literature from the student modelling community is disjoint from that of the broader user modelling community. There are important differences in the emphases of user and student modelling research. Some issues appear to be more important in student modelling.

- There is a greater focus on modelling knowledge, as distinct from preferences and goals.
- There is a further distinction between knowledge and misconceptions. We have already seen this in BGP-MS (Kobsa and Pohl, 1995) which distinguishes beliefs shared by the user and the system (agreed knowledge) from those held by the user but considered incorrect by the system (‘misconceptions’).
- An important sub-part of student modelling concerns *executable* models.
- Some student models are intended to have cognitive validity. For example, a model might simulate the reasoning of the student, including their buggy reasoning (Brown and Lehn, l980).

In assessing the breadth of applicability of a user modelling shell, we need to assess whether these aspects can be supported.

A student model is commonly categorised as:

- an overlay model (Carr and Goldstein, 1977): student knowledge is modelled as a strict subset of an expert’s model;
- a differential model (Burton and Brown, 1979, Clancey, 1987): focuses on the differences between the student’s apparent knowledge and that of an expert;
- a bug model: represents misconceptions and may be based on empirical study of student errors (Brown and Burton, 1978, Soloway, Ehrlich, Bonar, and Greenspan, 1982).

A user modelling shell with broad applicability should be able to support this range of student models.

2.4.1 TAGUS

TAGUS (Paiva and Self, 1995) was created as a user and learner modelling server in the context of intelligent teaching systems. It provides support for the full range of user modelling tasks: acquisition of observations of the user; generation of hypotheses to explain the user’s behaviour; changing the model in response to new information; providing consumers with information about the user; and particularly interestingly, supporting simulation of the learner’s reasoning.

The main elements of its architecture are:

- ULM, the underlying user or learner model, represented in an internal language that captures the user’s beliefs, goals, problem solving capabilities and strategies;

- core maintenance functions for adding to model, deleting from it and updating it;
- an acquisition subsystem;
- a stereotype subsystem with a taxonomy of stereotypes and mechanisms for classifying the user;
- a meta-reasoner for interpreting the ULM and reasoning within the model, including the execution of ULM rules that simulate the user's reasoning;
- a reasoning maintenance system, AMMS (Agent-Model Maintenance System);
- a communication module, which provides services to outside world;
- two interfaces to the model.

The ULM can be described on three levels: belief, reasoning and monitoring. The first represents first-order formulae to represent aspects of the user. For example

planet
earth
moon
planet(earth)
revolves-around(moon, earth)

reflecting that the user knows about the notions of planet, earth and moon, believes the earth is a planet and the moon revolves around the earth. The second level simulates the way the user might reason. For example, the user might apply *modus ponens* to infer additional knowledge. TAGUS also models the user's monitoring behaviour such as giving up when a task is too difficult.

The role of the TAGUS acquisition subsystem is quite broad. It manages all new information as it arrives, creating any additional hypotheses required to explain this new information. Then it adds all these to the ULM, while maintaining consistency. It also has a stereotype acquisition engine and rule-based diagnosis. So, for example, the stereotype subsystem can classify the user. Then the acquisition subsystem might add inferences from active stereotypes to the ULM. It has acquisition rules like: if a learner says X, in a particular situation, that fact should be added to the model.

The ULM distinguishes beliefs that are directly acquired from the user's behaviour from those that are internally inferred from stereotypes and acquisition rules. It also places a partial ordering on the reliability of the acquisition rules. This is important for dealing with conflicting information and noise in the process of updating the ULM. For example, when new information about the user is acquired, the acquisition subsystem applies stereotype-based reasoning. Activating the stereotypes may well generate many potential

models; TAGUS must select the best. The AMMS keeps the bases for each belief and uses them in truth maintenance.

Given our concern for scrutability, TAGUS's two interfaces are particularly interesting. These are designed for 'special users', such as the system developer. To interpret the displays, one requires considerable understanding of TAGUS.

In summary, TAGUS is a server for user and learner models, and it offers sophisticated reasoning and consistency maintenance. TAGUS can store models for several learners/users at the same time.

2.4.2 THEMIS

THEMIS (Kono, Ikeda, and Mizoguchi, 1992, 1994) focuses on inconsistency, especially the case where the student changes their mind or holds contradictory beliefs. It has been applied in teaching systems for geography and chemical reactions (Ikeda and Mizoguchi, 1994, Mizoguchi and Ikeda, 1991). It tackles the diverse causes of inconsistency affecting student modelling which the authors describe thus (Kono et al, 1992:451):

> No one has complete access to one's knowledge at all instances. Therefore, no one can guarantee the consistency of one's knowledge or reasoning processes. A person is apt to be ignorant about inconsistencies of his/her knowledge, and tends to arrive at conclusions which cannot be logically derived from his/her knowledge.

THEMIS distinguishes the four types of student model inconsistency listed in Table 2.2. Category A comes from student answers to set questions. So, for example, when the student attempts a task, their performance is assessed and this feeds directly into the student model. Category B is the consequence of the internal inferences. The last column shows that THEMIS treats most inconsistency as a problem to be addressed in a single world. In these cases, an ATMS belief revision process removes the inconsistency. For A3, the model needs to retain the inconsistencies and it does this by representing the student beliefs as multiple worlds. Within any one world an ATMS is used to maintain consistency.

The major parts of THEMIS are:

- HSMIS - Hypothetical Student Model Inference System which manages single world assumptions (and contradiction types A1, A2 and B);
- the concept discrimination tree plus multiple world model for contradictory beliefs, type A3;

Table 2.2. Types of inconsistency for student modelling in THEMIS

Type	Description	single- or multiple-world resolution
A	**Due to student**	
A1	change in student eg learning	single
A2	slips, careless errors	single
A3	contradictory beliefs held	multiple
B	**Due to system inference from changed ground beliefs**	single

- domain dependent heuristics to distinguish different classes of inconsistency and select the right mechanism to deal with it.

HSMIS describes the student model as a program in SMDL, the Student Model Description Language (Mizoguchi and Ikeda, 1991), an extended Prolog with four truth values: true, false, unknown or fail.

We now characterise the management of the various forms of inconsistency. The concept discrimination tree and multiple worlds reflect a two stage problem solving model:

1. decide which knowledge-world is relevant to this problem (for example, naive physics or formulae from school physics);
2. within that world, solve the problem.

To make the decisions of the first stage, THEMIS uses a *concept discrimination tree*, essentially an hierarchical structuring of concepts needed to select the world to use for this problem. The nodes of the tree are concepts used in the decision process and the leaves are sets of world knowledge. THEMIS generates a tree to match the student's problem solving behaviour, and this may match incorrect answers with a buggy structure. For example, the selection of a world may involve an over-generalised condition over concepts.

The leaves of the concept discrimination tree manage the second stage in the problem solving process. This can also model buggy knowledge. However, the beliefs within each world are internally consistent. Because the user's discrimination tree may model several worlds, the learner's inconsistent beliefs are separated into different worlds.

The third part of the system is the set of heuristics for identifying the type of inconsistency. For example, it presumes type A1 if relevant tutoring was done just before an apparent change. It decides on A3 if each contradictory belief has high certainty measures. Once this has been decided, the appropriate single or multiple world reasoning mechanism is activated.

In summary, THEMIS makes a contribution to the management of inconsistency. It

models the user who holds inconsistent beliefs, as is especially common when learners have partial grasp of concepts. At the same time, it manages more conventional sources of conflict in the user model.

2.4.3 SMMS - Student Modelling Maintenance System

The Student Modelling Maintenance System, SMMS, (Huang, McCalla, Greer, and Neufeld, 1991) was developed to model the student's knowledge, and it was used for teaching Lisp programming. It offers a similar range of facilities to those in general user modelling shells we have described. It is important for its exploration of the use of belief revision. It aimed to deal with conflict efficiently, by taking care with culprit selection: this is the problem of altering an inconsistent model so that it becomes consistent and so that minimal changes are made to the database. Its operates in a cycle:

- collects new observations and associated beliefs about the student;
- accepts these and updates the deductive knowledge base;
- performs revision to ensure consistency;
- checks active stereotype hierarchy
- activates or deactivates stereotypes.

SMMS has different representations for deductive and stereotypical knowledge. The former are modelled with justified beliefs and the later in a stereotype hierarchy.

Revision of beliefs is based on reason maintenance and formal diagnosis. This can also cause revisions of the stereotype knowledge in the latter steps of the reasoning cycle. Belief revision is performed by EBRS (Evolutionary Belief Revision Systems). This is a foundational revision system. So it holds only those beliefs with a current basis: either because they are 'base beliefs'; or because they are still valid 'derived beliefs' that can be deduced from current base facts. There are two forms of base beliefs: beliefs based upon observations; and the system's own inference rules.

The hierarchy of stereotypes is called the Default Package Network (DPN). This is a directed acyclic graph with lower nodes being subdomains of their parent. Each subdomain node holds a set of concepts and misconceptions, each represented as a propositional formula. Each of these has a value {novice, average, expert, unknown}. New stereotype inferences are caused by a propagating activation algorithm and deactivation is effected by a simple constraint satisfaction algorithm.

SMMS represents an exploration of the use of belief revision for modelling a learner's knowledge. It makes a clear distinction between the roles of deduction and stereotypic

inference and cleanly separates them.

2.5 Discussion and summary

The systems we have just described illustrate many of the common elements of user modelling shells as well as the different focuses and approaches of each system. None of these systems was created as a scrutable user modelling shell. We now review their approaches to the various tasks of a user modelling shell and discuss the way that scrutability concerns would affect those tasks. First, however, we review our scrutability goal in terms of the types of questions that a user might try to answer by scrutinising their user model. The first questions in the scenarios of Chapter 1 were:

- How did it know that I didn't know about undo?
- What else does it think I know (or don't know)?
- How did it decide I would be interested in a Flipper movie?
- Or a Carr-Boyd release?

Questions such as these highlight the need for *long term modelling*, as described in the Rich taxonomy (Rich, 1983, Rich, 1989). Most of the user modelling shells have focussed on *short term modelling*, perhaps limited to a single session. Systems such as those in our Chapter 1 scenarios operate over the long term, acquiring more information about the user over long periods of time.

We now discuss the five essential properties of user modelling shells, described at the beginning of the chapter. We review each in relation to the systems we have just described and scrutability concerns.

2.5.1 Scrutability and representation of user model components

Since the construction of user models can be a complex knowledge engineering task, it is not surprising that shells like BGP-MS, UMT and TAGUS which offer more sophisticated representations, have interfaces to support this task. These help the knowledge engineer formulate representations correctly and appreciate the larger scale knowledge structures. In a sense, these constitute support for scrutability; to build and modify the user model elements, the knowledge engineer needs to scrutinise them. However, in these systems, such scrutability support is for the *knowledge engineer* rather than the *user* being modelled.

As we have already mentioned, it is also natural that some consumer systems display some user model status information as in the case of UMT (Brajnik and Tasso, 1994).

Beyond this, Doppelganger has provided some tools to enable the user to see some aspects of their user model. It can accept email requests for changes to the user model. It uses email to report the model state as a natural language rendering of the contents of the model. In addition, the SPONGE representation is available to the user. This means that a user who can read the Lisp-like SPONGE representation could scrutinise their model. This is clearly not a form of scrutability that is useful for the average user. Nor was SPONGE designed for scrutability. It is unclear whether its organisation of information would make it easy to see what processes contributed to the values of the user model components. In this sense, it may be hard for even the system designer to understand a large and complex SPONGE model. In this sense, the TAGUS and BGP-MS interfaces for the knowledge engineer are likely to give greater scrutability support, at least for the knowledge engineer.

In summary, the user modelling shells described have some support for scrutability of the model. But none has been designed to support scrutability so that the 'ordinary' user can explore what is represented in the user model and so to answer questions of the form 'what does this system know about me?'.

2.5.2 Scrutability and external user modelling information

All the user modelling shells accept external evidence about the user. All but Doppelganger treat external information sources homogeneously, typically with each instantiation of the user modelling shell running within a single environment. The most common scenario seems to have the user model consumer as the primary contributor of external information about the user.

Part of scrutability includes access to the *processes* of user modelling so that the user can answer questions of the form 'How does the system come to believe this thing about me?'. So, for example, a user might be asked to attempt a series of maths problems over a period of time. If the user consistently gave incorrect answers to the series of problems, the shells we have described have facilities which could make the user model the user's ignorance of the aspect(s) tested by the problems. However, the user modelling shell systems do not enable the user to explore the way this process formed the user model. To support such scrutiny, the user would need access to the record of that series of incorrect answers to the set problems.

2.5.3 Scrutability and support for internal inference

This has been an important aspect of the work on user modelling shells. So, GUMS (Finin, 1989, Kass, 1991) refined stereotypic reasoning, GUMAC (Kass, 1991), established various forms of internal inference and BGP-MS (Kobsa, 1992, Kobsa and Pohl, 1995) has sophisticated knowledge representations that can support both of these forms of internal inference. Similarly, TAGUS (Paiva and Self, 1995) has sophisticated support for internal reasoning within the student model.

When we consider the issues of scrutability in relation to internal inference, we again see that none of the shells provides the user with access to the *processes* of internal inference that formed the values of the user model. Consider again the example of the teaching system where the student was asked a series of maths problems. It may be that the student's incorrect answer to a very easy problem was used to infer that the student was a rank novice in this area. It may have been further inferred that the student was unlikely to know any of the difficult aspects in this area. If the student is to be able to scrutinise their user model, they should be able to determine: that the system has concluded they do not know these difficult aspects; and that the process we have just described formed this conclusion.

Importantly, the user modelling shells have helped refine our understanding of the role of two major forms of internal inference, stereotypes and knowledge-based reasoning. The distinction is that stereotypes are generally regarded as low reliability defaults. Although there are different strategies for managing stereotypes, their wide use indicates they are a powerful basis for user modelling. They generally serve as default assumptions about the user. Scrutability concerns also arise here. There is an intuitively clear difference between stereotypic and knowledge-based reasoning. A scrutable user modelling shell should ensure that the user can scrutinise the reasoning processes to appreciate where each approach has been applied.

2.5.4 Scrutability and management of consistency

This has also been an important theme of several of the user modelling shells. UMT and TAGUS explored the use of truth maintenance and SMMS (Huang, McCalla, Greer, and Neufeld, 1991) similarly explored belief revision while THEMIS (Kono, Ikeda, and Mizoguchi, 1992, 1994) refined the notion of inconsistency and applied different approaches for its different forms. At the same time, none of the work has addressed scrutability of this aspect.

In supporting scrutiny of management of consistency, we enable the user to answer process questions of the form 'How does the system come to believe this thing about

me?'. Consider our example of the student who has done a series of maths problems. Suppose that after this student had answered the first series of problems incorrectly, they started to answer questions correctly. Suppose that later answers were consistently correct. The shells we have described have various ways to tackle the change in the values of parts of the user model as the user changes. Scrutability of the management of consistency would enable the user to see how the system might have used its conflicting information about the user to conclude whether they knew a particular concept at a particular time.

Our scrutability concerns and the associated need to focus on long term user modelling have an important impact on consistency management. In most of the systems we have described in this chapter, it made sense to deal with consistency of the model upon the arrival of each new piece of external information. This holds only if we assume the need to use the model at that point in time. In long term user modelling, where the user model is independent of any particular consumer and where its information may be reused by various consumers, it makes sense to defer costly reasoning about consistency until the time that the user model is needed.

For the systems we have described, the most common form of truth maintenance is a TMS-based approach (Doyle, 1979, Kleer, 1986, Kleer, 1987). This approach has some appeal for a scrutable system since it maintains information on the *justifications* for reasoning about the truth or falsehood of a component in the model. However, none of the systems have explored this possibility.

2.5.5 Scrutability and broad applicability

The breadth of applicability of a user modelling shell should be enhanced if it can offer power, flexibility and reuse of user modelling information. Scrutability gives the additional requirement that the user can still seek answers to questions about what is modelled and the processes underlying the conclusions in the user model. One can expect some tension between the goals of power and flexibility on the one hand and of scrutability on the other: if the 'ordinary' user, rather than the knowledge engineer, must be able to scrutinise their model effectively, that might limit the power of the user modelling shell.

One dimension of flexibility is the types of user model components that can be represented. Table 2.3 indicates some of this variety with examples from the systems we have described. We see that they include all the forms described at the beginning of the chapter: preferences, knowledge, goals and other user attributes. For scrutability, we are concerned with being able to explain what each component of a user model means as this

Table 2.3. Examples of components

Thing modelled is whether the user ...	**System**
is an athletic person	GRUNDY
is motivated by excitement	GRUNDY
has attribute of strength and perseverance	GRUNDY
is interested in sports	GRUNDY
knows the concept *otitis media* (with strength 75)	UMFE
has a goal to invest 10,000 dollars	GUMAC
wants to print a document	BGP-MS
believes that inkjet printers are printers	BGP-MS
and system have a mutual belief that Peter gave Mark a book	BGP-MS
nationality is French	UM-tool
has general interest in literature	UM-tool
has the name *Jon Orwant*	Doppelganger
is interested in the Olympics	Doppelganger
likes the source *USA today*	Doppelganger
knows about the notions of a planet and its moon	TAGUS
believes the earth is a planet	TAGUS
believes that moons revolve around a planet	TAGUS
believes the moon revolves around the earth	TAGUS*
believes Paris is in a torrid zone	THEMIS
believes the Lisp *car* function returns a list containing the first element of the given list	SMMS
knows the concept of recursion	SMMS

is part of the answer to the question ‘what does this system know about me?’ Examining the descriptions of aspects in Table 2.3, we see that this range of components can be described quite simply as in this table.

The design of each of the user modelling shells has been influenced by the modelling tasks it had to support. So, for example, BGP-MS has a rich representation of domain knowledge and TAGUS supports sophisticated simulations of the potential reasoning of the student. Our concern for scrutability will influence our choice of knowledge representation and mechanisms for inference and management of inconsistency. At the same time, this will be tempered by the need for breadth of applicability.

Chapter 3

Requirements and architecture

This chapter establishes the requirements for scrutable user models and associated user modelling tools. This is a critical first step in the design of our scrutable user modelling shell, which we call um. Our concern for scrutability has led us to distinguish two sets of requirements on um:

- *model* requirements ensure that um provides the user modelling shell's essential properties, described in Chapter 2;
- *scrutiny* requirements provide the framework for the user to scrutinise their user model.

Essentially, the *model* requirements take the perspective of the *programmer* who employs um for the user modelling needs of a um-consumer. By contrast, the *scrutiny* requirements focus on the needs of the *user* who is modelled. In our description of um requirements, we tag each with either *model* or *scrutiny* to clarify the role it serves.

Once we have described the requirements on um, we present the um *architecture*, then an overview of the *representation* in terms of an example user model and an overview to our scrutability support in terms of an example *scrutability support* interface.

3.1 Requirements

We now define the requirements for um is terms of the five essential properties of user modelling shells introduced in Chapter 2. The remainder of this section explains the requirements and their rationale, illustrating these from the following scenario.

> Several years ago we started modelling Linda Sculthorpe's preference for a particular piece of music, Carr-Boyd's Prelude. At that time, Linda was observed to start playing it on her digital music system and then stop it. She also used a rating interface and assessed it as *dreadful*. At this point, our user modelling system concluded that she disliked Carr-Boyd's Prelude.
>
> Starting a year ago, Linda has been observed playing this piece regularly. The user modelling system now concludes that she likes it.
>
> Recently, Carr-Boyd wrote a fugue. The user modelling system has no direct information that Linda has heard it or has any views about it. So the system has no external information about her preferences for it. Experts consider it very similar to the Carr-Boyd Prelude. So the user modelling system infers Linda probably will like the new Carr-Boyd fugue.
>
> At this point, two user model consumers make use of the model. One is an entertainment system which recommends new music. The other is a teaching system for modern Australian violin music.

3.1.1 Representation of user model components

The user modelling shell must represent a set of components. This establishes the *ontology* of a user modelling enterprise. The two requirements for representation of the components are shown in Table 3.1 as *model_ontology* and *scrutiny_ontology*. As we have noted, *model* requirements come from the perspective of the programmer and the um-consumer systems and *scrutiny* is for scrutability requirements that will enable user to scrutinise their models.

The *model_ontology* requirement calls for representation of the basic aspects which are to be modelled. We saw some of the range of different components in Table 2.3 which is a collection of examples from systems described in Chapter 2. We require that um be able to support modelling of such aspects of the user.

The corresponding scrutability requirement calls for an explanation to be associated with each component. We need this if the user is to be able to make sense of their user model as they scrutinise it. For example, some of the components that might be needed for the

Table 3.1. Requirements for representation for the user model components

model_ontology: $user_u$'s user model M_u is a set of components $M_u = \{component_c\}$
scrutiny_ontology: each $component_c$ has an explanation $explain\_component(component_c, [user_u], [consumer_a])$

above scenario include Linda's preferences for

- the Carr-Boyd Prelude,
- the new Carr-Boyd fugue,
- modern art music.

We can expect these three components to be part of a large model, with many components. Linda can only scrutinise her model in a meaningful way if she can determine what each component means. So, for example the model's internal representation for the Carr-Boyd Prelude may be a component called ''CBP_000342''. Linda's scrutiny of her model will be enhanced if she can access an explanation for ''CBP_000342 preference'' where that explanation may simply be ''Carr-Boyd's Prelude, written 1993 in Sydney, Australia''. We might argue that ''CBP_000342 preference'' is a poor name for this component and we could simply demand that all programmers select more meaningful names. Unfortunately, this would not provide adequate support for scrutability for two main reasons: one driven by implementation concerns and one related to users. The first, implementation-driven reason is that the most appropriate name for a component may be determined by factors outside the control of the user modelling system. We can see this in terms of our example: CBP_000342 may be the identifier for the particular recording of this music as lodged at an authoritative centre for music. From a programming perspective, there may be compelling reasons for maintaining consistency with this naming.

The second, and more compelling reason for requiring an explanation to supplement the meaningfulness of component names comes from consideration of the user's needs. We can expect that different users may be best served by different explanations of a component. We make provision for explanations to be customised for the user and the um-consumer. This is why *explain-component* shows these as optional - indicated by the square brackets $[\cdots]$. They are not always required optional because the explanation of many component meanings should be consistent across many users and um-consumers. However, they make it possible to customise such explanations to improve the understandability of the explanation.

Consider how this might operate in our music preferences scenario with $user_{Linda\_Sculthorpe}$. If Linda were a tertiary music student, we might explain the *Carr-Boyd Prelude* component in a manner most appropriate to an adult with music expertise. On the other hand, if Linda were nine years old, an appropriate explanation might be quite different. Such customisation of explanations (for example, C L Paris, 1989, Zukerman and McConachy, 1993, Milosavljevic, 1997) may be important to ensure the user appreciates the meaning of the component. In our scenario, a good explanation of the component ''CBP_000342 preference'' might be to play the piece so the user can listen to it and appreciate that the component represents the user's preference for precisely that piece of music.

Similar issues apply for the um-consumer argument and the combination of um-consumer and user. For example $consumer_{music\_entertain}$ when used by a nine year old might have different explanations from, $consumer_{teach\_aust\_modern\_violin}$ used by a tertiary student.

A more subtle issue is that a component may be regarded somewhat differently by different um-consumers. We can expect this to happen if the user model becomes established over a period of time and programmers of new um-consumer systems decide to reuse existing components. In our scenario, the model may have been created for $consumer_{music\_entertain}$ and later $consumer_{teach\_aust\_modern\_violin}$ might reuse it.

3.1.2 Support for use of external user modelling information

The rationale for the *model_externals* requirement in Table 3.2 is that external evidence sources must be able to contribute to the model. The scrutability requirements, *scrutiny_externals* ensures the user can scrutinise modelling *processes* based on external sources. Users should be able to find answers to two types of questions, cast here in terms of our music preferences scenario.

What were the sources that informed the system about my preferences for Carr Boyd's Prelude? In our description of the scenario, we reported external sources like observations of Linda playing the piece and rating it. The user should be able to scrutinise the list of external evidence contributing to the value of a component.

When was the information collected? As in our scenario, time may be important. People's tastes change, as does their knowledge and other personal attributes. Our scenario might be interpreted as the user having acquired a taste for Carr-Boyd's Prelude. External sources must tell um their identity as well as the details of the piece of evidence they contribute as indicated by the subscripts in the evidence e, p, and v. In our scenario, the first piece of evidence was that the user played the Prelude and the evidence source was the music-playing system. The required time stamp can be added by um.

Table 3.2. Requirements for external user modelling information

model_externals: each external source *e* can tell um a piece of evidence *p* contributing to the value *v*, *model_tell*($user_u$, $component_c$, $evidence_{e,p,v}$)
scrutiny_externals: user can ask for evidence about a component in their own model: *user_ask*($user_u$, $component_c$) each piece of evidence has an explanation *explain_evidence*($evidence_{e,p,v,t}$, $component_c$, $user_u$, [$consumer_a$])

The scrutiny requirements mean that for each source of external modelling information there must be an associated explanation which explains the evidence *p*, from the external source *e*, with value *v* at time time *t*. For example, Scenario 3.1 involved a system for playing music. This was also an external modelling source which could contribute evidence, for example that Linda played a Carr-Boyd Prelude on 15th May, 1978 or that she started it at 17.03.09, 20th May, 1978 and stopped it 10 second later. The scrutiny requirement means that the user should have access to just this information which was used to reason about her preference in the scenario.

We require that um record the time of the *model_tell*. Such a time-stamped history is a departure from the systems described in Chapter 2 but it is needed for explanations such as those above.

3.1.3 Support for internal inference

We treat internal inference similarly to external evidence sources. Essentially, an inference source is another provider of user modelling information. The critical distinction is that the external sources must be accepted as black boxes which provide information about the user. We can require only a general explanation for each piece of evidence they can contribute. By contrast, internal reasoning is based on a set of the system's beliefs, each based upon a set of components: {*SB*({$comp_c$})}. For example in our scenario, we inferred Linda's preference for the new Carr-Boyd fugue, an inference that might be expressed as

> if *the user likes Carr-Boyd's Prelude*
> then *the user likes Carr-Boyd's new fugue*

Here, support for scrutability involves two parts: access to this inference and to the components used in the inference (ground components for the inference). In this case, Linda's preference for the *Carr-Boyd's Prelude* is the ground inference.

The example we gave here happened to be a simple production or rule. We do not restrict the forms of inference allowed. The critical requirement is that any inference mechanisms used should be open to scrutiny by the user. The principle is that inference mechanisms should only be acceptable sources of internal inference if they can provide adequate explanations of their operation.

Table 3.3. Requirements for internal inference

model_internals: internal reasoning element *i* can contribute evidence *p* with value *v*, *model_tell*($user_u$, $component_c$, $evidence_{i,p,v}$)
scrutiny_internals: user can ask for component's evidence *user_ask*($user_u$, $component_c$) each piece of evidence has an explanation *explain_evidence*($evidence_{i,p,v,t}$, $component_c$, $user_u$, [$consumer_a$])

So requirements *model_internals* and *model_externals* in Table 3.3 look similar to their counterparts for external sources. However, the nature of the explanations coming from *explain_evidence* should be qualitatively different. For example, if the reasoning is based on a rule like the one just described, the explanation can include presentation of that rule as part of the detailed explanation of the internal user modelling processes. By contrast, external sources of user modelling information need only provide a general description of their operation.

3.1.4 Management of consistency

A um-consumer must be able to ask the value of a component. From the point of view of the um-consumer, this is the primary purpose of the user model. Table 3.4 shows two evaluation functions: *model_evaluate* for use with a program; and *user_evaluate* to report the value in a form suitable for the user.

Before discussing the arguments to the evaluation, we explain why there are two scrutability requirements. Essentially, these enable the user to answer two forms of questions. We consider examples from our scenario.

Does the system think I like the Carr-Boyd Prelude? This seeks the value of the component, provided by *user_evaluate*. In the scenario, the evidence available by the end of the period described, seems to indicate a positive answer to this question.

How did it work that out? This calls for an explanation of the process used to calculate that value. It is provided by *explain_evaluate*. In our scenario, the answer might be:

Table 3.4. Requirements for conflict resolution

model_resolve: For component $component_c$ there is at least one evaluation, $model\_evaluate(component_c, user_u)$
scrutiny_resolve: the user can ask for the value of a component $user\_evaluate(component_c, user_u, [resolver_r])$ each resolver has an explanation $explain\_evaluate(component_c, user_u, [resolver_r], [consumer_a])$

starting a year ago, Linda has been observed playing it frequently. (We might add that we also believe that people only play music repeatedly if they like it.) The scrutiny requirements mean that the system should make such explanations available to the user. Our evaluators optionally accept a *resolver* argument. This resolver encapsulates the process for reasoning from the evidence available about a component to the value for the component. Our design makes it possible to have a collection of conflict resolvers where each uses a different conflict resolution mechanism. The um-consumer can select from these or simply use the default resolver.

It may not be immediately evident why we would want this. Certainly, part of our motivation is to provide flexibility for the um-consumer. However, it is also appealing for scrutability. For example, given the time-stamps on all evidence, we can offer time series interpretations of user models. This means the user can ask about the value of components at nominated times in the past as well as at present (the default). This might be important for understanding the behaviour of a um-consumer over a period of time.

3.1.5 Broad applicability

The requirements summarised in Table 3.5.m. cover a number of different facets associated with achieving broad applicability for a user modelling shell. The first, *model_component_types*, reflects the importance of being able to support the variety of component types described in Chapter 2. We show the requirement for four types of components. The first two model the user's knowledge. *Knowledge* is for components like

> the user's knowledge that the Carr-Boyd Prelude is an example of modern art music.

By contrast, *non-mutual knowledge* is for components like

> the user's belief that the Carr-Boyd Prelude is classical music

or

the user's belief that Anne Boyd is the same composer as Anne Carr-Boyd.

We (and systems we might build) consider this to be untrue. Both types of knowledge have played an important role in user and student modelling. For example, in the scenario at the beginning of this chapter, these aspects may be used for the system which teaches about modern Australian violin music.

We also require a component type for user preferences. In our scenario, the system which recommends music may need such components. Indeed, the goal of such a system is to predict the user's preferences. There are fundamental differences between the user's knowledge and their preferences. Certainly, there is no need to consider any system beliefs for preferences. Only the user's preferences need be represented. We would expect this distinction to be significant for scrutability: for example, one explanation structure might be applicable for knowledge and another for preferences.

We then require another catch-all type for all other forms of component: for example, the user's goals, what they want and personal attributes like their age, whether they are athletic or motivated by excitement. In Chapter 2, we described systems with components beyond our basic three types for knowledge, belief and preferences. This final *attribute* type deals with all of these. Our requirements calls for support of just these four types of components.

Table 3.5.m. Summary of model requirements for broad applicability

model_component_types: component types include: *knowledge, non-mutual knowledge, preference, other attributes*
model_simplicity: Simple modelling task should be low cost and simple.
model_extensibility: Diverse modelling will be feasible as extensions.
model_privacy: Each action on the model should be controlled to protect the user's privacy.
model_structure: Components are structured in *contexts*: a component $comp_c$ is actually $component_{context,component\_name}$
model_scalability: The representation scales up in terms of model size and complexity.

The purpose of the model component types requirement is to ensure that we will be able to represent a range of the different classes of modelling information likely to be needed by a um-consumer application. The more primitive classes, knowledge, non-mutual knowledge and preferences clearly fail to deal with all possible modelling needs. To gain some sense of the various classes of user model content encompassed by our attribute

catch-all category, we can consider the categorisation of user model content for the set of papers at the 1997 User Modeling Conference (Jameson, Paris, and Tasso, 1997) which has the following: preferences, interests, attitudes, goals; knowledge and beliefs; proficiencies; non-cognitive abilities such as motor abilities; personal characteristics such as age or level of education; history of interaction with the system.

The requirement *model_simplicity* follows from the goal of broad applicability. At first glance, this may not be obvious. It follows from the fact that there is a range of user modelling sophistication demanded by um-consumers, with some having extremely simple needs and while others need quite complex, sophisticated modelling. The *model_simplicity* requirement calls for simple support for the simplest user modelling needs. Of course, one would expect that sophisticated modelling tasks would require correspondingly sophisticated user modelling. However, this requirement states that it is unacceptable to have large complex user modelling where the um-consumer has very simple needs.

To see the importance of this, we note that there seems to be great potential for user modelling that is quite simple. For example, effective customisation has been driven by just a few flags (Neal, 1987, Neal, 1989, Strachan, Anderson, Sneesby, and Evans, 1997). This should be managed simply and at modest cost.

One might argue that such simple demands do not merit a user modelling system at all. In fact, they do not. They could easily operate from a modest collection of set-up flags such as those commonly used to customise most operating systems, windowing environments and many, many powerful applications. However, our design goal is that even modest user modelling be able to operate within our framework so that it can be reused by other um-consumer applications.

This also introduces the possibility that the user model could support both *adaptive* and *adaptable* interaction. The distinction may seem subtle. Essentially, an *adaptive* system alters its own operation in order to improve the interaction. For example, a recommender or teaching system which tailors its actions to the individual user is adaptive. By contrast, an *adaptable* system can be customised by the user so that it operates differently. For example, a browser may have a preferences menu which enables the user to alter the fonts and other aspects of the presentation at the interface. This thesis is concerned with adaptive systems. At the same time, we note that adaptable systems are currently more widespread. Adaptable systems rely upon a collection of user preferences, flags or other information that could be regarded as a very simple user model which is normally set to default values but which can be adjusted explicitly by the user. If um can support both adaptable and adaptive systems, the same information can be used

across a range of systems. So, for example, the user's stated preferences for one application can be used as default values for others. This means that user modelling information can be reused across applications.

Although our approach is required to manage simple modelling tasks simply, the requirement *model_extensibility* ensures support for a range of interesting modelling tasks for realistic um-consumer's needs.

The *model_privacy* requirement acknowledges that user models are likely to have private information. So they must be protected, as argued elsewhere (Kobsa, 1990). This is essential for applicability outside research laboratories.

Our next requirement is *model_structure*. Essentially, we require that components be partitioned into *contexts*. For example, the components for the user's music preferences will be in a different context from those modelling their knowledge of a text editor. This seems natural. But it is also important for scrutability. It reduces the number of components that need to be considered at one time. It is also important for limiting the scope of component names so that the same component can appear in different contexts with different meaning. So, for example, a system that teaches about music and a music recommender can use the same component identifier but can have different meanings for it. This limiting of the scope of component names is essential if different programmers are to be able to work independently on different um-consumers, each with the possibility of having its own collection of user model components. Yet another important role for the structure is in the management of consistency. Structuring the components into partitions means that some parts of the model can be regarded as independent of others. Then, the consistency management mechanisms can operate on just the parts of the model that are relevant to the um-consumer. This should mean that the consistency management operates on small numbers of components and should operate more efficiently.

The last model requirement is *model_scalability*. The modelling process should scale in terms of the number of users, the number of contexts and um-consumers. Broad applicability should enable um to be used across many modelling tasks and by many users.

The *scrutiny_simplicity* requirement is essential for scrutability to work in practice. For a teaching system or music entertainment system such as in our scenario, there are likely to be many components. This constitutes a basic level of complexity and correspondingly a hurdle to understandability of the model. To counter this our design of the representation must strive for simplicity.

The *scrutiny_control* requirement is central to scrutability which is essentially about *access* and the user's right is to control their own user model. It may be that the user will

Table 3.5.s. Summary of scrutability requirements for broad applicability

scrutiny_simplicity: The model should be simple to understand.
scrutiny_control: The user u is able to alter any component c with a piece of evidence p with value v $user\_tell(user_u, component_c, evidence_{u,p,v})$
scrutiny_scalability: The support for scrutiny scales up in terms of model size and complexity.

make their model inaccurate. That possibility follows from their having control. The user might also perform *what-if* experiments with their model, altering it and seeing the effect. This may be part of the scrutability support for um-consumers where there is a complex chain of reasoning between the user model and the um-consumer actions.

We require *scrutiny_scalability* so that our support for scrutinising the model can operate on many contexts used by many um-consumers, large models with large amounts of external evidence and internal reasoning.

3.2 Architectural overview

Our architecture has been strongly influenced by the *software tools approach* (Kernighan and Plauger, 1976) which is central to several major successful software projects, perhaps the most notable being the Unix operating system. The appeal of this approach comes from its demonstrated power for creating sophisticated, extremely flexible and highly maintainable software systems. An essential element of the approach is that the designer must identify the basic tools that can be combined to perform a class of tasks.

On the basis of our requirements, we have defined a small number of classes of um tools. We now describe them in terms of the um architecture shown in Figure 3.1. The user is at the top of the figure and the persistent form of the long term user model at the bottom. Each of the four boxes across the lower middle of the figure represents one major classes of tools. Between these and the user are the um-consumers for which the user model is needed. The directed arcs indicate the main flow of user modelling information. We now describe how these interact to provide our architecture.

We begin with tools for *external* sources to fulfil the *model_externals* requirement. These collect information both from and about the user. We show one arc between the user and these tools. This corresponds to interactions where a tool *elicits* user modelling information from the user or *observes* the user. The arc from the um-consumers to the tools for external sources reflects the fact that a um-consumer may contribute user

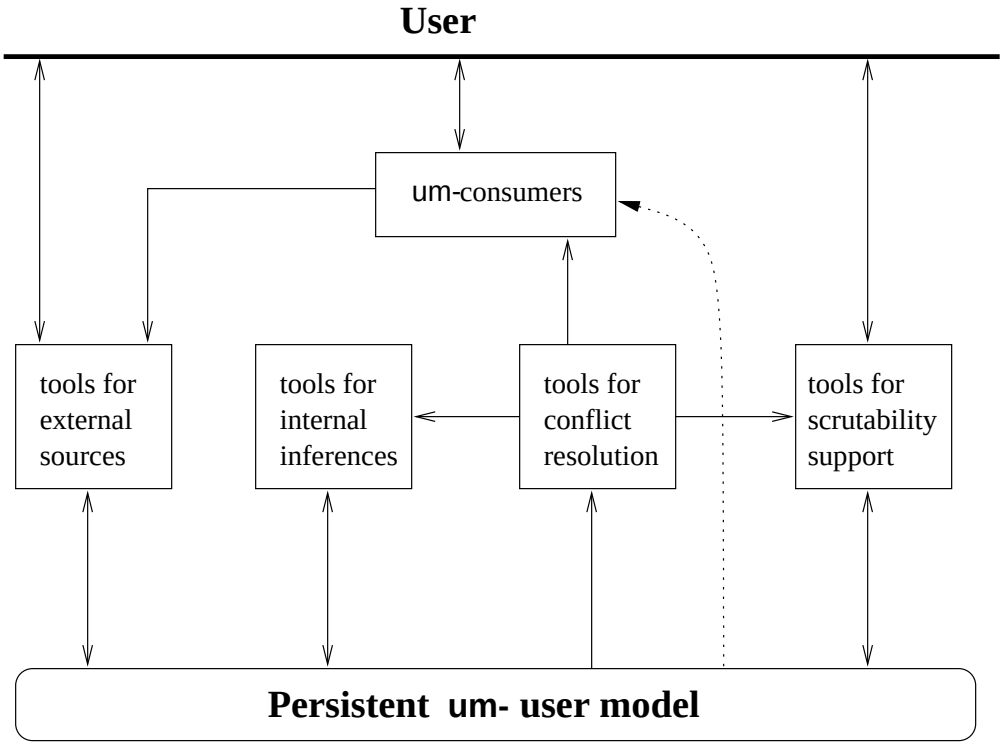


Figure 3.1. Architecture for um

modelling information which these tools can then add to the model.

Next we consider the third box from the left of the figure, tools for conflict resolution (requirement *model_resolve*). These *evaluate* components within the user model for use by the other tools. The flow of information shows how um-consumers normally access the model through a resolver. The figure also shows a dotted line directly from the model to the um-consumers. This reflects the possibility that um-consumers access information directly from the persistent um user model. This line is dotted to indicate that this is not the normal operation of the architecture. It is possible and may offer additional flexibility and, in the case of very simple parts of the user model, it can offer a faster and lower cost access mechanism (and so support *model_simplicity*).

In building the user model, the other main set of tools perform the *internal* inference (requirement *model_internals*). These tools reason from the existing user model state to new conclusions about the user. They can determine the current model state in two ways: taking component values provided from the conflict resolvers; or taking information directly from the persistent form of the model.

The final set of tools provide *scrutability* support explaining the four main parts of the user model and the processes that have formed it (requirements *scrutiny_ontology*,

scrutiny_externals, *scrutiny_internals* and *scrutiny_resolve*). These have access to the model via the tools for conflict resolution.

Note that each box in Figure 3.1 represents a class of tools. The architecture assumes there will typically be several different tools of each class. Each um-consumer may rely upon a selection of these: perhaps several tools that collect user modelling information of various forms; and a combination of a few resolvers with several inference tools. This supports the requirement *model_extensibility*. Finally, the user of such a um-consumer might be able to use a selection of scrutability support tools to study various aspects of their model.

Figure 3.1 is intended to show the main elements of the um architecture from the point of view of the programmer who constructs um-consumers. It does not show some of the additional housekeeping tools needed by um. One of these is the tool needed to create a new part of the user model, defining new components. Nor do we show the control mechanisms which are needed to start required collections of tools for each um-consumer.

The discussion to this point has touched on several requirements. The remainder will be treated in the detailed explanation of the um representation in Chapter 6 and our approach to scrutability support in Chapter 7.

3.2.1 Persistent storage form of the long term user model

Having outlined the tools in our architecture, we now consider the persistent form of the user model at the bottom of Figure 3.1. Essentially, the persistent form of the model is the repository of the long term user modelling information kept about the user, including a trail of the evidence from external sources and internal reasoning. From the point of view of both the user and um-consumers, this is the *long term* user model. We introduce our representation for the persistent form of um model in terms of an example, a component in a um model for the user's knowledge of a text editor called sam.

An example of the persistent form of a component

Figure 3.2 shows the part of the file containing the persistent form of the component, quit_k, which represents the user's knowledge of the quit command for the editor.†

The first line constitutes a declaration. It gives the component name quit_k. It also provides a string with a longer description of its meaning. Finally, it indicates the type of the component is knowledge. Next is a list of evidence indicating the component value is true: the value supported by each piece of evidence is indicated by the true in the first field of the evidence. The first piece of evidence is an observation from an external evidence source tool mcons reporting that the user had used the quit command 342 times and the evidence was provided at the time shown in the 5th column, here Thu Mar 18 12:25:01 1993. This piece of evidence actually involved the following stages.

```
// Declaration for component of type 'knowledge'
quit_k    "how to use the quit command"                knowledge

// List of evidence indicating the component is true
true      observation   mcons       quit              (Thu Mar 18 12:25:01 1993)  "COUNT=342";
true      rule          sam_morph   quit_anal         (Thu Jun 24 13:20:21 1993);
true      told          viewer      explain_quit_k    (Tue Jun 29 13:16:56 1993);
true      given         viewer      self              (Tue Jun 29 13:17:56 1993);

// List of evidence indicating the component is false
false     given         viewer      self              (Tue Jun 29 13:16:41 1993);
```

Figure 3.2 Hypothetical example of a persistent form of a component.

- Our version of the sam editor monitored each editing operation, logging it.
- After the user had been using sam for many months, the user modelling system started an external source tool called mcons (model constructor). This analysed the sam logs to count all successful uses of the quit command;
- Once mcons had completed its analysis, it produced the first piece of evidence in Figure 3.2. This was added to this user's model at the time shown, Thu Mar 18 12:25:01 1993.

The next piece of evidence comes from internal inference by a tool called sam_morph which considers the number of uses of the quit command and other means to quit and concludes whether the user appears to know the quit command. At the time shown it concluded the user knew the quit command.

The third piece of positive evidence was provided by the viewer program which is an interface to the user model so the user can scrutinise it. The evidence came from the part of the viewer that explained the quit command. This evidence means that the user used the viewer commands that give an explanation for the meaning of this command. This constitutes evidence the user has some knowledge of the command.

The fourth piece of supporting evidence was also provided by the viewer program. As

---

† This example is intended to indicate the form of the file. Actual models did not have the italicised comments and the order of the information is slightly different. In fact we have implemented several versions of um. All are similar to each other and the one in this figure. Appendix A shows an example of the version used for the work reported in Chapter 9.

the user interacted with the viewer, they indicated that they knew this command. This evidence is shown as coming from the user them*self*.

The last piece of evidence indicates the component is false. It also came from the viewer program and is almost identical to the other piece of given evidence. It looks as though the user first indicated not knowing the command, then indicated knowing it.

There are several ways to interpret this collection of evidence. It is the task of resolver tools to do just this. A resolver must deal with the conflicting indications about the user's knowledge of the quit command and conclude a value. Our design for um does not restrict the possible values that a resolver might be allowed to return. For example, one resolver might return boolean values only and it would need to analyse the evidence about a component's value and conclude either the value true or false. Equally, another resolver tool might return a numeric value in the range zero to one where zero corresponds to false and one to true and the values in between would correspond to intermediate degrees of truth or falsehood, as in MYCIN-like systems (Buchanan and Shortliffe, 1984). There are many other possibilities including, for example, the determination of a set of values for the component. For example, the resolver might return a pair of numbers in the range zero to one where these give the bounds on the truth value so that the pair (0, 0) corresponds to false, (1, 1) to true, (0.9, 1) to fairly confident of truth, (0, 1) to complete uncertainty about the value. It might use any of the range of techniques for uncertainty measurement (Motro and Smets, 1997). The important design issue is that the um architecture does not restrict the values a resolver tool can return for components.

Returning to the example in Figure 3.2, this approach might mean that there are several resolver tools available and each might deal with the evidence about this component differently. For example, resolvers might conclude the component is true in one of several ways. For example,

- one resolver may simply take the most recent evidence currently available, in this case, the user's claim to know the quit command;
- another resolver might define a partial order on the various sources of evidence and treat evidence from observations of the user as the most reliable source available and so these determine the component value;
- yet another resolver might also operate on a partial order on the various sources of evidence but it might treat the information from sam_morph as the most reliable source and so the value of the component would have been determined by the sam_morph evidence.

Many other resolvers are possible. Since resolvers are responsible for determining the

value of components, they play a pivotal role. The quit example illustrates the possibility of many different ways to infer the value of a component from a list of evidence. Our architecture allows for a collection of resolver tools so that a um-consumer can employ the one most suited to its needs.

To see why it is important to allow for different resolvers, we can begin by considering the difficulty of concluding that a user knows something. In the case of knowledge like the quit command, it might be reasonable to argue that the only conclusive indication of knowledge is repeated use of the command. For other types of knowledge, we might have quite different standards for judging what it means to know something. For example, one scale for self-assessment of knowledge (Tamir, 1984) runs from *heard of it* as the minimal assessment of knowing to *could explain it to a friend*. It seems likely that different um-consumers will demand different standards and forms of evidence for concluding that a user knows something well enough for a component to have the value true. This illustrates one motivation for allowing a range of resolvers. Similar arguments apply for concluding the value of user preferences or other types of components.

From the *model* perspective, the important point is that um offers flexibility in that it permits a variety of resolver tools and the particular um-consumer can make use of the one that suits its needs. From the *scrutiny* perspective, a critical issue here is that um must make the resolution process available for user scrutiny. In terms of the example in Figure 3.1, this means that the user should be able to see that the system concluded the true value indicated in the last line of the figure. In addition, the user must be able to scrutinise the mechanism used by the particular resolver which determined that value where that explanation might be a simple text rather like those in the three resolver process examples just described. This means that each resolver tool in a um toolkit must have associated explanations.

Our commitment to a collection of resolvers has important implications for scrutability. It introduces additional complexity to the modelling process, it adds the need for explanations associated with each resolver tool and it means that the user needs to cope with greater complexity than would apply if there were a single resolver process used in all um user model. However, we consider that the requirement for flexibility and power in the modelling processes outweighs this. Essentially, there is a tension between the requirement *model_extensibility* on the one hand and *model_simplicity* and *scrutiny_simplicity* on the other.

Design issues in the persistent external form of components

In the um representation, each component is stored as an ASCII file similar to that in Figure 3.2. The um user model has one file for a collection of related components. Within that file, the user modelling information about each component is expressed as text giving the definition of the component type and name as in the figure and then a sequence of lines, each with the evidence about the component, again similar to the figure. These files constitute the persistent form of the user model. They hold the information about the user that the systems can recall from one interaction to the next. So they contain the essence of the user model. The um design makes them simply a collection of ordinary files in the user's own file space. Such files are intended to be accessible by the user and they are designed with comprehensibility in mind. Although we believe that it is essential to provide scrutability support tools so that the user can readily explore their complete user model, we keep the fundamental information in a form that is intended to be read by the user. This means that we need a simple parser within the um toolkit so that programs can also understand the model.

This use of ordinary ASCII text files to hold the persistent form of the model is a departure from the approach taken by the systems described in Chapter 2 where the user model is a data structure within a program. In um user models, the user has read access to the user model files. So, the user can look at them with normal tools like text editors. Of course, this also means that the user could alter the files, and this means it is possible that the um tools would then be unable to parse the model. This difference from other user modelling shells is driven by concern for scrutability of the user model. One reason is that it ensures that nothing is hidden from the user. More important, the user has ultimate control over their model: it is simply a collection of ordinary files in their file space. They can see the exact model that programs access. We could easily store it on external storage media as Doppelganger does (Orwant, 1995).

Another effect of our approach is that we can use ordinary programming tools with the model, provided they can parse the text and have appropriate access permissions. This means standard programming tools can be used to create experimental tools for user modelling.

Although the various types of evidence in our example may have seemed arbitrary, they follow from analysis of interaction in systems with user modelling. We describe this in Chapter 4. In Chapter 5, we gives details of accretion representation underlying um and show how it can model interesting user modelling state information: changes in the user's knowledge, learning and forgetting; user moods; changing preferences; unreliable sources of information about the user; noisy sources of modelling information; and making other

inferences beyond pure boolean values.

3.2.2 Scrutability support tools

The example of the quit component's persistent representation gives an indication of the way that each piece of evidence from external sources or internal reasoning leaves its mark on the user model. It also illustrates how the resolver deals with internal conflict in the model. The remaining element of the um architecture is the scrutability tools.

We have just explained why the persistent form of the model is stored in a form that is accessible to the user. While this *permits* scrutiny in a sense, we must immediately face the fact that the example in the last section needed considerable explanation. It fails to meet the *scrutiny_simple* requirement.

We are convinced that practical and useful user models will be too large to be studied directly from the ASCII representation. The user will need support tools which present the information in forms that are easier to understand. We would expect that different tools would assist the user in exploring different types of models and different aspects of them.

Then the ASCII file is simply a final reference. The scrutability support tools should maintain consistency with the ASCII representation where there is no good reason to do otherwise. Then the user who wants to scrutinise the ASCII files can see the relationship between the user model as seen through the scrutability tools and the actual stored form in the ASCII file.

Figure 3.3 is a display from viewer, a scrutability support tool which gives an overview of the components in a model. It is a snapshot of the model for one user's knowledge of the text editor, sam. A dark node indicates the user knows that aspect and the white ones indicate lack of knowledge. Nested shapes, like the nested squares for *default_size_k* indicate there is conflicting information about this aspect and the viewer resolver could not assess whether the user knew it or not.

The circles represent non-terminal nodes. Each of these correspond to a context or sub-context. So, for example, the quit_k component is part of the collection of components classified as useful basic aspects of sam. At the particular point in interaction shown in Figure 3.3, the minimal and useful nodes have been expanded to show their components but the mouse node is not. If the user clicks on the circle for mouse, it too will be expanded. Correspondingly, clicking on the minimal circle will contract it, reducing the clutter in the diagram.

At this point, we present Figure 3.3 only as an illustration of one form of scrutability

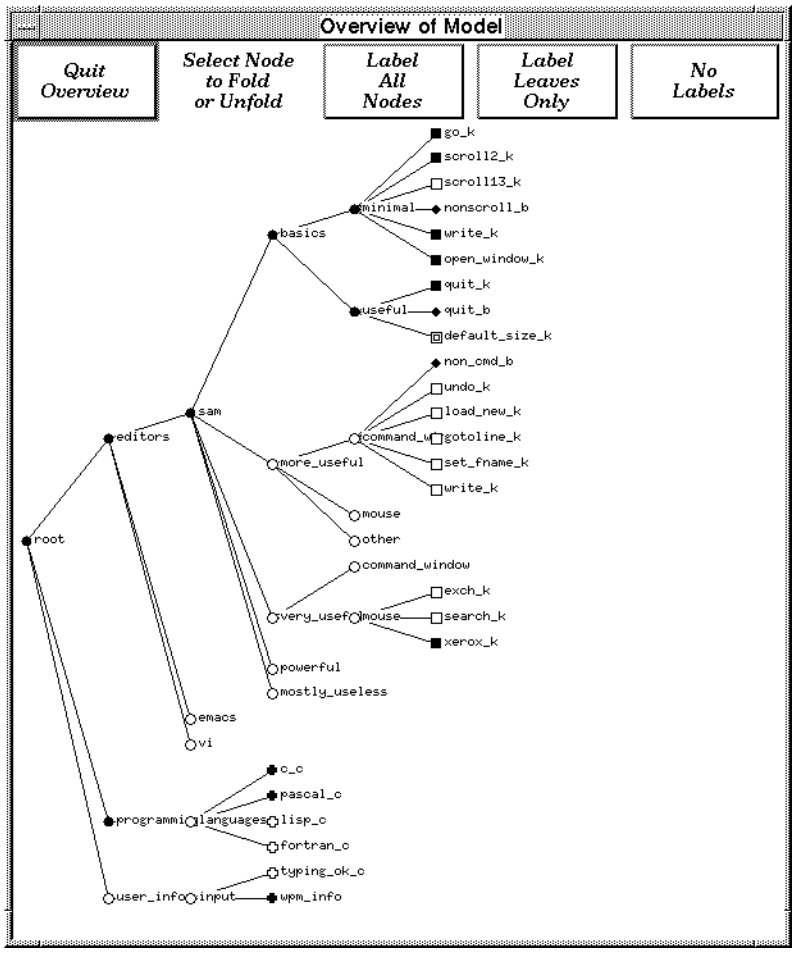


Figure 3.3. Example of the qv display

support. In Chapter 7, we describe the basic set of scrutability support tools necessary to fulfil our requirements.

3.3 Summary

Our design has been driven by the need to consider user modelling from two points of view, that of the

- user and
- the um-consumer program (and its programmer).

We have defined our requirements for a user modelling shell as a highly specialised database. Then, we introduced our approach and representation with an overview of the um architecture and a detailed description of one example of a component's persistent

form. We have discussed the relationship between scrutability and our choice of an ASCII representation for the persistent model and we have shown one example of a scrutability support tool.

Chapter 4

Scrutable-interaction model

> ‘‘The use of models of interaction helps us to understand exactly what is going on in the interaction and identify the likely root of difficulties.’’ (Dix, 1993:92)

As our first step in designing a scrutable user model representation and tools for user-adapted interaction, we develop a model of interaction. The serves several purposes.

- It identifies the *actors* in interaction involving user modelling. This establishes who might be responsible for elements of the interactions. This is a foundation for supporting scrutability in that it establishes the appropriate source of information for the user to scrutinise.
- It can serve as a tool for analysing interactions, especially to clarify the role of the user model in the interaction.

- It helps identify points where scrutability support should be provided.
- It identifies three distinct classes of external evidence about the user.

4.1 Overview of the model for user-adapted interaction

Our interaction model is called umps because it is based upon descriptions of the **u**ser, **m**achine, **p**rogrammer and **s**crutability. The strongest influence on the model comes from the work of Suchman (1987) in analysing interactions

> to find the sense of ‘‘shared understanding’’ in human-machine communication ... to compare the user’s and the system’s respective views of the interaction ... (Suchman 1987:115-6)

She used the following analytic framework for recording interactions.

THE USER		THE MACHINE	
Actions not available to the machine	Actions available to the machine	Effects available to the user	Design rationale

Figure 4.1. Suchman’s framework

She observed that ‘that the coherence of the user’s actions was largely unavailable to the system’ (Suchman 1987:116). This is because the machine had access only to the information in the second box from the left. However, the researcher observing the user could use the leftmost column to record observations and this could assist in interpreting problems in interactions. The centre pair of columns record the information that is available to both user and machine. When an interaction breaks down, the researcher can analyse the situation by comparing the outer columns with the inner ones.

Because we are concerned with a scrutable user modelling shell for supporting user-adapted interaction, we need to distinguish between the machine’s internal operation and the programmer’s design of that operation. Our framework is based on three actors:

- user: the user in the interaction (as in Suchman’s work);
- machine: the system, really the software driving this interaction;
- programmer: the person or people responsible for creating the machine.

We have added one new actor, the programmer, to Suchman’s framework. It is the programmer who will be responsible for ensuring support for scrutability of a user model. The user and machine are the only *active* participants at the time of the interaction. The programmer part of the model is used to record the programmer’s design decisions as well as relevant beliefs and intentions at the time they created the machine. These may need to be recorded within the machine so that the user can scrutinise them.

Throughout this chapter, we distinguish two classes of interactions: those concerned with user modelling are shown in bold; those of ordinary application are in normal Roman font. The distinction is important because we are concerned with supporting scrutability of the user model. To see the difference, consider examples of each. A pure user modelling interaction might involve the user telling the system that they like music by Carr-Boyd or that they know how to use the file-write command of a text editor. By contrast, a pure application interaction might involve playing music through a digital music interface or using a text editor.

We follow Suchman's approach in indicating whether information is mutually available or not. However, we describe this as follows:

- *private* : not available to the other parties in the interaction;
- *shared* : available to the other party at the interface.

At this point, we will use the framework provided by our model to analyse two hypothetical interactions. The first involves surreptitious logging of an unsuccessful interaction. The second is a user-adapted interaction such as our user modelling shell will support.

The umps interaction model as an analysis tool - example of an unsuccessful interaction with logging of user actions

Consider the following scenario:

> Rebecca has been using a text editor called sam. At the point that we begin this scenario, she has just finished typing an essay. She now wants to quit. Rebecca has been so concerned with the essay that she has unfortunately forgotten to write out the file. So it is currently in an unnamed editor buffer.

Figure 4.2 shows the use of our framework to describe the subsequent interaction. The column labels at the top of the figure show the parts of the umps model.

- $user_{private}$: shows Rebecca's private beliefs at each step of the interaction.
- $user_{shared}$: gives the actions she types at the interface.
- $machine_{shared}$: is sam's output.
- $machine_{private}$: has the machine's internal actions.
- $programmer_{private}$: documents the programmer's private beliefs in the design of $machine_{private}$.

The horizontal lines separate each of the actions during an interaction cycle. In a successful interaction, initiated by the user, this cycle has the following steps:

$user_{private}$	$user_{shared}$ $user_{action}$	$machine_{shared}$ $machine_{action-ext}$	$machine_{private}$	$programmer_{private}$
$user_{action}$ Have finished typing this important essay. Want to quit.	q			Quit command consistent with ed family of editors.
			$machine_{interpret}$ User wants to quit but there is an unwritten file buffer.	
			$machine_{action-int}$ **Log quit request with unwritten buffers**.	**Monitor commands that cause warnings - they may suggest user difficulties**.
		?changed files		Warnings should be terse and in style of ed family.
$user_{interpret}$ Funny, seems not to have quit.				
$user_{action}$ Try again.	q			As above.
			$machine_{interpret}$ User really wants to quit.	Operate as ed family. Assume user intends to quit.
			$machine_{action-int}$ Quit **Log quit with unwritten buffer.**	**Log warnings - they may suggest user difficulties**.
		sam quits		quit interface actions.
$user_{interpret}$ Good - it has quit as usual.				

Figure 4.2. Example of interaction analysis

- $user_{action}$
- $machine_{interpret}$
- $machine_{action-int}$

- $machine_{action-ext}$
- $user_{interpret}$

For the shared information, these are listed under the appropriate column header, $user_{action}$ with $user_{shared}$ and $machine_{action-ext}$ under $machine_{shared}$. The other actions are shown near the relevant descriptive text, in the $user_{private}$ column for $user_{action}$ and in the $machine_{private}$ column for $machine_{interpret}$ and $machine_{action-int}$.

Our documentation of the first action shows Rebecca's goal and the associated action. We see that the first part of $user_{private}$ records this goal as the need to save the work done. In a typical think-aloud protocol (Preece et al, 1994, Nielsen, 1993) the user might state such a goal and an observer could record it. The user acts on this goal by typing the usual quit command. In $programmer_{private}$, we show the reasoning behind the design of the quit command required by sam.

The second action is the machine's correct *interpretation* of the q command as a request to quit. We also show that the machine finds there is an unwritten file buffer.

Next we show the machine's internal actions. This is the logging shown in bold because it will contribute to a user model. The $programmer_{private}$ decision to log this is also recorded.

The machine then writes a cryptic warning message as $machine_{action-ext}$, and this is available to the user. The programmer's design of this is explained in $programmer_{private}$.

In this scenario, Rebecca does not notice the error message. Our analysis of the interaction simply shows that she recognises the quit has not happened. We see that Rebecca tries the same quit command again. This time the machine interprets this as a request to quit even though there is an unsaved buffer which will be lost. It does so. This is logged. Then the user sees sam has quit. At each step we have recorded the likely reasoning of the designer in $programmer_{private}$.

This figure documents a *dangerous state* (Dix, Finlay, Abowd, and Beale, 1993) in the text editor's design. Just as Suchman documented interactions with the four *user* and *machine* columns, so we use them to capture what is actually happening in this mismatch between user and machine 'assumptions'. Our model distinguishes $programmer_{private}$ from $machine_{shared}$ and captures the reasoning behind the behaviour of the machine. In this case, the underlying problems in the interaction follows from the $programmer_{private}$ decision to maintain consistency with the ed family of editors. In ed, a single quit generates an error and a second quit is acted on. This can be useful; for example, if one makes unintended changes to a file and realises the current saved version is superior. Underlying this design decision is a broad assumption the user really knows what is

happening in the interaction, and wants the machine to do as it is told without having to go through tedious dialogues.

From a user modelling perspective, there is simple logging of editing actions. From the scrutability perspective, the figure illustrates the need to distinguish $programmer_{private}$ from its manifestations in $machine_{private}$. If users are to scrutinise the behaviour of a system, we must be able to provide more than a trace of the machine's action (from $machine_{private}$). We also need to capture those $programmer_{private}$ aspects which explain $machine_{private}$.

In this example, the $programmer_{private}$ column documents two programmer's design decisions: the sam programmer and the logging programmer. If the logging is used to build a user model, the user scrutinising that model should be able to trace the source of information back to the logging processes recorded from this interaction. We now consider an example where umps model is used to document the nature of the user modelling in an interaction with a user-adapted system.

Example of interaction with user model central to machine actions

This scenario follows from the first.

> Rebecca has been using sam for several months. Today, she has some spare time and decides to try out the tutoring system for sam so she can become a more effective user.

The interaction is also rather more complex than the last. It involves a user-adapted system which can teach about sam. It relies upon a user model for Rebecca's sam knowledge.

Figure 4.3a shows the first interaction cycle between Rebecca and the tutor. She starts the samtutor. The machine interprets this as she intended. The machine then performs the sequence of actions shown. Since samtutor must determine what to teach, it builds a user model by analysing sam logs constructed from interactions like those in Figure 4.2. In $programmer_{private}$ we record the design decisions involved in each step. The machine decides to teach about the form of the write command required for unnamed buffers, and presents information about it.

We now consider the next interaction cycle, shown in Figure 4.3b. Rebecca does not understand why the system told her about this command and types the tutor's explain command. The machine interprets this as a request for the basis for its choice of the tutoring action. This has two parts, the frontier teaching strategy and the user model. It presents information about each. The part shown in Roman font is standard information presented by samtutor. This is the canned text about the teaching strategy. The bold

user $_{private}$	*user* $_{shared}$ *user* $_{action}$	*machine* $_{shared}$ *machine* $_{action-ext}$	*machine* $_{private}$	*programmer* $_{private}$
user $_{action}$ Want to learn more about sam.	samtutor			Command interface, tutor name.
			machine $_{interpret}$ Start sam tutor.	
			machine $_{action-int}$ • Determine what user knows. • **Analyse logs**. • **Construct user model**. • **Select tutoring action.**	Tutor initiates construction of a user model. **Log analyser counts occurrences of commands.**. **Uses command counts to add evidence to relevant components**. Select action of frontier of user's knowledge, **User model shows aspects knows, and not known**. Select most basic aspect not known.
		To write an unnamed file buffer, you have to use the command window and type the command w <filename>		Canned text about this command.
user $_{interpret}$ Wonder why it told me that.				

Figure 4.3a. Analysis of a user-adapted interaction - step 1

parts come from the user model. This begins by stating the *value* of the command-write component in the model. Then it reports the *processes* that contributed to this value. Three sets of observations are described. Together, these were interpreted as indicating that the user did not know the command-window write. These are the evidence for the component.

user $_{private}$	*user* $_{shared}$ *user* $_{action}$	*machine* $_{shared}$ *machine* $_{action-ext}$	*machine* $_{private}$	*programmer* $_{private}$
user $_{action}$ Ask tutor to explain.	explain		*machine* $_{interpret}$ Explain last action.	
			machine $_{action-int}$ Find most relevant parts of user model for choice of aspect tutored	Explanation needs reasons that aspect tutored was assessed as not known.
		The tutor selects the most basic command you don't yet seem to know. **You do not seem to know the command-window write.** This follows from your use of sam. **You used the button-3 menu to write an unnamed file buffer.** **You have never used the command-window write.** **You have also quit without having saved an unnamed buffer.**		Canned text from coach Canned text from coach + **user model** **Logs had several such warnings.** **Logs had no use of write.** **Logs had several such warnings.**
user $_{interpret}$ ok				

Figure 4.3b. Analysis of a user-adapted interaction - step 2

Using the model to describe interactions

In the short interactions just described, we have used the umps model to:

- document the relevant actor's reasoning and actions;
- clarify the *programmer* $_{private}$ design decisions behind the application and the user modelling;
- highlight the role of user modelling for adapting the interface to the individual;
- document a level of scrutability support where the machine can explain the value of the user model components and the processes that contributed to that value.

4.2 Description of umps interaction model

The examples in the last section serve as an informal introduction to the umps interaction model. We now describe its general form in terms of the Figure 4.4. This will clarify our understanding of the external sources of user modelling information. The leftmost column has aspects of user that are private, $user_{private}$. This corresponds to the messy shape we used to depict the contents of the user's mind in Figure 1.1. For example, consider the first scenario in this chapter, where Rebecca had finished typing an essay and wanted to quit the editor. This goal and the knowledge relevant to achieving it are all in the user's mind.

User		**Machine**		**Programmer**
$user_{private}$	$user_{shared}$	$machine_{shared}$	$machine_{private}$ $programmer_{shared}$	$programmer_{private}$
direct user modelling **user's mental models and other long term attributes**	**explicitly given user information**	**aspects of the model the user is told about**	**the user model and knowledge about users and making inferences about them**	**programmer's beliefs about users**
Application system user's mental models relevant to the task-at-hand, including current short term goals, awareness of recent interaction	user actions at interface with application	application actions	encoding of application system and current status information,	programmer's beliefs about the application domain and static user modelling

Figure 4.4. Interaction model overview: actors and belief sets

The next column has the user's shared beliefs, $user_{shared}$, the user's actions at the interface. Typically, this means the actual command typed by the user. The only information the machine has about the user is that which the user places in $user_{shared}$.

The next two columns have the corresponding beliefs for the machine. Aspects of the machine that are available to the user are in $machine_{shared}$ and private aspects are in $machine_{private}$. Successful interaction requires the machine to make a suitable interpretation of $user_{shared}$ information, combine that with its current state and do the required actions. Usually, the machine gives the user some feedback that it has done what the user requested. This is placed in $machine_{shared}$.

Finally, the rightmost column is the programmer's private beliefs, $programmer_{private}$. This represents the mind of the programmer and corresponds to the messy shape at the

right of Figure 1.1. This is the design decisions, knowledge and assumptions that underlie the creation of the machine. Similarly, programmer's design decisions define what the the user sees in $machine_{shared}$. An important part of HCI user modelling work assists the programmer improve assumptions about users.

The figure has double vertical lines around the *interface* which is composed of $user_{shared}$ and $machine_{shared}$. The information available in these shared areas is often described as the *dialogue history*, especially for natural language interactions. We use the term in a broader sense, including all actions by the user or machine that are available to the other party. Depending upon the richness of the interface, user actions may involve keystrokes, mouse actions, speech, gestures, eye movement, active badge data ... On the machine's part the actions will typically be information displayed on a screen but could include more exotic devices. The critical point is that private aspects are not directly available to the other parties in the interaction: this interface constitutes a narrow bandwidth of communication between user and machine.

4.3 Analysis of the transitions during interactions

Table 4.1 shows all potential transitions in the interaction model. Each transition has been given either a name or a number. The names should be familiar from Figures 4.2 and 4.3. The numbers give the row number then the column number. So, for example, the transition marked 21 is on the second row, meaning it is from $user_{shared}$ to the first column, meaning $user_{private}$.

Table 4.1. Summary of all transitions in the interaction model

To / From	$user_{private}$	$user_{shared}$	$machine_{shared}$	$machine_{private}$
$user_{private}$	–	$user_{action}$	13	14
$user_{shared}$	21	–	23	$machine_{interpret}$
$machine_{shared}$	$user_{interpret}$	32	$machine_{action-int}$	34
$machine_{private}$	41	42	$machine_{action-ext}$	–

We have omitted the transitions involving $programmer_{private}$. This follows from the fact that programming has been completed by the time the user interacts with the machine. The remaining mark of the programmer is in $machine_{private}$.

We have already used the named transitions when using the umps model to analyse interactions in Section 4.1. We will return to them in the next section. We now discuss the transitions that are either not meaningful or not relevant to this thesis. Most of the diagonal is marked with – to indicate that it is not a meaningful transition since it is not a transition at all. (One might argue otherwise for $user_{private}$ -> $user_{private}$ which

corresponds to the user reflecting: although that is important, it is outside the interaction.) The one transition we show on the diagonal is $machine_{action-int}$ which represents the action, $machine_{private}$ -> $machine_{private}$. This is the internal actions of the machine. These can be important for scrutability since such internal reasoning by the machine may provide conclusions about the user model.

Table 4.2 gives the meanings of the 11 numbered transitions. The first column is the transition number from Table 4.1. The second gives – for impossible transitions and • where the transition is meaningful but not relevant to our concerns for scrutable user models for user-adapted interaction.

Table 4.2. Impossible and irrelevant transitions

Transition number	– Impossible • Irrelevant	Comment
13	–	User's mind directly causes machine action!!
14	–	Telepathy from user's mind to machine's private beliefs
21	•	User reflects on their own interface actions
23	–	User action becomes machine action
32	–	Machine action becomes user action (see 23)
34	•	Machine reflects on its action (symmetric with 21)
41	–	Telepathy from the machine to user (symmetric with 14)
42	–	Machine directly causes user action (symmetric with 13)

We begin with the transitions from $user_{private}$. Both are impossible as they do not go through the interface. So, transitions 13 and 14 are impossible. The symmetric transitions from $machine_{private}$, 41 and 42 are also impossible because they avoid the interface.

Now consider the transitions from the shared areas, the second and third sets of transitions in the table. The symmetric transfers 23 and 32 are impossible. The remaining pair, 21 and 34 are outside the scope of this thesis. Transition 21 is the situation where the user reflects on their actions. So, for example, the user may realise they made a slip so that $user_{shared}$ is not what they had intended. In practice, such reflection commonly depends on the user also analysing $machine_{shared}$. For the times when it makes sense for the user to check whether they actually acted as they had intended at the interface, we show this transition as valid. However, it is outside the scope of this thesis. Similarly, transition 34 where the machine reflects on its actions is beyond the scope of this thesis.

Communication outside the interface is certainly possible. It is also of considerable interest for broader user modelling research where the programmer goes directly to the user to elicit their beliefs. However, it is not part of user-adapted interactions. We take

both the user and the programmer as they come and do not delve into how the programmer acquired their beliefs, $programmer_{private}$. For the user, we are deeply concerned with the role of interactions at the interface in contributing to $user_{private}$ but ignore the many other experiences and thoughts that created it.

4.4 Analysis of meaningful transitions

The next step in analysing our model is to identify the actions involved in successful interaction. These are the named transitions in Table 4.1.

The transition, $user_{action}$, allows the user to volunteer information in a user-modelling dialogue. In an application, the user's actions might be logged by the machine as in our first example in this chapter. There is a fundamental difference between information the user has explicitly given and that deduced from observations at the interface with an application system.

The next action $machine_{interpret}$ involves the machine interpreting the result of the user action in $user_{shared}$. This transition is very important especially as one of the central roles of a user model is to enable the machine to improve its understanding of the $user_{action}$. For example, a user model can help a natural language understanding system interpret user utterances.

The $machine_{interpret}$ action is also important in much more conventional interfaces. For example, consider a situation where a user of a Unix system wants to change directories and the correct command is `cd 1_intro`. If the user accidentally typed `cd 1_intr`, omitting the last letter of the directory name, the command may work if the command interpreter can still match the misspelt name with a directory. This is an example of a conventional interface making a helpful interpretation of a mistyped command.

Now consider the actions, $machine_{action-ext}$ and $machine_{action-int}$. Both will be affected by the current state of the machine and $user_{shared}$. The important distinction between the two is that $machine_{action-ext}$ produces as result which is available to the user while $machine_{action-int}$ is invisible at the interface. In a user modelling context, $machine_{action-int}$ relates directly to the user model. In the general application, the main effects should be those requested in $user_{action}$. Any user modelling is a side-effect. The other machine action, $machine_{action-ext}$, describes the machine actions that the user sees: for example, a message indicating a successful file write. Importantly for this thesis, the explicit user modelling interaction $machine_{action-ext}$ enables the machine to share parts of its user model with the user as happened in the second example we analysed in this chapter.

Finally, $user_{interpret}$ means that the user 'understands' the information presented by the

machine. This is important for scrutability in that the user will need to interact with their user model, scrutinising it and interpreting the information presented at the interface. If scrutability is supported, the user must be able to interpret the information presented.

We now see how the umps model can be used to identify the *sources* of user modelling information during interaction. Table 4.3 shows interface actions (from the full set of transitions in Table 4.1) in terms of the mechanism it provides for external information to contribute to a user model. Note that bold font indicates user modelling actions.

Table 4.3. Classes of external user modelling information from interaction

Action	**Context**	
	User modelling	**Application**
$user_{action}$	**given**	observation
$machine_{interpret}$	*(explain interpretation)*	–
$machine_{action-int}$	*(explain action)*	–
$machine_{action-ext}$	$\mathbf{told}_{user\_model}$	$told_{application}$
$user_{interpret}$	–	–

Given external information: $user_{action}$ in user modelling context

We call this source given because the information is given explicitly by the user. In polite discourse, the listener normally treats the speaker's claims about themself as true. Essentially, given modelling information is the user's claims about themself. So, for example, suppose the user claims to know the undo command. It would be polite to treat this as a reliable claim. Suppose the user scrutinises the user model component representing their user's knowledge of the undo command. If the machine is 'polite', it will show that the user knows the undo command and it should be able to explain that this is because the user said they knew it. This would seem the usual effect of given information in the user model.

If the machine disregards the user's self-assessment, the scrutable user model would need to explain why. For example, it may be that the user does consider they know something when there is considerable evidence they do not know it. A scrutable user model would need to explain how the weight of evidence indicates that the user does not know the undo command, even though they believe they do.

It may also be that that given information needs interpretation. For example, when one teaching system asked the user to self-assess various parts of the syllabus (Murray, 1991) the user was judged to be more reliable in assessing *not* knowing than knowing. In general, there are many ways that a user's direct information about themself may be unreliable. As in the above case, the user may not always be capable of assessing

themself. Equally, a user may decide to give silly answers to a program's questions.

In spite of the very real problems with interpreting given information, it will often be the most reliable source of user modelling information.

Observations as external information: $user_{action}$ in application context

Suppose the user is working with an application like a text editor. If these actions are **observed**, they can give user modelling information. In general, this is a low reliability source of information. Firstly, it may require interpretation that is at or beyond the leading edge of artificial intelligence research: it is difficult to determine user intentions on the basis of actions. This is even more difficult if much of the context has been lost by the time we are able to analyse the user actions. Even if we have the context, we cannot easily distinguish purposeful user activity from a multitude of other sources of noise: the user dropping a coffee cup on the keyboard; a friend making short term use of the user's machine; the user working with a very knowledgeable friend standing nearby telling them what to do; the user working under stress; and the user just deciding to be silly.

We would thus expect that observations are often of low reliability. Moreover, the reliability of the program(s) that make the observations and interpret them are important in assessing the reliability of observation-based user modelling information. However, observations are an extremely important source of user modelling information because they are readily available and low cost. They do not demand the user's attention and effort. It is often easy to collect observations while the user does their normal activities.

Internal actions and their scrutability implications

The two interpretation rows $machine_{interpret}$ and $user_{interpret}$ in Table 4.3 cannot contribute external information about the user: the first is entirely internal to the system and the second to the user. Similarly, $machine_{action-int}$ is entirely internal to the system.

From the point of view of scrutability of the user modelling, both the machine's interpretation and its internal actions are important. The table indicates the need for explanations for these internal aspects of the machine. It shows these in parenthesis and italics since they differ from the actions which can provide external user modelling information.

We have already seen cases where the explanation of $machine_{interpret}$ was an integral part of explaining user modelling processes. In the second example of this chapter, the explanation of the value of the write component depended on the way that the machine interpreted several of the user's actions: for example, it interpreted use of an inappropriate

write command as an indication that the user did not know the correct one.

Similarly, the interactions in this chapter illustrated the explanation of the machine's internal actions, $machine_{action-int}$. In the second example of this chapter, the conclusions about the user's knowledge of the write command relied upon a chain of the machine's internal reasoning about both write commands and quit commands.

Both $machine_{interpret}$ and $machine_{action-int}$ for user modelling actions must support scrutability of the user modelling. We do not show a similar requirement for conventional applications since this thesis is not concerned with their scrutability.

Told external evidence: $machine_{action-ext}$

This action concerns all the information the *machine* has given the user. At first, one might not consider this to be a user modelling information source at all. Perhaps it is easiest to see the role of the told source in a teaching context. If a teaching system has presented some information to the user, that is evidence that the user knows it. Indeed, we saw that BGP-MS (Kobsa, 1990, 1992, Kobsa and Pohl, 1995) and THEMIS (Kono, Ikeda, and Mizoguchi, 1992, 1994) used just this reasoning. As any teacher will know, this may be a very weak form of evidence. Table 4.3 uses the term told for both user modelling and application contexts because either user modelling or application interactions can equally provide this form of evidence about the user.

Another important reason for a program to keep track of what it has told the user is that it may improve the quality of the interaction. Just as people try to remember what they have already told a person and avoid repeating it, so a program can avoid repetition or find new ways to present the same information. Similarly, in a multi-step dialogue the machine-state records some of what has already happened so that subsequent dialogue can build on it. If this is recorded for the long term, and is associated with the user, we regard it as long term user modelling information.

It is useful to treat told as external information because it is related to actions at the interface and outside the user model itself. This is easy to see in the case of interactions with an application.

Summary of external user modelling sources and scrutability support

The analysis to this point has identified exactly three classes of external user modelling information: given, observation and told. We saw that no other transitions in Table 4.1 could contribute information to the user model. Indeed, we saw that most were outside the interaction altogether.

We also predict the relative reliability of the sources will frequently be:

given > observation > told.

This means that information the user gives directly is generally the most reliable external source. The next most reliable source will usually be based upon observation of the user. The least reliable source is the weak evidence that telling the user something will cause them to believe or know it. This partial ordering on reliability of information sources is a useful generalisation and simplification which can be useful for the design of our scrutable user modelling shell.

4.5 Related work

We now relate umps to other interaction models. None of these models has been concerned with the particular problems of user-adapted interaction and user modelling (let alone scrutable user modelling). So we would expect that none of the models would distinguish interaction concerned with modelling the user from that of conventional applications systems.

However, as the quotation at the beginning of this chapter notes, all were concerned with improving our understanding of interactions, especially so that we can appreciate the causes of problems. So it is more surprising that the programmer is not explicitly represented in any of the models.

We introduced the programmer into the umps model primarily because scrutability support must come from the programmer. Once the $programmer_{private}$ beliefs have been encapsulated within the details of $machine_{private}$, they are interwoven with all the other details of the code and machine state. The umps model clarifies which explanations the programmer should supply in support of scrutability of the user modelling processes. An associated benefit is that we document additional information that can clarify the source of interaction problems.

Suchman's framework for analysis of interaction failures

This framework (Suchman, 1987) has already been introduced in the description of the umps model. It highlights problems due to the *narrow bandwidth* (Browne, Totterdell, and Norman, 1990, Goldstein, l982) of communication between users and conventional machines with conventional input/output devices. It was used to document problems in user interactions with a photocopier.

The umps model can be seen as an extension of Suchman's framework, with refinement

of the distinction between $programmer_{private}$ and $machine_{private}$. We have also added the distinction between the user modelling and application interactions so that we can refine our understanding of the scrutability implications of interactions. In addition, we have explored the transitions that can occur.

Although Suchman notes the asymmetry in the relationship between user and machine, her model creates a view of interaction where $machine_{private}$ is as inaccessible as $user_{private}$. So the columns associated with just the *user* and the *machine* appear symmetric. An important part of the asymmetry between *user* and *machine* is the degree of scrutability that is possible as well as that which is desirable. By adding the *programmer*, the umps model makes it clear that scrutability is one-sided since all scrutability support must be in $machine_{private}$. If there is any additional information needed in order to understand interactions, the umps model shows it in $programmer_{private}$.

Norman's interaction models

In early work on mental models, Norman described user interaction in terms of several elements that map directly onto our model (Norman, 1983). His use of the term, *system image*, corresponds to $machine_{shared}$. His notion of user's *mental models* are firmly located in $user_{private}$. The definition of *user model* corresponds to a user model in $machine_{private}$. Norman has no equivalent of $user_{shared}$ since the focus is on the way that a user understands a system.

In later work, Norman developed the well-known seven stage interaction model (Norman, 1986) summarised in the left column of Table 4.4. The right column shows the corresponding umps model.

Table 4.4. Correspondence between umps and Norman's seven stage model

Norman's interaction model	umps **interaction model**	
1. establishing the goal	$user_{private}$	
2. forming the intention	$user_{private}$	
3. specifying the action sequence	$user_{private}$	
4. executing the action	$user_{private} \rightarrow user_{shared}$	$user_{action}$
5. perceiving the system state	$machine_{shared} \rightarrow user_{private}$	$user_{interpret}$
6. interpreting the system state	$user_{private}$	
7. evaluating the system state with respect to the goals and intentions	$user_{private}$	

This comparison makes it quite clear that Norman's model is far more concerned with the user's private thoughts. Five of its seven steps are sub-elements of our $user_{private}$! Of our

five actions, Norman represents only the two where the user is the active party, namely $user_{action}$ and $user_{interpret}$.

Note too that steps 4 and 5 of Norman's model map to umps only for successful transitions, where the user action is what they intended and when the user accurately perceives the machine's output. The comparison in Table 4.4 highlights the role of Norman's model as a means for clarifying *user's* problems at the interface. Since our concern is for scrutability of the user model in user-adapted interaction, our model refines our understanding of the machine's role in the interaction.

Abowd and Beale's model

This extends the Norman model to include the machine (Dix, Finlay, Abowd, and Beale, 1993, Abowd and Beale, 1991). Figure 4.5 show its relationship to the umps model. We use bold upright font for their terminology, and the rest of the labels use umps terminology placed to show correspondences. Abowd and Beale's model has two actors, the User and System. The User corresponds to our $user_{private}$ and the System is rather like Suchman's notion of machine. Input and Output correspond to our $user_{shared}$ and $machine_{shared}$, respectively.

We have used an asterisk to mark parts of the umps model that are not available in Abowd and Beale's model. As we see, these are the refinements of the $programmer_{private}$ as distinguished from $machine_{private}$. In the case of actions modelled, Abowd and Beale's model has no equivalent of our $machine_{action-int}$. These differences between our model and Abowd and Beale's correspond to the aspects that are important in supporting scrutability of the user model in user-adapted interaction.

PIE

Compared with the models described to this point, the PIE model (Dix and Runciman, 1985) represents an increased concern for depicting the machine's role. It distinguishes what the umps model calls $machine_{action-ext}$ and $machine_{action-int}$.

PIE has the three elements:

P the sequence of user actions (programs) - our $user_{shared}$

I the machine's interpretation - our $machine_{interpret}$

E the effect of a program, where two parts are distinguished, which it refers to as the *display* (our $machine_{action-ext}$) and the *effect* (our $machine_{action-int}$).

The model can be applied at different levels of abstraction and leads naturally to the

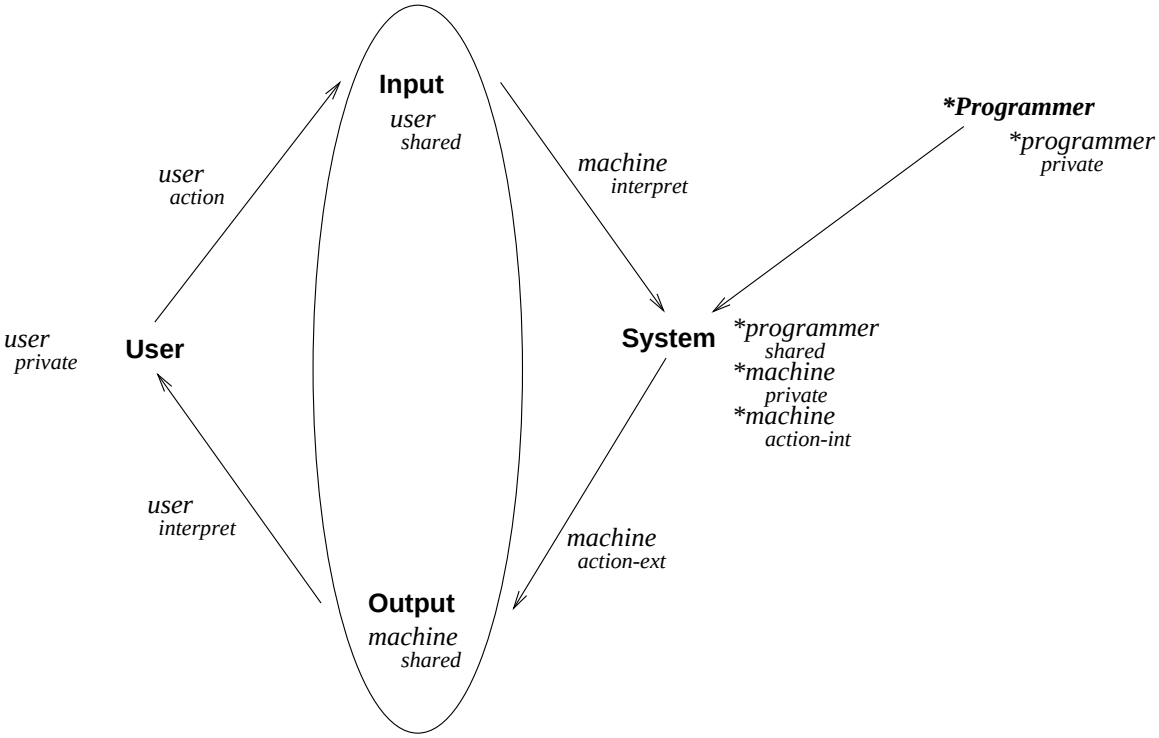


Figure 4.5 Correspondence between umps and Abowd and Beale model

principle that the aspects visible to the user ($machine_{action-ext}$) should reflect the internal state changes ($machine_{action-int}$).

In this model, there is no representation for umps's $user_{private}$, $user_{action}$ and $user_{interpret}$. Like the other models, the programmer does not appear. So this model too refines aspects of the interaction, but not those critical for scrutable user models for user-adapted interaction.

4.6 Summary

Our umps model of interaction provides foundations for the design of um, a scrutable user modelling shell. It clarifies the way that um must fit into the interaction which can contribute to the processes involved in user modelling. Further, by analysing umps, we have identified exactly three forms of external user modelling information: given, observation and told.

The umps model identifies three actors, distinguishing their private and shared beliefs:

- the user, who will be modelled on the basis of $user_{shared}$ since $user_{private}$ is not available to the machine;
- the programmer(s), whose private beliefs about users, user modelling and the domain(s) are held in $programmer_{private}$ and these beliefs should have been captured in $machine_{private}$;
- the machine at the time of the interaction, when $machine_{private}$ holds $programmer_{shared}$ combined with effects of $user_{shared}$.

A complete interaction has the stages:

- $user_{action}$
- $machine_{interpret}$
- $machine_{action-int}$ and $machine_{action-ext}$
- $user_{interpret}$

Our model highlights the points where there is potential for external user modelling information and the need to build support for scrutability:

- $machine_{interpret}$: is the process of the taking user actions and placing them in $machine_{private}$, a task for user modelling tools that collect and interpret $user_{shared}$ before providing it to the user model: we require that the user be able to scrutinise the role of such programs in the formation of their user model;
- $machine_{action-int}$: is the internal inference which enables the machine to reason about the user to add new inferences to the user model and the user will need to be able to scrutinise this basis for building the user model;
- $machine_{action-ext}$: is the aspects the user has been told by the machine, another source of external modelling information, provided by an application or user model scrutability support tool. It is also, of course, the way the machine communicates with the user and so is what the user sees when they scrutinise their user model.

The other two steps in the interaction, $user_{action}$ and $user_{interpret}$ are the user's side of the interaction. The machine does not need to explain these since scrutability does not require an explanation for the user's actions or their interpretations of $machine_{shared}$.

On the basis of the umps model, we have refined our understanding of the external sources of user modelling, identifying the three classes of external user modelling information:

- given - user modelling information given directly by the user;

- observation - where sensors or programs have observed the user's actions;
- told - where the user has been told something and hence may know about it.

We have also noted that there is often a partial ordering on the reliability of this information with

given > observation > told.

If a user model is to be scrutable, each aspect of it must be attributed to a programmer who is responsible for providing an explanation for the operation of their tool. Our umps model supports analysis of user interaction so that the system designer can document the mechanism for collecting user modelling information as was the case in the monitoring in the first example interaction of this chapter. The $machine_{private}$ must hold information needed to support scrutability. When we document an interaction using umps, the information shown only in $programmer_{private}$ is not available for the user to scrutinise. The requirements developed in Chapter 3 mean that information about the user model, its values and processes, must be available in $machine_{private}$, not just in the $programmer_{private}$ design decisions hidden within the machine.

Chapter 5

Scrutable accretion representation

This chapter introduces the *accretion* representation we have developed for scrutable user modelling. As we have seen from the previous chapters, there are typically many sources of information about the components of a user model. The sources are varied on many dimensions, most notably reliability. An essential task for the scrutable user modelling shell is to support a representation that can reason about the user from such varied sources. This problem of combining diverse sources of evidence has been described thus:

> When several sensors provide information, how do we recognise the nature of ignorance involved and select the appropriate model? How do we collapse them into more compact forms? How do we combine them? How do we manage redundancies, the correlations, and the contradictions? (Motro and Smets, 1997:245)

The authors also observe that little work has been done in this area. The accretion representation tackles this problem and, at the same time, supports the um requirements described in Chapter 3.

5.1 Overview of the accretion representation

To illustrate the properties of the accretion representation and operations, we make use of the example in Figure 5.1, which is an evidence list for a particular sam command called xerox.

value	source description	time (week #)
false	observation xan xerox	1
false	observation xan xerox	2
false	observation xan xerox	3
false	observation xan xerox	4
false	observation xan xerox	5
false	observation xan xerox	6
false	observation xan xerox	7
true	told coach xerox	7
false	observation xan xerox	8
true	observation xan xerox	8
false	observation xan xerox	9
true	observation xan xerox	9
true	observation xan xerox	10
true	observation xan xerox	11
true	observation xan xerox	12
true	observation xan xerox	13

Figure 5.1. Hypothetical example of evidence list

This component was constructed as follows. Each week, a program called xan analysed logs of the user's interaction with the sam text editor and identified interesting patterns of use in that week. If it found any error-free uses of the command in that week, xan provided a piece of evidence with the value true. For example, see the last five entries in Figure 5.1. Where there were no uses or any incorrect uses, a negative piece of evidence was created for that week. We can see this in the first seven cases in the figure. In addition, there is a coaching program which was run in week 7 of the example. It coached the user about xerox and provided the second piece of evidence for week 7 in the figure.

Let us now consider an intuitive approach to dealing with this collection of information about a component of a user model. One reading of the evidence in the figure is:

- there were 7 weeks where there were no error-free uses, suggesting the user either did not need xerox in this period or did not know it;
- in week 7, the coach taught the user about xerox;

- and then there were two weeks with a mixture of error-free and incorrect uses, suggesting that the user is in the process of learning of about xerox;
- and from week 10 on, there were exclusively correct uses which suggest that the user has learnt about the xerox command of sam.

This sequence of evidence characterises the type of information one might expect for a system which needs to deal with the user changing as they learn. The um representation needs to be able to manage such information and conclude about the value of the component, rather as we did in the last clause of each of the above bullet points. In addition, um must be able to support user scrutiny. So, for example, if the user were to explore this component up to the end of week 7, they should be able to find information which would enable them to see the value of this component and the evidence which was responsible for defining that value.

We note that this example is very simple. It has only external evidence. A typical component of a um model may have evidence which has been derived by various internal inference tools as well as multiple external sources of information. Even so, the example offers an interesting time series where the value of the component changes.

5.1.1 Basic accretion operations: accretion, resolution, destruction

There are three high level accretion operations. We now describe these in relation tot he example in Figure 5.1.

Accretion

This means the addition of a new piece of evidence to the list for a component. Each piece of evidence in in Figure 5.1 was added as one accretion operation. Because it is the the most basic operation, accretion gives the representation its name. Any source can volunteer information about a component. If the source is authorised to contribute to this component, its evidence is added to the component's evidence list. One view of the accretion representation is that each component has a steadily growing list of evidence, each piece provided by a source which authorised to contribute to that component.

Essential to the accretion representation is the goal of supporting scrutability in the form of explanation of all the evidence which was used to conclude about a component value.

Resolution

Since the accretion operation simply adds new evidence to a component list, there is no associated interpretation of that evidence. This is a significant departure from most of the shells described in Chapter 2. The addition of a piece of evidence does not trigger any conflict assessment or resolution. It is only when a um-consumer needs to know the value of a component that it is necessary to decide how to use a collection of possibly inconsistent evidence to conclude about the value of that component. That process of concluding a value is called *resolution*, since it has to resolve a value from the evidence available.

It is instructive to explore the nature of the resolution operation in relation to the example of Figure 5.1. We see that in the example, new evidence arrived each week. Suppose that the value of the component was only needed at weeks 6 and 13. In week 6, we suppose it was needed by the coach so it could decide whether to coach the user on xerox. If we examine the evidence available to the end of week 6, it seems that each item of evidence has the value false. Recall that each of these pieces of evidence was provided if there were either no uses of xerox or if the uses were incorrect. As we already noted, it may simply be that the user did not need to use xerox in that 6 week period. This case illustrates how difficult it may be to determine a value for a component.

Now consider the resolution of a value by the end of week 13. At that point there are 9 pieces of false evidence and 7 pieces of true evidence. Yet if we take account of the temporal information, we see that the last 4 weeks of evidence were exclusively true. Intuitively, we might resolve that the component is now true. There are many other ways to reason about this information. For example, the fact that true evidence began after the coaching might be important for concluding that the component is true at week 13. Just this type of reasoning was used by some of the systems described in Chapter 2. However, if we are very conservative about concluding that the user knows the component, we might conclude a component is true only after 8 weeks of correct use. This case illustrates the flexibility that may be required in resolution of a value for evidence in different situations.

The um scrutability requirements mean that the process of resolution of a component value must be available to the user. This means that the use should be able to access an explanation. For example, if resolution was based upon just the last four weeks of evidence, then at week 13, the explanation would convey the following information:

- the value of the component was determined on the basis of the information available over the last four weeks;

- in that period, the weekly analysis of your use of sam showed correct uses of xerox in each week;
- so it was concluded that you know the xerox command.

The accretion representation is based on the premise that there can be many different resolver operators. Intuitively, these corresponds to the fact that different people would assess a collection of evidence differently. For example, some people would conclude that a component was true whenever there was any evidence for this where other people would be more conservative in making such a conclusion.

Equally, people would assess a collection of evidence differently depending upon the situation. For example, if one were talking with a person who claimed they knew a fact, politeness might cause us to use this single piece of evidence to agree. On the other hand, in a safety critical situation, a more conservative interpretation would be appropriate.

These cases relate to the degree of conservatism in drawing a conclusion from evidence. There is another dimension of variability in interpretation based upon the evidence interpreter's trust in the individual pieces of evidence. Consider the evidence in Figure 5.1: most of it is based on observations that come from a single program and one piece comes from the coach. Some people would regard one of these sources as more reliable than the others. People might also judge the positive evidence from of a source differently from the negative evidence as, for example, in work by Murray (1991) who treated a user's claim of ignorance as more reliable than a claim of knowledge. In addition, we have timing information for each piece evidence. To conclude the value of the component at week 13, a resolver may treat the most recent evidence is more important than the older evidence.

In summary, there can be many resolution operators so that they can support a range of different reasoning about the component value indicated by its evidence list.

Destruction

This last operation destroys evidence, removing it from the evidence list for a component. In the case of Figure 5.1, suppose we knew that the resolvers only made use of the last 4 weeks of evidence. In that case, we can guarantee that all the earlier evidence can play no role in determining the value of the xerox component. So, we could remove all the earlier evidence without affecting the value of the component.

This is a dangerous operation, from the point of view of scrutability. For the example in Figure 5.1, suppose we deleted all but the last 4 weeks of evidence. The user is then unable to scrutinise the value of the model at weeks 6 or 7. Since those weeks were

important for determining and recording the actions of the coach, the user might well want to scrutinise that part of the model.

Destruction may also pose problems for future flexible use of evidence. For example, analysis of the evidence in Figure 5.1 suggests the user learnt xerox from the the coach. Similar analyses of the evidence for other components which were coached might show a similar pattern. This would then support a high level conclusion that the user learns from this coach.

Although we need to acknowledge the problems with destruction of evidence, efficiency may demand it. No representation can be allowed to grow continuously.

5.2 Accretion representation

The primitive objects our representation deals with are called *components*, the same term we use for the units of the user model. We refer to a component as $Comp_N$ when it has a list of N items of evidence. We define a function *elist* which returns the list of evidence associated with a component:

$$elist(Comp_{c,N}) = \begin{cases} \{\ \} \\ \{evidence_e\ \}, 0 < e \leq N \end{cases}$$

Each $evidence_e$ in $Comp_N$ has the following attributes:

- value: $\in \{\text{true}, \text{false}\}$ is the component value supported by this piece of evidence (in Figure 5.1 we saw examples of both);
- source_identifier: defines the source of the evidence (in Figure 5.1 these were xan xerox and coach xerox);
- time-stamp: records when evidence was added (in Figure 5.1 this is the week number);
- extra information: allows arbitrary additional information to be included in the evidence where this is to interpreted outside the accretion model (and there was none in Figure 5.1);
- scrutability information: enables a user to determine the meaning of the evidence: the component meaning, the source_identifier, time-stamp and the extra information (in the case of Figure 5.1, this would be a description of the meaning of the component xerox, a description of the xan xerox and coach xerox sources, explanation of the time stamp);

The evidence list has two roles: scrutability support and a representation of the value of the component.

5.3 Accretion operations

The accretion representation operations are shown in Figure 5.2. This shows the operations split into three categories we have already introduced: accretion, resolvers and destructors. We begin by describing the resolution operators which have the task of interpreting the list of evidence about a component in the system.

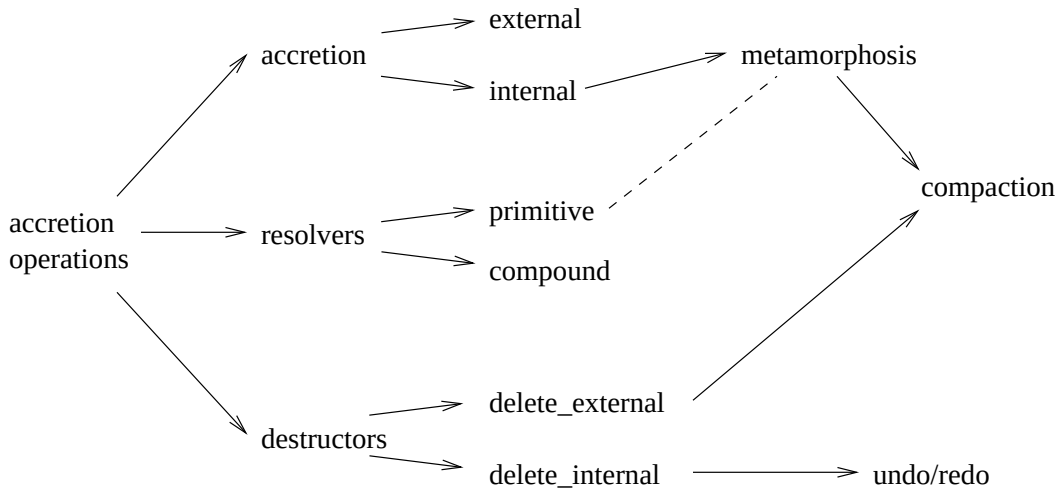


Figure 5.2. Accretion representation operations

5.3.1 Resolution operators

We distinguish the two forms of resolution indicated in Figure 5.2. The essential difference relates to matters of efficiency and richness of the machine's interpretations of a collection of evidence.

Primitive resolvers

These are the simplest resolvers. They determine the value of a component by taking the value of the *most reliable* piece or pieces of evidence in the list. Typically, this will be determined by simple characteristics of a piece of evidence: the time stamp; the source type (observation, given or told); the source identifier. So, for example, in Figure 5.1, suppose we use the following metric: the most reliable evidence is that added most recently. Then, in each of weeks 1 to 6, the last piece of evidence is false and so the value of the component would also be false. In weeks 10 to 13, the same metric returns the value true. For the weeks 7 to 9, there is no single piece of evidence and in each case, there are two pieces of evidence of opposite value. So the value returned would be

conflict.

A primitive resolver requires a function *max_rel* which maps from a list of evidence $max\_rel\{evidence_e\} = \{evidence_{MaxN}\}$
$\{evidence_e\}$ to a sub-list containing those pieces of evidence with highest reliability: where the resultant list has *MaxN* elements. Then, the resolver must determine the value from these. For example, one that returns a value for the component from a four-value logic:

$$resolver_{primitive}(Comp_N) = \begin{cases} true & \text{if } \forall\ i,\ 1 \le i \le MaxN,\ value(e_i) = true \\ false & \text{if } \forall\ i,\ 1 \le i \le MaxN,\ value(e_i) = false \\ no\_evidence & \text{if } MaxN = 0 \\ conflict & otherwise \end{cases}$$

Display 5.1. Primitive resolver

We describe this resolver because user-adapted interaction is often driven by systems which need to distinguish *no_evidence* and *conflict* from the pure boolean values. If a pure boolean value were adequate, one could easily define a similar resolver by folding the other two values into true and false. We would expect that the way this was done would depend upon the domain. For example, a domain where it was important to be conservative about concluding that a value is true might fold both *no_evidence* and *conflict* into false.

A primitive resolver provides an exceptionally simple mechanism. It should be very fast and low cost. It should be straightforward to explain to the user in terms of its process and the details of the metric for assessing reliability.

Compound resolvers

Against the benefits of a primitive resolver is its lack of power. One of the reasons for defining a representation based on evidence is that many items of low grade evidence should be able to collectively contribute to conclusions about a component. The multi-evidence resolvers determine the component value as a more complex function of the evidence list.

For example, consider the collection of evidence like that in Figure 5.1. We may wish to create a resolver tool which concluded the user knew the component only if they had used xerox correctly for at least 5 consecutive weeks and they had not made any incorrect uses in that period. This is a very tough test of knowledge. By week 13, we can see that the last 5 weeks do indeed reflect that the user has made 5 consecutive weeks of correct use. However, we can see that in week 9, there was also some incorrect use. So the collection of all the evidence for the last 5 weeks led to the conclusion that the user does not know

xerox.

In general, there may be many rich ways to reason about lists of evidence. A compound resolver is any tool which reasons about the value of a component by a mechanism other than taking the value of the single piece of most reliable evidence. Its reasoning more be more complex than the example give in the last paragraph. For example, it reason from several components' evidence.

An important reason for distinguishing a primitive and compound resolver arises in later sections when we will discuss compaction of the model by removing useless evidence. This is important for the efficiency and scalability of the modelling.

5.3.2 Accretion operators

Figure 5.2 shows the three forms of accretion: external, internal and metamorphosis. We now discuss these.

External sources

Evidence from external sources has to be accepted by the reasoning system as a fait accompli: there may be a complex process involved in collecting the evidence and determining its value, but this is not available to the reasoning system.

From the point of view of scrutability, an external evidence source is a black-box. Because the system does not have access to the processes internal to the source of external evidence, it can only require an ontological description of the evidence source.

In keeping with the geological metaphor in the names of accretion operators, we will also refer to these external beliefs as *ground* beliefs because they will act as foundations for the internal inference of the reasoning system which produced *inferred* beliefs. It is common for user modelling researchers to make essentially the same distinction between ground and inferred beliefs. The terms used vary. For example Self (Self, 1994) uses *basic* and *derived* beliefs and describes various systems with other terms like *explicit* and *implicit*.

Internal sources

By contrast, internal inference mechanisms lie within the accretion-based reasoning system and so are available to the system at all times. They can support user scrutiny of their processes by presenting the mechanisms and the way they operated to generate a particular piece of evidence. As Figure 5.2 indicates, there is a specialised form of

internal reasoning which we describe below.

Indeed, this is the point at which we require scrutability of the knowledge underlying the accretion-based reasoning system. This means $machine_{private}$ must be represented and managed in a manner that makes it available for scrutiny in $machine_{shared}$. This might be achieved by representing the knowledge explicitly, for example in a form like rules, where it is straightforward to enable the user to scrutinise the knowledge structures. Alternatively, the representation must be able to support some other form of explanation of the reasoning processes. We should be able to provide scrutability support for such inferences using the well established approaches from rule-based systems (Southwick, 1991) If the knowledge forms a complex knowledge base or if additional knowledge needs to be provided for the user to understand the system's knowledge base, a more sophisticated approach may be needed, as for example in (C Paris, Wick, and Thompson, 1988).

Internal inference creates a new piece of evidence derived from the resolved values of arbitrary components. For example, suppose the user is a sam-novice and an expert in vi, an editor in the same family as sam but without a facility like xerox. We may have an inference, which we express as a production:

sam-novice and vi-expert -> not xerox

meaning that if the left-hand-side is true, the inference creates a piece of evidence like this:

value	**source description**	**time** (week #)
false	inference, vi_expert_sam_novice, xerox	1

Internal-metamorphosis accretion

This is a specialisation of the inference operation. An important benefit of the accretion representation is that a collection of low reliability pieces of evidence about a component could be combined to give a higher grade conclusion. So, for example, in Figure 5.1 the early evidence showed no correct uses of xerox. Any one of these alone is a weak basis for concluding that the user does not know xerox. However, six consecutive weeks of this evidence may collectively constitute quite strong evidence that the user does not know xerox. The primitive resolver cannot do this since it makes a separate assessment for the reliability of each piece of evidence. Metamorphosis can coalesce several pieces of evidence into a new piece of evidence with higher reliability.

The metamorphosis process can be extremely simple. It may have a threshold for number

of positive evidence items required and the number of negative ones allowed and if this condition is passed, the new evidence is added to the component. For the example in Figure 5.1, a metamorphosis mechanism might use the last 5 weeks of evidence as follows. If at least three positive pieces of observational evidence occur in the last five weeks and no pieces of negative observational evidence in that time, it adds a piece of evidence like this:

value	**source description**	**time** (week #)
true	metamorphosis, many_successful_observations_over_time, xerox	13

The dotted line in Figure 5.1 links primitive resolvers and metamorphosis. This is because the representation offers two equivalent mechanisms:

- metamorphosis followed by use of a primitive resolver;
- a compound resolver which determines the value of an evidence list by aggregating evidence within the list.

We would generally favour the former mechanism since it has two steps, each of which should be easier for the user to scrutinise and understand.

5.3.3 Destruction operators

There are two destruction operators, one applying exclusively to external evidence and the other to evidence lists containing internal evidence. As we have noted above, an important difference between internal and external evidence relates to the support for scrutiny of the processes, with only internal evidence sources being required to support scrutiny of their processes.

To this point, we have described only accretion operators that make the evidence list grow monotonically over time. This may be unacceptable both in terms of the modelling processes and scrutability. Resolution will become increasingly slow as the list of evidence grows. Any other inference processes that deal with whole lists will also become slower. At the same time, it will become harder for the user to scrutinise the list: the user, like the resolvers, will need to study increasingly long lists of evidence to find those that actually affect the current value of the component. This could be a serious problem if the external sources can potentially provide huge amounts of very low grade information as, for example, in the case of sam monitor data.

Destruction of external evidence by compaction

Compaction replaces a component's evidence list a shorter one where:

- only external evidence is deleted;
- and that only if already metamorphosed;
- deleted evidence has no effect on the outcome from any of the resolvers applied to this component.

For example, suppose the only resolver used for the evidence in Figure 5.1 is a primitive resolver which takes the value of the most recent piece(s) of evidence. Suppose metamorphosis was used at week 13 giving a piece of evidence like this:

value	source description	time (week #)
true	metamorphosis, obs_xan, xerox	13

Suppose we now use a resolver which treats any metamorphosis evidence as more reliable than evidence of type observation. If we continue to use this resolver, the other evidence plays no part at all in the value of the component.

Essentially, compaction is a process that enables the component to forget about external evidence that has been coalesced into other, more reliable evidence. Compaction is a dangerous operation as it involves loss of information. This affects both scrutability and the modelling processes.

For scrutability, it means loss of the full set of external evidence that contributed to the existing evidence and current value. For example, the list of evidence in Figure 5.1 allows the user to scrutinise the complete set of evidence with time-stamps on each. The compacted form has less information available for the user.

For modelling, it means loss of flexibility in the future interpretation of the evidence list. It is only possible to compact a list in relation to a given set of resolvers. Future um-consumer systems may apply other resolvers which could have used the lost evidence.

Deletion of internal evidence as undo then redo inferences

This is another operation that enables the component to forget about irrelevant information, this time, information based upon internal inferences. Essentially, it relies upon the fact that the modelling system can re-apply its inference processes at arbitrary times. So, undo can remove all inferences from the model. Then the inference tools can be rerun. This is a simple mechanism for dealing with complex chains of inference that have been affected by changes in external observations.

For example, suppose that we had run an inference tool on the evidence in Figure 5.1 in week 5. For this discussion, the details of that inference are unimportant. Suppose it produced a piece of evidence like this:

value	source description	time (week #)
false	inference, observations_inferred, xerox	5

and suppose that we use a resolver which treats this as the most reliable evidence to that point in time. This single piece of evidence would then be used to conclude the component was false.

Now suppose that in week 13, we again want to assess all the evidence available to that point. The undo operation would remove the inference above. We could then rerun the inference mechanism which might now produce a piece of evidence like this:

value	source description	time (week #)
true	inference, observations_inferred, xerox	13

Intuitively, this corresponds to reasoning about evidence as needed, keeping inferences only until we are ready to review the reasoning about the external evidence available.

The undo operation is trivial. It simply means deleting all internal inferences. The redo operation requires re-execution of all the relevant internal inferences. In comparison with approaches like truth maintenance and belief revision, this is a low cost operation. Moreover, it produces evidence lists which are simple: this is important for efficient management of the model and for effective support for the user scrutinising their model.

5.4 Important user modelling representation problems

In Chapter 2, we saw common themes in the problems researchers need to address in user modelling shells. This section describes the way the accretion model manages a number of common user modelling situations that will typically cause conflicting information about the user:

1. long term changes in users as they learn or forget knowledge and change their preferences.
2. unstable user behaviour
3. erratic changes due to noisy evidence sources and inconsistency between external sources

4. changes in user's moods and other short term changes in the user
5. stereotype inferences overridden by more reliable evidence
6. stereotype trigger and retraction mechanisms
7. inconsistency in the user's beliefs
8. inconsistency between the system beliefs and the user's beliefs and the need for belief revision

This is not intended to be an exhaustive list of causes of conflict and uncertainty but it constitutes an interesting set of problems to be faced by conflict resolution tools and they illustrate how the accretion model deals with common user modelling problems. These types of problems have been important concerns for the systems described in Chapter 2. They have also been important in more specialised systems that have modelled users and they have been most important for epistemological modelling such as is the focus of student modelling. These are the types of problems that can be managed by the various classical and non-standard logics reviewed in (Self, 1994).

We now introduce a simplified notation for describing evidence lists. Figure 5.3 shows an abbreviated form of the evidence in Figure 5.1. It codes true as plus and false as minus. The evidence from each source is shown on a separate line. As before, it shows the evidence from xan as negative until the end of week 7 where there is evidence from the coach followed by two weeks with both positive and negative evidence. From then on, there is only positive evidence. By week 7 there is one piece of told evidence from the coach, as shown in the second row of Figure 5.3.

Time (week #)	1 2 3	4 5 6	7 8 9	10 11 12	13
xan xerox observation obs	− − −	− − −	− ± ±	+ + +	+
coach xerox told			+		

Figure 5.3. Hypothetical example of evidence list in time series

We will use this compact form for examples of evidence lists in subsections below.

5.4.1 Long term changes in users: learning and forgetting, changing preferences

An example of the evidence collected when a user was learning might be similar to Figures 5.1 and 5.3. Simple resolvers can determine the value of such evidence. For example:

- primitive resolution which operates in conjunction with a metamorphosis process to create a high reliability piece of evidence based on a weighted sum of the most recent external evidence;

- a resolver which returns true if there have been at least three recent, correct uses observed; false otherwise.

This example illustrates the role of domain dependent tools for resolution and metamorphosis. We cannot expect a single, simple, general theory for recognition of learning. The conclusion that a person knows something depends on the domain and um-consumer system since there are many possible standards of knowing: some may only require that the user knows *of* a concept and another might require that the user has been able to apply it in different contexts over long periods of time. Even within a single domain and um-consumer systems, we may need different criteria for judging knowing. For example, in our sam work, we applied one set of criteria for commands that the user had to *type* into the command window. Another, stronger set of criteria were used for *mouse*-commands that were easily to select accidentally.

A similar approach can apply to the user forgetting. A user who forgets may have an evidence profile like the xan line of Figure 5.3 with the + and − signs swapped. Then recognition of the user forgetting would be similar to that used to recognise learning.

It happens that in the xerox example in Figures 5.1 and 5.3, the external source provided both positive evidence on correct uses and negative information for non-use or incorrect uses. It is common in student modelling work to have sources contribute evidence only when the user does something. Then the same student as depicted in Figures 5.1 and 5.3 might have the profile shown in Figure 5.4 where blanks indicate no evidence was added in that week.

Time (week #)	1 2 3	4 5 6	7 8 9	10 11 12	13
positive only obs			± ±	+ + + +	+
coach xerox told			+		

Figure 5.4. Evidence list where only positives are recorded

In terms of the reasoning processes of the modelling system, we could simply treat the week where there is no evidence exactly as we treated those weeks with negative evidence in Figure 5.3.

Similar arguments can apply for changes in a user's preferences. A change might be reflected in new evidence that consistently indicates the change. In summary, a user learning can be expected to be characterised by a consistent period of negative evidence followed by a consistent period of positive evidence, possibly with an intermediate transition period. Primitive resolvers can handle this well. Forgetting is characterised by the opposite pattern and is also easily handled by a primitive resolver.

5.4.2 Unstable user behaviour

Another important problem for user modelling occurs when the user's behaviour does not follow a pattern. For example, a source might provide the evidence profile in Figure 5.5.

Time	1 2 3	4 5 6	7 8 9	10 11 12	13	
source	– + –	+ + –	– + –	+ – +	– + +	+

Figure 5.5. Evidence for unstable user behaviour

Depending upon the um-consumer system, we might use various resolvers to conclude the value of the component is:

- true: because the resolver uses the most recent evidence, or concludes the component is true if more than a third of the evidence is true - this would be appropriate where we favour assuming a component is true when there is some evidence for it truth;
- false: on the basis of the opposite of the reasoning just applied - the typical student modelling system tends to favour the conclusion that a user does not know something if there is a mixture of positive and negative evidence;
- conflict: because there are similar amounts of both positive and negative evidence both recently and over all the evidence.

The important point is that the domain determines which resolution strategy is appropriate. All are easily implemented as resolvers.

In addition, different resolvers might be needed for different users. For example, suppose we were modelling a student learning mathematics. If we know this student makes many careless slips, we might apply a different resolver from that suited to a student who rarely does this.

5.4.3 Noisy evidence sources

The profile of a noisy evidence source could well be similar to that in Figure 5.5. Then the management of this problem could be similar.

The important difference is that it should be more common for the programmer to know which evidence sources are noisy. Then the resolver would be chosen to deal with that noise. For example, if we know that the evidence source is likely to give false positive evidence the resolver would be different from that appropriate with some other distribution of noisy behaviour.

The accretion representation tags each piece of evidence with a source identifier. This means it has the potential to handle noisy and inconsistent evidence source by building

knowledge of the source's reliability into resolvers.

A similar approach applies for cases where there is inconsistency between different sources such as illustrated in Figure 5.6.

Time	1 2 3	4 5 6	7 8 9	10 11 12	13
source 1	– – –	– – –	– – –	– – –	– – –
source 2	+ + +	+ + +	+ + +	+ + +	+ + +
source 3	– – –	– – –	– – –	– – –	– – –

Figure 5.6. Evidence for inconsistency between evidence sources

There are several ways such evidence might be interpreted. If the programmer knew about some of the sources, the resolver could contain that knowledge. For example, some sources might give false positives but be reliable for negative evidence. Or, it might be that one evidence source is known to be less reliable than the others. If, in the example in Figure 5.6, it was known that source 2 was less reliable then the other sources, the resolver could easily treat it accordingly. There are arbitrarily many other possibilities. For example, the resolver might simply calculate the majority vote of all evidence sources at a point in time. The critical point here is that good interpretations of such conflicting evidence sources is straightforward to build into resolvers. The accretion representation's preservation of the history of evidence means that the resolver can make the best interpretation possible in terms of its knowledge of the domain.

5.4.4 Short term but stable changes in users

One important example of this problem arises where the user has stable preferences which are associated with particular situations or moods. For example, one set of music preferences might apply when the user is working and another when they are relaxing. Worse still, preferences might be affected by other people. For example, an adult user might have one set of movie preferences when alone or in the company of other adults. The same adult user may have a different set in the company of children.

This might be reflected in an evidence profile like Figure 5.5 or Figure 5.7.

Time	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
source	– – –	+ + +	+ – –	– + +	+ + +

Figure 5.7. Evidence for a mood related component

We cannot deal with this directly. What is needed is to convert this component into two or more components, one for each mood or situation. It might give the results shown in Figure 5.8.

Time	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
Component 1 source	– – –		– –	–	
Component 2 source		+ + +	+	+ +	+ + +

Figure 5.8. Evidence for a mood related component

At first glance, it is worrying that this problem is so similar to the others we have discussed. Essentially, what is needed is additional information, here coded by separate components for the user's different moods.

5.4.5 Stereotype inferences overridden by more reliable evidence

We saw in Chapter 2 that this has been an important problem for user modelling shells. The accretion representation can deal with it very simply: the resolvers treat stereotypic evidence sources as less reliable than others. This might be represented by the pattern shown in Figure 5.9.

Time	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
stereotype	–				
observational source				+ +	+ + +

Figure 5.8. Evidence for a mood related component

Here the stereotype source gives the first piece of evidence about the component and this indicates the component is umpsa. Some time late, the observation-based source starts providing evidence indicating the component is true. It is straightforward to define a resolver which can treat the observations as more reliable than stereotypes. Essentially, this is a special case of differential reliability of evidence sources. We have described it explicitly because it has figured in several user modelling systems described in Chapter 2.

5.4.6 Stereotype trigger and retraction mechanisms

This has also been an important problem in user modelling shells. Since stereotype-based evidence is a form of internal inference, the activation of a stereotype simply causes a piece of evidence to be added to each relevant component. Where retraction of a component is required, this is an undo operation affecting all inferred information, following by rerunning all inferences applying to this component. At this point, the stereotype will not be active and its inferences will not be added to the model.

5.4.7 User holds inconsistent beliefs

If the user holds inconsistent beliefs, the reasoning system may see this as instability in the user's behaviour. We have already discussed this so we now consider the more complex case where the values of different components are inconsistent. For example,

the user may claim to believe *x* and *x => y* but not agree that *y* is true. This problem has been a focus of the work by Kono, Ikeda and Mizoguchi (Kono, Ikeda, and Mizoguchi, 1992, 1994) who identified causes of inconsistency and responses to them, including the decision to allow the model to reflect the situation where the user maintains inconsistent beliefs.

In the accretion representation, this situation will be recognised if there is an inference tool which can infer from the two components *x* and *x => y* and add true evidence for *y*. At that point the evidence list for *y* might be like Figure 5.9.

value	**source description**	**time** (week #)
false	given, interview, y	1
true	inference, x_xImpliesy, y	2

Figure 5.9. Internal conflict in user information

This is another case where different conclusions might be drawn. We might use this information to drive a Socratic tutor which could present the student with the three sets of problematic information:

user claims to know *x*
user claims to know *x => y*
user claims to know *not y*

Then the user could be asked to resolve the problem. This is in line with much student modelling work. For example, Self (1994:213) points out that the the system might use situations like this 'to focus subsequent clarification dialogues with the student'.

On the other hand, a conservative and intuitive interpretation might treat the user's statements about their beliefs as more reliable than inferences derived from them. This amounts to accepting that the user is inconsistent: so the representation holds the beliefs *x*, not *y* and *x => y*.

Yet another possibility is that the inconsistency is due to differential reliability of evidence about each of these three components. A multi-component resolver might be created to reason across the evidence for several components in order to conclude about a consistent set of values.

5.4.8 Inconsistency between the system beliefs and the user's modelled beliefs

This is one of the major problems that has caused user modelling researchers to employ non-monotonic reasoning and belief revision systems. In the accretion representation, the

issues are rather different from those of the systems described in Chapter 2. One important difference follows from the availability of many resolution operators. Different resolvers will typically give some different outcome values. (Otherwise there would be no point in having more than one resolver). So we can only speak of inconsistency in relation to particular resolvers. This means that the whole issue of consistency only arises at the time that a um-consumer needs components evaluated. So, unlike the approach of a system like UMT or TAGUS, a um-consumer system would not normally invoke processing to deal with conflict as each piece of evidence becomes available.

Another important difference between accretion and the approaches taken by the systems of Chapter 2 is that accretion is based upon the assumption that much useful and interesting user modelling reasoning will be shallow. This means that the conclusion about a user will be based upon external evidence plus a short chain of reasoning. Much interesting user modelling will be feasible from a collection of evidence from external sources plus the resolver mechanism. Shallow reasoning is also important for scrutability since it should be simpler for the user to scrutinise. The problems of inconsistency is such situations are just those we have described the subsections above.

Nonetheless, there is an important class of inconsistency where the system's knowledge of the relationships between components enables it to detect apparent inconsistencies between component values. For example, suppose a sam model indicates the user knows none of the components in the basic sam commands but all of the the most sophisticated ones. There are two parts to dealing with this type of problem. First, a system must identify the inconsistency. Then, it must have a means for correcting the problem.

On appealing approach for managing such inconsistency would be based on a form of truth maintenance, as used by various systems described in chapter 2. In many ways, truth maintenance is closely related to an accretion style of reasoning since it keeps lists of the justifications for believing a component or for disbelieving it. Although truth maintenance is not a fundamental part of an accretion representation, we could include truth maintenance tools. We now describe three ways that this might be done in a um architecture. We emphasise that the same arguments could be made for other approaches to managing inconsistency, for example belief revision based reasoning.

Truth maintenance as a part of the um-consumer's responsibility

We assume that the um-consumer must have access to domain knowledge needed to identify the problem of inconsistency in the user model. After all, if a um-consumer is operating in a domain like coaching, we assume that it needs domain expertise to drive its own actions. Rather than duplicate all or part of this in the user model, we assume that

the um-consumer will use its domain expertise both in defining what it needs to model in the contexts it uses and in identifying problems like inconsistency.

To correct the problem, the most obvious course is for the um-consumer to interact with the user to clarify the situation. For example, it might ask the user to assess their own knowledge. Or it might start a tool which sets a diagnostic task to judge the user's knowledge. Similarly, in a preference domain where the model represents preferences for pieces of music. The um-consumer could use start a tool which asks the user if they the particular pieces of music in question. Such a tool can contribute evidence about the component as an external source.

This approach also has the merit that the um-consumer determines which components need to be consistent. It also determines which resolver to use in defining the notion of consistency. Further, it might determine whether to apply truth maintenance to the task of identifying inconsistencies and then whether to attempt to rectify them.

Truth maintenance as an internal inference accretion operation

This would operate in the following steps:

- select the set of components which need to be internally consistent within the frame of reference of a particular collection of knowledge;
- invoke a belief revision or truth maintenance tool over this set of component values;
- once it has selected an environment (consistent set of component values), it adds a piece of true evidence to each component that this process infers to be true and a piece of false evidence to each determined to be false. Different possible worlds can be represented by different evidence identifier parts.

For example, the first three pieces of evidence in Figure 5.10 are based upon observations by an external sources. An internal inference truth maintenance tool might have provided the last two pieces of evidence. The first of these is for one world, identified in the evidence of the figure as world1. In this world, the component is true. In world2, a second world provided by the same tool, the component must be false, as indicated by the last piece of evidence in the figure.

Of course at this point, the effect of this process on the actual value of a component will depend upon the resolver determining component values. So, for example, one resolver might treat evidence from the truth maintenance system as the most reliable form of evidence, and hence use it to determine the value of the component; another might not do this. In a case such as depicted in Figure 5.10, it would also need to operate within one

possible world.

value	source description	time (week #)
false	observation xan xerox	1
false	observation xan xerox	2
false	observation xan xerox	3
true	inference atms world1	4
false	inference atms world2	4

Figure 5.10. Hypothetical example of evidence list

5.5 Beyond resolvers: inferences from evidence lists

The accretion representation has been designed so that useful resolvers can be simple, using assessments of the relative reliability of evidence to determine a value for the component. Clearly, evidence lists permits much richer interpretations. For example, suppose the user does not want to learn about xerox. The trace of evidence might look like that in Figure 5.11.

Time (week #)	1 2 3	4 5 6	7 8 9	10 11 12	13
Xum xerox observation	– – –	– – –	– ± –	– – –	–
coach xerox told			+		

Figure 5.11. User who does not want to know about xerox

This might be explained as follows: the user was told about xerox, tried it, did not see any use for it and decided not to learn about it or use it. Such reasoning could provide evidence about components modelling the user's learning goals.

Analysing evidence across components could also be fruitful. For example, we might conclude from the evidence pattern in Figure 5.1 that the user learnt from the coach: the evidence list has negating items until the coaching and supporting ones starting after the coaching. This suggests the coaching occurred at about the time the user learnt to use this command.

A similar profile across other coached commands would suggest that this user was learning from the coach. If we used two coaches, each with a different teaching strategy, we could analyse the aspects taught by each to look for differential outcomes. Then we might reason about the teaching strategy that seems most effective for this user in this type of domain.

The point is that collections of evidence have the potential to be interpreted in many interesting ways. The accretion representation normally keeps all the evidence about a component in uninterpreted form, ready for such analysis.

5.6 Discussion

An important and unusual aspect of the accretion representation is the role of multiple resolvers and hence multiple interpretations of the same evidence lists. We now discuss how this late binding of a value to the evidence list affects the representation as well as its implications for scrutability.

Late binding and tolerance of inconsistency

The accretion representation applies late binding to the interpretation of evidence about a component and it can allow inconsistency between components. Consider the following typical situation in user modelling. Early in the modelling of the user we have quite incomplete sets of modelling information, much of it from rather unreliable sources. If this is contradictory, it may not be worth doing anything about it until more reliable information becomes available, possibly by metamorphosis. This approach is especially appealing if the um-consumer does not yet need the information and may not do so until there may be more reliable information. For example, our sam coach ran just once a week: it only needed to update the user model immediately before it generated its advice.

In addition, we consider that conflict across different domains may not be important. This means that we tolerate a situation where two contexts are each self-consistent but inconsistent with each other. This is similar to the approach taken by McCalla and Greer (McCalla and Greer, 1994) and Huang (Huang, 1994) who have explored various mechanisms for avoiding the need to aim for consistency across the whole user model.

Implications of multiple resolvers

There are also important scrutability implications that arise from supporting multiple interpretations of the evidence. The most striking is that we cannot give the user a single, simple statement about the value of a component.

We can, truthfully, report the evidence that we have about a component. We can then explain the mechanism used by *particular* resolvers. However, the user model, of itself, cannot be used to explain the way that arbitrary um-consumer systems will interpret the evidence.

The problem is not as bad as it might at first seem. We would expect that in most cases, one collection of components relevant to a particular domain would be interpreted by a small number of resolvers. In that domain, the user would need only to come to terms with those resolvers.

We could have designed the accretion representation with a small number of built-in resolvers. This approach might be appropriate if one were using accretion to represent a domain where the appropriate resolution processes were well understood. However, in generalised user modelling that can support diverse domains of user-adaptation, there is no such understanding of appropriate resolution mechanisms. So the accretion representation allows flexibility with the possibility of any number of resolvers.

5.7 Related work

Our representation is a form of endorsement based reasoning (Cohen, 1985) which he describes as follows (1985:176):

> endorsement based reasoning about uncertainty .. is based on the idea that we have *reasons*, called endorsements, for believing or disbelieving hypotheses and that these can be used
> • to weigh evidence;
> • to model changes in beliefs over inferences;
> • to decide whether a result is certain enough to be used in subsequent reasoning;
> • to represent evidential relationships (such as contradiction) between conclusions; and
> • to design new tasks with the goal of decreasing uncertainty.

The first three aspects are important for our representation. The second last is managed by internal inference tools. We leave the last to a combination of a special type of external knowledge source, the elicitation tool which is invoked when a um-consumer system finds that current beliefs of the system are too weak.

We might expect endorsement-based representations to have a particular appeal for user modelling. They have been used in a teaching system (Murray, 1991), and for modelling user preferences (Elzer, Chu-Carroll, and Carberry, 1994). Endorsements have also been applied in belief models (Galliers, 1992).

Accretion has much in common with the first endorsements developed by Cohen. Like him, we retain the source of the evidence since sources differ in reliability. Also common is the representation of the type of evidence since different types also have different reliability. The types of evidence from external sources are those identified in Chapter's 4 interaction model. For the internal inferred evidence sources, we distinguish stereotypes from other knowledge based reasoning, described in Chapter 2. We do not require an accuracy indication as Cohen does, but can support this as an extension to the basic representation.

We take a different approach from Cohen in the propagation of inferences. We maintain a tools approach. So we provide a selection of tools for ranking endorsements; no single one is built in. So, for example, we could use a tool which implemented a similar mechanism to Cohen's (1985:181); this does not differentiate between one poor and one

strong endorsement compared with two medium-reliability endorsements. But other tools would also be available to the um-consumer system.

The accretion representation can be combined with other approaches to management of uncertainty. Any one of these can be an internal source of evidence. For example, tools could support the many probabilistic models of reasoning (Shortliffe and Buchanan, 1975, Duda, Hart, and Nilsson, 1976, Szolovits and Pauker, 1978), or fuzzy logic (Zadeh, 1978). All of these coalesce a set of evidence into a single value. Since these can summarise much evidence succinctly, they might also be part of metamorphosis or compaction tools.

5.8 Summary

Accretion is a representation which is similar to endorsements in that it maintains the list of evidence which informs the value of a component. It has three basic operations.

- *Accretion* causes the addition of a new piece of evidence. We distinguish three classes of accretion operators:
  - *external* where the evidence added is from an external source;
  - *internal* where the internal reasoning of the system contributes the evidence;
  - *metamorphosis*, a specialisation of *internal* reasoning where the evidence added was derived from a two or more pieces of evidence within the list for a component $\{evidence_i\}$, giving a new piece of stronger evidence $evidence_j$, which summarises the collective strength of $\{evidence_i\}$.
- *resolution* operations interpret the evidence for a component to determine a value for a component. We distinguish two classes of resolvers:
  - a *primitive* resolver concludes the value of a component by taking the value of the single piece of evidence it determines to be the most reliable (where that piece of most-reliable evidence may well been generated by a *metamorphosis* operation);
  - any other form or resolver is called a *compound* resolver as it reasons on the basis of multiple pieces of evidence, be they from the evidence list for the component whose value is being concluded or other components. (Note that the effect of a compound resolver can always be achieved by a primitive resolver, combined with internal inference, often metamorphosis.)

- *destruction* operations remove information that is no longer useful. Here too, distinguish two classes of destruction operations:
    - *compaction* removes *external* evidence which has already be metamorphosed and which can have no effect on the outcome of any of the resolvers applied to this component' evidence list;
    - *undo* removes all the evidence from *internal* inferences where this will typically operate in conjunction with the *redo* operator which will generate the internal inferences anew.

This representation provides simple mechanisms for dealing with many interesting problems that arise in user modelling: changes in the user due to learning and forgetting, erratic user behaviour as well as noise in evidence sources, short term changes in users due to effects like moods, stereotype inferences where more reliable inferences override default inferences due to the stereotype and inconsistency in the user's beliefs.

The more complex problem of inconsistency between system beliefs and the user's beliefs can be managed by a range of different mechanisms managed by the part of the system that contains the system's domain knowledge - be that in um-consumer or the internal inference mechanisms.

Throughout this chapter we have treated the accretion representation only in terms of boolean valued components. The same approach can be applied for other types. Perhaps the most important is numeric types. The essential principle is that each piece of evidence contributes to a set of the allowable values for that component type. The task of the resolver is to deal with the collection of evidence and for numeric types, this would require a different approach from that applicable to booleans.

Chapter 6

um : the programmer's view

The programmer needs to be concerned with the details of the um representation, the tools and their organisation within the architecture. The representation is based upon the umps model described in Chapter 4 and and the accretion representation of Chapter 5. This chapter devotes one section to each of the main blocks in the architecture shown in Figure 6.1: the representation of the persistent form of the model, tools for external user modelling information; tools for internal inference; conflict resolution tools for managing of consistency, certainty, noise and change; and um-consumers which control the modelling tools.

The remaining block in the figure is the tools for scrutability support. These are primarily for the user's view of the um model, and are so important as to deserve a separate chapter (Chapter 7). However, we also discuss scrutability issues throughout this chapter because all the major elements of um have been designed with concern for supporting scrutability

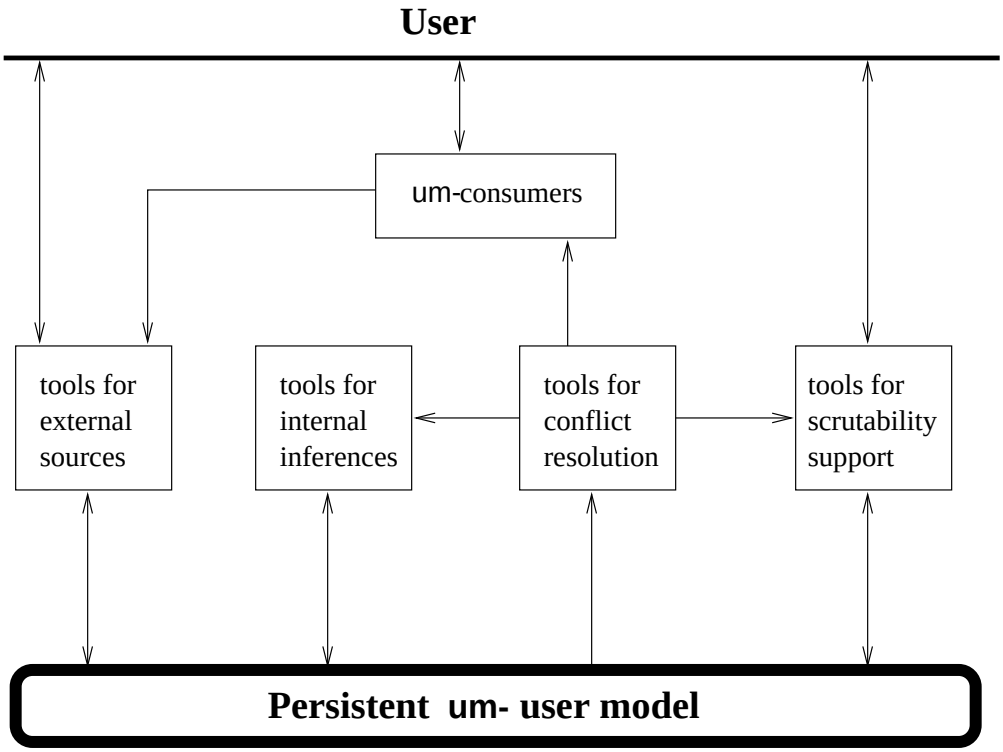


Figure 6.1. Persistent user model within architecture

in addition to the base user modelling requirements.

To support the descriptions in this chapter, we use examples taken from two domains where um has been applied, the samcoach which sends advice to users about the text editor sam and a movies recommender.

6.1 Representation for the persistent form of the user model

The persistent user model is shown as a heavy ellipse in Figure 6.1. We begin by describing the elements of a *component* definition. This consists of two sets of elements: *individual* and *central*. The individual aspects are associated with an individual user while the central elements are part of the um definitions that apply for many users.

The basic element of a um model is the *component*. Table 6.1 shows the parts of the representation in the left column and an example is given in the right column. The bold headers in the left column indicate the main parts of a component. Those shown in square brackets are optional. The double horizontal line separates the central modelling *definitions* above from the individual *values* below. The remainder of this section describes the elements of Table 6.1.

Table 6.1. Representation of a component

Component element	Example
Central elements	
Declaration elements	
name	quit_k
description string	how to use the Quit (q) command correctly
explanation	*The quit command is available in the* sam *command window. It has the form* q <return> *Normally, this causes* sam *to quit.* *However, if there are problem, it may give a warning report instead.* *For example, if you have not written out all your files, it will tell* *you so. You should read these warnings carefully.* *If you quit again,* sam *will assume you really mean it and it will quit* *even if some files have not been saved.*
type	knowledge
[extra]	alt_expln explain editors/sam general/learn_prefs
Individual elements	
[**Value information**]	
name	quit_k
[extra]	
[**Value list**]	
value	true
resolver	tutor_resolver
time	741323816
[extra]	certainty 0.8 to 0.9
[**Evidence list**]	
each evidence-item has	
value supported	true
evidence class	observation
source identifier	mcons
part identifier	quit
time	Sat Feb 7 22:55:11 EST 1998
[extra]	COUNT=342

6.1.1 Central component declaration elements

The *declaration* elements define the concepts in the user modelling ontology. These are central elements since a component should have a consistent meaning across the models of different users.

Name, description and explanation

The first parts of a definition specify a *name* and a *description*. The name is primarily an identifier for use by um tools. The description should be longer and more meaningful. It is also intended for use by um-consumers: so a coach can generate text like '*You appear to know* how to use the Quit(q) command correctly'.

In addition to this, we require a fuller *explanation* of the component such as the example in the table. This is for use by um-consumers and the scrutability support tools that need to explain the meaning of the component to the user. The example in Table 6.1 was handcrafted.

In general, the explanation can be provided by a subsystem which might generate explanations. As noted in Chapter 3, the explanation subsystem can optionally be provided arguments specifying the user and um-consumer. For example, in the sam work, one explanation subsystem took the user's sam-expertise to customise its explanations. It gave terse, abstract explanations containing jargon for the expert sam user. By contrast, the novice explanations were simpler, less complete, included more examples and avoided jargon.

Type

We distinguish four component types:

- *preferences* like a movie advisor's components for the various genres and subject preferences;
- *knowledge* for most of the components in sam user models, including the example in Table 6.1;
- *beliefs* are non-mutual beliefs;
- *attributes*, a catch all for anything else.

The type is used in generating text descriptions of a component. So, a knowledge component value can be described with '*You appear to know* how to use the Quit(q) command correctly', where the italics stem comes from the type and the rest from the component description. A preference component value can be described by a corresponding stem, as in this example: *You appear to like* Starwars.

Another reason for distinguishing the four types is as an aid to comprehensibility of the model. For example, scrutability support tools can display each differently.

Our use of the term 'belief' deserves comment. Firstly, we need to explain why we use it for a concept that already goes by various names, including *misconceptions*, *bugs* or *mal-knowledge* in the student modelling community (as for example, in Brown, Burton, and Larkin, 1977, Johnson and Soloway, 1985, Wenger, 1987).

Consider the problems with a term like 'misconceptions' in an example from the sam domain. We model whether the user *know*s how to use the quit command. We also model whether the user appears to *believe* that the best way to quit is to kill the sam

window. This is a faster, but somewhat riskier method of quitting than the quit command. The major benefit of the proper quit command is that one is warned about various potential problems before quitting makes them irreparable. Many users ignore warnings anyway! For such users, kill-to-quit method may be better. Although it is the norm for student modelling to refer to cases like this as bugs, mal-knowledge or misconceptions, we prefer a more neutral name.

A second difficulty with our use of 'belief' is due to its common use for a much broader concept in the research in modelling users and agents: for example, one might model the system's beliefs that the *user wants to print a document*, or the *user likes Star Wars*, or the *user knows the* sam *quit command*. The difference between our knowledge and beliefs correspond to BGP-ms partitions for user beliefs shared by the system and those that are not (Kobsa and Pohl, 1995).

Countering these difficulties, our design makes use of *belief*, because it seems natural. This is important since we make the persistent form of the model available to the user and need to express it naturally. Although our choice is at odds with the technical use of the term in the belief revision community's research, it seems intuitive even there. For example, the theoretically rigorous treatment of belief in SMMS is accompanied by informal descriptions of misconceptions like the following examples (Huang, McCalla, Greer, and Neufeld, 1991:95) where we have added bold font for emphasis:

- 'the student **believes** the Lisp *car* function returns a list containing the first element of the given list'
- 'if the student **believes** that *append* is the same function as *list* then she/he must not know that *append* requires lists as its arguments'

and correct knowledge is described informally as:

- The student **knows** the concept of recursion'
- 'if the student **knows** the function *mapcar* then she/he must also know the function *car'*

Attribute components can hold numbers, strings and enumerated types in addition to booleans. So, for example, they might be used to model a goal, like 'wanting to learn the minimal amount about sam': we do not have a separate component type for the user's goals. Numeric attribute components are used in cases like *wpm_info*, an estimate of the user's typing speed in words per minute. Attribute components must store any additional information they need in the optional extra field for the component.

Optional extra field

This *extra* part is for anything that a tool needs to lodge in this component. It can hold information needed by *extended* representations. The example in Table 6.1 shows its use for the name and arguments of a program that should be executed to explain the component: it could equally well hold a program. Whatever this field means, it is interpreted outside the basic um framework.

6.1.2 Individual component value elements

We now move to aspects in the *individual* user model. These determine the value of the component. All are optional. This means that a component may be defined within the modelling system but not be used for a particular user.

Individual components are kept on a per-user basis. Normally, this information is kept in the user's own filespace. In demonstration systems or other cases where the user does not have a filespace, the um-consumer creates a password protected individual user model filespace in a directory for that user.

Name

Where a component is modelled for an individual, its name must be specified. This must match the corresponding component definition name. The name also constitutes the minimal representation for a component.

Associated with this is name is an optional extras element. This permits arbitrary information to be kept with the individual component.

Value list

Associated with the name is an optional list of values. Each is represented as a tuple: the value; the name of the resolver-tool used to determine the value; the time it was calculated; and the optional catch-all *extra* slot. Resolvers normally calculate a value from the evidence about a component. Retaining this value as part of the component is optional. It may be useful where efficiency is important. For example, a um-consumer may resolve all the components it needs and then use the stored values. This will be important if the resolver is slow, response time critical and much of the model changes very slowly.

A value element can optionally hold extra information about the value. For the example in Table 6.1, this is a certainty range determined by the resolver.

6.1.3 Evidence lists

Lists of evidence are the bases for reasoning about a component value. The example evidence of Table 6.1 records that this user was observed to use quit 342 times up to the time on that evidence.

Value supported

The single piece of evidence in Table 6.1 contributes to the conclusion that this component is true. The basic um accretion representation manages only boolean components. In this case where the component is of knowledge-type, this means it models whether the user knows it or not.

Evidence class

There are five evidence classes. These come from either external user modelling sources or from internal inference tools.

Chapter 4 identified three distinct sources of modelling information available at the interface: given for information information given directly by a user; observation for observations of the user as in our example in Figure 6.1; and told for information the machine has presented to the user.

The two classes of evidence that come from internal inference are stereotypes for stereotypic default assumptions about the user and rules for other knowledge based inference tools.

Evidence source and part information

The evidence in Table 6.1 was added by the *quit* counting part of a tool called *mcons* (the Model CONStructor tool that analyses sam monitor logs to construct a model). Each piece of evidence must be tagged with the identity of the tool that contributed it. The *source identifier* indicates the program that added it and the *part identifier* qualifies this, so that a program which has several mechanisms for establishing evidence can indicate which was used for any particular piece of evidence. This information is used as the link to explanations of the source of each piece of evidence.

Evidence time stamp

The time stamp on evidence is provided by the user modelling system at the time that the evidence is added to the model.

Evidence extra information

The extra field is optional. In this case it records the number of times that mcons observed the user quitting.

6.1.4 Minimal representation

To match the requirement *model_simplicity* from Chapter 3, the simplest form of individual model representation is a *name*. Of course, there must also be a corresponding um definition for the component. For example, tracking the movies that have been recommended by a movie advisor is a very simple modelling task. It can be managed in the minimalist form. The um definition must provide the information down to the double lines in Table 6.1. Then, the name-identifier of each movie recommended to a particular user is stored in their individual model. To determine if a movie has been recommended to the user, the resolver tool checks for the presence of the movie identifier in this user's model.

The next level of complexity allows for a *name* plus an *arbitrary string* in the extras field for the individual component in Table 6.1. The extra string is interpreted outside um.

6.1.5 Scrutability support

In keeping with the tools approach of um, explanation management is a separate tool. We expect there could be several explanation subsystems, each offering different services. A user or um-consumer can employ the one most suited to its needs.

Since explanations are available across different users, they must be part of the central definitions. So each tool that can contribute evidence must have a central definition for its explanation.

A user should be able to scrutinise the model in the following ways:

- component meaning comes from the explanation slot of the central definition or from an alternate explanation subsystem given in the extras slot;

- explanation of each piece of evidence is in terms of central definition for the tool (for the example in Table 6.1, this is an explanation for the quit-command part of mcons);
- resolver operation explanations are in the central definitions (for the example in Table 6.1, this is an explanation for tutor_resolver).

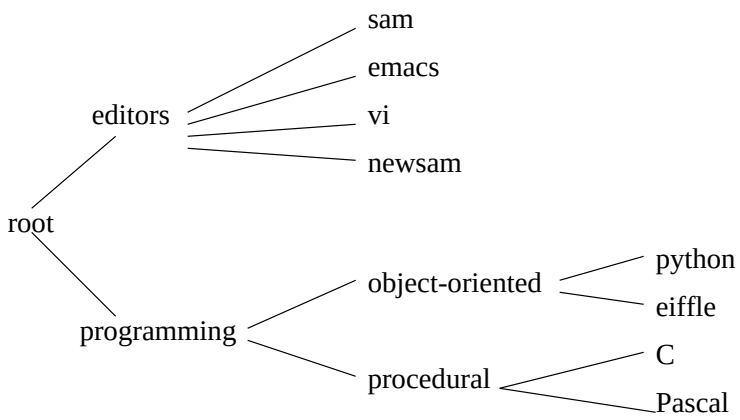
6.1.6 Structure: contexts and partial models

The *model_structure* requirement calls for a mechanism for limiting the scope of component identifiers and groups of related concepts. This is essential for scalable user modelling from the programmer's point of view. For example, a programmer building a model about sam should not need to know about details of the user model associated with a movies advisor. Conceptually, the context of an identifier is one user modelling domain.

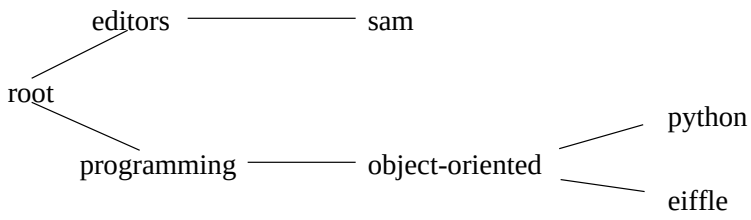
Contexts are organised in a hierarchy. The central um hierarchy is mirrored in the individual user models as shown in Figure 6.2. The top tree depicts a central hierarchy with a number of contexts and sub-contexts. Below are corresponding hierarchies for two users: neither of whom have models for all the contexts. Each has only some of the contexts in their user model. The central context contains the definitions needed for the model: component definitions including explanations for components and the various tools that operate in this context. The components and tools to be used within a context should be defined either within that context or at a parent context. Since many tools will need domain-specific knowledge, it seems likely that most definitions will either be at a leaf context or the root of the context tree.

Note that Figure 6.2 shows two contexts for sam models, one called sam and the other newsam. This might happen when two sets of modelling work happen to operate over the same real-world domain but have been created independently, potentially having different and conflicting ontologies. Or they might reflect early work on sam with one set of tools being used and later work in newsam being driven by different tools.

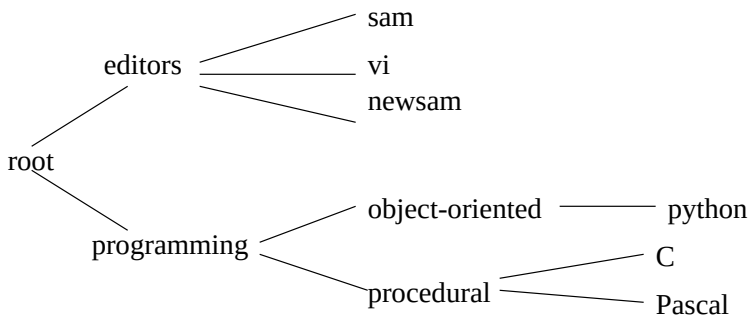
The context is useful for high level structuring but we also need a mechanism that can structure the components within a context. For example, consider a context for a movies advisor. At one point in time, suppose this models the user's preferences for 300 actors, 100 directors, 50 genres, 100 subjects and 1000 movies. It makes sense to structure these, separating the components for actors, directors and so on. We do this by defining *partial models*, each with a collection of related components. This additional structure aids both the programmer and scrutability. In both cases, it makes it easier to understand the model because we can structure the components into a hierarchy.



Central context hierarchy



Context hierarchy for user 1



context hierarchy for user 2

Figure 6.2. Context structure centrally and individually

Consider the context hierarchy in Figure 6.2. Within each of its leaf nodes, there would be a collection of partial models. For example, the first level of partial models in our sam models has basics, more_useful, very_useful_mouse, powerful and mostly_useless. Within basics, is a simple collection of components. Other partial models contain further partial models. For example, more_useful has separate partial models for mouse and keyboard commands. We saw a depiction of this tree in the model displayed in Figure 3.3 at the end of Chapter 3.

In the movies model, we used this minimalist form of partial model, with one partial model for the movies the user had seen, another for the movies that had been recommended and yet another for those rated. More detailed forms of partial model, similar to that in Table 6.1, modelled the user's personal information and preferences for various movie genres and subjects.

There is no pre-defined semantics attached to contexts or partial models. However, a um-consumer or user might be able to associate semantics. In the case of the editors context, the sub-contexts sam, emacs and vi are specialisations, each being a particular editor.

The context and partial model structures form a directed acyclic graph. This means that partial model can be shared by two or more contexts. For example, one partial model in the sam context represents the user's knowledge of regular expressions. Some of the concepts involved in this sub-context are identical to regular expressions in Unix. In one experiment, we created models for users' knowledge of Unix and these used the same partial model for the common regular expression concepts.

Partial models and granularity

An important role for the context structuring is to make the model more understandable. It ensures that we can limit our focus to a small number of closely related components. This meshes well with work on modelling at various levels of granularity (Greer and McCalla, 1989). For example, there are times when we want to consider the components which model the user's knowledge of every single sam command. At other times, we want to operate at a higher level. For example, we might simply want to consider the five general categories of sam knowledge: basics, more_useful, very_useful_mouse, powerful and mostly_useless.

To do this, we need tools that can evaluate a partial model. Since the partial model is a collection of components and sub-partial models, this means evaluating each of them. The components can be evaluated by a nominated resolver tool; and each partial model by recursive evaluation.

Once the components of a partial model have been evaluated, a mechanism is needed to determine the value of the whole partial model. One simple approach is to take a majority vote and deem a partial model to have the value held by the majority of its components. This is what was done for the sam models displayed by various viewer tools.

Consistent with the tools approach, we allow other tools for evaluating partial models. For example, if we needed to certify a user's knowledge in a safety-critical domain, a majority vote would be inappropriate. The evaluation algorithms would determine a partial model to be true only if all sub-partial models and components were true. Moreover, the resolver used would need to be suitably conservative about judging a component true.

Partial models as consistency partitions

Another important role for the partial model is as the basis for assessing consistency. For example, a particular um-consumer may require that three particular partial models be consistent. Since the basic representation treats each component as an atom, the um-consumer must include some knowledge source which is able to identify inconsistencies. As we discussed in the last chapter, this knowledge might be part of an inference tool, the resolver or the um-consumer. The importance of the structuring mechanisms is that they provide a mechanism for forming collections of components upon which a um-consumer can impose consistency requirements.

Context as basis for privacy control

Privacy control operates at the context level. By default the whole model is available only to the user and any programs they run.

We have implemented each context as a Unix directory. This means that we can use the operating system for management of access control. If the user wishes to make parts of the model more widely available, they can alter the access to the relevant directories and hence contexts. This means that changes to access control can be done either using standard utilities or one could implement a simple tool that provides an interface for the task.

6.2 Support for use of external user modelling information

The tools responsible for providing evidence from external information sources are highlighted in Figure 6.3. These must be able to contribute to the user model. These tools either observe or interact with the user to collect user modelling information. They then contribute this to the user model.

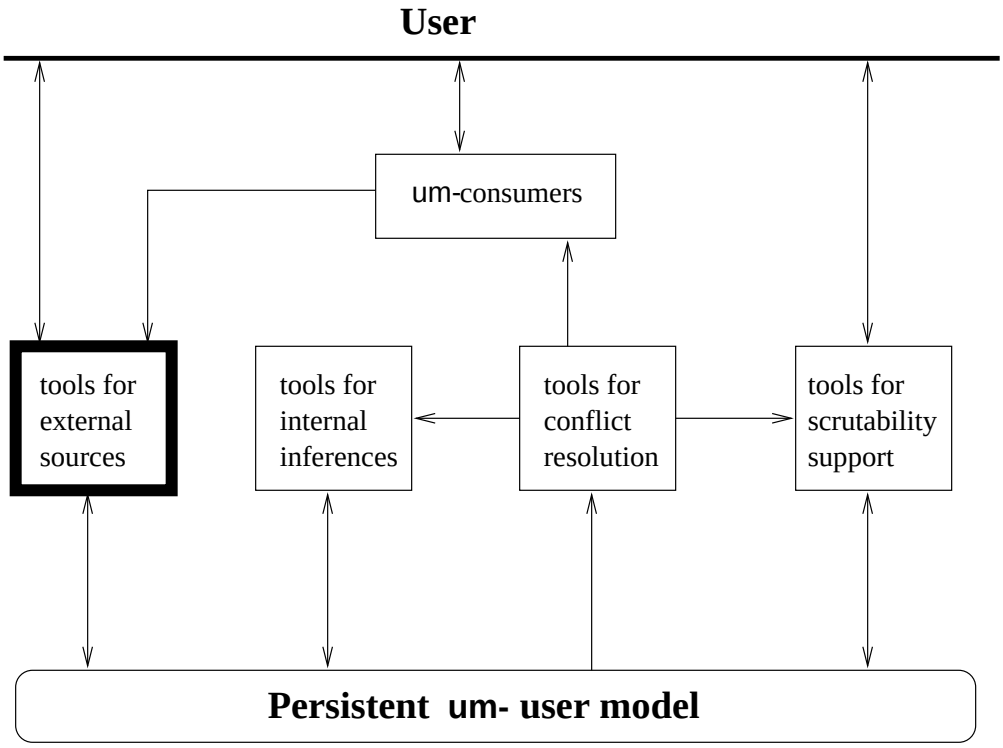


Figure 6.3. Tools for external sources within architecture

The quit component in Table 6.1 had evidence from an external source. The name of the program and part within it which contributed the evidence act as hooks for explanations which must be provided by an explanation subsystem.

6.2.1 External evidence classes

At this point we draw upon the analysis in Chapter 4 where we identified three distinct classes of evidence from user interaction: given, observation and told. We preserve the classification of these categories of external evidence because our analysis indicates they are the fundamentally different classes of information available about the user from interaction.

Evidence type given

The form of given evidence is

value given *ToolId SourceId (time) [extras]*

This class is for evidence from $user_{private} \rightarrow user_{shared}$ in user modelling interactions where the user *actively* gave information. For example, this is the class of evidence provided by MovieRater, an interface for rating movies. It might produce evidence like this

true given MovieRater self (Sun Feb 1 08:20:46 EST 1998) rate=4

where the user, themself had given a rating indicating liking the movie with the strength of that rating being 4.

For simplicity, the umps model dealt only with the modelled-user's interactions. Beyond this, um needs to accept other user's beliefs about the user modelled. Suppose, for example, we were modelling *userX*. Suppose too, that information about *userX* was volunteered by another user, *userY*. This is the action, $userY_{private} \rightarrow userY_{shared}$. This captures the beliefs of the user, *Y* about beliefs about the *userX*. We show its class as given since it is direct modelling information. Of the three evidence classes derived from interaction, given is the only one where an external source tool makes use of evidence from users other than the subject of the user model. (observation and told interactions should only be important for modelling the user being observed or told.)

For example, suppose we want to model the user's Pascal programming skill. If one tool provides an interface for use by the student's lecturer, the assessment will be represented as the given evidence class. So given captures the broader class of external evidence that comes from explicit user modelling interaction. The persistent form of the evidence holds the identity of the person giving the evidence.

Taking this one step further, suppose that the lecturer of a first Pascal course lodges course results in a database. If a tool consults this database and then contributes evidence about the user, this is represented as given. It might look like this:

false given StdScan C1001LectJsmith (Sun Feb 1 08:20:46 EST 1998)

where the tool that scans the student database (StdScan) took the information about a particular course (C1001) from the lecturer (jsmith).

Evidence type observation

The form of observation evidence is

value observation *ToolId PartId (time) [extras]*

where the *ToolId* is the name of the programme that collected the observations of the user and *PartId* identifies the part of it which produced this evidence. When these two identifiers are provided to an explanation subsystem, it should produce an explanation for the particular mechanism within *ToolId* which gave this evidence. For example, a monitor-based tool might collect observations about all the user's sam activity. In our work with such data collection, the part which analysed xerox usage had a special set of rules it applied to assess whether each use of the command seemed likely to be correct. The specialised explanation for this process was available in the explanations for the xerox part of the monitor analysis tool.

This class of evidence corresponds to $user_{shared} \rightarrow machine_{private}$, where the user's main purpose was to use an application system. User modelling information comes as a side-effect. In our modelling of sam users, this class of evidence comes from analysis of the sam-monitor logs, as for example, the evidence about the sam quit-command in Table 6.1.

Evidence type told

The form of told evidence is

value told *ToolId PartId (time) [extras]*

and the form is almost identical to that for the other forms of external evidence.

The transition $machine_{private} \rightarrow machine_{shared}$, is designated as told because, from the machine's point of view, it has told the user something. The term told derives from the comment often made by teachers 'I told them ...' which usually indicates that the teacher expects the students should know what they were told about. The evidence has the name of the program that did the telling and the identifier for the particular part of it involved. For example, the samcoach is one source of told evidence. When it sends a message to the user about a particular aspect of sam, it adds a piece of evidence to relevant components in the user model.

true told samcoach quit_k (Sun Feb 1 08:20:46 EST 1998)

The evidence shows the coach as the *ToolId* and quit_k is the *PartId*. When a single coaching action explains aspects modelled by several components, it adds a piece of evidence to each of these.

6.2.2 Scrutability support

Explanations for external sources of evidence are essentially at an ontological level, giving a description of the source and the relevant part of it. Table 6.2a shows each class of external evidence and an example explanation. Default explanations are available at the root context. These are strings like those in the table. Each context may redefine the explanations that apply in that context. This can also take the form of simple strings or it may be a more sophisticated explanation subsystem which generates an explanation for the argument presented to it.

Table 6.2.a. Generic explanations for evidence by class

Class of evidence	Example of an explanation
given	This information was explicitly given by someone
observation	You were observed to do something which shows this
told	You were told what this component means

Table 6.2.b has examples of explanations for each of the classes. The explanation have two parts: a description of the tool and the qualifying part. For evidence class given, ToolId is the tool which interacted with the user and then placed the evidence in the model. The second argument here is actually the identifier for the person who gave the information. The other classes, observation and told, essentially use both *ToolId* and *PartId* as two arguments to the explanation subsystem.

The told class only applies to knowledge and beliefs. If the source which told is 'reliable' and 'honest', it will only tell the user things that um and the um-consumer programmer consider to be correct. So we would normally expect told evidence to have the value true for knowledge components and the value false for belief components.

We also use the evidence class told for information given by any agent. For example, in the sam work, the students were studying a course that ran in parallel with the coaching work and the lecturer for that course told the class some aspects of sam. It could be represented with evidence like this:

true told PPlect8 xerox_k (Sun Feb 1 08:20:46 EST 1998)

indicating that the eighth lecture in a course (PPlect8) was used to inform students about the command. The explanation for PPlect8 is 'Jan Lennon gave information about this command in the week 8 PP class lecture', where the lecturer was 'Jan Lennon' and 'PP' is the name for the course.

These explanations are kept either at the root context or as part of the local context definition. This means that some tools can have generic explanations or specialised ones. For example mcons is a general tool that analyses log files to identify and count patterns.

Table 6.2.b. Example explanations for external evidence

Class	ToolId, Source/PartId	Example of an explanation
given	ToolId: MovieRater	This information came from the MovieRater interface
	SourceId: self	You gave this information yourself
given	ToolId: StdScan	This information came from the student data base
	SourceId: C1001LectJsmith	The person providing this information was Jan Smith (id: jsmith)
observation	ToolId: mcons	This information came from mcons a program that analyses logs of sam usage to assess whether you know each command or not.
	PartId: quit	This checks for at least 20 uses of quit in the last three months of sam use
told	ToolId: samcoach	This information came from samcoach which sends advice about useful things to learn abut sam
	PartId: quit_k	This explained why you should learn and use q in the command window to quit

A general explanation could be kept at the root context. The part description would then be specialised for the current context. Note that although the the examples in Table 6.2 are very simple, an explanation subsystem could be arbitrarily sophisticated. For example, it might use multi-modal information such as movie clips, displays of simulations of the actions of the tools or generated natural language explanations.

6.2.3 Examples of tools for external information

The primitive tool for adding information to the persistent form of the model is *model_tell* described within the requirements in Chapter 3. Essentially, this adds evidence to the model. This means the source must be properly defined within this context. This in turn means that there must be explanations available for it.

Above this primitive level, there are three classes of tools, each corresponding to one of the evidence classes. These can each take many different forms.

For given, one class of tools is designed to collect information about the user directly from that user. These interfaces ask or elicit information. For example, an interface might ask the user to rate movies. A more sophisticated interface might use other techniques, for example concept mapping, to elicit the user's understanding of a domain (Kay, 1990).

Where the given source is anyone other than the user, the tools may be interfaces that enable that source-person to volunteer information. Alternately, as in our example above, the tools could seek information from conventional databases.

Tools that can provide observations of the user include a broad range of possible sensors

such as those used by Doppelganger (Orwant, 1995). These might include relatively exotic active badge sensor data indicating the user's location or eye tracking sensors. At a more mundane level, it could be data from tools like the sam monitor. In student modelling, observations of the student at work on tasks in the domain fit this category. In comparison with given information, this class of information is characterised by its high volume, easy availability and low quality. One of the challenges for user modelling is to exploit such information so that systems can model the user unobtrusively and without making extra demands on the user.

External information tools have two main elements: sensor data and an interpretation phase. The latter is sufficiently challenging that there may be many possible tools applied to any one set of sensor data. For example, there may be several tools analysing the sam logs, which amounts to several observers of the user's sam activity. Each might embody different ways of seeing the user, collecting different perspectives of their actions and contributing different information to the user model.

Finally, told external evidence has the potential to capture information about the many aspects the user has been told about.

6.3 Support for internal inference

The internal inference tools are highlighted in Figure 6.4. The evidence they contribute to the persistent form of the model has a similar form to that of the external information sources just described. The important difference is that the accretion representation handles internal inferences quite differently. Evidence from external sources constitutes the ground assumptions which the modelling systems must accept. Except for cases where the number of external inferences poses an unacceptable cost, they remain in the model.

By contrast, the internal inference tools:

- contain the system's knowledge about the domain and about users in that domain;
- combine this knowledge with the current state of the user model to infer new evidence;
- and add this evidence to the model, to define a revised model state.

The significant point is that in the accretion representation, the inferred evidence is short term. It will be normal to undo the effects of internal inference tools by removing their evidence.

On the basis of the existing work in user modelling shells, we distinguish two classes of

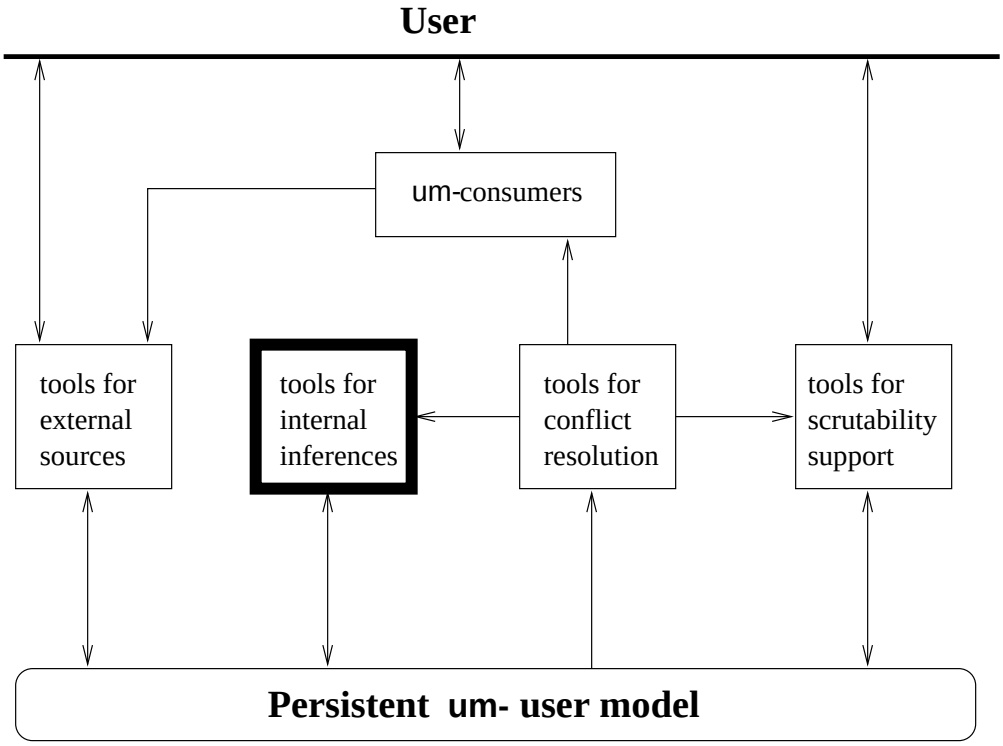


Figure 6.4. Tools for internal inference within architecture

inferred evidence: stereotypes and rules†, our term for other knowledge based inference. Since no single definition of stereotype emerges from existing work, we need to establish the distinction we make between stereotypes and other rules. We do this on the basis of our concern for scrutability. An intuitive reading of the terms stereotype and rule has connotations that characterise each from the point of view of the user.

At that intuitive level, there is an important difference between stereotypes and rules. the term rule captures the strong sense in which the *meaning* of novice is a person who knows some of the simpler things in a domain but not the more advanced aspects. This reasoning process should be very reliable.

By contrast, an intuitive understanding of stereotype is a reasoning process that is gives default assumptions, based on statistically-based inference. So, for example, we would expect that it is statistically true that novices typically do not know any of the advanced aspects and do know several of the basics. In fact, the sam work grew from monitoring large numbers of users over several year. So we might build stereotypes whose accuracy

---

† We use Helvetica font when referring to um stereotypes and rules and Roman font for the general terms, stereotype and rule.

can be quantified like the following:

if novice
then
not undo (.73)

which would reflect that analysis over thousands of novice users. This reasoning process is reliable in a statistical sense and such inferences are useful predictors for a population. However, some users will not fit the pattern. For example, a particular novice may know undo; this person is one of the 27% of novices who know undo.

In practice, stereotypes seem likely to follow the emerging practice in existing user modelling work where they serve as default assumptions to be overridden as soon as more reliable information becomes available.

Of course, the reliability of a conclusion also depends upon the reliability of the base information: in the case of our example of a rule, these were the components modelling the user's knowledge and for the stereotype it is the reliability of the value for the novice component.

Evidence class stereotype

Stereotype evidence in the persistent user model has the form:

value stereotype *ToolId PartId (time) [extras]*

For example, evidence created by a stereotype based on the user's educational level and an inference of their preference for the science fiction movies might look like this:

true stereotype PersonalData EducScifi (Sun Feb 1 21:16:11 EST 1998)

where the PersonalData tool has an inference from *edu*cation to a preference for *sci*ence *fi*ction.

Evidence class rule

The form of rule evidence in the persistent user model is:

value rule *ToolId PartId (time) [extras]*

For example, a rule inference tool which reasons from ratings of one movie to movie attributes might add the following piece of evidence to the component for the user's preference for science fiction.

true rule movie2attr movie_171 (Sun Feb 1 21:16:11 EST 1998) value=9

where the rule inference tool called movie2attr reasoned from the user's liking a particular science fiction movie, movie_171, to produce a piece of evidence indicating the user likes science fiction.

6.3.1 Internal inference tools

We now describe the collection of different categories of internal inference tools.

Accretion tools for internal inference

Both rule and stereotype can be expressed as simple productions, with the trigger expressed as a boolean function based upon the values of one set of the user's components and the result being a set of pieces of evidence, one each for a set of components. An internal reasoning component i could then contribute evidence p_j to J components, c_j, where each has the value v_j

$$f(model\_state_u) \rightarrow model\_tell(user_u, component_j, evidence_{i,p_j}, v_j),\ j = 1..J$$

We have implemented only the simple stereotype and rule tools in our work in the domain of a movies recommender. The um architecture permits arbitrary tools. So, for example, there could well be an inference tool that implements a sophisticated hierarchical structure of stereotypes. Equally there could be knowledge based inference tools which hold the type of domain knowledge described in several of the user modelling shells in Chapter 2. The critical constraint is that each piece of evidence they add must have an explanation.

Metamorphosis tools

These tools enable a collection of low reliability pieces of evidence to be tallied up and produce a new piece of higher reliability evidence. For example, in some experiments with the sam models, we simulated the effect of a weekly analysis of the monitor logs. After many weeks, this produced a sequence of observation evidence items. A metamorphosis tool could take such a list and add a new piece of higher reliability observation evidence.

Compaction tools

Where the model size is important, a compaction tool can remove useless evidence from external sources. Consider the example of the sam models above where a metamorphosis tool has created a piece of evidence. Since all resolvers in the sam-context treat this new

evidence as more reliable than the evidence it summarised, a compaction tool could remove this now-useless evidence. As we noted in Chapter 5, this process destroys information that was used to create the user model; scrutability seems better served if compaction is avoided unless the design constraints of the particular modelling task demand it.

Undo/redo tools

These tools operate on evidence that was created by internal tools: stereotype and rule evidence. The undo tool removes all such evidence in a given partial model or context. The redo initiates each of the internal inference tools for that partial model or domain. This process deals with the effects of changed information from external sources.

One important situation case to consider concerns stereotypes. Typically, activation of a stereotype will cause default inferences to be added to many components. This is similar to the production shown above. Each conclusion of a stereotype provides evidence for one component as in this example:

> true stereotype sam_stereos novice (Wed Dec 31 7:25:52 EST 1997)

where the same stereotype for novices produced this evidence. It is in the nature of a default inference that the value of the component will be altered if later, more reliable evidence becomes available.

In addition, it may be necessary to retract the whole stereotype. For example, a stereotype for novice may initially be triggered. Subsequently, it may be determined that the user knows many sophisticated aspects of sam. Then the inferences for the novice stereotype should be retracted. There are two main ways to treat this situation.

First, there can be a simple accretion approach, with the addition of an opposing piece of evidence like

> false stereotype sam_stereos novice (Wed Dec 31 13:25:52 EST 1997)

so that the original stereotype inference is balanced. The other approach is to undo the stereotype's evidence. This involves removing all inferences from the named stereotype. The merit of the former approach is that the user can scrutinise their user model as it was at times past.

6.3.2 Scrutability support

One important aspect of internal inference is that the processes involved are within the user modelling system. So we can expect more than ontological explanations. We can also expect detailed explanations of the *process* of the tool's reasoning.

We have implemented only simple inference structures. For example, in the movies domain we used inferences based on stereotype reasoning like this:

> educated -> not violence
> female -> drama
> female -> not violence

and rules like this:

> movie_171 -> drama
> movie_182 -> drama
> not movie_193 and not movie_132 and not movie_7777 -> not violence

Suppose we have an educated, female user who likes movie_171 and movie_182 and dislikes movie_193, movie_132 and movie_7777. Then all the above productions fire and each adds a piece of evidence to the model, either for the component modelling preference for violent movies or for dramas. The explanation for each piece of evidence is a single production. The user may also want to scrutinise the details of the components that caused the productions to fire (educated, female and the various movies).

For more complex inference structures, one needs techniques such as those applied in explaining knowledge structures in expert systems, for example, those reviewed in (Southwick, 1991).

6.4 Resolvers for managing consistency

The resolver tools, highlighted in Figure 6.5, must evaluate a collection of evidence about components. The inference tools and um-consumers may need the resolved value for a single component, a complete partial model or several partial models.

6.4.1 Resolver tools

In a case like quit_k in Table 6.1, it should be easy to decide that the component has the value true: there is just one piece of evidence, that the user was observed to use the quit command 324 times! In general, we have longer evidence lists, with conflicting evidence on the value of the component. This section describes some simple resolvers we have constructed for the sam and movies systems.

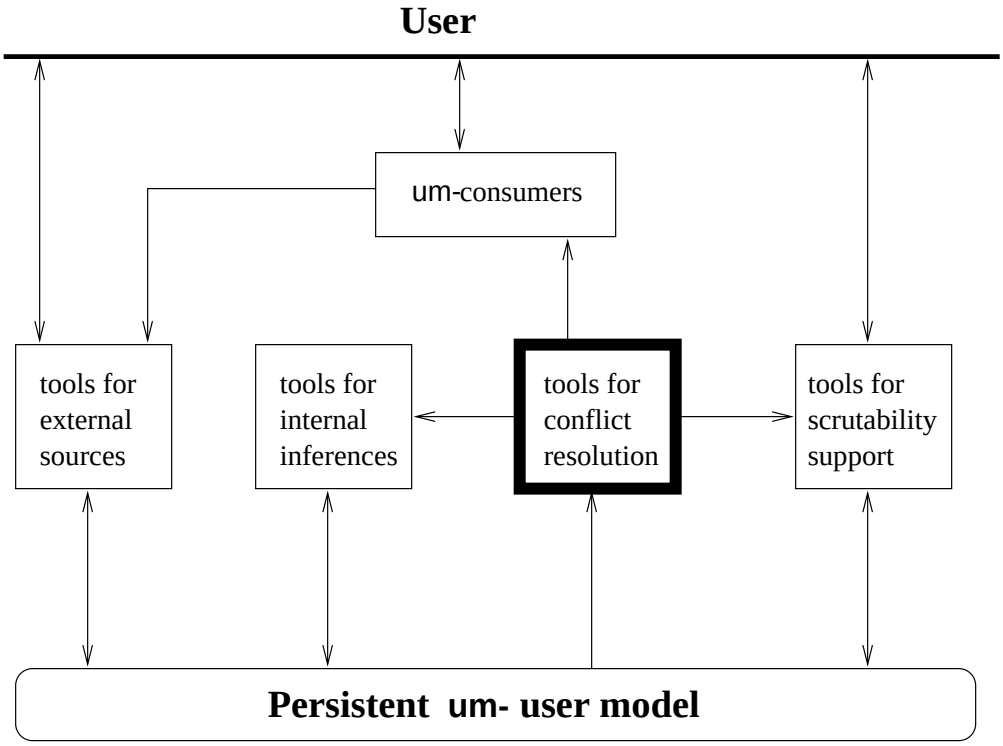


Figure 6.5. Tools for conflict resolution within architecture

Because the evidence contains the identifier for the person giving it, resolvers can treat some people's evidence as more reliable than others. A um-consumer might allow the user the option of deciding whether a resolver should ignore certain evidence sources.

Primitive resolvers

A primitive resolver needs to evaluate the reliability of each piece of evidence. For example, one primitive resolver we used for the user's knowledge in sam models depends upon the class of the evidence and the time it was added:

given > rule > observation > stereotype

If the most reliable pieces of evidence have the same reliability, we accept the most recent. If there are two or more of these with the same timestamp and different values, we conclude the value is conflicting. If we have no evidence at all, we resolve that component to be false.

Note that we have omitted told from the list above. This is because it was ignored. This

resolver did not deem the user to know an aspect of sam just because they have been told about it. Another resolver, used by a sam coaching program has a different partial ordering:

false, given-by-user > rule > observation > true, given-by-user > stereotype

where the user's given assessment of their knowledge is regarded as the most reliable indicator that the user does *not* know something but their own assessment of knowing something is treated as less reliable than observation-based evidence. A similar approach has been taken elsewhere, for example by Murray (1991).

The accretion representation and um architecture are designed to allow for a multitude of resolver tools. Each um-consumer can select the resolver that suits it. This might mean that one um-consumer will use a resolver that regards an external source s_1 as more reliable than another source s_2. A different resolver may have the opposite assessment.

6.4.2 Scrutability support

As with other explanations, we require at least an ontological explanation of the resolver. This can be a simple handcrafted text explaining what the resolver does, rather like our explanations above.

Like the other explanations, there is a generic explanation at the root of the central model. This could be used for the two primitive resolvers described above, where the mechanisms for deciding the value of an evidence list is domain independent.

There can also be more specialised explanations associated with the context. For example, suppose a resolver treated information from one tool, s_1, as more reliable than any other information. Suppose also that s_1 was only available in one domain which is modelled in one context. Then the resolver and its explanation would only reside in that context.

6.5 Control

Having described the types of tools that contribute external evidence, those that perform inference, and the resolvers, we now discuss the control mechanism which ensures a set of them will operate properly to perform a modelling task. This is a task for the um-consumer. It is responsible for invoking the particular tools at the times they are needed.

This makes um quite different from most of the systems in Chapter 2 since control might

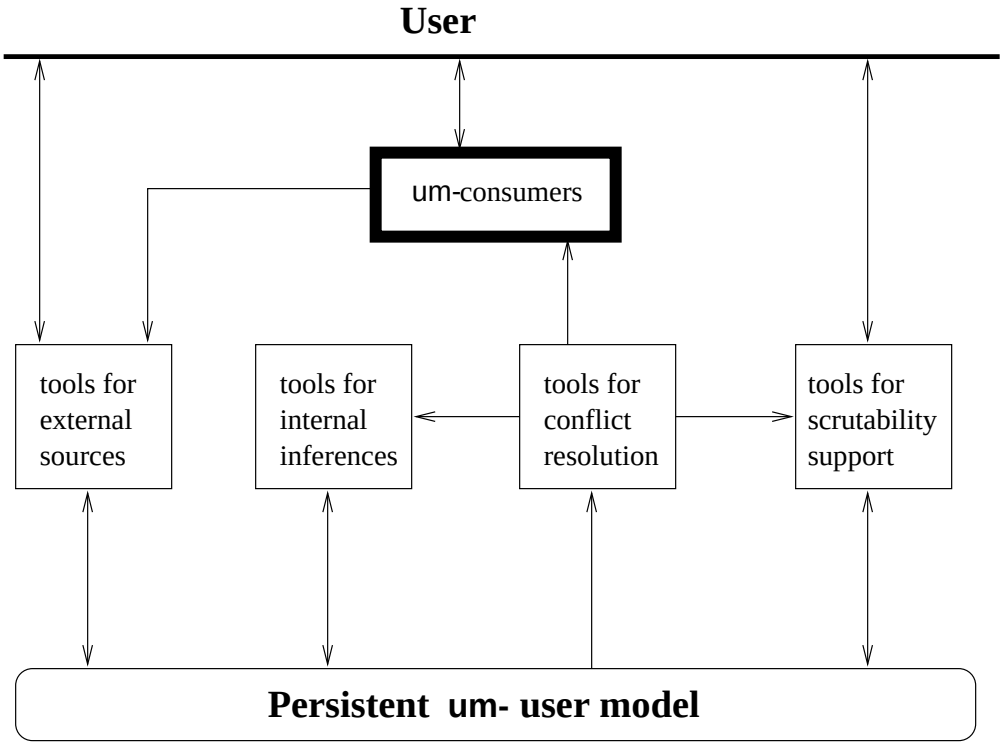


Figure 6.6. um-consumer for control within architecture

seem, at first glance, to be a responsibility of the user modelling system. Our approach means that the control mechanisms for different um-consumers can be entirely under their control.

For example, consider an extreme example of a um-consumer that operates on a large model that changes quite slowly. Let us also assume that the tools for updating the model are costly to run, consuming large amounts of machine resources. This um-consumer might update the model only infrequently, perhaps once a week.

As the other extreme, a fast changing model that needs to be up-to-date should operate more like the systems described in Chapter 2, where each new event recognised by the um-consumer should be followed by updating the model with any new external evidence and running the internal inference tools. Note that if such a um-consumer has to run quickly so that it can interact satisfactorily with the user, that um-consumer will need to use a carefully chosen set of tools. These would either need to run quickly or have mechanisms for curtailing their operation if it is taking too long.

At the same time, we note that two or more um-consumers might use the same partial models and operate over the same period of time. For example, one um-consumer might generate coaching advice about sam and another could generate customised

documentation about sam. These may have quite different demands, just as we have described in the two extreme cases above.

The um architecture is intended to allow such flexibility. Only the um-consumer can have the knowledge of its needs and hence the basis for making the tradeoff decisions needed to control the user modelling tools.

6.6 Summary

The fundamental element of the representation of um models is the *component* with its list of *evidence*. Components are organised into contexts and partial models.

The underlying representation for components is the accretion representation described in Chapter 5. In addition, there are slots for extra information about each aspect of a component and about a piece of evidence. These are needed for numeric reasoning or the various other approaches for dealing with uncertainty, change and inconsistency.

Basic um modelling tools:

- add evidence from direct evidence sources, whose class can be given, observation or told;
- add evidence from inference sources, whose class can be stereotype or rule;
- resolve the list of evidence about boolean components.

There is late binding of an interpretation for evidence lists. This means that different tools applied to the same model will enable different interpretations and these are computed at time-of-use.

We now review the requirements on the modelling process, summarising the ways that um deal with each:

- *model_ontology*: provided by the central definitions for components;
- *model_externals*: provided by the external evidence sources;
- *model_internals*: provided by the tools that perform internal inference;
- *model_resolve*: provided by the resolvers;
- *model_component_types*: provided by the builtin types: *knowledge*, *non-mutual beliefs*, *preference* and other *attributes*;
- *model_simplicity*: provided by the minimalist representation which can be just the component name or that name with an arbitrary string;

- *model_extensibility*: has been a concern of the design throughout and is demonstrated by applying um to tasks which represent diverse user modelling needs - we address this issue in Chapter 8 which describes the two domains in which um has been extensively used;
- *model_privacy*: provided by the context structure and its implementation method which provides access control;
- *model_structure*: provided primarily by the context structure which essentially controls the scope of component names, tools for external sources, internal inference and resolvers as well as all their associated definitions and explanation - additional structure within contexts is provided by the partial models;
- *model_scalability*: has been a concern throughout the design but needs to be demonstrated by deployment of the um and evaluation of the scalability.

At this point, we can see that our concern for scrutability has taken um in a different direction from most of the modelling shells described in Chapter 2. The primary reason for this follows from the role of the long term persistent user model as the basic part of the model and the accretion representation's evidence lists for the reasoning about the value of a component. This design was driven by the goal of scrutability. It means the the system's conclusions about the user derive from only two sources:

- external evidence about the user;
- internal knowledge of the system which enables it to interpret that evidence, make additional inferences from it, identify conflicting information about the user and finally, to manage conflicting information about the user.

Such evidence is the basis for the individual user's model. Each piece of evidence must have an explanation which will support scrutability of the model.

This approach has meant that um's architecture differs from most of the systems described in Chapter 2. We show the comparison in Figure 6.7. Figure 6.7.a characterises most of those systems in Chapter 2. To its right, Figure 6.7.b shows the um architecture, omitting um's scrutability tools since we are focusing on the other, core aspects of user modelling support. Also, we have labelled this diagram to show the different classes of System Beliefs, **SB**, at different points in the architecture.

The heavy horizontal lines mark the boundaries between the outside world and the runtime user modelling system and application. Since the user is outside the system, the top lines indicates the interface between the user and the system.

The um architecture shows the persistent user model and explanations outside the run-time user modelling system and application. A distinguishing feature of um is the

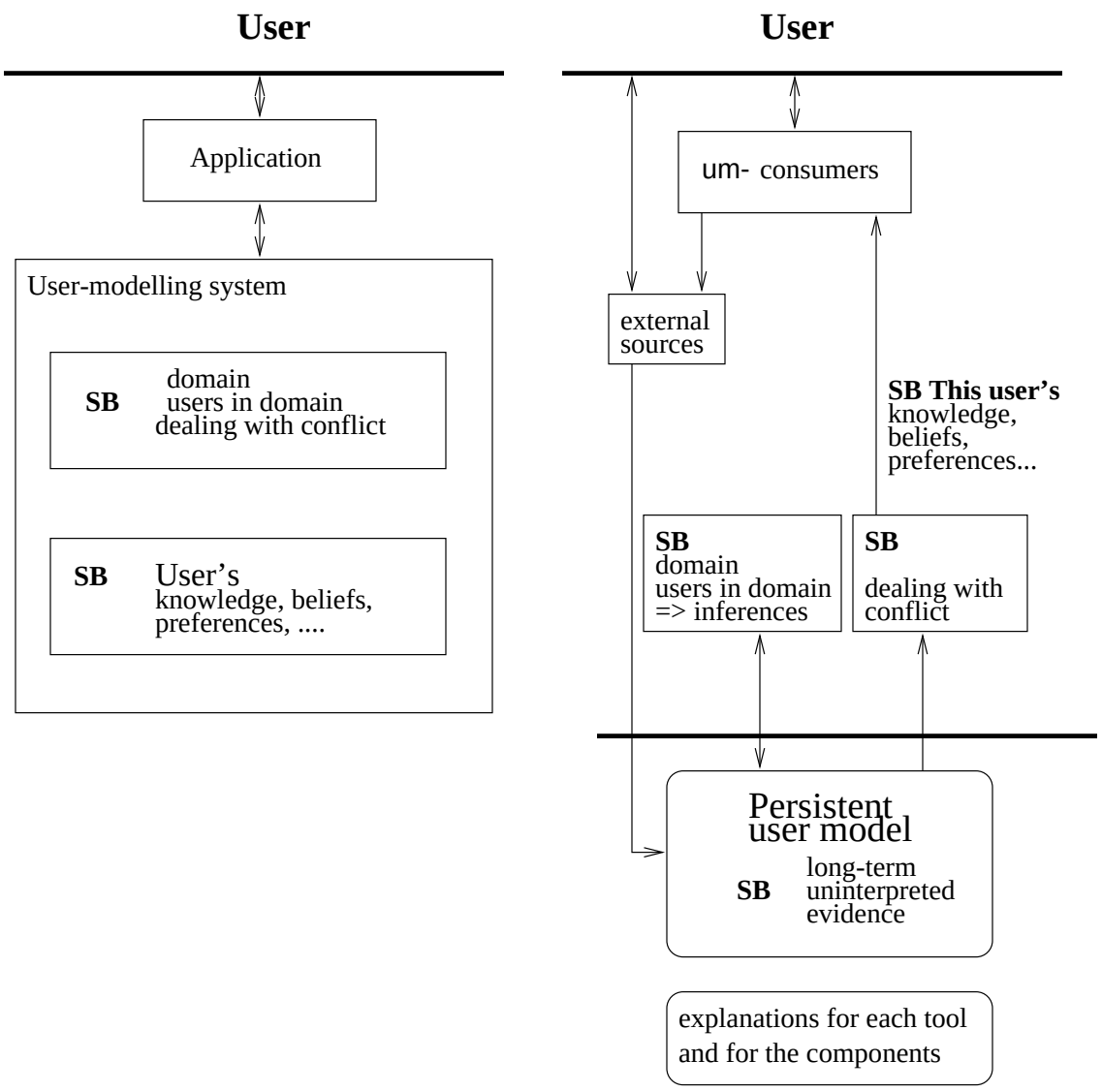


Figure 6.7.a. Model-embedded architecture

Figure 6.7.b. um architecture

Figure 6.7. Architecture comparison for user modelling shell systems

required explanations for both the tools that produce evidence and for the meanings of the model components. This difference is unsurprisingly since we have a focus on scrutability and the other user modelling workers had other focuses.

The persistent user model constitutes a less obvious difference in the architectures. The model is not bound to any um-consumer or collection of tools. It belongs to the user and

is available to any um-consumer they want to use.

This serves our goal of reuse of the user modelling information. There may be several um-consumers which use the same parts of the individual user's model. Each might interpret it differently and each can operate independently. So, for example, one um-consumer might be working to recommend books about music and another might be selecting pieces of music for the user to listen to. Both might use some of the same components. Each makes independent interpretations of the evidence for those components. Each might be composed of quite different tools. Importantly, the simple form of this model makes it simple for arbitrary programs to make use of it. Further, the user can examine it directly.

Contrast the external persistent um user model with the architectures depicted at the left. The block corresponding to the persistent model is the **SB** user's knowledge, beliefs, preferences ... This is the system's beliefs about this user. It is embedded within the user modelling system: it is a data structure within the system. Undoubtably those shells can store this data structure and reuse it between sessions. However, this mode of operation appears not to have explored.

Figure 6.7 identifies differences in the internal architectures. Essentially, these relate to the tools approach. Consider how this affects the management of the various aspects of the system's beliefs:

- **SB** domain, for example knowledge that some sam commands are simpler than others, that some are available in the command window and others are mouse based.
- **SB** users in the domain, for example that sam beginners usually know the simplest commands and none of the harder ones and most stereotypes.
- **SB** dealing with conflict to resolve conflict, for example knowing some evidence sources are more reliable than others.

In the architectures of Figure 6.7.a, all such knowledge is in parts of the user modelling system. In a system like BGP-MS, it is possible to disable some of the more complex parts of this structure where the application calls for it but essentially all of this knowledge comes with the user model. More importantly, the same modelling support structures are part of the shell in its deployment for different tasks. By contrast, um's approach is that the um-consumer assembles just the tools it needs for the task at hand. So, for example, a particular um-consumer may use three resolver tools where another um-consumer may use one of these and one other.

This leads to an interesting observation on the relationship between these two approaches.

The architectures in Figure 6.7.a could be incorporated within that at the right as a tool within the architecture. So, for example, BGP-MS might be a um internal inference tool. Given one collection of resolved component values, it could apply its internal knowledge of the domain and users in the domain to infer more about the user and then to return the values it determined for a collection of components. From these, um would produce pieces of evidence to add to the relevant components in the persistent model. Equally, shells like TAGUS, UMT, THEMIS and SMMS have developed sophisticated support for truth maintenance and so could be used as tools for that part of a um system. Of course, any such use of one of these shells will only be allowed if it could provide explanations for the evidence produced. Since these are quite complex system, it is not clear how or indeed, if, this would work.

The outlier from Chapter 2 is DOPPELGANGER (Orwant, 1995) which is most similar to um. It is evidence based and it takes a toolkit approach somewhat similar to um. However, its two major focuses were:

- tools for collecting external information from a diverse collection of sensors;
- machine learning to build community models over many users' persistent user models, with the community models at any time being based upon the current collection of individual models.

If we were to draw its architecture, it would be similar to um's on the right of Figure 6.7 in that the persistent user model is kept independently of the the user modelling system. However, its construction of community models relies upon a collection of individual user's models. So, in a sense, the individual models are part of the system as a whole even if it is possible for some individuals to store their models on private storage devices. Also, it has a heterogenous representation for the persistent user model. This means it needs a collection of scrutability support tools. For example, it had such tools for community models in the domain of news preferences, for Markov models displayed as graphs and a histogram graph display for predictions of login behaviour. Other aspects would need other tools or, as Orwant suggests, the user could examine the Lisp-like code.

It is not surprising that the different goals of the various user modelling shells have led to different outcomes. The scrutability concerns of um have led to continuing concern for the long term modelling issues. There has also been a striving for simplicity and for the definition of the three sets of tools described in this chapter. It is important that the various parts of the um architecture are loosely coupled so that the user can scrutinise the model focusing on one aspect at a time, be that a part of the persistent user model itself or the processes that formed it.

Chapter 7

um : the user's view

The design of the persistent form of the user model enables the user to see the components in their model and the evidence about each. Each part of the user model is simple. But it is a challenging task to really understand the model for two main reasons.

- It is hard to visualise a large and complex model, which will have many components, organised into many different contexts each with many partial models. The *inherent complexity* of what is modelled creates the need for additional support for scrutability.
- Our design places the user model's explanations centrally rather than with each individual user's model. This means that the user needs to look in one place for their user model and in another for the explanations.

Put simply, user models can be large and complex and explanations add even more information for the user to absorb. So an essential element of an effectively scrutable

user model includes tools that aid the user in navigating around the user model, its explanations and justifications. This chapter describes two classes of user model viewing tools.

- *Navigation* tools assist users in visualising and navigating around complex user models.
- *Explanation sub-systems* assist the user's detailed exploration and scrutiny of components in their model, explanations for component meanings as well as the processes involved in collecting evidence and resolving the value of a list of evidence. The explanation sub-system also supports the user in changing the value of the components in their model.

To make this chapter easier to understand, we will illustrate the um approach to scrutability support in terms of examples of actual tools.

7.1 Navigation and overview tools

The role of this class of tool is to enable the user to see a user model as whole, to identify parts that are likely to be interesting and to explore those. We identify the following requirements for such a tool.

1. *Overview*: The user is able to see an overview of large parts of the model, with the display reflecting the structure of the model.
2. *Relevance*: The interface tool assists the user in identifying and examining *relevant* parts of the model.
3. *Navigate*: The user can navigate around the model displaying more detail in some parts and less in others.
4. *Component types differentiated*: The display should distinguish the different component types as well as differentiating components from structuring mechanisms of contexts and partial models.
5. *Component values*: The display will show component values.

For the first of these, the context and partial model structure make it natural to display the model as a tree. The um form of a user model is a directed acyclic graph. However, it is easy to map this to a tree by repeating parts of the model. This simplifies the display of a pleasantly laid out model without crossing lines and since we expect that most models will, in practice, be trees, the limitation should not be a serious problem for accuracy of presentation.

As soon as the model is too big to fit easily on a typical display, the interface should support the user in identifying relevant parts of the model, the second requirement. This calls for a mechanism for displaying a subset of the model by collapsing some contexts or partial models to a single node-display while presenting others in full. Of course, the third requirement means that the user should be able to override this default, expanding and contracting contexts or partial models and so to navigate around the whole user model.

We discuss the other requirements in relation to an instance of such an overview interface, qv. Figure 7.1 is gives an example of the qv display for the model of a user's knowledge of the sam text editor. This model is organised in terms of the sophistication and utility of the components. So the display places the most basic aspects of sam near the top. Moving down the screen, the aspects modelled become more sophisticated and esoteric.

The qv tool can also be used to view selected parts of a user model by invoking it with a specification of the context that should be the root of the tree to be viewed. This is a simple but important mechanism for our second requirement. It means that a um-consumer launching qv can focus attention on the 'relevant' parts of the model. The root of the displayed tree defines the 'relevant' context. In the case of Figure 7.1, this is the whole model for this user (starting at the 'root'). We could have displayed just the part of the model starting, for example, at sam.

In addition to this rather limited notion of 'relevance', we need a mechanism for selecting the actual nodes to display. For example, the display in Figure 7.1 was that presented initially to the user starting qv. Had we displayed the full tree, it would appear as in Figure 7.2. It might be argued that the user can study this to find the 'relevant' parts. We can see from Figure 7.2 how this can pose problems: the tree is large, confusing and potentially intimidating. If the starting node given to qv has a very large subtree with many components, it may be infeasible to display all of them as the screen would be too cluttered or the print too small to be readable. the alternative followed by qv is to group components within a partial context so that it is easier to see the structure of the model. The interface then collapses contexts or partial models that are least likely to be 'relevant'.

Using qv in the context of work on the sam, we have developed a set of heuristics for collapsing parts of the model that are less likely to be interesting (Cook and Kay, 1994). It is based on the assumption that a context or partial model is good candidate for collapsing if all its components have the same value. Where all the components are true, this constitutes showing the user that the more general notion is true. In the sam work,

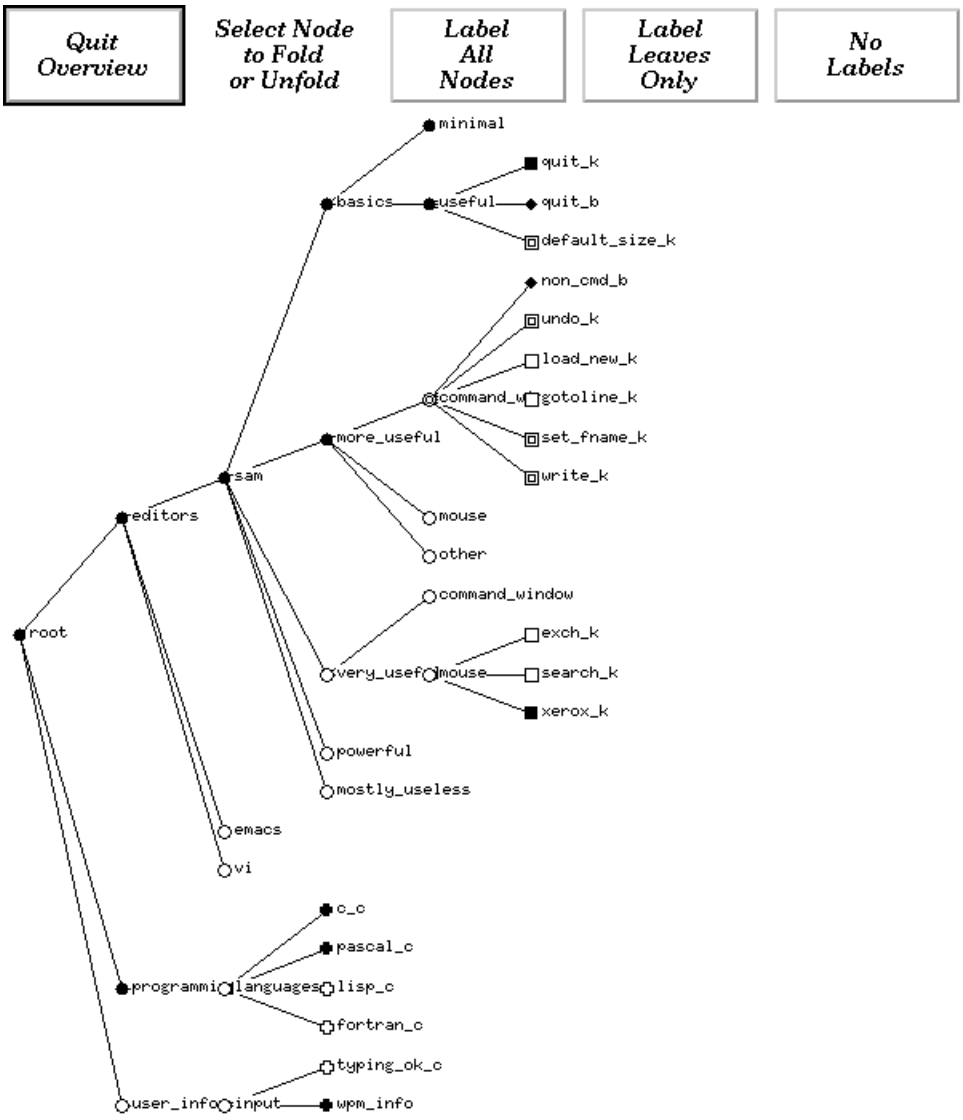


Figure 7.1. One example of a user's qv display

we were even more concerned about the opposite situation where the user knew none of the components in various partial models. This is the case when a beginner has only made small use of sam and knows little about it. We were concerned that a screen showing large numbers of unknown nodes would be discouraging. Moreover, a user who knows only the simpler aspects of sam is unlikely to be interested in sophisticated features. In the case of sam, this principle is then used to drive display heuristics.

Initially, all partial models with any true components are tentatively marked as visible. If the number of visible components is too small, further partial models are marked as

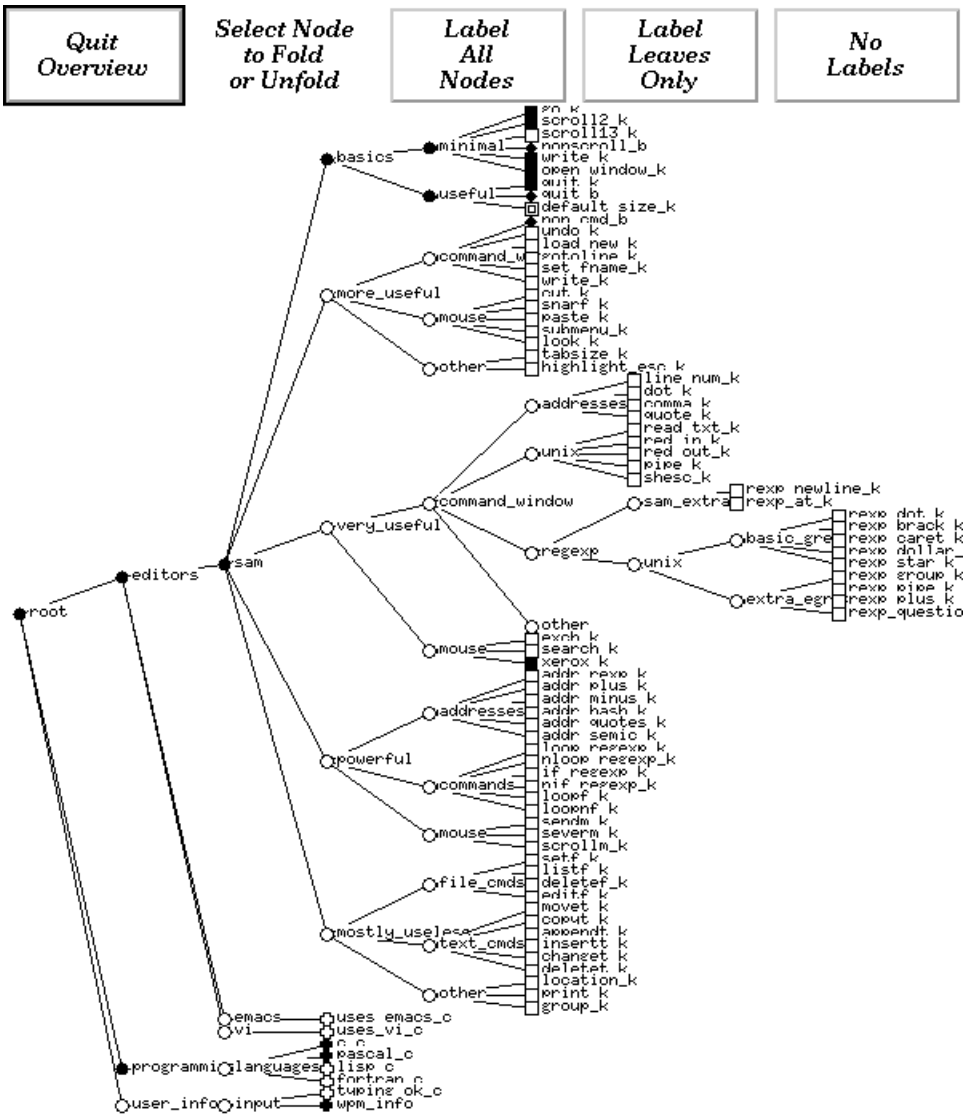


Figure 7.2. Fully expanded qv model

visible until a predefined "quota" is filled, favouring the partial models for the simplest aspects. If the number is too large, wholly true contexts or partial models are collapsed.

Of course, these heuristics are for the *default* display at start up. The third navigation requirement means the user must be able to expand and reduce the detail of the parts of the model. In qv, clicking on a node whose sub-nodes are not displayed causes the next level down the tree to be displayed. Conversely, the user can collapse displayed sub-nodes by clicking on a parent node. For example, had the user clicked on the node representing the partial model for command_window within more_useful in Figure 7.1,

the display would appear as in Figure 7.3, with the components in that node contracted.

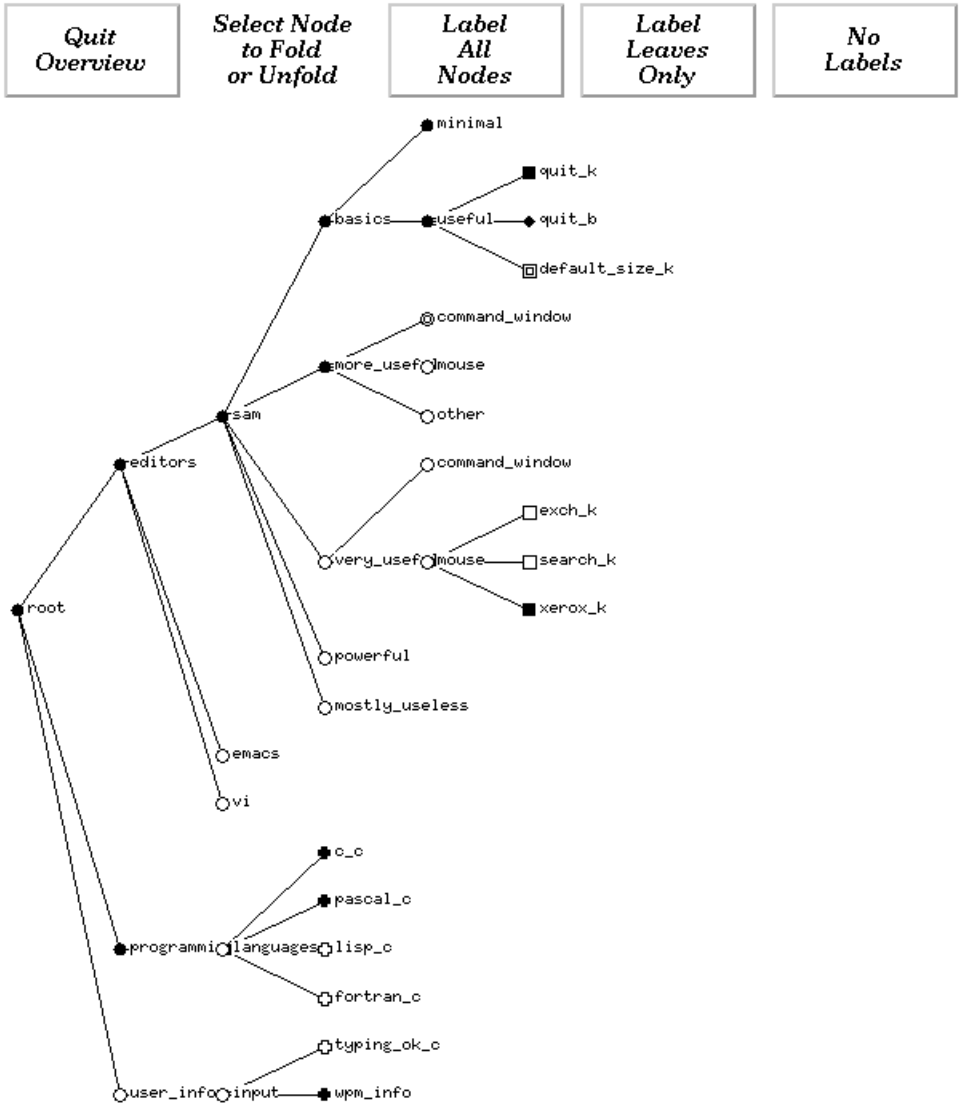


Figure 7.3. Contracting a node of the qv display

The next requirement is achieved here by using a circle for contexts and partial models. Clicking on a circle that is a terminal node in the display expands it, showing its sub-models or constituent components; conversely, clicking on an expanded node collapses it. These navigation operations do not distinguish between contexts and partial models: the differences are essentially internal to the user modelling. Until there is a compelling reason to highlight this distinction, there seems no reason to create additional complexity.

Different shapes are also used to mark the type of the component to meet this fourth

requirement. In qv, we have used the following codings: squares for *knowledge* components; diamonds for *beliefs*; crosses for all other attributes; The merit of distinguishing component types is that it helps identify the parts of the model. For example, most of the upper part of Figure 7.1 models knowledge components, as indicated by the many squares marking components. The belief-components are noticeable because of their diamond shape. By contrast, the bottom of Figure 7.1 has several characteristics-components such as the representation of the user's typing speed (wpm_info).

Our sixth and final requirement is that the display distinguish the values of the components. Our research has focused on boolean components and it is natural to indicate boolean values with a simple coding. In the case of qv, for all but belief-components, true is indicated by black, false by white. The opposite convention applies for beliefs. This convention ensures that a user who is modelled as having perfect knowledge will have all knowledge components displayed in black. The model for a user with completely incorrect knowledge will have all components displayed as white, indicating that they do not know any of the knowledge components and they do hold all the (incorrect) beliefs modelled. Nested shapes are used to indicate the resolver could not determine the truth of the component.

7.2 Explanation system tools

The navigation and overview scrutability support tools enable the user to see the structure of the user model and the values of the components and partial models. Scrutability support must go beyond this so that the user can delve into the details of the model. This support for scrutiny of the details is the task of explanation subsystem tools.

These support the user's understanding of the model ontology and the processes that contributed to each component. The user must be able to focus on each part of the model and explore the evidence and associated explanations.

An example of a tool for such detailed scrutiny of the user model is Xum, an X windows viewer for um models. A startup screen displays the root and immediate child nodes as in Figure 7.4. The user can select nodes that are leaves in the visible tree, expanding that part of the tree until the leaf components are available as is the case in the example of Figure 7.5. (As with qv, the converse navigation applies and leaves can be collapsed by clicking on their parent node.)
Also like the qv display, Xum displays the value of each node. Clicking on a leaf nodes pops up the menu shown in Figure 7.6. It enables the user to select from three main actions:

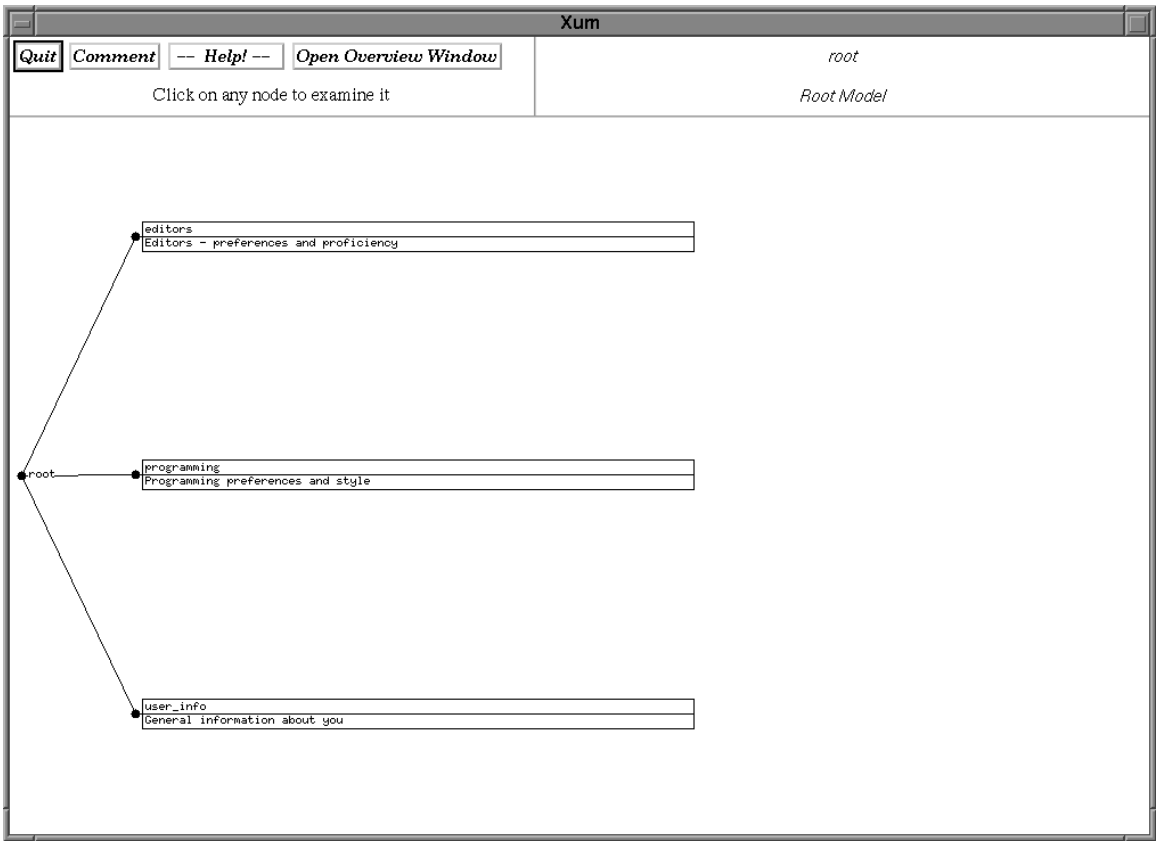


Figure 7.4. Start up screen for Xum.

- *justify* the value of the component;
- *alter* the truth value of the component;
- *explain* the meaning and purpose of this part of the model.

We now discuss each of these in the following subsections.

7.2.1 Scrutiny of the value of components

The um representation is based on evidence in order to support scrutability. So the natural form for explanation for the value of a component should be expressed in terms of its evidence list. The explanation subsystem presents such evidence in a format similar to that of the persistent form of the um model. This ensures that the scrutiny support tool serves as an aid to understanding the actual representation.

For example, selecting ‘justify’ in Xum, gives a display of evidence like Figure 7.7. (This display has been contrived to illustrate a range of the forms available.) This is a direct

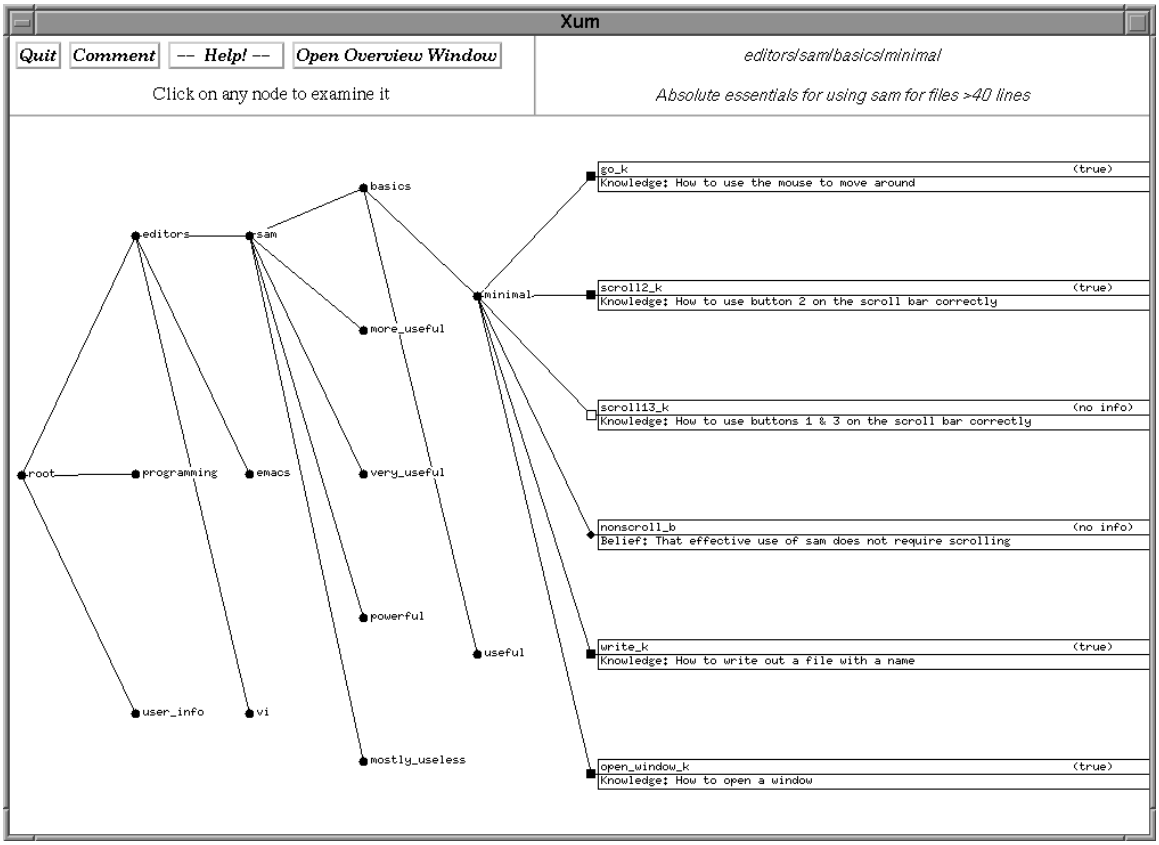


Figure 7.5. Xum display showing leaf nodes.

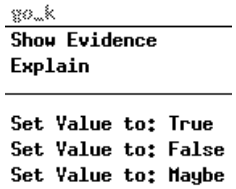


Figure 7.6. Menu available at Xum leaves for components

mapping from the contents of this part of the user model. Each line shows one piece of evidence about this aspect of the user's knowledge. The first column shows whether the evidence supports the truth of this component (Supporting) or not. The second column shows the type of evidence source. The last column gives the time the evidence was added to the model and the intermediate columns provide information that differs for different forms of advice.

For example, the first item of evidence is of type given meaning that the user themself

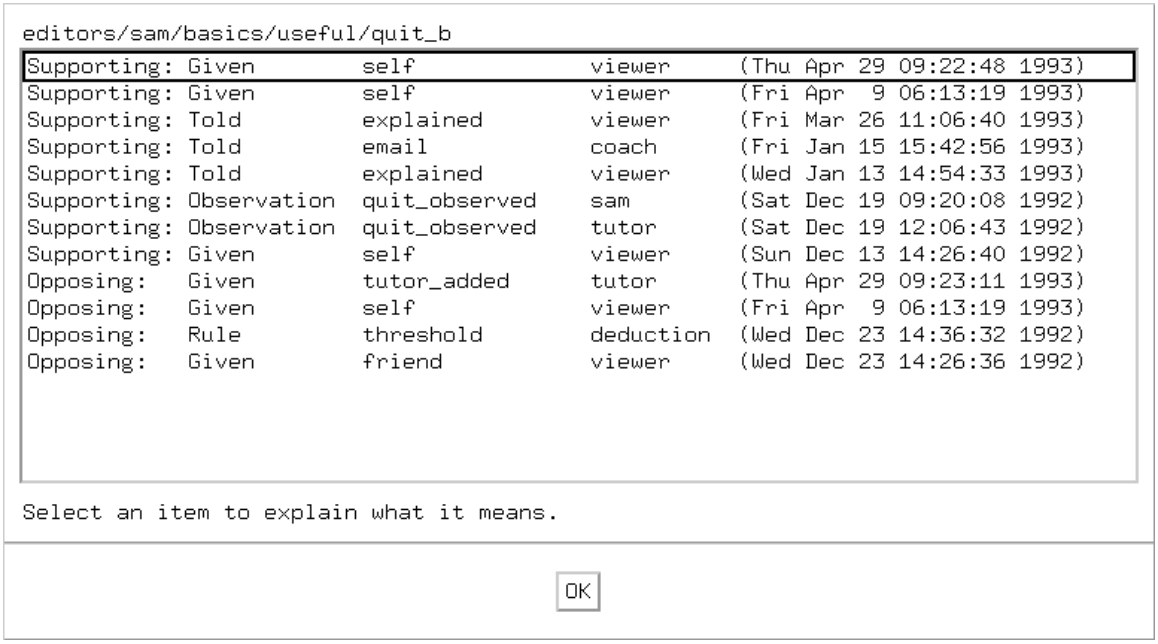


Figure 7.7. Sample display for the evidence about a component

gave that evidence while using the program called viewer. The third to fifth items are all of type told, indicating that the user was told about this by the programs named in the third column. The observation type evidence was provided by a program that observes the user. The second last piece of evidence was produced by a rule tool called deduction using a threshold to draw the inference.

7.2.2 Explanation of the process of resolving components values

The resolver assesses the evidence list for a component to conclude its value. The um requirement *scrutiny_resolve* means that the explanation subsystem should explain that process applied by the resolver to determine the component value. An important aspect of a user model is its interpretation. We require an explanation for the operation of each resolver to be lodged with the explanation subsystem. For the simple resolvers we have used to date, this can be managed with a text explaining the partial ordering of reliability of evidence.

This leaves a problem inherent in our goals. A major motivation for building user modelling shells and toolkits is to support reuse at all levels. In particular, we would like a single user model to be able to serve several um-consumers. At the same time, we intend that different um-consumers might use different resolvers for the same evidence.

Therein lies our difficulty: the value of a component depends upon the resolver used and so, no single display of the user model tells the whole truth about the component values.

If the accessibility of the user model is to be most effective, we need to link the Xum program to the um-consumers. So a tool like Xum must be invoked by the relevant um-consumer, with its resolver nominated. Then the values displayed and the explanations of the resolver will match the actions of that um-consumer.

7.2.3 Explanation for evidence

Presenting the evidence almost directly from the underlying representation may be adequate for the user who has some experience with the format of evidence in um's persistent form. However, for the novice, more information is needed.

An example of a simple explanation for the first line of Figure 7.7 would have been lodged by the viewer and simply reports that the viewer program provided this evidence when the user indicated they considered the component was true.

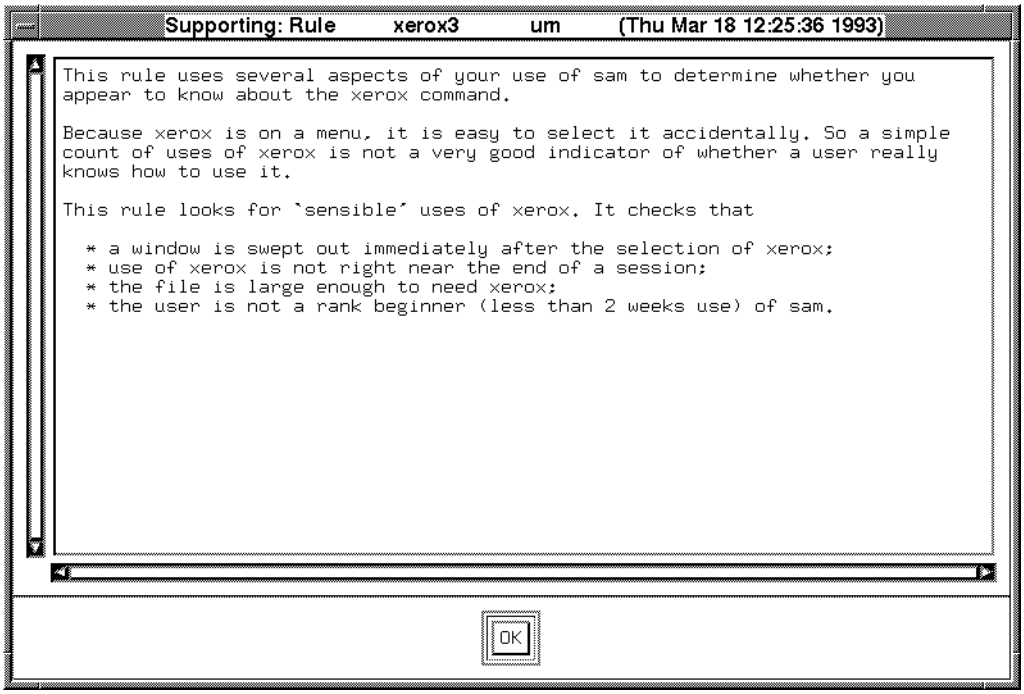


Figure 7.8. Explanation of an evidence item

A rather more interesting example of such evidence is shown in Figure 7.8. This is an explanation of a rule which combines a series of observations to conclude about the user's

knowledge of xerox, a sophisticated command. As the explanation indicates, this rule has to deal with the fact that it is easy for a user to select it accidentally because it is on a mouse menu. So, for example, it ignores the chaotic activity of the first two weeks of use of sam because that period is characterised by accidental selections of many menu-based commands.

7.2.4 Explanation of component meanings

The user model may be incomprehensible if the user cannot determine what each component is supposed to represent. For example, if the user does not recognise the term go_k in the context of the sam text editor, they need access to an explanation for it. That explanation might be a simple canned text.

As we have already noted, the design of um is based upon classes of tools, of which the explanation sub-system is one. Although our requirements do not demand it, Xum can customise its explanations. We found this useful for the sam domain where we saw a need for quite different explanations for sophisticated users from those provided to beginners. For example, Figure 7.9 shows an explanation aimed at a user with a low level of sam expertise.

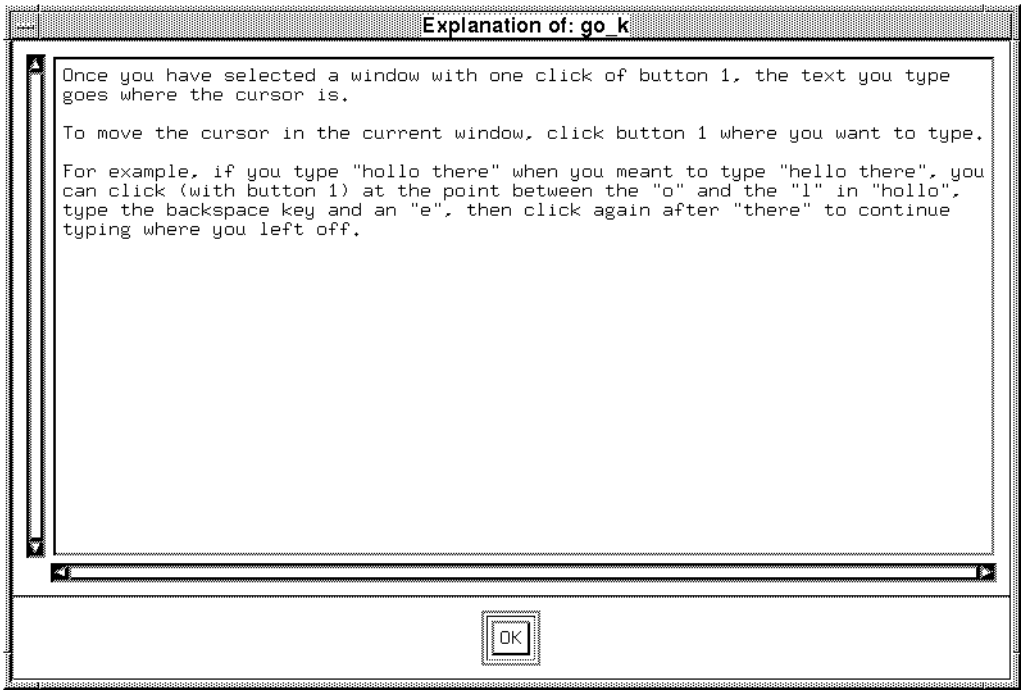


Figure 7.9. Example explanation intended for a novice

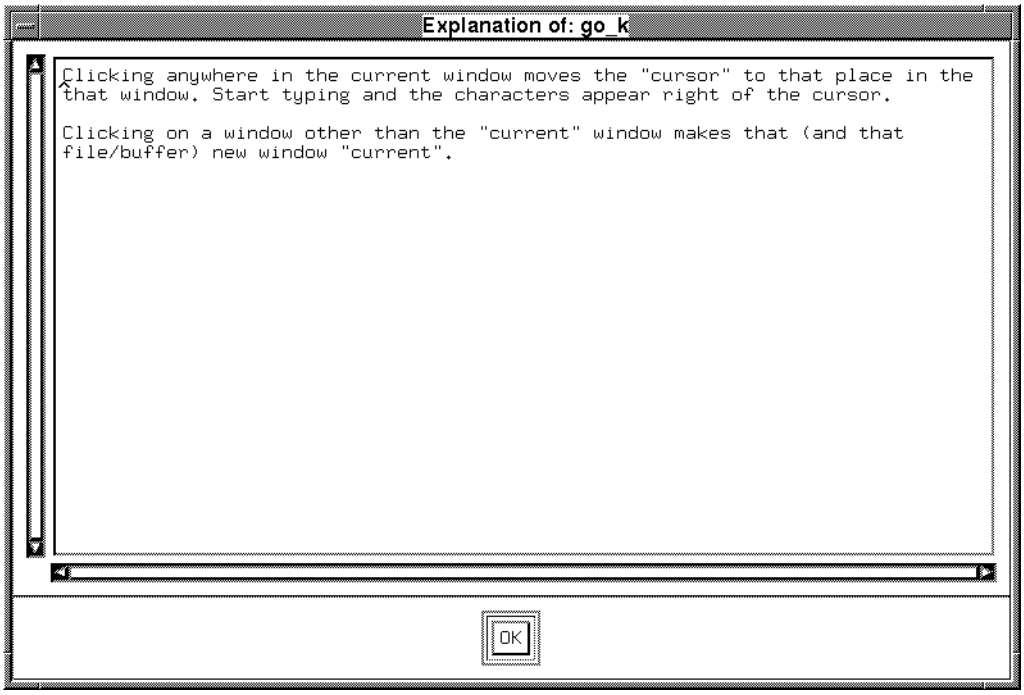


Figure 7.10. Example explanation intended for a sophisticated user of sam

Figure 7.10 shows an explanation of the same aspect for a more sophisticated user. These are handcrafted texts with the novice explanations being characterised by greater use of examples and avoidance of jargon even at the cost of longer explanations.

7.2.5 Altering a value

Our um design means the user could use a simple text editor or similar tool to alter the externally stored representation of their model. However, this is dangerous. If it is done incorrectly, the model will be corrupted and um tools will be unable to read the models.

It is far safer if the user can alter their model through a scrutability tool which also supports an undo operation. This is generally a desirable property for interfaces, as argued, for example, by Thimbleby (1990). It is particularly important in the case of scrutability support tools where the user should have a sense of control. If the user can alter their model, safe in the knowledge that this can readily be undone, then they can experiment with changes in order to explore their effect and be safe in the knowledge that the changes can be undone.

Xum allows the user to set the component value to *true*, *false* or *uncertain.* The actual effect of this is to add a new piece of evidence such as in the given items of evidence shown in Figure 7.7. If the user asks that the value of a component be set to *Uncertain* Xum adds two equally weighted evidence items with identical time stamps, one supporting and one negating. The source for these is the viewer.

Since the user's value-setting is simply another piece of evidence, its effect on the component value depends upon the resolver. Our first resolvers treated given evidence as the most reliable form of evidence. If a particular um-consumer uses a resolver that behaves differently, this should be clear to the user if they scrutinise their model using Xum when launched by that um-consumer.

7.2.6 Side effects of the scrutiny tools

We have described two role for these tools:

- to scrutinise the model;
- to alter the value of components.

Another kind of change occurs when the user requests an explanation. In this case, a told evidence item is added. This has a very low weighting, enough for default resolvers to push a model with no evidence (tentatively *false*) to *uncertain*. In Figure 7.7, the third and fifth evidence items were added after the user viewed the explanation for that component.

7.3 Summary

This chapter has described two classes of scrutability support tool:

- high level scrutability tools like qv which support navigation provide an overview of the model;
- fine-grained scrutability tools like Xum which give explanations of the ontology and modelling processes.

The particular scrutability support tools introduced in this chapter were developed and evaluated in the context of the sam domain. This is a substantial domain representing a significant and useful body of knowledge. However, a domain like a movie advisor is likely to model preferences for hundreds of actors and movies, dozens of directors and genres and many other aspects. We would expect to need a quite different interface for giving an overview of such a large model with looser structure.

By contrast, the fine grained scrutability support of Xum is close to the detailed

representation of the persistent form of the um model. It seems likely to be more broadly applicable. Even so, the um architecture allows a collection of such tools, as well the explanation generation sub-tools that provide the actual text the user will see.

The scrutability support tools must fulfil the scrutiny requirements identified in Chapter 3.

- *scrutiny_ontology*: provided by the explanation subsystem's explanation of components.
- *scrutiny_externals*: comes from the explanation subsystem's presentation of the external evidence and the explanations for these external sources.
- *scrutiny_internals*: is handled similarly to the external evidence except that it should provide fuller detail on the processes used by the internal inference tools.
- *scrutiny_resolve*: comes in part from the overview displays since they show the value of the components but these are explained in the explanation subsystem, typically as canned text describing the way the resolver operates.
- *scrutiny_simplicity*: is intended to make the model simple to understand, a goal that relies upon the overall um approach of managing user modelling with a collections of tools, each of which is individually simple and so, more straightforward to understand in isolation. In addition, the overview display assists the user in seeing the way that all the tiny parts fit together.
- *scrutiny_control*: is primarily supported by the interface operation that alters the value of a component.
- *scrutiny_scalability*: supported mainly by the overview interface tools since they enable the user to select the level of detail visible, to navigate around the model and so to cope with potentially large user models.

A fundamental aspect of the um representation is the use of uninterpreted evidence lists as the primary source of user model values. This means there is no single value for a component. A component only has a value in relation to a particular resolver. Essentially this means that the user needs to appreciate that a list of evidence is amenable to different interpretations in different situations. We believe that users will find this intuitively acceptable. However, it still leaves a problem for the user who wants to understand their model. As we have mentioned in this chapter, this is best addressed by using the scrutability tools in conjunction with a um-consumer.

We believe that this problem will not be as serious as it might seem at first. It seems likely that in practice most user model components, partial models and contexts will only

be used by a small number of different um-consumers. Moreover, it seems likely that most of these will tend to use a small number of resolvers. In such cases, there will be very few interpretations of the evidence in a single context, very likely only one or two. If this is the case, the user will generally need to scrutinise the modelling context with one or two resolvers.

In the small number of situations where a context is used by many um-consumers with many associated resolvers, the user's task will be far more difficult. We postulate a comparative scrutability tool for this case. This would accept a list of resolvers and help the user see the different effects of each. For example, these tools would display similar information to that in qv and Xum except that a set resolved values would be displayed with codings to indicate differences in resolver outcomes.

Chapter 8

Deployment of um

The last four chapters have described the um representation and architecture and its design foundations. The interaction model in Chapter 4 and the accretion representation of Chapter 5 provide the abstract foundations for our scrutable user modelling shell. In Chapter 6, we showed these applied to the design of um persistent model representation. Also in Chapter 6, we described tools constituting the essential functionality for the user modelling processes. Then in Chapter 7, we described the scrutability tools in Chapter 7. Now we move to more concrete demonstrations of um in two deployed systems.

These systems have quite different user modelling demands:

- one is based epistemological components, modelling the user's knowledge of sam, as a basis for a coaching system;

- the second models the user's preference for a movies advisor.

Together, these systems demonstrate um's deployment in quite different um-consumers. They support um's claim of flexibility, scalability and extensibility.

These systems are examples of the general architecture introduced in Chapter 3 and shown again in Figure 8.1. We produce the figure here so that the reader can more easily relate it to the similarly formatted depictions of the architecture of each system presented in this chapter.

Both were used extensively, with large numbers of users. In the case of the sam coaching system this involved 352 users in authentic field conditions. The movie advisor is a more complex system which evolved in several stages as we experimented with the um representation and architecture. Some of the main versions were deployed in the sense that they were made available to a large number of users. For example, one trial included 75 causal users.† In the coaching system, the primary modelling task is to represent the user's evolving *knowledge* of a text editor. This form of user modelling, commonly called student modelling, is a major part of user modelling work as evidenced by the many papers on student modelling in user modelling conferences as well as the user modelling journal (User Modeling and User-Adapted Interaction) and the book that preceded it(Kobsa and Wahlster, 1989). At the same time, student modelling is important in Intelligent Teaching and Learning Systems research where it is well represented in the early books such as (Polson and Richardson, 1988, Lawler and Yazdani, 1987, Mandl and Lesgold, 1988, Psotka, Massey, and Mutter, 1988, Kearsley, 1987, Wenger, 1987) as well as in the major conferences (AI-ED, Artificial Intelligence in Education, ITS, Intelligent Teaching Systems) and more recently a conference and subsequent book devoted to student modelling (McCalla and Greer, 1994).

The movie advisor is driven by a model of user *preferences* for various aspects of movies. It is representative of the fast growing number of systems that perform filtering and information retrieval tasks based on user models, including personalised news'papers' (Kamba, Bharat, and Albers, 1995, Shardanand and Maes, 1995, Morita

---

† This work involved a team of people. Ronny Cook worked through the main sam experiments and was the main programmer for the various viewer and model management tools in those experiment. The coaching was mainly the work of Noroja Parandeh-Gheibi but the experimental design also involved Katherine Crawford, Richard Thomas, Takashi Kato and David Benyon. Gary Butler was also part of this team and he worked on parallel Unix coaching experiments, also using um. The actual monitoring was implemented by Richard Thomas. The movies work had several people involved in eliciting knowledge for the stereotypes and building the various interfaces. In its later use, Ronny Cook built the troubleshooter interface.

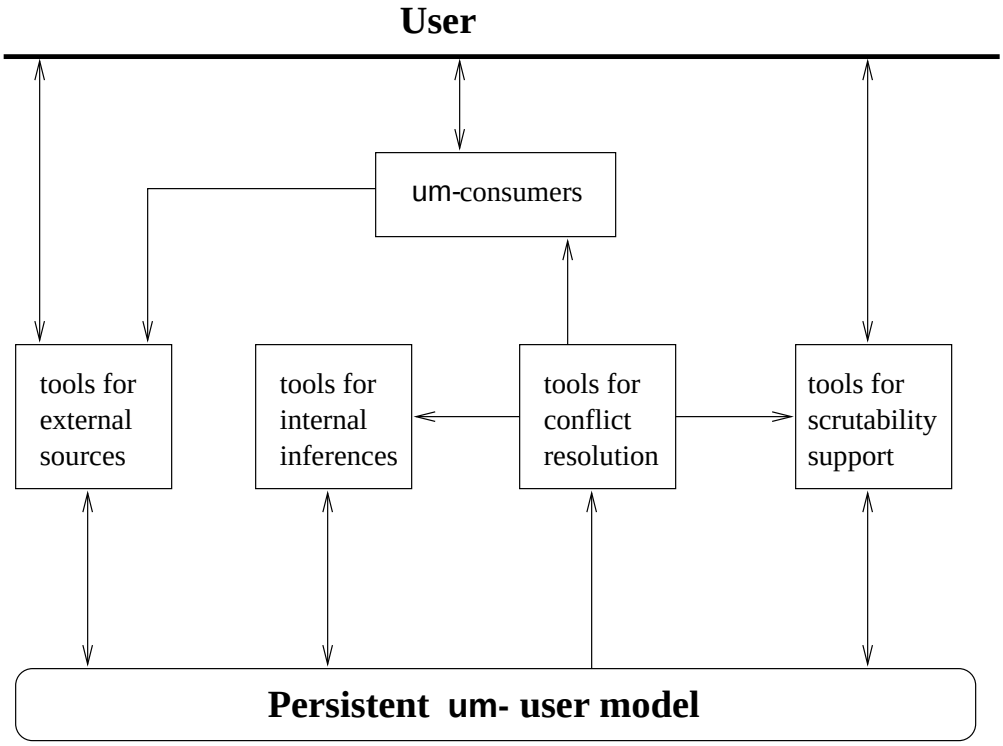


Figure 8.1. um architecture

and Shinoda, 1994, Mock and Vemuri,) and recommender systems (Resnick and Varian, 1997).

8.1 Coaching system

The motivation for the work on a sam coach follows from the observation that users of computer tools like text editors typically learn only a small part of the functionality of the tool (Jordan, Draper, MacFarlane, and McNulty, 1991, Carroll, 1990, Mack, 1990, Rosson, 1984). We have experimented with several coaching systems to help undergraduate students learn more about the text editor, sam (Pike, 1987). Our sam was modified to monitor each user's editing actions at the command level. For a detailed description of the monitoring, see (Cook, Kay, Ryan, and Thomas, 1995).

Initially, we just observed student's use of sam with no coaching system at all. Analysis of the sam monitor logs for our students showed that they developed quickly in their earliest periods of using the editor. A plateau was reached, after which their knowledge and skills developed extremely slowly, if at all. However, one striking counter-example to this pattern occurred when the class first had to write a program which was too big to

be fully visible in one sam window. At this point, we posted a message explaining how sam can display a specified line-number in a file. In the period just after this, the monitor logs showed a dramatic and sustained rise in the use of that facility.

We designed coaches to mimic this effect by sending electronic mail with 'timely' advice to each user about sam facilities that could make them more effective in doing their current tasks. Because the users were our students, we knew a good deal about their high level goals and tasks. Using this, we defined a timely 'syllabus': at the point when a facility should have first become very useful for the student's tasks, it appeared in the syllabus.

We now describe the user modelling architecture associated with one coach. Note that the system design was strongly influenced by the scale of its operation: we were modelling 352 users and the user model was built from logs of eighteen months of sam use. The coaching then ran for two months. These users were only able to use sam on our computers; so the monitor log for the population was a complete record of use.

Figure 8.2 gives an overview of one coach's architecture. A user is shown at the top and the persistent user model at the bottom. The elements of the coaching system, including the various tools, are in the middle in positions corresponding to those in the architectural overview in Figure 8.1. An example of an excerpt from a sam user model is shown in Appendix A. The main processes for constructing the user model are based on external evidence as shown at the left of the figure. The primary source of information about each user comes from observational evidence based on monitoring use of sam. This could have been added directly to the model. However, the two stage process shown in the figure was more flexible. This enabled experiments with various analysis tools over the same data.

The figure shows analysis done by the mcons, the Model CONStructor to create the evidence to add to the model. For example, seemingly correct uses of the xerox command were interpreted from the monitor logs and supporting evidence for the user knowing xerox was added to the model.

Some of the analyses could be quite simple, just counting command occurrences in the log and adding evidence once a threshold was reached. Other analysis had to be more sophisticated as, for example in the case of the xerox command whose explanation we discussed in Chapter 7. The xerox command creates multiple edit windows on one file. Because it is on a menu, it is easy for the user to unintentionally select it. To identify uses of xerox that were likely to be intentional and correct, mcons only counts uses of xerox which occur in a plausible context: the user followed the xerox command by sweeping out the new window needed; the file was big enough to exceed the size of one

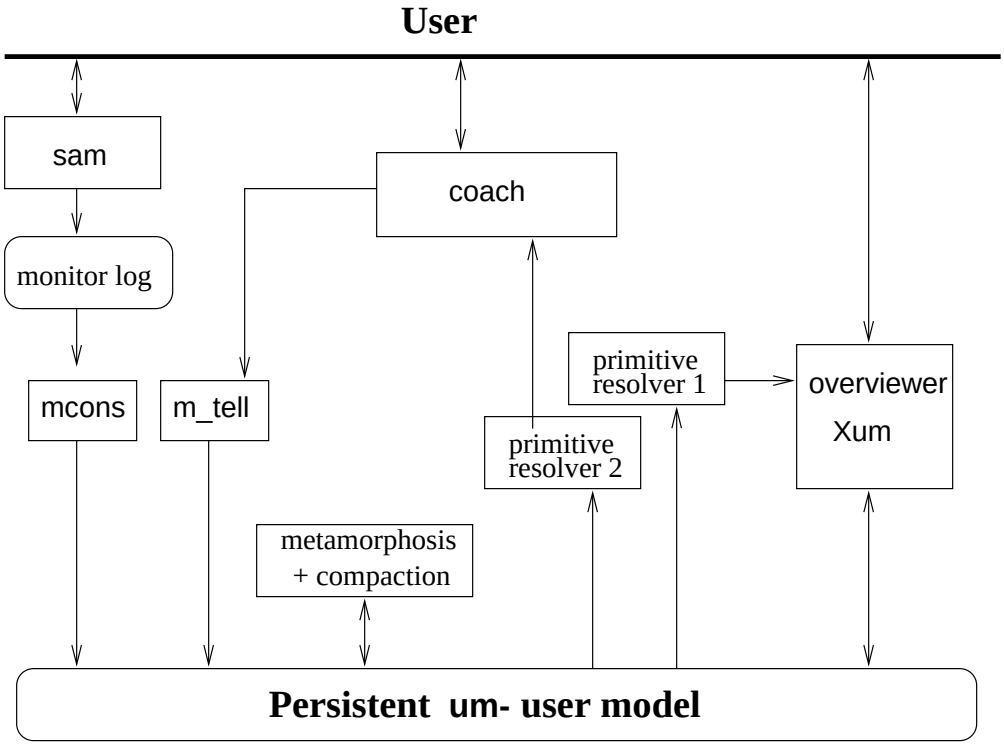


Figure 8.2. Architecture of a coach

window; the action was not right near the end of an editing session. Also, as the first period of editor use was so chaotic, it was ignored.

The next source of external information come from the coach via m_tell, a simple tool which can add information to the user model. The coach uses this to provide told evidence for each aspect it coaches.

The upper middle of Figure 8.2 has the um-consumer, the coach. It has two main tasks: selecting a coaching action and generating advice. The first was largely driven by the individual's user model. It selected a coaching goal only if the user model indicated the user did not know it. Ideally, it could select an aspect that met two conditions: it should be in its syllabus for the current time; it should also be at the frontier (Goldstein, l982) of the user's knowledge, meaning that the user does not know it but does know the knowledge required to understand it. This is not possible if the user knows all the current syllabus goals or if the user's frontier of knowledge is so far short of the current syllabus that the user does not know far simpler concepts and commands. Where the ideal was not possible, the coach favoured the frontier coaching strategy over the syllabus.

The metamorphosis and compaction tools were not used for the deployed versions of the coach. They were not necessary for the experiments which created an initial model for

the user after they had used sam for eighteen months and where the model was updated weekly for eight weeks. However, we experimented with approaches that might be applied in longer running systems where mcons was run each week. Like the examples in Chapter 6, the metamorphosis inferred from many pieces of observations to a stronger rule inference. Then compaction could be used to remove the trail of lower reliability evidence.

The last elements of Figure 8.2 are the scrutability support tools, qv and Xum shown in the last chapter. Note that these tools use a different resolver from the one used by the coach. We did this because we felt that the viewer tools should resolve the model with the user's own assessment of their knowledge having the highest reliability. Essentially, this mirrors polite human conversation where we tend to agree with people's self-assessments when we deal with them face-to-face. However, when it came time to create the coach, we decided that we trusted observations of the user more than their claim to know things. In hindsight, we consider the use of two different resolvers to have been a mistake. However, this is the architecture we created for the main sam coaching experiments.

8.2 Movie advisor

The movie advisor project was a testbed for tools to support individualised information retrieval and filtering. It operated from a database of movies with their attributes, suggesting movies that should appeal to the user. The movie database recorded many attributes of the movie, including the cast, awards it has won, director, its genre and subject matter. Some examples of the latter two are: Romance, Murder, Families and Family Life, Marriage, Death and Dying. The overall goal of the advisor was to model the user's preferences for movie attributes and to use this information to rate each movie in the database. Then it could recommend the highest rating movies.

Consider the issues in modelling the user preferences. Table 8.1 shows the distribution of different descriptors for genre and subject matter in the database of about 6600† movies. The first line indicates there were nine descriptors which applied to at least 500 movies.

---

† We designed the system with the goal of applying it to the full database. Our experimental work was on a subset of 406. For this subset, we carefully reviewed the information about each movie to ensure that it was reliable. So, for example, spelling errors in descriptors were corrected. In addition, we added importance weightings to each descriptor. So, for example, if a movie had the description 'Romance' but this was only a minor aspect of the movie, it was given a low weight. On the other hand, if this was the dominant feature of the movie, it was given a high weight.

There were 1665 different descriptors in all: the last row shows how many occurred at least once. It is not useful to model all of these low frequency descriptors as they cannot be used to predict the appeal of many movies. It seems likely that at least the most

Table 8.1. Cumulative frequency of descriptors

Number of descriptors	occur at least in this many movies
9	500
31	250
103	100
172	50
315	20
468	10
633	5
688	4
775	3
940	2
1665	1

commonly occurring 100 and probably 300 to 500 would be useful. Now, consider the potential mechanisms available for modelling shown in Figure 8.3. This figure summarises the relationships between stereotypes and domain knowledge. At the lower right is our goal, a model of the user's preferences for the attributes that describe movies.††

The three lines labelled G1, G2 and G3 represent different sources of given information. The arcs labelled S and R combine with these to achieve our goal. The combination of G1 and S uses personal attributes combined with stereotype-based reasoning about our GOAL. This requires an interface for G1 to collect user attributes like age and then a set of stereotype inferences like 'teenage males tend to like action movies'.

The other indirect route to the GOAL is G2 with R, where we use assessments of movies following by rules. For example, if the user indicates loathing the movies *Blade Runner*, *2001* and *Startrek*, we might infer that they dislike *science fiction* movies. This inference cannot be certain (since other aspects of these movies may be responsible). Such rules might be written in a general manner, based on analysis of the metadata for the movies in the sample. However, in our system, each rule was handcrafted, with the attributes of the

---

†† Although we limit discussion to this goal, we acknowledge other goals might have been defined. For example, we might seek to directly deduce preference for the movies themselves, without inferring this from the preferences for attributes of movies. This might be done with a rule such as 'liking $movie_1$ suggests you will like $movie_2$'. This gestalt approach is valid. However, our filtering is based on metadata about the movies and the assumption that the user's attitude to attributes of a movie is useful in predicting their response to the movie as a whole.

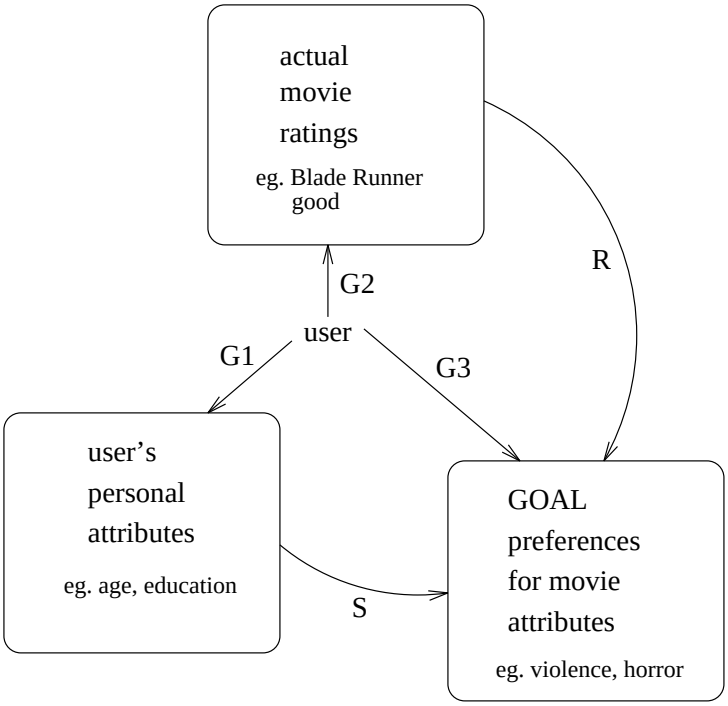


Figure 8.3. Sources of information used for movie attribute preferences

movies in the initial rating set built into the rules.

To make G1 and G2 useful, we must capture stereotype knowledge in S and rule knowledge in R. For our movies domain, we handcrafted both, the former from interviews of experts and the latter from the intuitive inference based on attributes of the set of movies selected for initial rating.

The remaining route to our goal is G3. This is the most obvious approach: simply ask the user to report their preferences for attributes of the movies. The bounds of user's patience makes this feasible for only a small number, far less than the hundreds of movies descriptors that Table 8.1 shows to be quite common and useful. By contrast, the indirect route is very appealing because of its leverage: it enables inferences on many of the goal-characteristics from a small number of user answers.

Figure 8.4 shows the architecture of the movie advisor and the tools that perform the different elements shown in Figure 8.3. As in Figure 8.1, the user is at the top, the persistent model at the bottom, the various tools between. An example of an excerpt from a movies user model is shown in Appendix A.

The advisor controls the system. On the first use of the advisor, the user is invited to answer some simple questions using the personal information tool. These are multiple-

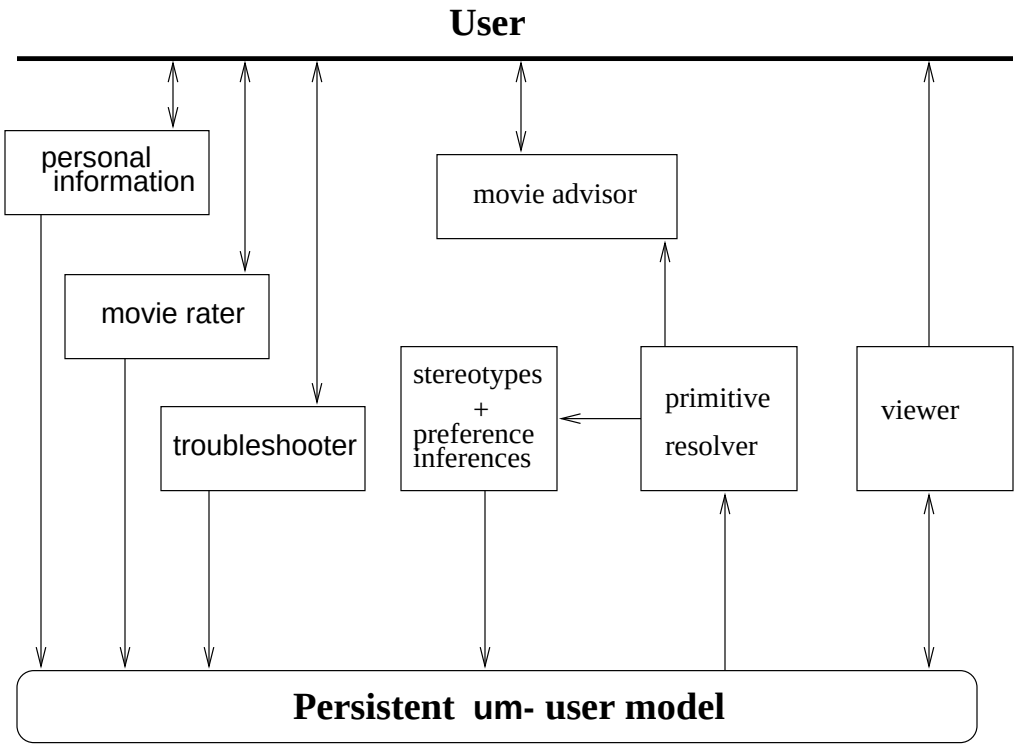


Figure 8.4. Elements in the movie advisor

choice questions about the user's age, gender, education level, and how often they see movies. This is the G1 phase of Figure 8.3. Answers contribute given information to the model.

Next the advisor runs stereotypes which infer from the personal information to a collection of preference assumptions. As Figure 8.4 shows, the stereotypes are part of a tool which accepts component values determined by a simple resolver. Each applicable stereotype adds evidence of class stereotype to the user model.

The next tool in Figure 8.4 is the *movie rater*. If the user agrees to rate some movies, this presented a screen like Figure 8.5. (This is G2 in Figure 8.3.)

At the upper right is a list of movies. Below is information about the movie selected. This helps the user assess even movies they have not seen, as well as prompting memories of ones they have seen. The left half of the screen has two sliders for the ratings: the upper one for movies the user has actually seen; the lower one for others.

Once the user selects the 'Press here when finished' button, the movie rater completes stage G2 of Figure 8.3 and enters the R inference phase. The general character of these inferences is simple: if the user likes movies which share a particular attribute (eg

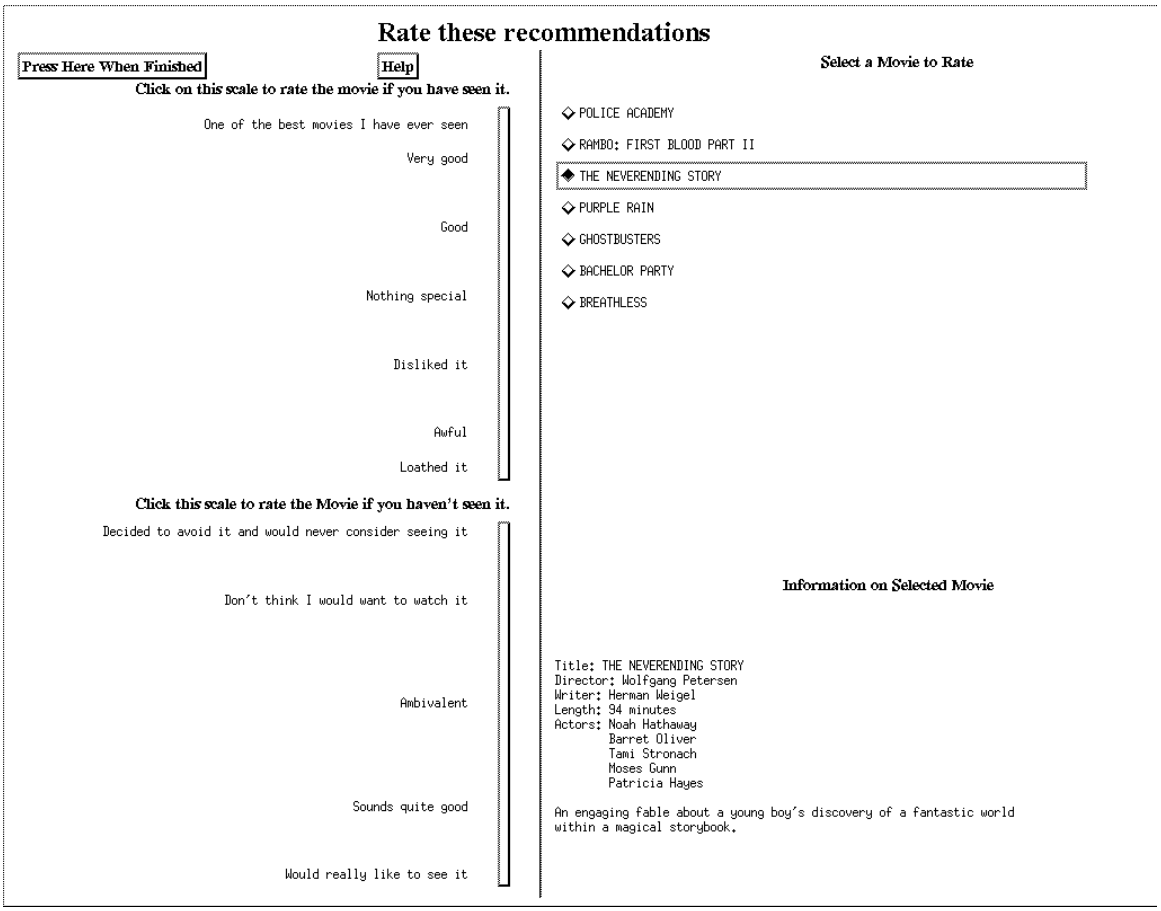


Figure 8.5. Start up rater interface

violence), the rules add evidence supporting the belief that the user likes this attribute in movies. A corresponding inference applies to disliked movies and their attributes. Several heuristics determine the strength of the inferences:

- the greater the strength of the user's rating of each movie, the greater its effect;
- ratings for movies the user has actually seen are given greater weight;
- the more rated movies with this characteristic, the greater the effect;
- the higher the importance score on an attribute the stronger its effect;
- the full set of ratings for a user is scaled to give a uniform range: this was to account for different ranges of responses users give (with, for example, some people using the extreme positive and negative ratings and others avoiding these entirely).

The movies the user is asked to rate were chosen to give a range of attributes but to repeat

common ones.

Note that the movie rater adds two sets of evidence to the model. First, it recorded whether movies had been seen or not. Secondly, it adds evidence from the user ratings. The slider of the rating interface looks continuous. Given the vagueness of any such rating, we actually represent the rating as an integer in the range 1 for extreme dislike to 10 for extreme liking. This is kept in the extras field of the evidence. In both cases, the evidence source is the movie rater.

This completes the initial phase. Thereafter, the movie advisor settles into its normal mode of operation, selecting movies to recommend and allowing the user to rate the recommendations. These are also presented by the movie rater interface. Although it was actually offering recommendations, the user might have seen these - they never included movies that the user was known to have seen. The user could rate these or simply quit; perhaps to go and see a movie!

When they next started the advisor, the user would be asked to rate the last set of unrated recommendations. Suppose that the user disliked one of the recommended movies, and indicated this through the rating interface. Then, the advisor had a problem: it recommended the movie because the information about the user and the movie indicated the user would like it. There are two general approaches to such problems: create mechanisms that draw the best conclusions they can; or ask the user to help. Given our focus on user control and scrutability, we use an interface which asks the user to help resolve the problem. The is the troubleshooter tool.

There are several possible sources for an incorrect prediction:

- the user model may be wrong about the user's movie attribute preferences;
- the movie database may have incorrect attributes listed, or their importance may be wrong;
- important attributes of the movie may be missing;
- the process that combines user preferences and movie attributes may have be flawed.

If the user agrees, the *troubleshooter* identifies problems from the first three sources. It presents a two dimensional grid like Figure 8.6. The vertical dimension indicates the user model's assessment of how much the user likes this attribute. So 'Crime' near the bottom of the grid indicates that the user dislikes crime movies. The horizontal dimension indicates how important the attribute is in this movie. The initial position comes from the movie database. The figure shows 'Crime' about a third the way from the left, as it is a fairly minor aspect of the movie.

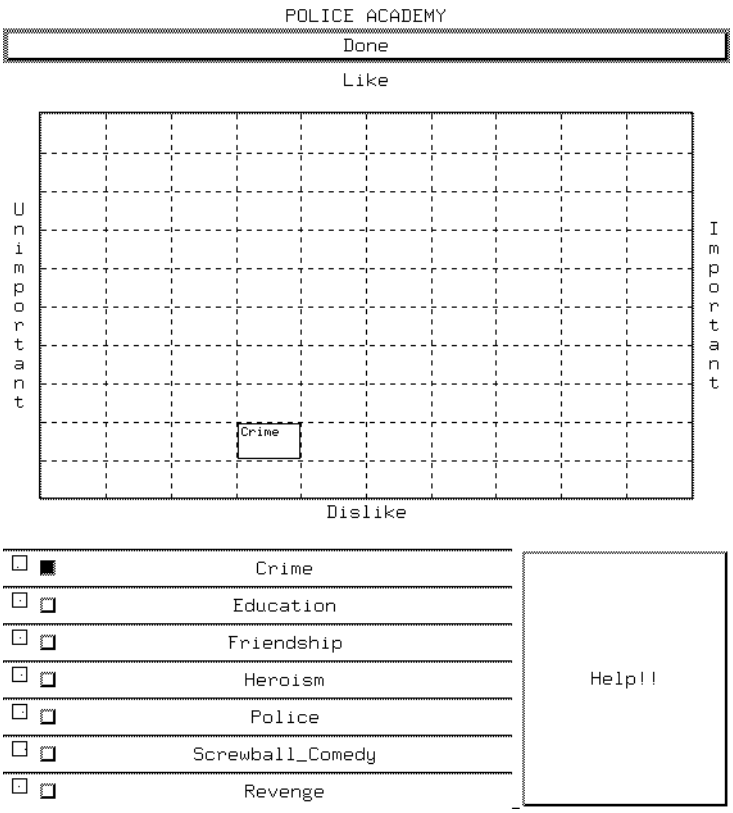


Figure 8.6. Dealing with unexpected user ratings of a movie

The user is then free to move the attribute. Vertical shifts update the user model with a piece of evidence indicating the new rating. If the user moves it horizontally, the information could be used to adjust the predefined ratings in the database as, for example, Rich (1983) and Goldberg, Nichols, and Oki (1992) do. We avoid this for several reasons. We took care to check the ratings in the database and were reluctant to allow their values to drift arbitrarily as users tried the system. We also consider that the effective use of such information requires considerable care and additional research that is a diversion from our user modelling focus. We simply logged the information.

If after going through all the recorded attributes of the movie, the user finds no disagreement with the attribute ratings or their own user model, they are asked to offer a descriptor for the aspect of the movie that they dislike. This is added to the user model and to movie database (although distinguished from other attributes). This means new movie attributes, and the corresponding new components, might be created during system use.

The movie advisor had to control this whole process, starting the various tools. In addition, it analyses the database of movie metadata to filter out movies it predicts the

user will not like. This relies on movie attributes the user *dis*likes. If a movie passes the filter stage, a rating is calculated with liked-attributes adding to the rating and dislikes detracting from it. The recommendations are the movies with the highest predicted ratings.

The final element of the architecture is a user model viewer. This could have been the same tool described in the sam work. The experimental work on the movies advisor was actually done with earlier versions of viewer tools.

8.3 Summary

The coach and movies advisor are examples of consumer systems for our scrutable user models and user modelling tools for user-adapted interaction. Our scrutable user modelling representation, architecture and tools are intended to support a broad range of user modelling consumers. The coach is an example of the systems that model the user's knowledge and the movies advisor is a recommender system which must model the user's preferences. We chose these quite different consumers to ensure our design work would be tested against different classes of modelling tasks and so that our approach might transfer to a broad range of user modelling domains. The coach's elements should be applicable to teaching in other domains, using strategies other than just coaching. Similarly, the movies advisor is intended to provide a basis for work in other filtering and selection domains.

These two um-consumers were built in parallel with the development of the um. They include some of the tools constructed in the process of defining and refining the um representation and architecture. This means that they were experimental in the sense that they were built so that we could explore the elements of architectures for user modelling. They provide examples of the four classes of tools in the architecture depicted in Figure 8.1, with tools for;

- external sources: mcons for adding evidence-based on the sam monitor logs, m_tell for adding evidence when the coach gave advice, and in the movie advisor the interfaces for gathering personal information, the movie rater and troubleshoot;
- internal inference: compaction rules for combining observations of sam command use and metamorphosis for concluding about aspects like the user's expertise level on the basis of commands known and not known; also, in the movie advisor, the stereotype inferences from personal information to preferences for movie attributes and knowledge-based inference from the movie ratings to those same movie attributes;

- conflict resolution: using a simple partial ordering of evidence class combined with source;
- scrutability support: um and Xum in the sam coach and their predecessor versions in the movies advisor.

Given our focus on scrutability and the relationship between the user and user modelling system, our strongest focus has been on tools for collecting external evidence and for scrutability support. This is in contrast to most of the systems described in Chapter 2 where the primary focus was on internal inference and conflict resolution.

Chapter 9

Evaluation of scrutability

Our primary goal has been to create a user modelling framework and toolkit to support scrutable user models, ones that can be scrutinised by the person they model. In this chapter, we describe two experiments which assessed um's support of scrutable user modelling.

The first experiment was small-scale, formative and qualitative. It was intended to inform the design of um and its tools. It also served as a basis for the design of the second experiment which involved a field trial with a um user model and two scrutability support tools.

Domain

The domain is the sam text editor. This is a substantial and complex piece of software, with approximately 90 different user actions or classes of commands. The corresponding model for a user's knowledge of sam is of sufficient size and complexity to constitute a significant user modelling task. In the two experiments, there were about ninety components representing the user's knowledge of various aspects of the editor.

The last two chapters have shown several examples from this domain, including the scrutability support tools used in these experiments and excerpts from the models. The last chapter described the architecture of one sam coach. The evaluation work in this chapter is based upon user models which were primarily based upon analyses of sam monitor logs. The experiments used a small set of tools:

- model construction tools that analyse log files of the form used for sam monitor logs and construct an individual's user model;
- primitive resolvers described in Chapter 6;
- the same scrutability support tools used to illustrate the principles described in Chapter 7.

Model construction

The sam models for these experiments were largely based upon data from monitoring use of sam at the keystroke level. This form of user modelling information has considerable long term promise because there are many situations where monitoring could be a source of low-grade but plentiful data about the user.

The sam monitoring and building of the user models are described elsewhere in detail (Cook, Kay, Ryan, and Thomas, 1995, Thomas, 1998). Essentially, we performed keystroke level monitoring of sam use. The monitor logs were analysed by mcons, an awk-based tool which counted command uses. Once a threshold number of uses was attained this constituted a piece of evidence that the user knew the command. At the same time, inactivity caused a decay in the count. So a command that was not used for a long period of time would fall below the threshold.

Essentially, both experiments involved creation of a user model after we had collected several months of monitor data for each user. This was partly dictated by the timing of our experiments. However, it was also due to the nature of monitor data: the low quality of such data means that an accurate user model can only be constructed if we have substantial amounts of data. At the very least, this is needed to account for people allowing others to use their accounts and people getting help from tutors and peers. Over

the long term, one would expect these effects to be swamped by genuine activity by the user. In addition, for mouse-based commands which can be accidentally selected, one can only make confident claims about the use of a command over a long period of monitoring.

Resolvers

Both experiments used the primitive resolvers described in Chapter 6. As we will see, one of the outcomes of the first experiment was a refinement to the resolver.

Scrutability support

The two experiments used different scrutability support tools. The one used in the first experiment was called viewer and it supported detailed scrutiny of the user model, in the same style as the examples of Xum shown in Chapter 7. Essentially, the viewer was a prototype for Xum which was used in the second experiment in conjunction with qv, which is an example of an navigation and overview scrutability support tool and was used for the examples in Chapter 7.

The user models

The two experiments describes in this chapter represent two points in the development of um and its tools. So there are some differences between details of the models in each of the experiments. They are also some significant differences from the general representations described in the preceding chapters.

The detailed form of the user model for the experiments is indicated by the sample in Appendix A. It differs from that described in the earlier chapters of the thesis. One obvious difference is that times were kept as a large integer (the number of seconds since Jan 1st 1970). This was simpler to manage than a more intuitive string format and at that stage of our work, it was the way we kept time in the persistent user model.

Another important difference relates to the design of our experiment. We were experimenting with the models and wanted to be sure that we could alter the modelling at short notice. So, some of the aspects that Chapter 6 describes as being kept centrally were kept within the individual models. These are the component definitions and specification of the locations for explanations of the user modelling information sources. These were kept at the individual level so that we could alter models at that level if the need arose.

There is another important difference between the experimental conditions and those we

have described in Chapter 6. In the experiments, all the individual models were stored centrally. This meant that the actual model files were not accessible directly to the students. This is not consistent with our philosophy that um models should be kept in the user's filespace and under user control. However, in this experiment we needed to maintain control over the models.

We considered keeping the main model in the user's own file space and shadowing this so that we could detect cases where users directly changed the model files. This possibility was rejected for two main reasons. Firstly, it would have added an additional level of complexity to the experiment and introduced the potential for greater difficulty in both managing the experiment and interpreting the results. Secondly and more critically, our students had quite restricted allowances of disk space. Had we kept the user models as files owned by the users in their own filespace, we feared causing antipathy in our user population when some exceeded their disk quota.

At the same time that we were refining our um representation, we were also developing our understanding of the modelling of knowledge in the sam domain. So the detailed structure of the sam model evolved between the two experiments.

For example, the form of the sam model we used at the time of the first experiment is slightly different from that used in the second experiment. The main model used for Experiment 2 is shown in Appendix B. Although there are small differences between these, they are of a similar size and complexity. The differences are:

- The second set of sam models was constructed for the coaching experiments that involved both sam and unix. So the common elements of sam and the unix shell regular expressions were organised so that they would be common for both models. This meant that the common components were placed in a separate partial model.
- The later model included belief components where the first did not. This was largely due to our growing awareness of some of the user misconceptions. For example, many users who disliked sam seemed to be unaware of the importance of the command window and the powerful commands that are only accessible from it. So the second experiment included a component representing the user belief that sam can be effectively used without the command window.

9.1 Formative experiment on scrutability

Since the goal of um was to support scrutable user models, this formative evaluation assessed whether users could indeed scrutinise and understand their um models. Our approach derives from work on usability. For the viewer interface and underlying user model, we express our goal as follows:

> that the user should be able to to use the viewer to explore the full range of explanation and justification facilities without instruction.

This requires that the scrutability tools be easy to learn and use, aspects typically listed as central to usability. See, for example (Newman and Lamming, 1995, Nielsen, 1993).

We asked users to assess the accuracy of their model. This task gave the users a reason to scrutinise their model. In addition, the task of checking the accuracy of the user model is one of the primary reasons that we want to support scrutability of a user model.

9.1.1 Experimental design

This formative experiment assessed how users responded to the interface, whether they could scrutinise all aspects of their models and how we could improve the interface or um in light of the user's responses. We refined this broad formative goal to a set of sub-goals:

- Goal 1: Interface performance goal: to assess whether users could navigate through the interface to explore their user model since this is clearly a pre-condition for scrutinising it;
- Goal 2: Formative goal: to inform the longer term development of the interface, including the details of screen design and instructions and other text presented to the user;
- Goal 3: Affective goal: assessing user responses to the overall notion of scrutinising a user model;
- Goal 4: Evaluative goal for scrutability: assessing whether users could evaluate the accuracy of their user models. Note that this conflates two distinct issues. It relates to scrutability of the user model representation itself, *the* central concern of this thesis. However, it also depends in part on the model's accuracy being good enough to make this task meaningful.

Methodology

The experiment employed two main usability assessment tools, think-aloud protocols (Preece et al, 1994, Nielsen, 1993) and open-ended questions about affective issues.

Participants

Primary concerns in selecting participants were that we should have a diverse pool of users and that we should be able to construct a user model for them. Our methods for constructing the user models required that the users had been monitored for a significant period of time. So we selected participants from a class of students near the end of their first year of using sam in the first year computer science course.

Within the constraint that all our users are first year computer science students, our selection mechanism sought to ensure diverse approaches to learning and using computers. We selected participants as follows:†

- We posted a machine message to all Computer Science 1 students to say we were interested in student learning of sam and we invited students to perform approximately one hour of psychometric tests on various aspects of learning style.
- From these, we selected twelve regular sam users who represented a diverse range of learning styles.
- Each was invited to participate in a one hour interview and was told they would be paid a small honorarium. Seven were able to participate.
- The students negotiated a time for the experimental period, and these ranged from 29th October to 26th November 1991, corresponding to just before or just after the final examination period for that year.

For this type of study, this number of participants was beyond the minimum of 4 +/- 1, recommended by Nielsen (Nielsen, 1994) for think-aloud studies that are intended to be formative, identifying the problems in an interface.

---

† This aspect of the sam work was part of a study that investigated relationships between learning preferences and learning outcomes with sam. The work was done by a team which included Dr Kathryn Crawford and Dr Richard Thomas in addition to the author.

Form of the experimental session

The interviews† had the following structure:

1. *Introduction*: The interviewer introduced herself and explained the overall purpose of the interview was to help us assess software we had developed. We emphasised that the interview and sam monitoring were unrelated to the assessment for the Computer Science course. We then explained that we had constructed a model of the user's sam knowledge and we would like them to check it to see if they considered it accurate.†† We briefly outlined the interview structure.

2. *User experience of* sam *and learning it*: We asked the user how they felt about sam, learning it and using it. This was an open-ended question to give users an opportunity to express views that were not tainted by the experience of using the viewer or answering our questions.

3. *User view of knowing*: We asked the user to explain how they assess when they know something, especially something in computing. This is critical for Subgoal 4 since we had to be confident that we understood the user's perception of knowing something when they assessed the accuracy of the interface. Given the context of this question, we expected that users would answer the question in terms of their 'knowing' sam and similar classes of knowledge.

4. *Initiate use of* viewer: We showed the user how to invoke the viewer. Then, we asked them to start exploring it and to explain what they were doing as they used it. If a user failed to talk about what they were doing, we made simple prompting queries asking what they were reading, doing or the like. Essentially, we asked them to *think-aloud*, while we observed. This follows the recommended role of think-aloud as a clarification for the observer about what the user is doing or trying to do (Newman and Lamming, 1995).

---

† Dr Kathryn Crawford, Faculty of Education, University of Sydney, conducted all but one interview (User A). She had extensive experience in performing such interviews and had previously been driving the cognitive profiling work. Also, it was intended that she conduct the interviews so that the students would be dealing with an outsider who they did not know and who had no role in their assessment. In the event, there was one interview for which she was not available and the author did that.

†† After the first few interviews, we decided to add monitoring to the viewer. Once we had done this, the introduction included a statement that we would monitor their use of the viewer as an additional source of information.

5. *Initial, observed use with think-aloud*: We stayed for approximately ten minutes, watching the user. This was partly to ensure we would be on hand in case the software or the system failed or the user wanted us to help them. †
6. *Private use*: We asked them to continue using the viewer program, checking out their model while we left them alone. This was to allow them to feel less inhibited about exploring any aspects they chose. (For users D, E, F, G monitoring was in place so this private use was monitored.)
7. *Final assessment of the* viewer *and model*: We returned near the end of the allotted one hour and asked for any comments, prompting for comment on the accuracy of the model and whether they thought they would use the viewer.

9.1.2 Results

The written interview summaries by the observer are in Appendix C. Here we give summaries of the answers to our questions as well as our observations.

User view of sam and learning it

The first question invited users to indicate their feelings about sam, using it and learning to use it. The responses are summarised in Table 9.1. They indicate that some users had difficulty learning sam (D, and G) and others referred to other editors they liked to use (A, B, C, D). Only one user actually said they liked sam (user C). Three users (A, E, F) either said or implied that they found sam easy to learn. However, their comments through the interview indicate that they knew only some of its more powerful facilities, which might make it more attractive than other editors mentioned.

User view of knowing

This question is important in enabling us to understand user comments on the accuracy of their model. The responses are summarised in Table 9.2.
The most consistent response is that *use* of a command or aspect was central to knowing (Users A, D, E, F, G). For a learning domain like the sam text editor, this is a reasonable criterion for 'knowing'. Only user B did not mention this, instead focusing on difficulties in conceptualising sam and an associated lack of confidence in remembering it.

---

† In the event, the only such intervention was in the first interview where some access permissions had to be adjusted.)

Table 9.1. Summary of user experience of sam and learning it

User ID	Answer
A	Found sam easy to learn and thought it similar to the Macintosh. Likes some facilities of sam but prefers vi and mainly uses it.
B	Prefers vi but uses sam for some required assignments.
C	Likes sam but also uses vi and xedit.
D	Found it hard to learn and feels very pressured, with little time to spare to learn sam. Sees no advantage of sam over xedit
E	Mainly commented on learning by word-of-mouth. Found basics easy to learn.
F	Mainly commented on other people's confusion and difficulty in learning and implied own learning was not a problem.
G	Found it difficult to learn. Seemed to prefer home editor.

Table 9.2. Summary of user views of knowing

User ID	Answer
A	Likes deep understanding and likes to learn by doing.
B	Expresses difficulty in conceptualising sam
C	Has a big picture of the editor ... talks in terms of exploration, curiosity and comfort in using sam
D	Use once at least
E	Knows [something] when they can use it and feel comfortable
F	They try it out. Then it's added to their repertoire.
G	'I know what I use often.'

Initial, observed use with think-aloud

In this period, all users were observed to make a dutiful effort to explore the model. This period proved very useful in identifying a number of small improvements in the interface. For example, some instructions were improved. Also, user's comments and non-verbal reactions to some of the explanation texts made it clear that there was room for improvements in these.

In terms of the primary scrutability thrust of this experiment, the central finding was that the users were able to use the viewer to explore the values of the components. As one would expect in a prototype, this period of use gave use many ideas for improving the interface.

An important change we made during this experimental period was the addition of monitoring to the viewer itself. Once this was in place, we were able to track in detail the aspects users explored.

A significant interface enhancement conceived during this evaluation was the qv interface. At the time of this experiment, the viewer interface provided access to the

detailed view of the model similar to that for Xum as described in Chapter 7. So the user could explore the model by expanding one node of the model at a time. As in Chapter 7, they could then change the value of a leaf node, see the component meaning or explore the list of evidence for a component and the explanations for that evidence. However, there was no interface support for the 'big picture', the overall look of the model. As we watched users explore the detailed model, we conceived of the need for this additional scrutability support interface.

Private use

The initial design of this experiment had not included any tracking of the user's private activity with the viewer. The purpose of the private period of use was to enable the user to decide which parts of the model they wanted to explore and to allow them to work with the tool without being observed by us. This would mean that their comments at the final stages of the interview could be informed by activity that was not constrained by our presence.

After the first set of interviews (Users A, B, C) we decided to supplement the other interview data with data from monitoring. The monitoring is summarised in Table 9.3. Columns with two numbers indicate the total number of actions of that type first and then the number of unique components the command was applied to. For example, User D sought 56 explanations for components, of which 44 were for different commands.

Table 9.3. Summary of viewer-monitor logs

User ID	Counts of activities by this user				
	Navigation	**Explain**	**Evidence list**	**Evidence explain**	**Change actions**
D	115	56, 44	39, 27	20	9, 3
E	45	14, 9	2	-	1
F	38	4, 3	9, 7	3, 2	1
G	37	5, 5	4, 3	1	7

The log for User D runs from 09:36 to 10:48 (Mon Nov 4). It is characterised by a quite extensive exploration of the model, including all the major parts of the model, explanation for 44 different commands and the evidence lists for 27 different commands. Of the repeated evidence lists explored, the wpm component which modelled the user's typing speed was selected four times and the commands for which evidence lists were explored three times were pascal_prog, modelling that this user was a Pascal programmer, snarf_k and write_k, two sam commands. The 9 changes to the model involved only three components, wpm (changed 6 times), subst_k twice and write_k once.

User E had a relatively modest sized log for the period 15:15 to 15:34 (Wed Nov 20).

Notably, this user never sought an explanation for the evidence. Of the 14 explanations this user selected, there were three for quote_k, two each for comma_k, line_num_k and search_k and one for cut_k, dot_k, subst_k, undo_k and xerox_k, all aspects of sam. The change to the model was for the component xerox_k.

The log for User F is rather different from the others. This user was interviewed on Tue Oct 29 at which time a first, simple form of viewer monitoring was implemented. The information from that monitoring was in a form that is not comparable with that for the later interviews. However, this user made subsequent use of the viewer after the interviews on Wed Oct 30, Thu Oct 31, Fri Nov 1 and Wed Nov 20. Only the last of these uses was tracked by the form of monitoring used in the other interviews reported in Table 9.4 and so it is the only part of their use reported in that table. The full log for this user had 175 actions. The part summarised in the table is simply an indication of the user's behaviour in using the viewer, exploring all the facilities it offers to examine explanations for components, evidence and its explanation.

User G was interviewed on November 26. Although this user might appear to have been less active than the others, they did explore all the main parts of the model. This user is unusual in not having revisited any of the component explanations. For three of the commands they changed the values, they also selected the explanations. These were dot_k, listf_k and rexp_dot_k, all aspects of sam. This user's pattern of actions was to navigate around the model, then change a component value and then explore component explanations and evidence lists.

Final comments

This final stage of the interview had two stages. First, users were asked if they agreed with the model. The responses are summarised in Table 9.4.

Table 9.4. Summary of final assessment of the viewer

User ID	Answer
A	yes, reflects knowledge
B	generally accurate
C	generally thought accurate
D	OK
E	generally OK
F	generally accurate
G	generally agreed

All users stated that they considered the model was essentially accurate.

One important exception related to the way that the resolver worked during use of the

viewer. In our initial resolver, we had decided that the action of viewing the description of a component of the model constituted weak (told) evidence that the user knew that concept. We felt it was polite to assume that once a user had 'seen' an explanation of a concept, we should regard them as knowing it, though not necessarily well. As our users explored the model, and the explanations of components, user 2 user expressed surprise that just looking at an explanation of a component changed the value of that component model, showing them as knowing it. This user was quite adamant that this was most inappropriate. They pointed out that they read the component descriptions to convince themselves that they did *not* want to know it.

This outcome is consistent with several users' statements that one only knows something if one has put it into practice repeatedly. Although the other users did not bring up this issue, there was strong agreement that use of a sam aspect was required before one could judge it known. In light of this, we decided to alter the viewer's resolver to ignore all told evidence in assessing whether a user knows a concept in sam.

The concluding question asked the users whether they would use the viewer. Responses are summarised in table 9.5.

Table 9.5. Summary of whether the user would use the viewer

User ID	Answer
A	by end of interview, said yes but would like it for vi and unix rather than sam
B	no - does not want to learn more
C	probably
D	no - dislikes sam
E	yes
F	yes
G	may save some time

Three answered yes (user A, E, F) and three indicated lack of interest in learning about sam (user A, B, D). Overall, users seemed to show polite interest in having access to their user model.

9.1.3 Discussion of Outcomes

The strongest outcome was that our users disliked sam. This is not relevant to our central concerns so we will not discuss it further. We report the major outcomes in terms of our four evaluative goals.

Goal 1: Interface performance goal

All users were able to use the viewer for the purpose of assessing the accuracy of the sam model. Without any instruction from us except in how to start the viewer, they could move around the interface and see what the system assessed them as knowing or not.

The monitor logs support our observations in concluding that the users were able to perform all the major actions available for the interface: navigation around the model, seeking component explanations, listing evidence about a component and examining the explanations for the evidence.

Goal 2: Formative goal

There were two important outcomes. The first was in relation to the design of the viewer interface available. It was during this evaluation that we identified the need for an overview display of the model. We also identified tuning changes to the interface and the need to improve texts explaining the components.

For the underlying user modelling representation, there was an additional important outcome in that we changed the resolver used by the viewer.

Goal 3: Affective goal

The users made no negative comments about the interface. Since we were paying them (a very modest honorarium) and all were our students, we cannot be confident that they would be entirely honest in assessing something we obviously cared about. On the other hand, the users did seem to feel able to show quite clear dislike for sam. So, perhaps we can have some confidence that the users would have communicated significant dislike of the viewer.

For this goal, the formative study gave no clear negative feedback but cannot be regarded as a strong reassurance that users generally liked the viewer and the notion of viewing a user model. Perhaps the most positive feedback was User A's comment that they would like a similar model for things that interested them (vi and unix).

Goal 4: Evaluative goal scrutability

The users indicated substantial agreement with the model. They seemed able to assess what it showed and to judge its accuracy. Bearing in mind that they did not have the overview interface, users had to explore the model in many steps to see what it showed. The changes users made to the model seem to have been partly to explore the viewer and

the effect of changes on the evidence list.

Overall, the users comments and actions (both observed and monitored) indicate they were able to scrutinise their models. The most significant impediment to this was that the component explanations were hard for these users to understand. This meant that it was not easy for the users to learn about a new command from the explanations. From the users comments and actions, it seems that for the sam aspects the users did know, the explanations were sufficient for the task at hand, judging the accuracy of the model.

9.1.4 Conclusions

Essentially, this experiment had two goals. Firstly, it was to assess whether users would be able to scrutinise their models and understand them well enough to make a meaningful evaluation of their accuracy. In our experimental design, we expressed this as:

> that the user should be able to to use the viewer to explore the full range of explanation and justification facilities without instruction.

In terms of this goal, the experiment was promising. In the quite artificial conditions of this experiment, the users were able to explore their models in detail. In response to the task we set of assessing the accuracy of the model, the users were able to use viewer as a scrutability support tool to find the components modelled, their meanings, the evidence defining their values and for most, the meanings of some evidence.

Our second goal was formative. We intended that this experiment should inform the future development of scrutability support tools. It did this in the following ways.

- The resolver should ignore the evidence based on the user being 'told' about a component in terms of a viewer explanation for a component was selected. This applies for the sam domain where it is reasonable to conclude that a user only knows sam aspects if they can successfully use them. This is consistent with the approach taken by Chin (D Chin, 1989) where using a command properly indicates the user knows about this command. It also accords with the user's answers to the questions about their own way of judging when they know. We conclude that in this domain, the user should have actually made use of a command before we judge it known.
- We needed to improve the explanations for the components. After this experiment, we concluded that it would be desirable to customise component explanations to match the user's level of expertise. The explanations in this experiment were appropriate for an expert sam user but not so for the actual user population in this experiment.

- Observing the user navigate around the details of their models, we concluded that there should be a scrutability support tool which helped users see the overall structure and values of their user model. This was the genesis of the overview-type tools and, in particular qv.
- The monitoring for the viewer use aided our understanding of the way that people used the interface. This experiment helped inform the design of monitoring for the next experiment.

9.2 Field test evaluation of scrutability

Our primary goal in this thesis is that um user models and associated user modelling tools should be able to drive user-adapted interaction for systems like advisors, consultants, help systems, recommender systems and systems that tailor information presentation. These all share a similar relationship with the user model: the user model is an enabling tool for the user-adaptation.

Our first experiment was helpful and promising. It was conducted in an artificial, experimental setting where the user (a student, who most likely had little income) was paid to use some software. Moreover, it was based upon the user model as a primary focus for the user. It was reassuring that under these circumstances, the users explored their model and applied their understanding of it to assess its accuracy. However, we want to evaluate the scrutability of um user model in a more authentic environment. We now describe an experiment designed to assess the scrutability of um user models in a 'natural' setting.

9.2.1 Experimental design

Chapter 8 described the architecture of a sam coach.† Essentially, this experiment involved making the coach's user model available to users.

It is important to understand the context of this experiment from the user's perspective:

i. the central task for most users was to learn computer science, and at the particular point in time of this experiment, this meant the students had to learn the programming language C and various unix programmer's tools like make and awk (among other learning goals);

ii. to program in C and write scripts for various unix tools, the students needed to create source code files;

iii. sam was one of the text editors available for the task of editing source code;

iv. the sam-coach was intended to help them learn to use sam more effectively;

v. the sam-user-model drove the sam-coach;

vi. and finally, two scrutability support tools called qv and Xum enabled users to see and change their user model.

---

† The coach was build as part of a larger study of sam users (Kay and Thomas, 1995) and of users learning from various forms of coach (Gheibi and Kay, 1993)

In summary, viewing the user model (vi) was several levels removed from the user's central task (i). We considered that this constituted a fair but tough evaluation of several aspects of the scrutability of um user models.

We define the goal for our second experiment in terms of a set of increasingly demanding subgoals.

1. Field test the use of scrutiny support tools: Essentially, this is a prerequisite for the scrutability of the um model. If users are simply not interested in their models, they will not explore them. This possibility is not strictly part of our brief. However, if we are to provide effective scrutability support, we need to be aware of it.

   Essentially, this subgoal seeks to answer the question: *will users scrutinise the user models at all?*

2. Field test the non-trivial use of scrutiny support tools: This is a more demanding subgoal than the last but it is still a prerequisite for scrutability. If users do start using the scrutability support tools but then make only minimal use of them, they cannot scrutinise their models. The reasons users might do this are be complex and varied. It seems likely that the quality of the scrutability interface is likely to be important for this aspect.

   Essentially, this subgoal seeks to answer the question: *for those who make some attempt to scrutinise their models, will some examine them more than perfunctorily?*

3. Field test the extensiveness of user scrutiny of their model: This subgoal will only be meaningful for those users who move beyond the levels assessed in the preceding two goals.

   Essentially, this subgoal seeks to answer the question: *for users who spend some time scrutinising their model, will they scrutinise all the major elements: the model structure; the meaning of the components; the evidence that defines the component values; the explanations for the evidence; and will users act to alter the value of user model components?*

Methodology

Our experimental method for this experiment was a field test (Newman and Lamming, 1995, Preece et al, 1994) with passive observation by monitoring. As Borenstein (Borenstein, 1991) notes, passive observation is 'simple, generally nonintrusive, and natural - the tasks observed are the ones that are actually useful'. Moreover, if we are

interested in whether users will try to scrutinise their model and whether they will give up exploring it, a laboratory evaluation is inappropriate. This has been observed, for example by (Eason, 1984)

> one of the most telling indicators of non-usability is when a user stops using a system. How do we arrange for our subjects to visit our labs in order not to use our systems. (Eason 1984:885)

Note that by the time of this experiment, we were well aware that sam was not very popular with some of users. By this point, we did not have the option of changing this situation and judged it acceptable since it merely made sam-coach and associated um user model less attractive and so the experiment becomes a tougher test of scrutability.

Participants

The participants in this study were enrolled in their fourth semester of computer science study, most having begun their first year course in February 1991. We randomly selected 81 students from the whole cohort of 352 students. Our criterion for selecting the students was based on the basis of the second last digit of their official student identification number. This was chosen since it gave a random distribution on student's previous computer science performance (Butler, 1992).

The sam coach sent these students weekly mail about sam. Other students were in different control or experimental groups for other experiments (Butler, 1992, Gheibi and Kay, 1993). This was intended to ensure that no student was advantaged or disadvantaged because of our studies: every student was coached on something of similar relevance to their studies. Of the 81 such users selected, one was User E from the first experiment.

These 81 users were also mailed information on how to start Xum. This was the only assistance provided. Use was completely discretionary. Since the experimental group constituted only 23% (81 in 352) of the class, they worked in an environment where the majority of their peers did not see their own sam model.

Duration

The coaching experiment ran for 8 weeks starting at the beginning of August 1992. Our analysis here is limited to use of the Xum and qv from that time to the end of the year. In practice this meant until student use of these accounts stopped at the end of October. The first log starts on Tue Aug 4 11992 and the last ends on Tue Oct 27 1992.

9.2.2 Results of analysis of monitor data

The data available from this experiment is the logs from Xum and qv. We performed two main forms of analysis. First we summarised the overall usage profile for all Xum and qv users. This was to explore Subgoals 1 and 2 which assessed whether the users tried the scrutability support tools at all and, assuming some users did so, whether they made non-trivial use of them.

Then we performed a detailed analysis on the subset of the data for students who made heaviest use of the Xum. These were the users who were likely candidates for Subgoal 3, assessing whether user activity indicated scrutiny of the various aspects of the model.

Summary results for overall use of Xum

The 81 students in the experimental study were sent mail about the Xum. Of these, 32 students (40%) started it at least once in the eight weeks of monitoring. In addition, 14 other students made some use of Xum! Since these students were not sent mail about Xum, they must have learnt about it by some other means, most likely from students in the experimental group. As mentioned in Chapter 8, there were 352 students in the class. Table 9.6 summarizes the actions of the 46 who used Xum. The full set of data for each user is in Appendix D.

Table 9.6. Summary use of Xum

Description of usage	**Average per user** N = 46
sessions	3.0
navigate around model	30.5
select explanations	8.2
select evidence list	9.2
change value	5.0
total	55.8

The first row is the number of sessions, or invocations of Xum. As one might expect with an average number of sessions being 3, there were many users who only used Xum once. The distribution of the data is shown in Figure 9.1. The x-axis ranks the users according to the number of session with Xum. So, for example the first users had the smallest number of sessions, in fact just one. The right most points correspond to users who had the most sessions. Eighteen users (39%) started it only once. Of these three did no actions within Xum. Clearly these users are part of the population envisaged in Subgoal 2

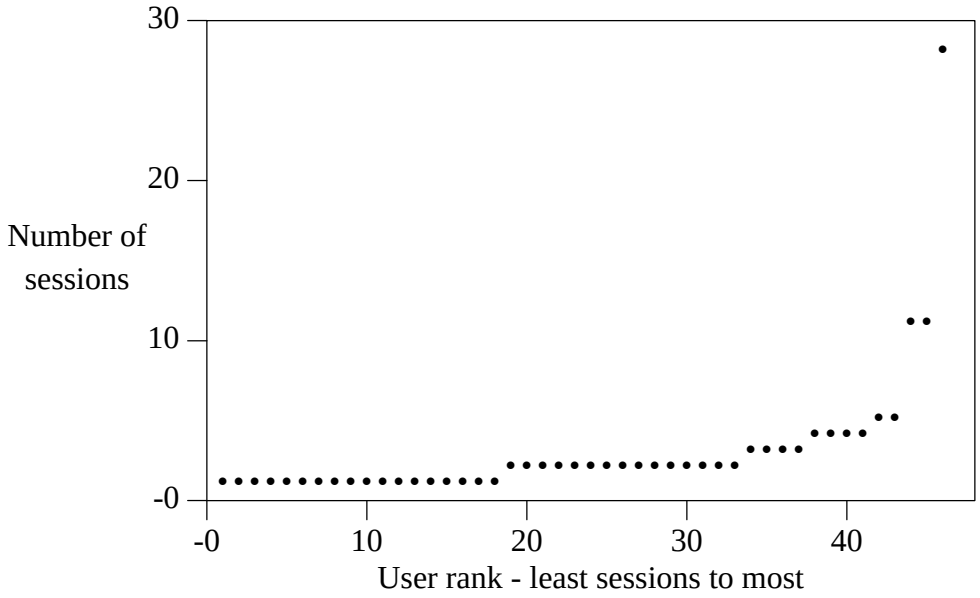


Figure 9.1. Number of Xum sessions

where the users did start Xum but then made no significant use of it. On the other hand, the majority of users who ever started Xum did return to it at least once more. Thirteen users (28%) had at least three sessions and one user had 28 sessions with Xum.
The second row of Table 9.6 records the number of Xum commands involved in moving around the model, expanding and contracting nodes to navigate through it. These constitute over half the total commands monitored. Figure 9.2 shows the distribution of the navigation actions. With the exception of the three users who simply started Xum, 8 more users did less than 10 navigation actions. There were 8 users (17%) who took at least 40 navigation steps. The heaviest user performed 188 navigation actions.

Next we consider the use of the explanation facility, with the average at 8 and the distribution shown in Figure 9.3. There were 8 users who selected none and 7 users only selected one. At the other extreme, one user selected 46. Selecting explanations enables the user to determine what the model is intended to represent. This may be part of the user's scrutinising the model or it might be a way to learn more about sam. Figure 9.3 shows that only 12 users sought at least 10 explanations and 19 users selected at least 5. so, it seems that these users typically made many navigation steps compared with the number of explanations explored.

This difference might have been due to relative lack of interest in the explanations or the limited need for them. There is support for both these hypotheses. Firstly, the aspects which interested the user can be partly gauged by the parts of the model whose

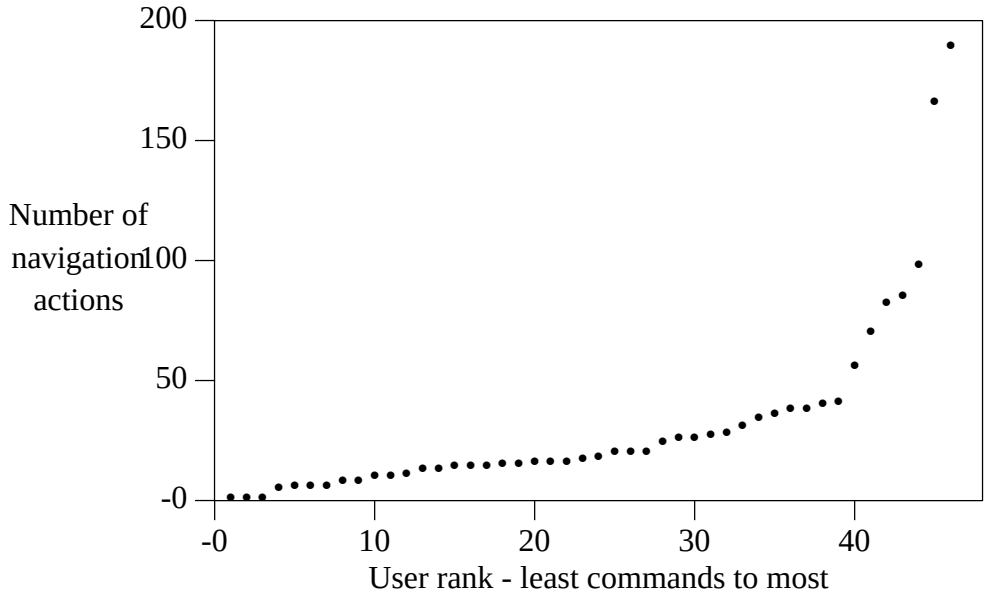


Figure 9.2. Distribution of navigation actions

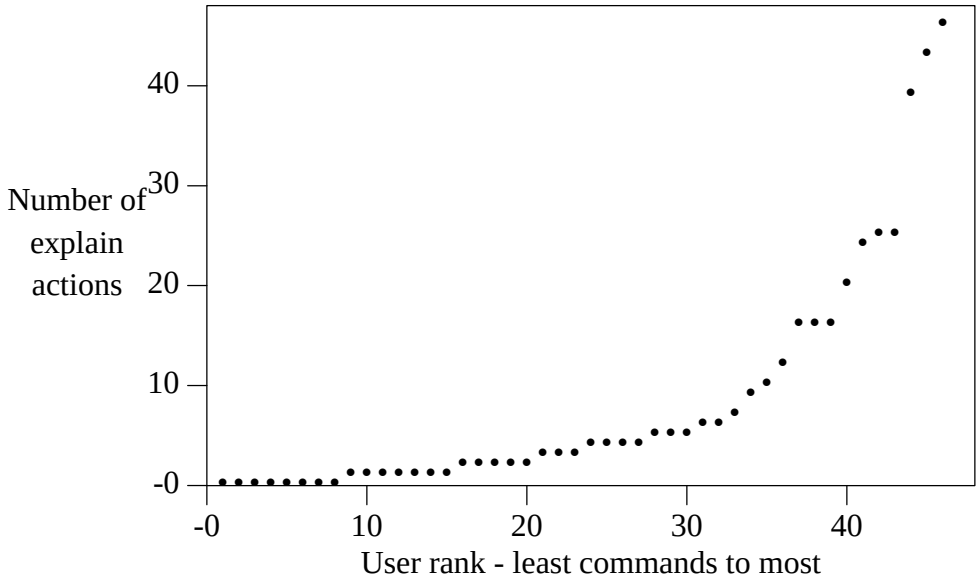


Figure 9.3. Distribution of explanations selected

explanations were most often selected. Analysis of this aspect confirms that this population has limited interest in sam.

We analysed the total number of selections of each component explanation. The most common explanations sought were for the typing speed aspects modelled, with wpm_info

selected 42 times and typing_ok_c selected 36 times. The next most popular commands related to programming languages and other text editors (pascal_c selected 14 times, c_c selected 15 times, and uses_vi_c 17 times and somewhat lower, uses_emacs_c was selected 5 times). The other two programming language components fortran_c and lisp_c were selected 7 and 8 times respectively. There were 8 aspects of sam which were selected at least 5 times: default_size_k 9 times; go_k 8 times; quit_b 8 times; nonscroll_b 8 times; minimal/write_k 7 times; non_cmd_b 6 times; open_window_k 6 times; write_k 6 times; scroll2_k 6 times; scroll13_k 5 times; subst_k 5 times; highlight_esc_k 5 times; set_fname_k 5 times; read_txt_k 5 times; and comma_k 5 times. It seems significant that the only three belief components in the model were high on the list of aspects of the sam model. It is likely that a belief is a less intuitively obvious part of the model. The default_size_k is an example of a concept which normally is not named by users. It represents the action of clicking to create the default sized window (rather than sweeping out a window of a different size). Similarly go_k refers to the action of clicking the mouse at a point in a window to move the current position of editing to that point. The other commands listed here include the more sophisticated ones which most of these users did not know. From these patterns of explanation selections, it seems that users are very interested in aspects like their modelled typing speed and their use of programming languages. Beyond that, the users have explored components which were probably unfamiliar to them.

Now, we consider the other reason for limited use of explanation: the meaning of the components may have been clear without the need for an explicit explanation. Note that each component's name was presented as part of the Xum display and the longer string for it was also available without the full explanation. For example here are some examples of component names and the corresponding descriptions:

Component name	**Description available at** Xum **interface**
scroll2_k	How to use button 2 on the scroll bar correctly
quit_k	How to use the Quit (q) command correctly
quit_b	That killing sam's window is usually the best way to exit sam
exch_k	How to use <exch> on the menu for mouse button 2

For a sam user who knows these aspects, these are quite self-explanatory. Many are also adequate for a user who does not know the aspects but can see, for example, that the button-2 menu in the command window does indeed have an entry labelled exch. One would expect that users would not need to read the longer command explanations unless they wanted to check it, perhaps confirming their understanding of what the component means. For example, if the user thinks they know an aspect but find that the model shows it as unknown, they might well check the full explanation to see if they had

misunderstood what the command really meant. In the case of components the user does not know, the component explanations might to be used as a means of learning about new aspects of sam or deciding whether aspects seem worth learning.

In Figure 9.4, we show the use of explanations against navigation actions. The spread of the data points indicates that some users explored their models extensively, seeing the structure and the values of many components. Others examined many explanations but did not navigate around the model as much. Essentially, the spread of points reflects the different ways that the more active users used Xum.

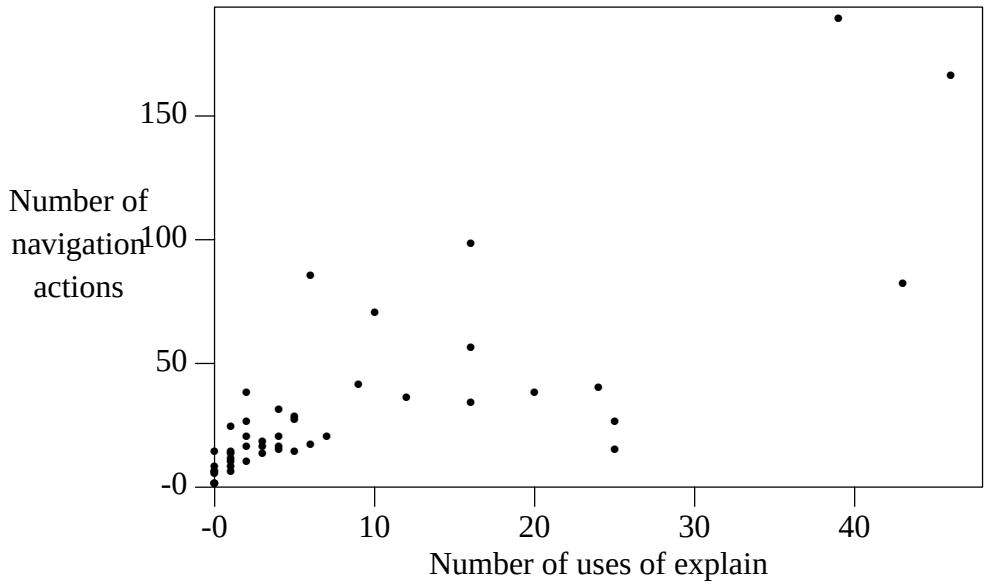


Figure 9.4. Navigation actions against explanations

The next aspect shown in Table 9.6 is the selection of evidence lists. This is a strong indication the user is scrutinising the user model. The average number of such actions was 9, slightly higher than the number of explanations for component meaning. The distribution is shown in Figure 9.5. There were 10 users who selected none and another 4 users selected only 1. At the other extreme, the most active user selected evidence lists 63 times.

The components for which evidence lists were selected most often show a similar pattern of interest to the explanations, with the most popular relating to typing speed (wpm_info selected 62 times and typing_ok_c 59), followed by programming languages and other editors. The sam aspects which were selected at least 5 times were: scroll2_k 7 times; go_k 6 times; quit_b 5 times; and paste_k 5 times.

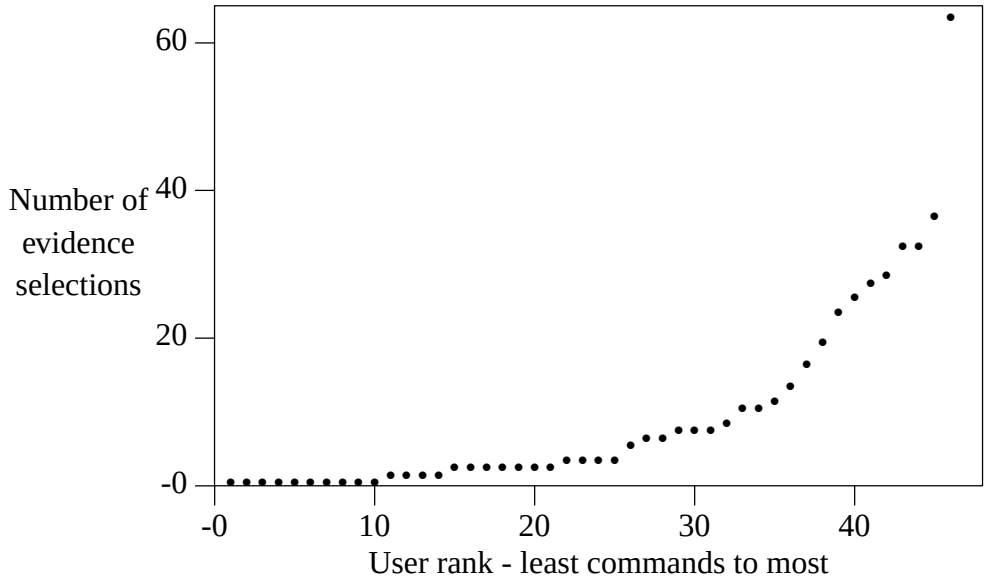


Figure 9.5. Distribution of evidence selections

In conjunction with the evidence lists, there were 53 selections of the explanations for the parts of the evidence. The most common was for told (19 times) and given (16 times). Often the user would examine the evidence for a component, then alter the value and the review the evidence list and the explanation for the given, the type of the new evidence that would have been added.

The last facility available enabled the user to alter their model. As Table 9.6 indicates, the average user tried this 5 times. The distribution is shown in Figure 9.6. There were 22 users who made no changes at all and the maximum number of changes made by a single user was 38.

It remains to summarise the total activity of users. Figure 9.7 shows the number of actions each user made in Xum.

As one would expect, the users who had a large number of sessions tend to appear towards the right of the graph as they performed larger numbers of Xum actions. These graphs suggest that many of these users did make perfunctory use of Xum but at the same time, there is a group of users who made quite extensive use of it.

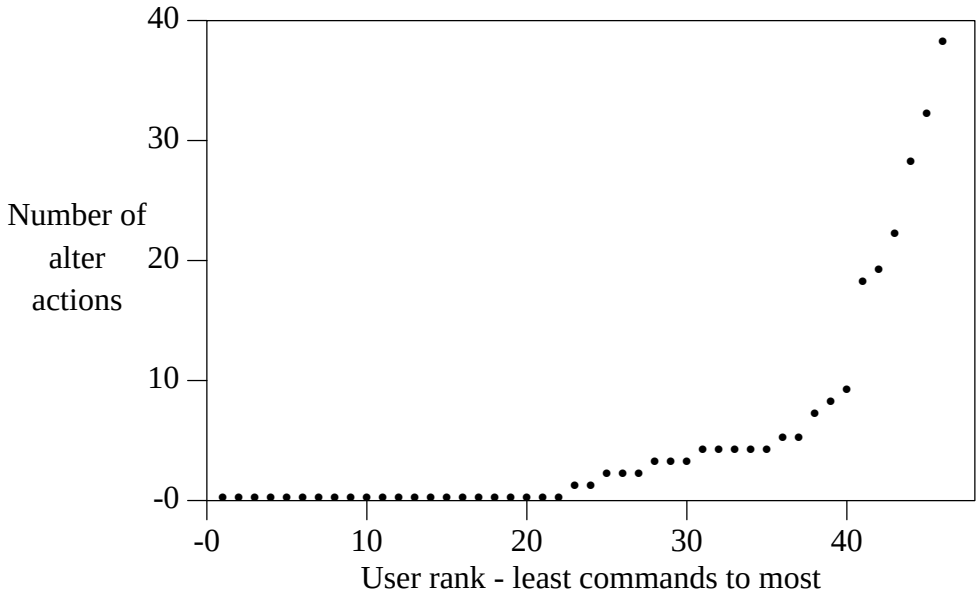


Figure 9.6. Distribution of actions to alter a component value

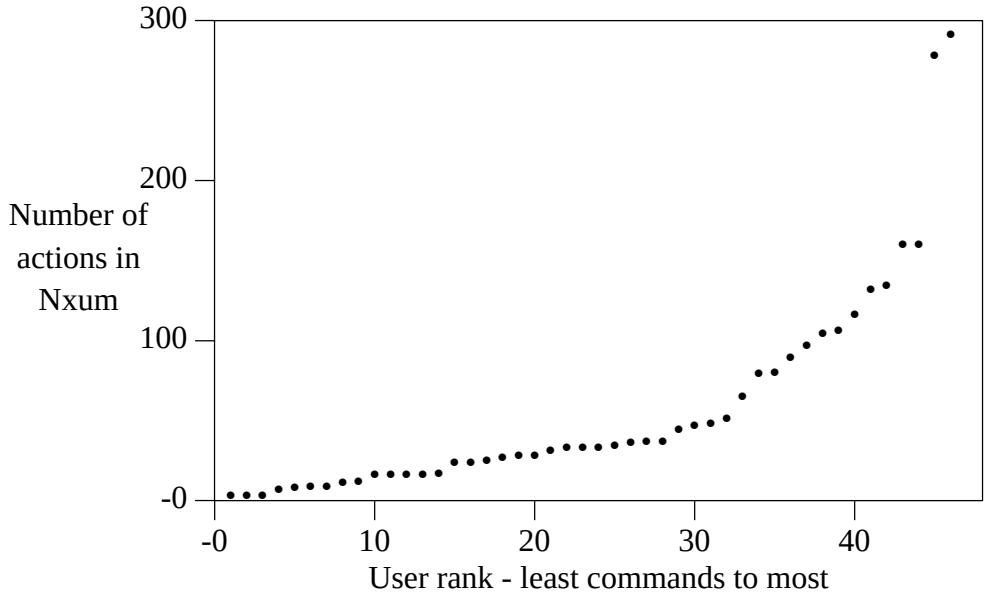


Figure 9.7. Number of actions within Xum

Summary results for use of Xum by heavier users

Our design for this experiment distinguished three classes of users: those who never even started the scrutiny support tools; those who did so but made perfunctory use of them; and thirdly, those who spent some time time scrutinising their models. It is this third

population that we now consider.

As Table 9.6 shows, the average number of Xum commands was 56. There are fourteen users above this level. These users seem to be candidates for classification as non-perfunctory users of Xum. Of these users, 11 were from the sample group of students who were mailed about Xum; the other 3 were not.

Table 9.7. Summary counts for use of Xum for heavier users

Description of usage	**Average per user** N = 14
sessions	5
navigate around model	68
select explanations	21
select evidence list	23
change value	12

We now study their activity with Xum: their average use is summarised in Table 9.7. As one would expect these values are all higher than those for all users. In fact the various actions all have averages between 2.3 and 2.7 times those for the whole group. So the relative number of uses of each of these classes of commands was similar for this population and the whole population.

We now examine the activities of each of these users in more detail. We allocated identifiers to each of the users. The allocation is random so there is no systematic way to relate the identifier to the actual users. The user identifiers are the letters A to P, with I and O omitted.† Table 9.8 shows the command use for each of these 14 users. † The values marked with an asterisk indicate cases where that user made more than double the average number of uses for that type of action compared with the full population of 46 people who used the viewer. The double asterisk is the maximum value in a column.

We analysed the logs for these fourteen users, assessing just which aspects of the user model they explored and changed. The full notes appear in Appendix E. The overall counts of the xum command use gives an indication that most of these users were making repeated use of most of the facilities described. An example of part of a user's log is shown in Appendix F. This has been edited to remove any indications of the user's

---

† User E from Experiment 1 appears as User J in Experiment 2. Users F, H and M were the three users who were not part of the experimental group: they were not sent the mail inviting them to explore their sam model using Xum.

Table 9.8. Summary counts for use of Xum

user id	sessions	navigate	explain component	evidence list	evidence explain	alter value
A	5	69*	10	8	0	38**
B	1	14	25*	36*	10	2
C	2	25	25*	3	0	32*
D	4	55	16*	23	0	4
E	11*	165*	46**	32*	0	22*
F	1	39	24*	63**	36**	5
G	2	33	16*	25*	12	19*
H	11*	35	12	19*	0	0
J	2	37	20*	27*	11	28*
K	28**	188**	39*	16	0	18*
L	2	81*	43*	28*	2	4
M	4	40	9	10	0	0
N	3	84	6	11	2	0
P	5	97*	16*	32*	6	8

identity. A summary is in Table 9.9.

All these users made non-trivial use of xum and most made some use of qv. A number of different patterns is discernible from analysis of the logs. We can characterise the behaviours of the users into a small number of interesting categories. These are shown in Table 9.10.

Given our concern for scrutability, one of the most interesting behaviours relates to users who do many explorations of evidence lists. This was the case for users B, F, G, J, L and P. In addition, users B, H and M seemed to explore the evidence in conjunction with the facility to change the value. So they would change values then look at the evidence list. This is consistent with the sort of scrutiny of the model that we intended in the design of um.

Users D, G, H, J, M and P explored non-sam aspects of the user model in detail. These are the components for the user's typing speed and use of programming languages. This behaviour is consistent with users having little interest in sam.

One behaviour we had expected was exhibited only by user A who set large parts of the model to true, to indicate they knew many aspects of sam. A slightly different behaviour was exhibited by user C who took considerable care in setting a small number of values in the model, them resetting them. These suggests the user experimented with the model but then set it back to its original state.

A particularly interesting behaviour was exhibited by users F, E, K and L who seemed to be exploring the explanations for a number of concepts that were at the boundary of their

Table 9.9 Summary of user activity

User Id	Summary of actions with viewer
A	Sets large parts of the model to true - seems to be playing. Uses qv once.
B	Explores variety of more basic commands with ‘Explain’ and ‘Justify’ pairs, makes change then ‘Explain-evidence’ (total 10 times). Uses qv once.
C	Made many changes to model, 6 components being set more than once. With just 3 uses of ‘Justify’, this user seems to be taking care in setting, thinking and resetting the model. Uses qv once.
D	Mainly looks at non-sam aspects, with only about one third of the ‘Explain’ uses for sam aspects, with dominant actions being the paired ‘Explain’ and ‘Justify’. Used qv twice.
E	This student seems to be working to get the model right. The student appears to be looking most at the aspects likely to be useful to learn and they may well have been using the viewer as a learning tool. Used qv three times.
F	User explores many aspects likely to be useful to learn and explore the various xum facilities. Did not use qv.
G	Explored the xum aspects and focused on non-sam aspects. Used qv three times.
H	Explores the typing aspects with ‘Explain’ and ‘Justify’. Used qv three times.
J	Explored the viewer but had been involved in earlier experiment, mainly explored non-sam aspects and explored ‘Explain-evidence’ eleven times in tandem with ‘Explain’, ‘Justify’ and ‘Change’. Used qv twice.
K	Many uses of ‘Explain’ in areas of sam useful to learn and appears to be using this as an aid to learning). Used qv seven times.
L	This user explores more ‘Explain’ compared to other commands and the choice command-explanations explored is consistent with use for learning. Did not use qv.
M	Makes small use of xum, mainly looking at non-sam aspects, mainly typing speed aspects, exploring the ‘Explain’ and ‘Justify’ Used qv four times.
N	Has explored various parts of the model, moving around it and checking ‘Explain’ and ‘Justify’. Used qv once.
P	Made use of all xum facilities but focused on non-sam aspects. Used qv twice.

knowledge. This seems consistent with using xum as a tool to learn about sam, or even to learn about what they might learn about sam (this latter being a way to decide on learning goals as distinct from actually trying to learn). This is indicative of an interesting and important potential role for the scrutable user model. Since it is not

Table 9.10 Interesting categories of user behaviour

Characterisation of activity	Interpretation of activity	Users
Evidence in association with component explanations	Exploring the values in the model	B, F, G, J, L, P
Navigation and altering component values	Essentially focus on values of the model	A, C
As above but with evidence examined in association with changes to model	Explore evidence in association with altering the model	B, H, M
Explore mainly non-sam aspects	Mainly interested in non-sam aspects of model	D, G, H, J, M, P
Exploring component explanations at knowledge boundary	May be using model as a learning tool	E, F, K, L

central to our current concern for design of scrutable user models, we will not explore it further here.

Summary

We now examines the questions associated with the three goals of this experiment.

Subgoal 1: Will users scrutinise their models at all?

The experiment gives a clear affirmative answer to this question. In total 46 students started xum at least once. of these, 32 were from the sample of 81 who received mail informing them about xum. This represents 40% of those sent mail. It also indicates an interesting social effect: some users who learnt about xum tried it. This indicates a significant level of interest in the model.

The only information we gave these students was mail stating how to start xum. We have already explained that the users in this experiment had as their primary focus the learning goals of their Computer Science course: this was several levels away from the user model that drove the coach that taught about sam, one of several text editors available for creating source code files. Also the students had a substantial work load. This would have discouraged students from leisurely exploration of sam facilities for some long term goal of more effective editing ability. Users under pressure have less inclination to learn about tools they use (Carroll, 1990). Moreover, we know that many students did not like sam. Even more than this, some of the 81 users may not have even read their mail. At the time of this experiment, we did not use electronic mail for any of the formal communication with students and many students did not use it heavily. We did not track

the number of people who read the mail.

On the other hand, there were factors favouring our experiment. First is Hawthorn Effect (Bullock and Stallybrass, 1977) where our observations affected the outcome. The students knew we were very interested in their learning sam. For example, some made comments indicating pleasure at our interest in them and their learning. It is to be expected that this may have influenced them to try the xum. In addition, one of the 14 heavy users (User J) was part of the interview group of Experiment 1 (User E). This observational effect would probably have been significant for that user.

Subgoal 2: Of those who do scrutinise their models, will they do so more than perfunctorily?

The experiment also gives a clear affirmative answer to this question. Although many users made only minimal use of xum, those in our sample of 14 heavier users made substantial use of it. The 11 heavy users within the experimental group constituted 14% of the 81 who were told about the xum.

Subgoal 3: Of those who scrutinise their model more than perfunctorily, will some explore the the full range of facilities?

Each heavy user made use of each of the xum facilities. Although the navigation commands are the largest group, the next largest is the request for justifications and this was followed by requests for explanations.

In fact, most users used most facilities. There were only two classes of scrutability support that were not used by every user in this group. There were three users who did not use the *change value*. Since this is on the same menu as the *explain* and *justify* commands that they did use, it seems likely that these users did not want to alter their model.

The other facility that was not used by all was the explanation of evidence. As can be seen from Table 9.8, exactly half the heavy users tried this facility and of these, two users used it just twice. Use correlates strongly with the number of times that users explored evidence lists: most users who made relatively small use of evidence lists were those who made little or no use of the evidence explanation facility. Also some of the modes of behaviour were not consistent with exploring evidence explanations. For example, the users who appeared most interested in exploring the meaning of the model made extensive use of navigation and component explanations.

Looked at differently, scrutiny of the evidence explanations constitutes extensive

exploration of the user model and its foundations. Four users examined ten or more such evidence explanations. This is 28% of the heavy user population, 9% of those who made any use of the editor.

9.3 Discussion

This chapter described two experiments for evaluating the scrutability of um model. The first was formative. It was promising since the users did indeed scrutinise their models in order to assess their accuracy. Their scrutiny went beyond the simple values of the components of the model. They explored the evidence for components and explanations for components. The users were politely positive about the viewer as a basis for exploring the machine's model of their knowledge. At the same time, the first experiment served to inform the design of later generations of scrutability support tools. It also informed the design of the monitoring for those scrutability support tools.

Another important outcome from the first experiment was that the model constructed for that experiment were deemed accurate by the users. This gave us confidence that the processes used to create the models should be reliable for a more exacting field trial run over a semester later, with additional monitor data for the model construction.

The second experiment was summative, assessing whether users would scrutinise their user models in their normal working environment with the usual pressures of deadlines. The experiment involves several aspects:

- the effectiveness of the design for scrutability in the user model;
- the usability and effectiveness of the scrutability support tools;
- the quality of the user model built;
- the quality of the explanations for the components and evidence.

The first pair of these have been discussed extensively in the descriptions of the design of um in Chapter 6 and its underpinning in Chapters 3 to 5. The others deserve clarification.

Consider the quality of the sam user model built. One could not expect users to spend time on a poor model. At the extreme, a very poor model would not make any sense. For example, if it were completely inaccurate, the user would be more likely to dismiss it. If the structure of the model and the components chosen for representation were poor, the users would have had difficulty relating these to the domain. Here too, the user would be more likely to dismiss it. The point is that we evaluated just one, particular instance of a user model created within the um framework. The evaluation combines assessment of both the scrutability support in the um framework and the quality of the particular model

implemented for participants in the sam experiments.

The explanations within the scrutable user model are clearly critical to its scrutability. We noted that in the first experiment, the explanations were found wanting for the particular level of user involved in the experiment.

The critical point we make here is that although this evaluation is intended to assess the general um framework, it inherently involves many other details. Any one could have contributed to failure in the experimental use of the scrutability user model. So, for example, had we found that no users scrutinised their user models, it might have been due to any of these factors as well as others like user motivation to learn about a text editor that was not generally liked. Equally, had we found that users did start scrutinising their model but make only perfunctory use of the tools, this might have been due to a range of factors including um. The fact that users did scrutinise their models and that some users did so repeatedly indicates adequacy in all the elements.

Limitations of the evaluation

The conclusions we might draw from this study are limited by the fact that it involved just one domain, sam with the samcoach and that the user models created for the experiment are largely based upon one class of user modelling information, the sam monitor data, interpreted by analysis tools.

The use of a field trial also has limitations in that there may be many factors that affected the outcomes and we may not be aware of them. This is an inherent disadvantage of a field trial but, as our goal is to provide scrutable user models, an experimental evaluation creates artificial constraints which make the outcomes less meaningful.

The timescales of the experiment must also be noted. We constructed the initial user models on the basis monitor logs which had been constructed from many months system use. To build models starting from the very first use of sam, different techniques would be applicable. Similarly, the um-consumer was a coach which ran for just eight weeks. Longer term or different um-consumers are likely to create different effects.

Another important limitation of the experiment is that it involved only computer science students. They are clearly not representative of the broader public. One would hope that they had a greater interest in understanding computing tools as well as greater confidence in their ability to scrutinise a user model.

9.4 Conclusions

Experiment 2's major strengths are its scale and the use of um in a significant field trial. The experiment's also evaluated use of the scrutability support tools in an authentic and demanding environment.

At the pure user modelling level, it demonstrates that um can support significant experimental use. The um tools were able to manage user models for the full class of 352 students. The models constructed were interesting and significant. They represented the main elements of sam, a useful text editing tool with a quite rich and powerful set of operations.

Supporting basic user modelling is the sine qua non for a user modelling shell. However, our additional requirements on um concern scrutability. The experimental work indicates that users could scrutinise their models. It further indicates that users in a stressed working environment did use the scrutability support tools to explore their models.

To appreciate the extent of user interest, it is important to see that sam was just one of many elements of these user's computer environments. This means that our experiment parallels the operation many of the um-consumers for which um is intended to support the user-adapted interaction. As mentioned in the description of the experimental design, the um user model is well removed from our user's primary goal. This is illustrated in Figure 9.8. At the top, we show the user's primary goals, in this case related to the course goals. Of course, different students will have perceived those goals differently. For example, most would have had at least some concern for the examination requirements, as indicated in the figure. At the same time, the overall learning outcomes of the course were intended to involve acquisition of skills in the effective use of various programmers tools, such as the examples shown in the figure.

In order to achieve their primary goals, the students needed to create source code files. They could use any of the many tools available for this. Many students used various editing tools for different circumstances. Most used sam at least in some situations because it was the editor that was explicitly taught in the first year class that almost all the users had taken. As we know, many students did not like sam. Even for those who did like it, sam was is subsidiary concern to their primary focus on achieving course goals.

One level further from the user's central concerns was the samcoach. It sent electronic mail advice to students in the experimental group. We believe that many students did not read their electronic mail. In addition, we know that many students failed to appreciate that they might benefit from taking the time to learn to use sam more effectively. This means that the coach is a um-consumer which has value for the users but that value is well removed from their primary goals.

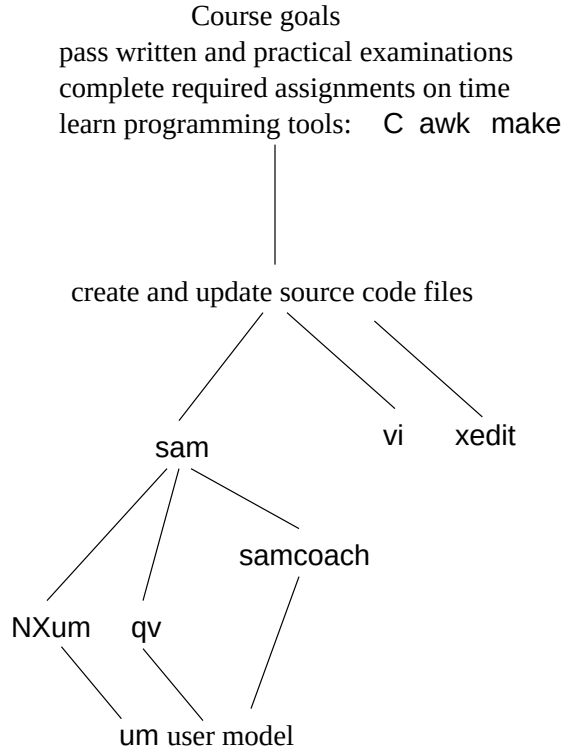


Figure 9.8. Relationship between um model, samcoach, sam and users' central goals

We need to appreciate this context when we observe the level of interest that users appeared to show in their model. Of users who were informed about xum, 40% started it at least once. In addition, word of the user model and scrutability tools spread to so that 14 other students tried it.

Analysis of the actual use of xum indicates that the users explored the full range of facilities, selecting explanations, evidence list justifications for the value of components, altering component values and navigating around the model.

Overall, Experiment 2 was a tough evaluation of the user modelling support as well as the scrutability of the sam user model, supported by the scrutability support tools xum and qv. The effectiveness of the toolkit software is indicated by the effective management of a large user population, hundreds of users, over a significant time period, two months. We have described the results of our analyses of xum logs. These indicate that a significant minority of the potential users scrutinised their user models in considerable detail. Overall, the experiment indicates that um and its associated scrutability tools provided users with user models which they could scrutinise.

Chapter 10

Conclusions

The design and development of um can be seen as part of growing recognition of the potential value of giving the user access to their user model. We now explore the context of um's design in terms of both the related trends towards inspectable user models and by explaining the context and history of um's evolution. These serve to place in context the work we have described in this these.

Some examples of the work which reflects an appreciation of the importance of inspectable user models includes the following:

- Probably the first paper explicitly recognising the potential benefits of making the user model available to the student user is (Self, 1988).
- In Chapter 2, we noted that UMT (Brajnik, Tasso, and Vaccher, 1990, Brajnik and Tasso, 1992, 1994) displays parts of the user model used to select advice for tourists.

Strachan's system (1997) mentioned in Chapter 3 also makes its user model available to the user. There appear to be an increasing number of such systems.

- Murphy and McTear (1997) describe a system in which the user has access to the system's estimate of the learner's proficiency in the domain and they can alter this.
- Brusilovsky and Schwarz (Brusilovsky and Schwarz, 1997) describe a teaching system which makes the user model available. They also note that the interface to the user model requires further work as some users found it hard to find and understand.
- An adapted hypermedia system (Hook et al, 1996) refers to a 'glass box' approach which enables users to inspect the adaptivity mechanisms and control their effects.
- Csinger (Csinger, 1995) makes the user model available to the user in order to make his system scrutable.
- Paiva, Self and Hartley (Paiva, Self, and Hartley, 1995) argue for making the user model available to learners.
- An even more central role for the user model is taken in PHelpS (Collins et al, 1997), a system which helps people find others who can help them with tasks.
- And finally, Bull has done extensive work based upon the importance of inspectable student models. She has developed and evaluated several systems which make their student model available to the learner (Bull, 1997, Bull and Brna, 1997, Bull and Broady, 1997, Pain, Bull, and Brna, 1996, Bull and Pain, 1995, Bull and Smith, 1995, Bull, Brna, and Pain, 1995, Bull, Pain, and Brna, 1993) and these played a pivotal role in supporting the learning. The accessibility of the student model is central to Bull's work.

In systems like these, we see a growing appreciation of the need to make the user model inspectable.

Scrutability extends this notion of inspectability and access to the user model. It enables the user to see the model, to find the meaning of the elements of the model and the processes that contributed to the model. The task of um is to provide a representational framework as well as tools for managing scrutable user models and, at the same, supporting reuse of the user modelling information and the um tools across different um-consumers.

Our work began with a concept mapping interface designed to elicit the user's knowledge of a domain (Kay, 1990, Kay, 1991, Kay, 1986). This interface was based on the philosophy that the elicitation was a co-operative enterprise between the user and the system, with the user in control at all times. The system provided a helpful interface and

framework. That work was restricted to elicitation of knowledge. Moreover, it was intended particularly for eliciting the user's *conceptual* understanding. Otherwise, it was domain independent. Since the concept mapping interface was intended as a source of user modelling information, we wanted to maintain the same philosophy of co-operative user modelling in the earliest versions of um (Kay, 1990). This concern about the relationship between the user and the machine which provided the initial motivation for a scrutable user modelling shell toolkit.

An important influence on the design of um's scrutability support was the potential roles that the user model might serve in encouraging learner's metacognitive development and assist the learner and teacher in achieving shared learning goals (Crawford and Kay, 1992, Crawford and Kay, 1993). This was one prong of the large scale experiments involving study of users of the sam text editor. Another important aspect of that work was to explore the role of huge amounts of extremely low quality monitor data for building high fidelity models of the user (Benyon, Kay, and Thomas, 1992). This work demanded careful design of the monitoring (Cook, Kay, Ryan, and Thomas, 1995) to ensure smooth monitoring of 2273 users for at least a year, 880 for at least two years and 316 for the full three years 1991 to 1993. This led to an improved understanding of the problems of nurturing learning of a tool over the long term (Kay and Thomas, 1995) as well as the characteristics of long term system use (Thomas, 1998, Kay and Thomas, 1996). During this period, the team in the sam project conducted several experiments involving simple coaches (Gheibi and Kay, 1993, Butler, 1992).

A parallel development was a series of experiments in modelling user's preferences for movies so that an advisor system could make recommendations to the user (Kay, 1995). Although this work was never evaluated on the scale of the sam experiments, it was important. It provided an opportunity to explore the very different demands of a domain in which the user's preferences were the focus. In addition, this domain has a huge number of movies. This domain needs to model user preferences for these as well as the large number of attributes of movies. The modelling and scrutability support issues for this scale of task are quite different from those for a relatively small domain of learning where there are fewer components and more structure.

In addition to the focus on the two domains, sam and movies, there was a continuing exploration of approaches to supporting the user in scrutinising their model. From the very first (Kay, 1990) um models were designed as ASCII files to be kept in the user's filespace so they could traverse the various partial models to see the details of their user model. The concept mapping interface was the first tool which provided external evidence; its evidence included the details of the user actions, stating the concepts and links from the user's concept map. The issues involved in viewing and visualising user

models were explored over a period, starting with a simple text-based tool for exploring a um model (Cook, 1991) in the sam domain. These were also applied, with some adaptation, to the movies domain. On the basis of this experience, we experimented with a series of user model viewers (Cook and Kay, 1993, Cook and Kay, 1994).

The evolution of um has continued with modelling user's knowledge of the C programming language as a basis for customising a hypertext (Kay and Kummerfeld, 1994) and modelling user's interests and preferences for music (Kay and Kummerfeld, 1997).

10.1 Further work and limitations

The directions for um are divided between those which might support more effective or powerful modelling and those which will improve and extend the scrutability support.

10.1.1 Evaluations

The foundations for future evolution of um should be based upon more extensive evaluations. Certainly, the sam work has involved large numbers of users but future work should include a broader range of domains and modelling needs. The design of um has led to a very simple representation. It would interesting to explore the use of um in further domains, with varied user modelling demands and from this to assess its adequacy in these. Such studies would improve our understanding of the need for more complex mechanisms for representing user models and reasoning about them. Of course evaluations must assess both the efficacy of the user modelling and the scrutability support.

10.1.2 Extensions to the representation

The current um representation has been carefully designed to be simple. Many extensions need to be explored. For example, um is primarily based on boolean components, with the addition of values to represent no-information and unresolvable conflicting evidence. Future work should include other types which might more naturally represent components in some situations. For example, fuzzy values or even simple enumerated values seem promising.

Another interesting direction is generative user models. Some shells can compute new components and their values by making inferences within the model. We saw this in the case for TAGUS and BGP-MS. Many other systems treat the user model as a database of model components. The database approach has the merit of simplicity and directness

between the actual model representation. This would seem to be an important issue for scrutability. At the same time, generated models have the potential to be very powerful. For example, if the domain modelled is well suited to such inference, as in the case of artificial sciences like mathematics, we can build a quite small but powerful user model. This could be argued to support scrutability since a small model should be easier for a user to understand than a large one. Another potential benefit of generative models is simulation of the student's reasoning. This is especially important in cognitively valid user models intended to model the actual thought processes of the learner. On the other hand, the understandability of such models is severely hampered by the fact that one must appreciate the full power of possible inference to understand the model. The development of scrutable and understanding generative models is a very interesting direction for future work.

10.1.3 Additional user modelling tools

It is in the nature of um that we expect to see many more tools developed. For example, there are likely to be new resolvers specifically built for many um-consumers. Equally, new tools should provide external information and internal reasoning. In addition to more of these tool types we have described in the um architecture, we will explore the role of additional classes of tools. For example, an alert tool, available in BGP-MS could send messages to um-consumers when certain situations arise. The current um architecture would make it natural to implement this as a um-consumer. Nonetheless, as we explore new domain which require such tools, we will evaluate whether they should be implemented as part of the um architecture or whether a new class of tools is justified. There are many possible tools which might be considered, for example those performatives described by Paiva (1996, and Kobsa (1996).

Another interesting possibility is to create a communication channel between um models and those managed by other shells. Then each shell could support the type of modelling it is best able to support. The support of scrutability would need to be explored.

10.1.4 Visualisation support

Large user models involve inherently different problems from small ones for scrutability. We have already observed that modelling user preferences for movies involves a large number of similar components. For example, there are hundreds of actors and our prototype placed them all in the same partial model. We could well structure them in ways that the current um approach supports. For example, they could be grouped by the year they appeared in movies, with a special category for long-lived actors. Many other

structures are equally feasible. It seems likely that a fruitful direction for research into understandable models of this type should be built from the various approaches to visualisation.

10.1.5 Views

The current implementation of um represents just one conceptualisation of a domain. For example, the sam part of the models we showed through Chapter 7 were organised in terms of the utility of each aspect of sam for typical editing tasks. Many other organisations are possible.

One important direction for um is to support a multitude of *views* which operate at the context level. Any one view should select components and structure them as appropriate for it. For example, in the sam context, suppose the um-consumer teaches about sam in terms of editors the user knows. Then, we could usefully construct one view of sam aspects for a user who knows vi and a quite different one for someone who knows Word. There are many ways to structure such models but one natural part of the structuring would distinguish the sam aspects that map closely to the user's familiar editor from the others.

A logical extension of adding alternate views of the structure of a user model is to allow the user to define the structure themself. This could be supported with an interface like the concept mapping tool (Kay, 1986, Kay, 1990, Kay, 1991).

10.1.6 Scrutability of um-consumers

This thesis has focused on the scrutability of the user model and user modelling processes. It might be easy to loose sight of the fact that most user models exist as adjuncts to a um-consumer. We need to explore support of scrutability in consumers. This links directly to scrutability of the user model since part of the explanation of a um-consumer's operation is due to the user model. However, it is important to explore support of um-consumer scrutability beyond this. The role of the user model scrutiny support may turn out to be even more significant than it first seems. For example, if the model is mainly intended for a particular consumer, it may be reasonable for the user model explanations to give some hint of the real impact of the model for the consumer in the customised explanation. For example, suppose the model represents the user's expertise in the Pascal programming language. If this is used to drive the customisation of a hypertext for teaching other programming languages, it may be reasonable for the explanation subsystem to be able to inform the user that when this component is true, it may be used to present information in relation the user's assumed Pascal knowledge.

10.2 Contributions

The contributions of this thesis correspond to the chapters in the body of the thesis.

Definition of requirements for a scrutable user modelling shell in Chapter 3, with one set of requirements on the modelling power and efficiency and another for the scrutability of the model.

umps **interaction model** described in Chapter 4, with three main actors in interactions where adaptivity is driven be a user model: the user, who will be modelled on the basis of $user_{shared}$; the programmer(s), whose beliefs about users, user modelling and the domain(s) are captured in $machine_{private}$; the machine at the time of the interaction, when $machine_{private}$ combines with effects of $user_{shared}$. The model highlights the points where there is potential for external user modelling information and the need to build support for scrutability: $user_{shared}$ is evidence about the user, given in user modelling dialogues or observations of the user in an application; $machine_{interpret}$ is the process of taking user actions and placing them in $machine_{private}$, a task for um tools that collect and interpret $user_{shared}$ before providing it to the user model: we require such programs to be identified and to provide explanations of their interpretations; $machine_{action-int}$ is the internal inference with our primary concern being due to um inference tools; $machine_{action-ext}$ is the aspects the user has been told by the machine, another source of external modelling information, provided by an application or um scrutability support tool.

Accretion representation described in Chapter 5, which provides a simple representation which serves scrutable user modelling. It is based on representing components with a list of evidence and it defines the basic operations on such lists. Accretion simply means the addition of a new piece of evidence to the list for a component, with a distinction between evidence from internal and external sources. Resolution is the process of interpreting the evidence list to conclude a value for the component. We distinguish between primitive resolvers which determine the value from the most reliable piece of evidence in the list and compound resolvers which may perform arbitrary calculations. Destruction is the process of removing unwanted information from the user model: for external evidence it is called compaction; for internal evidence, undo. That chapter described how accretion deals with many of the common and interesting problems in user modelling.

um **toolkit supporting reusable user models and reusable tools** described in Chapter 6 in terms of the the um architecture and its basic tools for collecting external evidence,

resolving the value of a component and the performing internal inferences based upon either stereotypes or rules.

Scrutiny support tools described in Chapter 7, support the user in scrutinising their user model and the user modelling processes that formed it. There are two types of tools. An overview tool enables the user to see the broad brush view of the user model. A detailed scrutability support tool enables the user to see and alter the details of the model. This gives access to explanations for the component meanings and enables the user to see the justification for the value of the component in terms of its evidence list. It can give explanations of each piece of evidence. And, it allows the user to alter the value.

Two consumer systems and their associated um tools were described in Chapter 8: one for the sam text editor and the other, the movies advisor. These were chosen to ensure that um should deal with two important but very different classes of user modelling application systems. The sam work constitutes an example of modelling:

- user's knowledge (and misconceptions);
- parts of the model were reused, being needed by two, independent coaching systems, one for unix and the other for sam;
- in a highly structured model with several levels of hierarchy; a real domain;
- and an authentic task (coaching) was supported.

The movies domain is an example of:

- representing the user's preferences and dislikes;
- with large numbers of modelling components (hundreds of actors, directors, descriptors ...);
- where there is little obvious or helpful structure (so that the partial model for actors is a single, large, flat collection of components);
- and it was a prototype system.

Evaluation of the scrutability described in Chapter 9 in terms of a large-scale field test where users were monitored in their exploration of their sam user models. This demonstrated that even under very unfavourable conditions, a significant number of individuals chose to explore their user models and of these, some scrutinised the full range of explanations, justifications and the possibility to alter component values. The experiment demonstrates the robustness of the user modelling software and indicates that the scrutability support interfaces enabled users to see the model and the information

about the processes that formed it.

The goal of this thesis was the design of a scrutable user modelling shell for supporting user-adapted interaction. This requires that the system be able to support the user's exploring their user model and the processes which formed it. Our goal has meant that we have analysed the elements of interactions involving user modelling and we have designed a representation for user modelling. Scrutability is not an add-on in um. It is fundamental to the design of um's architecture and the underlying accretion representation. At the same time, the design of um reflects the requirements for the functionality necessary for representation of reusable user models. The um architecture and its classes of tools reflect the same concern for a robust and effective user modelling shell.

In Chapter 1, we discussed the importance of scrutable and inspectable user models. This was in terms of several issues.

- A user model holds private information and so the user should have access to it and control over it. The user can explore um models, see their structure, the values of the um components, the evidence which defines the component values, and explanations for each element of the user model and the user modelling processes. The user can alter the user model, correcting inaccuracies or exploring the impact of changes.
- Programmer accountability is supported by a system which enables the user to explore the machine's processes which formed the user model. In um, this means the user can determine which evidence sources are responsible for conclusions in the user model.
- The inherent asymmetry of the human-machine relationship is reflected in the underlying principles for the design of um. The umps interaction model clarifies the nature of that asymmetry in user-adapted interactions.
- The potential of user models as an aid to reflective learning is an important and exciting role for scrutable user models. This role has informed the design of um and been an element of our evaluation.

The abstract framework of um are the accretion representation and um architecture. These define the general form of a um user modelling shell. The particular, concrete implementations of this abstract um have enabled us to evaluate um and to demonstrate that it can support interesting, useful and robust user modelling. Moreover, it has enabled us to demonstrate that um can support scrutable user modelling in a large field-trial where

users scrutinised their models within an authentic, working environment where the user model was created for a coaching system which gave user-adapted advice. We have demonstrated that um is scrutable user modelling shell which can serve the user's need to explore the user model which drives user-adapted interaction.

Appendix A - Example models

An excerpt from a sam user model.

```
model: sam_useful "Extremely useful but not absolutely essential items in sam";
files:
        sam "sam" "Miscellaneous rules concerning sam in um";
        viewer "viewer" "Rules concerning sam in um";
        tutor "tutor" "Suggestions from the user's tutor";
        coach "coach" "Observations & evidence from the automatic coach";
        deduction "rules" "Things deduced solely from other evidence";

components:
[
        know: quit_k "How to use the Quit (q) command correctly";
        believe: quit_b "That killing sam's window is usually the best way to exit sam";
        know: default_size_k "How to create a window of the default size";
]

values:
        quit_k: true (741323816);
                support
                        observation sam used_quit (732421532) "COUNT=2100";
                        rule deduction quit_weights (732421532);
                        told viewer explained (740892021);
                        told viewer explained (741320963);
                        told viewer explained (741322608);
                        told viewer explained (741323039);
                        told viewer explained (741323810);
                        given viewer self (741323816);
                negate
                        observation sam noused_quit (732421532) "COUNT=92";
                        given tutor tutor_added (741323801) "PERSON=ronny";
        quit_b: false (741323816);
                support
                        given viewer self (724217200);
                        observation tutor quit_observed (724727203) "PERSON=jsmith";
                        observation sam quit_observed (724717208);
                        told viewer explained (726897273);
                        told coach email (727072976);
                        told viewer explained (733108000);
                        given viewer self (734299999);
                        given viewer self (736039368);
                negate
                        given viewer friend (725081196);
                        rule deduction threshold (725081792);
                        given viewer self (734299999);
                        given tutor tutor_added (736039391) "PERSON=jsmith";
        default_size_k: conflict (741323816);
                support
                        told viewer explained (268444112);
```

An excerpt from a movies model.

```
model: test "a test partial model";

files:
        movies_all "/usr/staff/ronny/.um/movies/Movies/Movies.all" "movies database";
        rules "rules" "Miscellaneous rules including stereotypes";

components:

        violent is "a preference";
        drama is "a preference";
        Abortion is "a preference";
        Children_and_Child_Rearing is "a preference";
        Horror is "a preference";
        Murder is "a preference";
        Police is "a preference";
        Religion_and_Spirituality is "a preference";
        Supernatural_Beings is "a preference";
        American_South is "a preference";
        American_Civil_War is "a preference";
        Death_and_Dying is "a preference";
        Doctors_and_Medicine is "a preference";
        Marriage is "a preference";
        Romance is "a preference";

values:

        violent: false (715669345);
                negate
                        stereotype rules violent_age (715668590);
                        stereotype rules violent_gender (715668592);
                        stereotype rules violent_educated (715668598);

        drama: true (715669345);
                support
                        stereotype rules drama_gender (715668592);
                        stereotype rules drama_educated (715668598);

        Abortion: no_info (715669345);

        Children_and_Child_Rearing: true (715669345);
                support
                        stereotype movies_all movie_6063 (715668795);

        Horror: no_info (715669345);

        Murder: true (715669345);
                support
                        stereotype movies_all movie_171 (715668795);

        Police: false (715669345);
                negate
                        stereotype movies_all movie_5016 (715668795);
```

Religion_and_Spirituality: conflict (715669345);
	support
		stereotype movies_all movie_1722 (715668795);
	negate
		stereotype movies_all movie_137 (715668795);

Supernatural_Beings: false (715669345);
	negate
		stereotype movies_all movie_137 (715668795);

American_South: true (715669345);
	support
		stereotype movies_all movie_175 (715668795);
		stereotype movies_all movie_6063 (715668795);

American_Civil_War: true (715669345);
	support
		stereotype movies_all movie_175 (715668795);

Death_and_Dying: false (715669345);
	negate
		stereotype movies_all movie_5096 (715668795);

Doctors_and_Medicine: no_info (715669345);

Marriage: no_info (715669345);

Romance: false (715669345);
	support
		stereotype movies_all movie_175 (715668795);
	negate
		told (715669322);

Appendix B - User model structures for experiments

This appendix shows the
structure of the user model for sam as used in the two experiments
described in chapter 9.
The partial model names are and structure are indicated at left.
So, for example, within the sam context, there is a partial model called
basics and within it is a partial model called minimal
and within that is the command go_k.
The strings at the right are the descriptions associated with the
components.
These descriptions were available on the leaf displays of xum.

Model structure for Experiment 2

This model is very similar to that used in experiment 1.
The only differences are:

- The regexp partial model was simpler in Experiment 1 since we did not distinguish the aspects that common to the unix shell.
- The belief components were not used in Experiment 1.

basics		
	minimal	
		go_k — How to use the mouse to move around
		nonscroll_b — Believe scrolling is not really necessary to use sam well
		open_window_k — How to open a window
		scroll13_k — How to use buttons 1 & 3 on the scroll bar correctly
		scroll2_k — How to use button 2 on the scroll bar correctly
		write_k — How to write out a file with a name
	useful	
		default_size_k — How to create a window of the default size
		quit_b — Believe killing sam is a good way to quit
		quit_k — How to to the Quit (q) command correctly
more_useful		
	command_window	
		non_cmd_b — Believe sam should be used without the command window
		gotoline_k — How to use addresses to go to a line
		load_new_k — How to use the load new files (B) command
		set_fname_k — How to use the set-filename (f) command
		undo_k — How to Undo errors
		write_k — How to use the write command
	mouse	
		cut_k — How to use the cut menu option
		look_k — How to use look on the menu for mouse button 2
		paste_k — How to paste text back into the document
		snarf_k — How to snarf text without deleting it first
		submenu_k — How to use the right button (button 3) submenu
	other	
		tabsize_k — How to set the tab size when invoking sam
		highlight_esc_k — How to highlight text by pressing ESC
very_useful		
	command_window	
		addresses
		comma_k — How to use the comma (,) address operator

```
        dot_k           How to use the dot (.) address
        line_num_k      Hoe to use line number addressing
        quote_k         How to use the quote (') address mark
    unix
        pipe_k          How to use the pipe (|) command
        read_txt_k      How to use the read (r) command
        red_in_k        How to use the redirect-in (<) command
        red_out_k       How to use the redirect-out (>) command
        shesc_k         How to use the shell escape (!) command
        regexp
            unix
                basic_grep
                    rexp_brack_k    How to use brackets [] in regular expressions
                    rexp_caret_k    How to use a caret ^ in regular expressions
                    rexp_dollar_k   How to use a dollar sign in regular expressions
                    rexp_dollar_k   How to use a dollar sign in regular expressions
                    rexp_dot_k      How to use dot (.) in regular expressions
                    rexp_star_k     How to use an asterisk in regular expressions
                extra_egrep
                    rexp_group_k    How to use parentheses () in regular expressions
                    rexp_pipe_k     How to use pipe | in regular expressions
                    rexp_plus_k     How to use plus + in regular expressions
                    rexp_question_k How to use a question mark ? in regular expressions
            sam_extra
                rexp_at_k           How to use regular expression to find expressions in a file
                rexp_newline_k      How to use backslash-n in regular expressions
        other
        mark_k          How to use the address marking (k) command
        subst_k         How to use the substitute command

    mouse
        exch_k          How to use <exch> on the menu for mouse button 2
        search_k        How to search for the last regexp using button 2
        xerox_k         How to use xerox on the menu for mouse button 2
powerful
    addresses
        addr_hash_k     How to use hash (#) for addressing
        addr_minus_k    How to use minus (-) for relative addressing
        addr_plus_k     How to use plus (+) for relative addressing
        addr_quotes_k   How to use quotes around a regexp for addressing
        addr_rexp_k     How to use regular expressions for addressing
        addr_semic_k    How to use the semicolon (;) for addressing
    commands
        if_regexp_k     How to use the regexp conditional (g) command
        loop_regexp_k   How to use the loop-over-regexps (x) command
        loopf_k         How to use the loop-over-matching-files (X) command
        loopnf_k        How to use the loop-non-matching-files (Y) command
        nif_regexp_k    How to use the absent-regexp conditional (v) command
        nloop_regexp_k  How to use the loop-between-regexps (y) command
    mouse
        scrollm_k       How to use the scroll/noscroll option on button 2
        sendm_k         How to use the send option on button 2
        severm_k        How to use the sever option on button 2
mostly_useless
    file_cmds
        deletef_k       How to use the Delete-files (D) command
        editf_k         How to use the edit-file (e) command
        listf_k         How to use the list-files (n) command
```

	setf_k	How to use the set-file (b) command
text_cmds		
	appendt_k	How to use the append-text (a) command
	changet_k	How to use the change-text (c) command
	copyt_k	How to use the copy-text (t) command
	deletet_k	How to use the delete-text (d) command
	insertt_k	How to use the insert-text (i) command
	movet_k	How to use the move-text (m) command
other		
	group_k	How to use the grouping {} construct
	print_k	How to use the print-dot (p) command
	location_k	How to use the (=) command

And other contexts also available for this experiment were:

input	Information on how the user communicates with the computer
wpm	Typing speed in words per minute
programming	Computer languages which you use on this system
c_prog	You program in C
pascal_prog	You program in Pascal
lisp_prog	You program in Lisp
fortran_prog	You program in Fortran

In Chapter 9 we refer to work on a Unix coach. This was managed in a separate context. Since the users involved in Experiment 2 did not have this as part of their model, we do not include the details here (Butler, 1992).

Appendix C - Summaries of interviews

User A

Comments about sam**, learning:**
Likes to learn by experimentation but likes an introduction first. Like to know generally what is going on [moves hands in large circle]. Finds documentation hard to use. Prefers Macintosh as no need to learn from documentation - learn by doing. Found sam easy to learn and similar to Mac. Learnt throughout the year.

Has used computers a lot and can learn more quickly than most students. Things liked about sam: multiple files, speed of startup, simple things. Prefer vi generally and used it in the prac exam. Used vi except when several files needed or moved large chunks of text [so they do not know vi very well].

Has a Mac at home. Has used many word-processors on it (Word perfect, Word, MacWrite). Began using vi for library. No arrow keys [in sam] irritating. Likes choice of mouse/keyboard commands as on Macintosh.

Did program development and thinking and home and then used uni machines, library and dept.

User view of 'knowing' something
Likes to understand why they are doing things *rather than just how*. Hates a step-by-step blind outline because you don't remember. Likes a reference. Liked vi because it was a challenge: found sam simplistic.

Do you agree with the model?
Agrees it fairly reflects their knowledge. Observed 'undo' and did not know it (as model showed). However, used 'look' more than 'xerox'.

Would you find the viewer **helpful if it were on-line?**
Felt that qv needed more explanation. Needs to know what it is all about. Notes that qv helped in navigation and gave an initial idea of what things there were. Liked qv way of collapsing wholly unknown parts. *Would be more interested in a similar* viewer *for vi and unix.*

By the end of the interview, thought they would use it: had earlier not thought so because not so interested in sam. [It showed greater potential for sam - that it is not just simple as they had thought]. Would have liked more information on viewer. Would like evidence in lower window. Prefer 'help' in larger font and perhaps not so much of it.

Liked explanations of what they did not know. Would like to know how to use sam more efficiently.

Review notes
This student is keen to learnt thoroughly and likes challenges. The viewer helped them see some of the potential of sam that they had not recognised in the time they had spent with sam and its documentation.

User B

Comments about sam**, learning:**
Uses vi usually. It is faster and more accurate. Likes cursor keys. Would like a menu on sam. Uses sam for required assignments only. Better for multi-files. Would improve sam to mac interface standard with cursor keys, page up and page down, highlights, ...

User view of 'knowing' something
Expressed difficulty conceptualising sam. ''Even if you remembered how it [sam] worked it would still be hard after three years ... it is consistent but hard to get to really advanced commands. You need to use the mouse and I wouldn't have confidence that I could get back to it.''

Do you agree with the model?

Found the model interesting and useful. Generally accurate.

Would you find the viewer **helpful if it were on-line?**
Not necessary for the essential. [Not interested in learning more than that.]

Review notes
This is a student who achieved excellent results in the course and clearly used sam only as a necessary evil for situations where it was really necessary. This user was really not interested in sam and in turn was only polite in use of the viewer.

User C

Comments about sam**, learning:**
Likes sam generally. Would like xedit facilities for moving cursor [a common comment]. Uses vi for quick data files. Found sam better for 'multi-tasking'. Is used to the mouse. Finds it annoying to use backspace.

User view of 'knowing' something
Has a *big picture* of the editor. Talks about it in comparison with xedit and vi. Talks in terms of *exploration, curiosity and comfort.*

Do you agree with the model?
Generally thought it accurate.

Would you find the viewer **helpful if it were on-line?**
Probably but like to fiddle about and have not had great difficulty with sam.

Review notes
This user made only limited exploration of the model and seemed uninterested in having a very limited knowledge of sam. Seemed satisfied with current knowledge and not very interested in viewer.

User D

Comments about sam**, learning:**
Commands were a problem.
I had to read the manual.
Text editor not easy to learn.
Mouse different from MS-DOS version at home.
Three buttons a major adaptation.
Never sure how to save a file.
I tend to rush in.
Time is a great pressure.
Most work done at home.
Not enough time in the the practical class even if the program is fully written out.
Thinks of editor as a tool that works.
Not interested in how.

User view of 'knowing' something
Read the manual and *use once at least*.
Unix not user-friendly.
Prefer to telnet files from home.
Not interested in exploring or changing usual methods.
Works mostly at home.
Had trouble with the first practical exam as not used to working at uni.
Prepared for second prac exam by learning syntax.
Sees no advantages in sam over xedit.

Commands actually mostly harder to learn, with harder sequences.
Can't find line [meaning the last line of the command window where commands should be typed].
Like using arrow keys in xedit.
[sam does not support such operations.]

Do you agree with the model?
OK [then more about own editor]

Would you find the viewer **helpful if it were on-line?**
not very interested in using the viewer

Review notes
Strongest message being communicated was negative response to sam. Clearly did not make the move from familiar editing tools to sam and its very different approach.

User E

Comments about sam**, learning:**
Learnt about sam, and everything else, mostly by word of mouth. Not very much explained by the tutor. Basic functions easy to learn. For more, need to understand the editor.

User view of 'knowing' something
I know *when I can use it and feel comfortable.* Telling is not enough [referring to the change in status after use of the explain]. Uncomfortable with regular expressions. Need to sit down and use these to really understand how they work.

Do you agree with the model?
Generally OK. Queried the 'You probably don't know' category. Somewhat reluctant to explore 'explain'. Commented that the print was too small.

Would you find the viewer **helpful if it were on-line?**
Yes [actually did so later].

Review notes
This user seemed quite curious about the viewer and interested in their model.

User F

Comments about sam**, learning:**
At the beginning, read notes fast, skimming, then started straight away. People in the course were confused about the windows and interface.

User view of 'knowing' something
Someone tells me and I try it out. Then it's added to my repertoire. I like commands that save work. Remembers commands that are useful.

Do you agree with the model?
Generally accurate. But the explanations! A bit complicated for the uninitiated. Found the qualifiers a bit confusing [referring to 'you know how to' and 'you probably don't know' ...]. interesting in the weighting.

Would you find the viewer **helpful if it were on-line?**
Yes. Would use it for feedback. Needs a menu to assist in search for commands. But sometimes not clear what the explanation means. [Would you look at the explanations?] No, I talk to others. 'It would be useful if you had an incentive to use it.' *Commented on the change of knowledge assessed after looking at explanation. Felt strongly that they still did not know the commands as they had not used them.*

Review notes
This user was very curious. In fact, they later used the viewer three times: Wed Oct 30 17:30:42 1991; Fri Nov 1 10:16:11 1991; Wed Nov 20 15:03:18 1991. This user explored all aspects of the viewer.

User G

Comments about sam**, learning:**
Took almost six months to learn the commands. Was very ill and found it difficult to find time for the editor. "Frustrated because it didn't fall into place". Used sam for most work.

No on-line help and documentation difficult to understand. Command names not meaningful, eg snarf. Save in Wordstar is 'S'. Mouse use is different. Three buttons tool a while to master. Was used to non-mouse-based systems. Finds keyboard commands easier. Menus too small for mouse. Cannot place it accurately as required. Liked being able to move whole blocks of text as it saved typing. Windows a new thing to master.

User view of 'knowing' something
I only know what I use fairly often. Things I do fairly often, for example, 'w' write. When you want to learn something new, it's a bit of a problem. There is no time to look things up ... tend to have a bash ... guess ... command names are not as intuitive as other word processors. Why not use 'copy' instead of 'snarf'. ''You know something when you can use it correctly without looking it up.''

Do you agree with the model?
essentials .. *generally agreed*
... interested in weighting
undo .. not known ... saw no point ... ''easier to delete and type again''
finds scroll bars difficult ... [evident to interviewer]
found explanations difficult to understand
expressed strong need to understand the logic of the viewer and of sam
slow typing speed

Would you find the viewer **helpful if it were on-line?**
''May save some time
... took a long time to learn to use sam ... also meant to be programming ... would help''. ''Tells what the commands do ... don't use many commands editing programs ... use error messages after compiling ... [meaning that the student edits a program in sam, compiles it in another window, looks at compile errors which are gives in relation to the line numbers, goes to the relevant line number and edits that, repeating the

cycle]. BASIC experience not a help for other programming languages ... blocks development.

Review notes

This student had been extremely seriously ill during the year and this had clearly made all study stressful. They had clearly found sam difficult to master and this is mentioned frequently in the interview. In fact, it is the dominant message from the student, overshadowing their response to the viewer. The response to the viewer is politely positive, essentially capturing the message that the student is not interested in sam and, by implication, the viewer displaying a sam model is not interesting. The student could use the viewer to check the accuracy of the model and was at least politely interested in the exploration. They found the text of the explanations hard to understand.

Appendix D - Summary of logs of usage of viewer

The following table shows the counts for the various actions taken by the 46 users who made any use of xum. Values with * are more than double the mean for that column.

The first column is the number of component explanations sought. The second column is the number of change actions on the model. Note that these often involved repeated changes to the same component. The third column is the total number of sessions. The fourth column is the number of times the user requested the evidence about a component. The second last column is navigation steps around the model. The final column is the total number of actions.

Chapter 9 shows the overall summary statistics and focuses on the 14 users whose total number of xum actions was above the average for the whole group.

	explain component	alter value	sessions	show evidence	move	total
column total	377	228	136	421	1403	2565
column average	8.2	5.0	3.0	9.2	30.5	55.8
	10	38*	5	8	69*	130*
	3	4	1	2	12	22
	5	7	2	5	27	46
	4	0	2	7	19	32
	1	0	1	3	9	14
	25*	2	1	36*	14	78
	0	0	1	0	0	1
	1	3	2	7	13	26
	0	0	1	1	13	15
	4	4	2	2	14	26
	4	1	4	3	30	42
	25*	32*	2	3	25	87
	2	1	2	1	25	31
	16*	4	4	23	55	102
	46*	22*	11*	32*	165*	276*
	0	0	1	0	4	5
	24*	5	1	63*	39	132*
	3	2	1	2	17	25
	16*	19*	2	25*	33	95
	2	0	1	2	9	14
	5	0	1	13	26	45
	1	0	2	1	10	14
	0	0	1	0	0	1
	0	0	2	0	5	7

This page continues the table with counts for actions taken by the 46 users who made any use of xum. As before, values with * are more than double the mean for that column. Overall column totals and averages are repeated on this page.

	explain component	alter value	sessions	show evidence	move	total
	12	0	11*	19*	35	77
	20*	28*	2	27*	37	114*
	1	0	1	1	7	10
	6	0	1	0	16	23
	1	0	1	0	5	7
	2	4	1	3	19	29
	2	0	3	7	37	49
	2	5	3	6	15	31
	39*	18*	28*	16	188*	289*
	1	0	1	0	12	14
	4	2	3	10	15	34
	7	3	4	2	19	35
	0	0	1	0	0	1
	43*	4	2	28*	81*	158*
	0	0	1	0	5	6
	9	0	4	10	40	63
	0	0	2	0	7	9
	1	3	2	2	23	31
	3	9	2	6	15	35
	6	0	3	11	84	104
	5	0	2	2	13	22
	16*	8	5	32*	97*	158*
column total	377	228	136	421	1403	2565
column average	8.2	5.0	3.0	9.2	30.5	55.8

Appendix E - Heavy users of xum - log summaries

This appendix summarises the qualitative analysis of the logs for heavy users of the viewer. These notes have removed user identification data. Since we are concerned with the way that each student used the 'Explain'/'Justify'/'Change' facilities of the viewer. References to commands mean these three classes of command only. The following text refers to parts of the sam model that was build. For the structure of that model see Appendix A.

There were separate logs for the main viewer and the qv displays and both these are summarised below.

User A

First 18 commands looked at various aspects near the top level of the model and doing both 'Explain' and 'Justify' for each. Then the next 5 commands used 'Change' to set components to FALSE and the remainder of the uses were 'Change', setting large parts of the model to TRUE, including belief-components. This was the only user to do this playing with the model.

They used qv once.

Essentially this user was playing, setting the model all to TRUE and looking at the viewer.

User B

The first nine commands are pairs of 'Explain'-'Justify' then similar command pairs for a sequence of minimal, then useful, then more_useful commands (about 10) then a sequence of 'Explain' for 6 more_useful and then back to the 'Explain'-'Justify' pairs for 3 more_useful.

Next was 'Change' paste_k to MAYBE followed by 'Justify' for it and then examine the basis for the evidence with 4 'Explain-evidence'. Then 'Change' it (paste_k) TRUE and 'Justify'. This is an interesting depth analysis and exploration of the viewer.

The remainder of the use involved exploration of 4 more commands with the 'Explain-evidence' as well.

They used qv once for about 30 seconds.

Overall this user altered only paste_k first to MAYBE then to TRUE. They used 'Explain-evidence' 10 times, 'Justify' for basics and more_useful, each different except non_cmd_b (2) paste_k (3) typing_ok_c (3) wpm_info (2). Aspects explained were mainly from basics and more_useful, all different except go_k (2) typing_ok_c (2) wpm_info (2).

Essentially this user appears to be exploring the viewer.

User C

Starts with 'Justify' (submenu_k) and then sets 2 aspects (cut_k and snarf_k) to MAYBE. Then uses 'Explain' for submenu_k. Next are several 'Explain', starting with line_num_k and comma_k and moving to the languages for explanations followed by the setting of their values. Then a sequence of several (6) 'Explain' 'Change' pairs where the user appears to check the meaning of the component and then set it for various minimal-editor aspects, then 1 'Change' for quit_k (fairly obvious without 'Explain'), then quit_b which is set FALSE. Next they go through the more_useful commands in the same way either using 'Explain' then 'Change' or, for obvious ones, just 'Change', mainly setting them TRUE (they set only set_fname_k FALSE)

They seem to take some trouble at this as they set dot_k first to MAYBE and then FALSE and similarly, read_txt_k twice to MAYBE and then FALSE. Only at comma_k do they use 'Justify'.

They used qv once, trying two variants on labelling.

In all they made many changes to the model, with considerable numbers being done more than once: cut_k (2) snarf_k (2) submenu_k (2), dot_k (2), read_txt_k (3 with 2 the same), fortran_c (2 the same). This seems to suggest they were thinking about and revising their self-assessment. The 3 uses of 'Justify' were for submenu_k, comma_k, line_num_k. The uses of 'Explain' were single for each aspect, except comma_k

This user seems to indicate browsing and the focus seems to be on getting the model right. (2).

User D

Starts with 'Explain'-'Justify' for typing aspects, then vi and emacs, then basics of sam and 'Change' quit_b TRUE, MAYBE then FALSE and then did 'Justify'-'Explain''Change' non_cmd_b FALSE. Interesting that they focussed on the beliefs. This could be because they are different. Then they look at more_useful a bit with 'Justify'-'Explain'. They then move to programming languages and typing aspects with the same pattern. Overall they seem to be looking around a bit but have little real interest in sam.

They 'Change' quit_b (3 different values) and non_cmd_b. The evidence explored was emacs, 6 basics (1 being quit_b twice) 1 more_useful and 2 very_useful, then 1 vi, 2 languages (pascal_c twice) typing aspects, typing_ok_c (3 times) and wpm_info (4). They used 'Explain' for 3 basics, 1 more_useful, 2 very_useful, vi, 4 languages, 5 on the 2 typing aspects.

They used qv twice, the first time trying seven variants of labelling the leaves.

This user seems to be exploring the viewer and the things that interest them. For example, only about 1/3rd of the uses of 'Explain' were for sam - the rest being for other parts of the model. Makes

User E

Since this student made so much use of the viewer, we mailed to ask about it. Student comment in reply about use of the viewer

> I thought the model was very useful for finding out about sam, and exploring the available features

then noted the need for on-line help in sam and short cut keys, and/or cursor keys - monitor logs shows them to be a regular sam user.

Starts with sequences of 'Explain' usually accompanied by 'Justify' for several aspects from very_useful and setting the values some of the commands, mainly to TRUE. Then they move to using 'Explain' and 'Change' for most aspects in command_window and minimal. Next a sequence of 8 'Explain' of basics, 'Change' for 2 to TRUE. Then a series of 'Explain' for more_useful and 5 'Change', all to FALSE. Next explored vi and languages using 'Explain' and both 'Explain' and 'Justify' for typing_ok_c then more languages and 'Change' to set some to FALSE. Then they re-examine 'Justify' for 10 more_useful, 'Change' for one and 'Explain' for one. Next they do 3 'Explain'-'Justify' sets on aspects from other editors and typing. Then there is a series of 4 'Explain', then more 'Justify' for typing aspects and a sequence of 8 'Justify' for languages and typing aspects.

Overall, there were changes to 22 components, each only affecting one component and 3 from basics (all set TRUE), 6 from more_useful (5 set FALSE and 1 TRUE), 11 from very_useful (9 set to TRUE, 1 FALSE and 1 MAYBE) and 2 from languages set FALSE.

Requests for 'Justify' were 1 emacs, 13 from more_useful 4 from very_useful with each being a different component except paste_k, tabsize_ k, comma_k each being twice. There were 6 'Justify' for languages (c_c and pascal_c twice) 4 'Justify' each for typing_ok_c and wpm_info.

The use of 'Explain' had uses_emacs_c, 10 basics (of which default_size_k occurred twice) 7 more_useful (in which non_cmd_b and undo_k appeared twice) 16 very_useful with each different except pipe_k (2)

read_txt_k (3) as well as 3 ‘Explain’ for uses_vi_c 7 languages (with 2 of 3 and 1 of the other) and 1 each of 2 typing aspects.

They used qv three times, each spaced from the other and each with the labels altered to Leaves, None, All.

This student seems to be working to get the model right. This should have helped the coaching program. The student appears to be looking most at the aspects that are likely to be useful to learn and they may well have been using the viewer as a learning tool.

User F

First 32 commands were ‘Justify’. The typical pattern was a ‘Explain’ followed by ‘Explain-evidence’ (usually twice). Then 6 ‘Explain’ in more basic aspects, then pairs of disagree (toggling to and back) quit_k and go_k. Then sets one ‘Change’ and shifts to a pattern of ‘Explain’ then ‘Justify’ and ‘Explain-evidence’. Then a burst of 10 ‘Explain’ about regular expression, mainly 20 seconds apart, enough time to skim the text. Then there is a diversion to ‘Justify’ and ‘Explain-evidence’ followed by 4 more of the useful-level aspects, 1 ‘Justify’ and another ‘Explain’.

In all they used ‘Change’ 5 times (on 3 components), ‘Explain-evidence’ 36 times and ‘Justify’ 28 times with cut_k, go_k, line_num_k, open_window_k, undo_k all twice and quit_k, scroll13_k scroll2_k three times. The 24 uses of ‘Explain’ repeated rexp_newline_k, rexp_plus_k, rexp_question_k, scroll2_k twice and scroll13_k three times. This looks like an exploration of the viewer or an attempt to learn or at least judge the utility of the aspects of sam.

User G

First half of commands are in basics. First is ‘Justify’-‘Explain’ for go_k then a lone ‘Justify’ for scroll2_k and then ‘Explain’-‘Change’ for go_k and for scroll2_k. Then they use ‘Explain’-‘Change’ scroll13_k (TRUE) and nonscroll_b (FALSE). They appear to be setting the model to correct values and the ‘Explain’ is to check the meanings. They next explore write_k with ‘Justify’ and ‘Explain-evidence’, then ‘Explain’ and ‘Change’. Then is a sequence with ‘Explain’-‘Change’ for 4 components, exploring further the ‘Justify’ for 2, then return to ‘Justify’ for another. Next is ‘Justify’ for typing and ‘Explain-evidence’ and a similar pattern for pascal_c and c_c, setting the latter to TRUE and looking again at ‘Justify’ for pascal_c and then using ‘Justify’-‘Explain’-‘Change’ for lisp_c and then making ‘Change’ 4 times for fortran_c. Then is a similar type of series on typing aspects and their evidence.

They used qv three times, each 15-35 seconds.

This user appears to be tinkering with the model and the viewer but seems not terribly interested in sam. They used ‘Change’ on 9 basics components (all different), then 7 languages components, 3 typing aspects. This suggests relatively low interest in sam. They looked at ‘Explain-evidence’ of various sorts: 2 s_percent_suff, 3 s_total_suff, 1 given, 1 no_info, 2 observation, 3 told. Their use of ‘Justify’ had 6 from basics, 4 from languages, 3 typing. There were 9 aspects from basics ‘Explain’-d, one twice (go_k) and 3 each for languages and typing aspects. This seems consistent with interest in the model but not so much in sam.

User H

The whole log in interspersed uses of ‘Explain’ and ‘Justify’ for typing aspects. There are 8 ‘Justify’ of typing_ok_c and 11 of wpm_info and 5 ‘Explain’ and 7 uses of ‘Explain’ for them.

They used qv three times.

This user seems rather aimless and is really a non-user of the viewer.

User J

(This user was interviewed in first set.) Starts with a series of 4 ‘Explain’-‘Justify’ then a sequence of 2 ‘Explain’-‘Justify’-‘Change’ sets for typing and languages aspects, setting both lisp_c and fortran_c FALSE. Then goes through several minimal aspects of sam doing ‘Justify’-‘Change’ pairs. Then they start to do mainly ‘Explain’-‘Change’ pairs for more of the minimal and useful aspects from basics. Next is a series of actions in more_useful mainly aiming to ‘Change’ but some of the time doing ‘Explain’ and in most of this setting things as known (or beliefs false)

Next shifts attention to editors setting uses_emacs_c FALSE and uses_vi_c through a series MAYBE, ‘Justify’, MAYBE, FALSE, TRUE, MAYBE then ‘Justify’ for it and ‘Explain-evidence’ for given. Then a similar series in aspects of typing and languages. This series seems to be thinking about this aspect then looking in some detail at the model and basis for operation.

They used qv twice, the first time trying options for labelling the tree.

In summary, they set uses_emacs_c FALSE, basics elements go_k, scroll2_k write_k default_size_k quit_k TRUE nonscroll_b quit_b FALSE and scroll13_k MAYBE (3 times). This means they were careful about getting knowledge aspects set TRUE and beliefs FALSE to indicate that they knew the basics. They set more_useful aspects gotoline_k cut_k paste_k snarf_k submenu_k TRUE, load_new_k look_k FALSE, tabsize_k MAYBE. As noted above they set uses_vi_c 5 times (differently), 2 languages aspects FALSE typing aspect 2 (different) values. They used ‘Explain-evidence’ 11 times (5 given, 1 observation and 5 told). The ‘Justify’ was used 3 times for sam aspects (scroll2_k twice), and for vi twice, languages 5 times, 6 times for typing aspects. So they explored the role of evidence mainly outside the sam domain. They used ‘Explain’ 6 basics and 6 more_useful aspects, 4 different programming languages, 3 times for typing aspects.

This student looks to have explored the viewer but I suspect that also as they were part of the interview group, they should be seen as different. They were not very interested in sam.

User K

Starts with 13 uses of ‘Explain’, all in one session. The sam aspects examined were the more sophisticated ones. Then follows a series of ‘Explain’ and ‘Justify’ pairs, setting typing_ok_c TRUE, then ‘Justify’, changing it to MAYBE, looking again at ‘Justify’, then setting it FALSE and doing ‘Justify’ and returning in a later session to ‘Explain’ and ‘Justify’ for it. Then looks at more sophisticated aspects, mainly looking at ‘Justify’. Seemed to regularly use ‘Justify’ to check it. Sets quit_b FALSE (three times in one session) Did bursts of looking at ‘Explain’ (12 in a row over different sessions.) One session has a burst of 6 ‘Change’, mainly to TRUE. Set nonscroll_b TRUE then FALSE - people may have some trouble with the notion of beliefs rather than knowledge. Finally did a series of ‘Explain’ for very_useful and more_useful aspects. So, overall this user had a large number of commands that involved moving around the model and other quite purposeful actions that involved studying the commands in sam and amending the model as well as checking the ‘Justify’ for values of components of current interest.

They used qv seven times well spaced.

Overall, altered one belief-component to FALSE and back, another to FALSE three times. They set one basics-component to TRUE, 9 more_useful commands to TRUE (without having first used ‘Explain’ for 6 of them - gotoline_k, undo_k, write_k, cut_k, paste_k, submenu_k - mainly fairly obvious from the string in the xum display what each of these means if the user knows the commands). For look_k, they read the explanation at time 715211772 (Aug 31) and then set it TRUE at time 715473219 (Sep 3) Similarly for snarf_k at times 714018647 (Aug 17) and 715473206 (Sep 3) as well as for xerox_k at 713669048 (Aug 13) and 715473391 (Sep 3) respectively. They also altered typing_ok_c through FALSE, MAYBE, TRUE.

Read explanations once, except load_new_k, typing_ok_c (3 times - set_fname_k, highlight_esc_k, tabsize_k, sendm_k, subst_k, pipe_k, exch_k, wpm_info (twice each). This is consistent with trying to use this as a learning tool: the student refers back to the explanations of sam and they are looking at aspects all of which are quite useful.

User L

Starts with sets of 'Explain' and 'Justify' for several basic commands. Then has 9 'Explain' in a row (address_cmd_k, read_txt_k, red_in_k, red_out_k, pipe_k, shesc_k, mark_k, gotoline_k, mark_k again). Next 6 'Justify', some on the above and others too, then several more 'Justify', one Disagree ('Change' equivalent), many more 'Explain', one 'Explain-evidence' for undo_k (twice) and then Disagree undo_k (twice - so toggled)

The 4 uses of 'Change' were setf_k, xerox_k and toggling undo_k. Heavy use of 'Explain', once per command except gotoline_k (2), mark_k (2), quit_k (3), scroll13_k (2), scroll2_k (2), setf_k (2), subst_k (3) and undo_k (2). Overall use of 'Justify' was only once per command, except for gotoline_k (3), load_new_k (2), quit_k (2), subst_k (3), undo_k (4).

This user explores more 'Explain' compared to other commands and the choice command-explanations explored is consistent with use for learning.

User M

This user had only 'Explain' and 'Justify' for two commands, wpm_info and typing_ok_c. In all they did 4 'Justify' for typing_ok_c and 6 for wpm_info, 'Justify' 4 and 5 times respectively.

They used qv three times in close together, the first time trying various amount of labelling. Later they used it once more.

They did not seem interested in sam.

User N

Starts with 'Justify' and 'Explain' for typing aspects then does either 'Explain' or 'Justify' for 6 very_useful aspects and the 'Explain'-'Justify' for 2 and 2 'Justify'-Evidence and one more 'Justify'.

They used qv for once 25 seconds.

This user appears to have moved around the model quite a bit and explored the viewer a bit and dot_k, line_num_k, subst_k, xerox_k, search_k, highlight_esc_k, subst_k all from more_useful

User P

Starts with a series of 'Justify' and some pairs of 'Explain' with 'Justify', mainly in the areas of the programming languages used but also some basics-components. Then sets uses_vi_c TRUE followed by 'Justify', sets three languages FALSE and one (C) TRUE. Some activity using 'Change' and 'Justify' for wpm_info, typing_ok_c and 'Explain' and 'Justify' for C and Pascal (checking they are TRUE). Series of 4 'Explain-evidence'. Then some 'Explain' and 'Justify' of some command_window aspects. Used 'Justify', 'Explain' and 'Change' for wpm_info. This user seems interested in the viewer as they have explored a number of its facilities. They do not seem to have been exploring sam with it though.

Note that the first 'Change' they used was to set uses_vi_c TRUE. This user may well have little interest in learning sam.

They used qv twice, setting different labelling levels the second time - both uses in the one session.

Overall they set 5 aspects TRUE, 3 FALSE and these were read_txt_k, uses_vi_c, c_c, typing_ok_c,

wpm_info all TRUE and fortran_c, lisp_c, pascal_c all FALSE. They looked at 3 ‘Justify’ of given-evidence and 3 of told-evidence. They made heavy use of ‘Justify’, uses_emacs_c (twice), minimal sam aspects (2), command_window aspects (3, 1 twice), uses_vi_c (4 TIMES), languages (8), typing aspects (4)

Appendix F - One user's viewer log

This log has been edited to remove any indications of the user's identity. In particular, each line of the real logs has the user's login identifier.

713668155 Thu Aug 13 11:09:15 1992
713668155 XUM:Start
713668155 XUM:Rescan
713668158 XUM:Scan editors
713668160 XUM:Scan sam
713668168 XUM:Scan more_useful
713668171 XUM:Scan mouse
713668177 XUM:Backing up by 1 level
713668177 XUM:Scan mouse
713668177 XUM:Backing up by 1 level
713668177 XUM:Scan mouse
713668195 XUM:OpenComment
713668199 XUM:CloseComment
713668201 XUM:Help
713668390 XUM:CloseHelp
713668398 XUM:Quit
713668398 Thu Aug 13 11:13:18 1992

713668407 Thu Aug 13 11:13:27 1992
713668407 XUM:Start
713668407 XUM:Rescan
713668412 XUM:Scan editors
713668414 XUM:Scan sam
713668420 XUM:Scan more_useful
713668423 XUM:Scan mouse
713668433 XUM:Quit
713668433 Thu Aug 13 11:13:53 1992

713668452 Thu Aug 13 11:14:12 1992
713668452 XUM:Start
713668452 XUM:Rescan
713668465 XUM:Scan editors
713668468 XUM:Scan sam
713668470 XUM:Scan more_useful
713668473 XUM:Scan mouse
713668481 XUM:Quit
713668481 Thu Aug 13 11:14:41 1992

713668490 Thu Aug 13 11:14:50 1992
713668490 XUM:Start
713668490 XUM:Rescan
713668494 XUM:Scan editors
713668499 XUM:Scan sam
713668505 XUM:Scan more_useful
713668509 XUM:Scan other
713668519 XUM:Explain:
editors/sam/more_useful/other/tabsize_k
713668538 XUM:Explain:
editors/sam/more_useful/other/highlight_esc_k
713668559 XUM:QVOpen
713668588 XUM:QVClose
713668591 XUM:Quit
713668591 Thu Aug 13 11:16:31 1992

713668600 Thu Aug 13 11:16:40 1992
713668600 XUM:Start
713668600 XUM:Rescan
713668602 XUM:Scan editors
713668604 XUM:Scan sam
713668608 XUM:Scan very_useful
713668611 XUM:Scan mouse
713668619 XUM:Explain:
editors/sam/very_useful/mouse/exch_k
713668662 XUM:Backing up by 1 level
713668662 XUM:Scan mouse
713668668 XUM:Backing up by 1 level
713668668 XUM:Scan mouse
713668669 XUM:Backing up by 1 level
713668669 XUM:Scan mouse
713668672 XUM:Backing up by 2 levels
713668672 XUM:Scan very_useful
713668676 XUM:Scan mouse
713668714 XUM: Explain:
editors/sam/very_useful/mouse/exch_k
713669007 XUM:Explain:
editors/sam/very_useful/mouse/search_k
713669048 XUM:Explain:
editors/sam/very_useful/mouse/xerox_k
713669065 XUM:Backing up by 2 levels
713669065 XUM:Scan very_useful
713669071 XUM:Backing up by 1 level
713669071 XUM:Scan powerful
713669074 XUM:Scan mouse
713669103 XUM:Explain:
editors/sam/powerful/mouse/sendm_k
713669142 XUM:Explain:
editors/sam/powerful/mouse/severm_k
713669161 XUM:Explain:
editors/sam/powerful/mouse/scrollm_k
713669232 XUM:Backing up by 2 levels
713669232 XUM:Scan powerful
713669235 XUM:Backing up by 1 level
713669235 XUM:Scan mostly_useless
713669237 XUM:Scan file_cmds

713669247 XUM:Explain:
editors/sam/mostly_useless/file_cmds/setf_k
713669266 XUM:Explain:
editors/sam/mostly_useless/file_cmds/listf_k
713669308 XUM:Explain:
editors/sam/mostly_useless/file_cmds/deletef_k
713669499 XUM:Backing up by 1 level
713669499 XUM:Scan other
713669504 XUM:Backing up by 3 levels
713669504 XUM:Scan sam
713669508 XUM:Backing up by 2 levels
713669508 XUM:Scan editors
713669511 XUM:Backing up by 2 levels
713669511 XUM:Rescan
713669515 XUM:Scan user_info
713669522 XUM:Backing up by 2 levels
713669522 XUM:Rescan
713669538 XUM:Scan user_info
713669540 XUM:Scan input
713669548 XUM:Explain:
user_info/input/typing_ok_c
713669566 XUM:Evidence:
user_info/input/wpm_info
713669573 XUM:Explain:
user_info/input/wpm_info
713669694 XUM:Evidence:
user_info/input/typing_ok_c
713669706 XUM:Explain:
user_info/input/typing_ok_c
713669726 XUM:Evidence:
user_info/input/wpm_info
713669743 XUM:Evidence:
user_info/input/typing_ok_c
713669749 XUM:Change:
user_info/input/typing_ok_c->TRUE
713669752 XUM:Evidence:
user_info/input/typing_ok_c
713669767 XUM:Change:
user_info/input/typing_ok_c->MAYBE
713669770 XUM:Evidence:
user_info/input/typing_ok_c
713669777 XUM:Change:
user_info/input/typing_ok_c->FALSE
713669781 XUM:Evidence:
user_info/input/typing_ok_c
713669788 XUM:Quit
713669788 Thu Aug 13 11:36:28 1992

713669999 Thu Aug 13 11:39:59 1992
713669999 XUM:Start
713669999 XUM:Rescan
713670001 XUM:Quit
713670001 Thu Aug 13 11:40:01 1992

713670140 Thu Aug 13 11:42:20 1992
713670140 XUM:Start
713670140 XUM:Rescan
713670145 XUM:QVOpen
713670219 XUM:QVClose
713670222 XUM:Quit
713670222 Thu Aug 13 11:43:42 1992

713681453 Thu Aug 13 14:50:53 1992
713681453 XUM:Start
713681453 XUM:Rescan
713681456 XUM:Scan editors
713681457 XUM:Scan sam
713681459 XUM:Scan powerful
713681461 XUM:Scan commands
713681468 XUM:Explain:
editors/sam/powerful/commands/nif_regexp_k
713681477 XUM:Backing up by 3 levels
713681477 XUM:Scan sam
713681479 XUM:Backing up by 2 levels
713681479 XUM:Scan editors
713681487 XUM:Backing up by 1 level
713681487 XUM:Scan user_info
713681490 XUM:Scan input
713681497 XUM:Explain:
user_info/input/typing_ok_c
713681505 XUM:Evidence:
user_info/input/typing_ok_c
713681513 XUM:Evidence:
user_info/input/wpm_info
713681523 XUM:Explain:
user_info/input/wpm_info
713681545 XUM:Backing up by 3 levels
713681545 XUM:Rescan
713681548 XUM:Quit
713681548 Thu Aug 13 14:52:28 1992

713749318 Fri Aug 14 09:41:58 1992
713749318 XUM:Start
713749318 XUM:Rescan
713749332 XUM:Scan editors
713749337 XUM:Scan sam
713749341 XUM:Scan very_useful
713749344 XUM:Scan mouse
713749353 XUM:Evidence:
editors/sam/very_useful/mouse/exch_k
713749359 XUM:Explain:
editors/sam/very_useful/mouse/exch_k
713749407 XUM:Backing up by 3 levels
713749407 XUM:Scan sam
713749409 XUM:Backing up by 3 levels
713749409 XUM:Rescan

713749411 XUM:Quit
713749411 Fri Aug 14 09:43:31 1992

714018565 Mon Aug 17 12:29:25 1992
714018565 XUM:Start
714018565 XUM:Rescan
714018566 XUM:Scan programming
714018568 XUM:Backing up by 2 levels
714018568 XUM:Rescan
714018576 XUM:Scan editors
714018582 XUM:Scan sam
714018587 XUM:Scan powerful
714018590 XUM:Scan addresses
714018597 XUM:Evidence:
editors/sam/powerful/addresses/addr_rexp_k
714018604 XUM:Backing up by 2 levels
714018604 XUM:Scan powerful
714018608 XUM:Scan mouse
714018612 XUM:Evidence:
editors/sam/powerful/mouse/severm_k
714018622 XUM:Evidence:
editors/sam/powerful/mouse/scrollm_k
714018627 XUM:Backing up by 2 levels
714018627 XUM:Scan powerful
714018631 XUM:Scan commands
714018638 XUM:Backing up by 2 levels
714018638 XUM:Scan powerful
714018641 XUM:Backing up by 1 level
714018641 XUM:Scan more_useful
714018643 XUM:Scan mouse
714018647 XUM:Evidence:
editors/sam/more_useful/mouse/snarf_k
714018652 XUM:Quit
714018652 Mon Aug 17 12:30:52 1992

714106346 Tue Aug 18 12:52:26 1992
714106346 XUM:Start
714106346 XUM:Rescan
714106350 XUM:Scan editors
714106351 XUM:Scan sam
714106587 XUM:Scan very_useful
714106589 XUM:Scan mouse
714106593 XUM:Backing up by 2 levels
714106593 XUM:Scan very_useful
714106594 XUM:Backing up by 1 level
714106594 XUM:Scan powerful
714106596 XUM:Scan commands
714106600 XUM:Backing up by 2 levels
714106600 XUM:Scan powerful
714106602 XUM:Scan mouse
714106607 XUM:Backing up by 2 levels
714106607 XUM:Scan powerful
714106609 XUM:Backing up by 4 levels
714106609 XUM:Rescan
714106612 XUM:QVOpen
714106625 XUM:QVClose
714106626 XUM:Quit
714106626 Tue Aug 18 12:57:06 1992

714348133 Fri Aug 21 08:02:13 1992
714348133 XUM:Start
714348133 XUM:Rescan
714348161 XUM:Scan editors
714348164 XUM:Scan sam
714348174 XUM:Scan powerful
714348178 XUM:Scan addresses
715211477 XUM:Start
715211477 XUM:Rescan
715211478 XUM:Scan editors
715211482 XUM:Scan sam
715211486 XUM:QVOpen
715211518 XUM:QVClose
715211520 XUM:Scan basics
715211522 XUM:Scan minimal
715211626 XUM:Backing up by 2 levels
715211626 XUM:Scan basics
715211628 XUM:Scan useful
715211641 XUM:Change:
editors/sam/basics/useful/quit_b->FALSE
715211647 XUM:Change:
editors/sam/basics/useful/quit_b->FALSE
715211649 XUM:Evidence:
editors/sam/basics/useful/quit_b
715211657 XUM:Change:
editors/sam/basics/useful/default_size_k->TRUE
715211667 XUM:Change:
editors/sam/basics/useful/quit_b->FALSE
715211673 XUM:Backing up by 2 levels
715211673 XUM:Scan basics
715211677 XUM:Backing up by 2 levels
715211677 XUM:Scan sam
715211679 XUM:Scan more_useful
715211681 XUM:Scan command_window
715211694 XUM:Evidence:
editors/sam/more_useful/
command_window/gotoline_k
715211700 XUM:Change:
editors/sam/more_useful/
command_window/gotoline_k->TRUE
715211707 XUM:Change:
editors/sam/more_useful/
command_window/write_k->TRUE
715211712 XUM:Change:
editors/sam/more_useful/
command_window/undo_k->TRUE
715211726 XUM:Explain:

editors/sam/more_useful/
command_window/load_new_k
715211749 XUM:Explain:
editors/sam/more_useful/
command_window/non_cmd_b
715211760 XUM:Backing up by 2 levels
715211760 XUM:Scan more_useful
715211762 XUM:Scan mouse
715211772 XUM:Explain: editors/sam/more_useful/mouse/look_k
715211799 XUM:Backing up by 1 level
715211799 XUM:Scan mouse
715211803 XUM:Backing up by 2 levels
715211803 XUM:Scan more_useful
715211805 XUM:Scan other
715211808 XUM:Explain:
editors/sam/more_useful/other/tabsize_k
715211821 XUM:Explain:
editors/sam/more_useful/other/highlight_esc_k
715211841 XUM:Backing up by 2 levels
715211841 XUM:Scan more_useful
715211843 XUM:Backing up by 2 levels
715211843 XUM:Scan sam
715211845 XUM:Scan very_useful
715211847 XUM:Scan command_window
715211849 XUM:Scan unix
715211858 XUM:Explain:
editors/sam/very_useful/
command_window/unix/pipe_k
715211876 XUM:Explain:
editors/sam/very_useful/
command_window/unix/red_in_k
715211886 XUM:Backing up by 2 levels
715211886 XUM:Scan command_window
715211887 XUM:Scan regexp
715211891 XUM:Scan unix
715211895 XUM:Backing up by 2 levels
715211895 XUM:Scan regexp
715211898 XUM:Scan sam_extra
715211903 XUM:Backing up by 2 levels
715211903 XUM:Scan regexp
715211905 XUM:Backing up by 1 level
715211905 XUM:Scan other
715211912 XUM:Explain:
editors/sam/very_useful/
command_window/other/subst_k
715211922 XUM:Backing up by 3 levels
715211922 XUM:Scan powerful
715211923 XUM:Scan commands
715211928 XUM:Backing up by 1 level
715211928 XUM:Scan commands
715211931 XUM:Backing up by 2 levels
715211931 XUM:Scan powerful
715211935 XUM:Scan mouse
715211942 XUM:Explain:
editors/sam/powerful/mouse/sendm_k
715473029 XUM:Start
715473029 XUM:Rescan
715473049 XUM:Scan editors
715473052 XUM:Scan sam
715473094 XUM:Scan very_useful
715473105 XUM:Backing up by 3 levels
715473105 XUM:Scan programming
715473107 XUM:Backing up by 1 level
715473107 XUM:Scan editors
715473109 XUM:Scan vi
715473116 XUM:Explain: editors/vi/uses_vi_c
715473123 XUM:Backing up by 2 levels
715473123 XUM:Scan programming
715473124 XUM:Scan languages
715473131 XUM:Backing up by 2 levels
715473131 XUM:Scan programming
715473132 XUM:Backing up by 1 level
715473132 XUM:Scan editors
715473150 XUM:Scan sam
715473152 XUM:Scan more_useful
715473153 XUM:Scan command_window
715473162 XUM:Explain:
editors/sam/more_useful/
command_window/load_new_k
715473174 XUM:Explain:
editors/sam/more_useful/
command_window/set_fname_k
715473195 XUM:Backing up by 1 level
715473195 XUM:Scan mouse
715473206 XUM:Change:
editors/sam/more_useful/mouse/snarf_k->TRUE
715473209 XUM:Change:
editors/sam/more_useful/mouse/cut_k->TRUE
715473213 XUM:Change:
editors/sam/more_useful/mouse/paste_k->TRUE
715473219 XUM:Change:
editors/sam/more_useful/mouse/look_k->TRUE
715473223 XUM:Change:
editors/sam/more_useful/mouse/submenu_k->TRUE
715473227 XUM:Backing up by 2 levels
715473227 XUM:Scan more_useful
715473229 XUM:Scan other
715473239 XUM:Backing up by 2 levels
715473239 XUM:Scan very_useful
715473242 XUM:Scan command_window
715473248 XUM:Scan addresses
715473257 XUM:Backing up by 1 level
715473257 XUM:Scan unix
715473266 XUM:Backing up by 1 level
715473266 XUM:Scan regexp
715473269 XUM:Scan sam_extra

715473274 XUM:Backing up by 2 levels
715473274 XUM:Scan regexp
715473276 XUM:Backing up by 1 level
715473276 XUM:Scan other
715473282 XUM:Explain:
 editors/sam/very_useful/
  command_window/other/subst_k
715473382 XUM:Backing up by 2 levels
715473382 XUM:Scan mouse
715473391 XUM:Change:
 editors/sam/very_useful/mouse/xerox_k->TRUE
715473393 XUM:Backing up by 2 levels
715473393 XUM:Scan mostly_useless
715473398 XUM:Backing up by 4 levels
715473398 XUM:Rescan
715473400 XUM:QVOpen
715473418 XUM:QVClose
715473421 XUM:QVOpen
715473460 XUM:QVClose
715473462 XUM:Quit
715473462 Thu Sep 3 08:37:42 1992

715558196 Fri Sep 4 08:09:56 1992
715558196 XUM:Start
715558196 XUM:Rescan
715559788 XUM:Start
715559788 XUM:Rescan
715559817 XUM:Scan editors
715559819 XUM:Scan sam
715559821 XUM:Scan basics
715559823 XUM:Scan minimal
715559836 XUM:Explain:
 editors/sam/basics/minimal/nonscroll_b
715559912 XUM:Change:
 editors/sam/basics/minimal/nonscroll_b->TRUE
715559917 XUM:Change:
 editors/sam/basics/minimal/nonscroll_b->FALSE
715559928 XUM:Backing up by 1 level
715559928 XUM:Scan minimal
715559938 XUM:Backing up by 1 level
715559938 XUM:Scan minimal
715559942 XUM:Backing up by 2 levels
715559942 XUM:Scan basics
715559944 XUM:Scan useful
715560188 XUM:Backing up by 2 levels
715560188 XUM:Scan more_useful
715560190 XUM:Quit
715560190 Fri Sep 4 08:43:10 1992

715989643 Wed Sep 9 08:00:43 1992
715989643 XUM:Start
715989644 XUM:Rescan
715989647 XUM:Quit
715989647 Wed Sep 9 08:00:47 1992

715991042 Wed Sep 9 08:24:02 1992
715991042 XUM:Start
715991042 XUM:Rescan
715991049 XUM:Scan editors
715991051 XUM:Scan sam
715991057 XUM:Scan basics
715991059 XUM:Scan useful
715991064 XUM:Backing up by 1 level
715991064 XUM:Scan useful
715991066 XUM:Backing up by 1 level
715991066 XUM:Scan minimal
715991078 XUM:Backing up by 2 levels
715991078 XUM:Scan more_useful
715991081 XUM:Scan command_window
715991088 XUM:Explain:
 editors/sam/more_useful/
  command_window/load_new_k
715991098 XUM:Explain:
 editors/sam/more_useful/
  command_window/set_fname_k
715991150 XUM:Backing up by 2 levels
715991150 XUM:Scan more_useful
715991153 XUM:Scan other
715991158 XUM:Backing up by 2 levels
715991158 XUM:Scan more_useful
715991160 XUM:Scan mouse
715991164 XUM:Backing up by 2 levels
715991164 XUM:Scan more_useful
715991166 XUM:Backing up by 1 level
715991166 XUM:Scan very_useful
715991169 XUM:Scan command_window
715991203 XUM:Scan addresses
715991211 XUM:Explain:
 editors/sam/very_useful/
  command_window/addresses/quote_k
715991221 XUM:Explain:
 editors/sam/very_useful/
  command_window/addresses/comma_k
715991235 XUM:Explain:
 editors/sam/very_useful/
  command_window/addresses/dot_k
715991252 XUM:Backing up by 2 levels
715991252 XUM:Scan command_window
715991254 XUM:Scan unix
715991267 XUM:Explain:
 editors/sam/very_useful/
  command_window/unix/pipe_k
715991298 XUM:Backing up by 2 levels
715991298 XUM:Scan command_window
715991301 XUM:Scan regexp
715991303 XUM:Scan sam_extra

715991307 XUM:Backing up by 2 levels
715991307 XUM:Scan regexp
715991309 XUM:Scan unix
715991313 XUM:Backing up by 1 level
715991313 XUM:Scan unix
715991315 XUM:Backing up by 2 levels
715991315 XUM:Scan regexp
715991317 XUM:Backing up by 3 levels
715991317 XUM:Scan very_useful
715991319 XUM:Backing up by 2 levels
715991319 XUM:Scan sam
715991320 XUM:Backing up by 3 levels
715991321 XUM:Rescan
715991322 XUM:Quit
715991322 Wed Sep 9 08:28:42 1992

716420567 Mon Sep 14 07:42:47 1992
716420567 XUM:Start
716420567 XUM:Rescan
716420964 XUM:Quit
716420964 Mon Sep 14 07:49:24 1992

716446767 Mon Sep 14 14:59:27 1992
716446767 XUM:Start
716446767 XUM:Rescan
716446768 XUM:Quit
716446768 Mon Sep 14 14:59:28 1992

716446781 Mon Sep 14 14:59:41 1992
716446781 XUM:Start
716446781 XUM:Rescan
716446783 XUM:Scan editors
716446785 XUM:Scan sam
716446792 XUM:Scan very_useful
716446793 XUM:Scan command_window
716446798 XUM:Scan other
716446801 XUM:Backing up by 1 level
716446801 XUM:Scan regexp
716446803 XUM:Backing up by 1 level
716446803 XUM:Scan regexp
716446807 XUM:Scan unix
716446812 XUM:Backing up by 2 levels
716446812 XUM:Scan regexp
716446826 XUM:Backing up by 3 levels
716446826 XUM:Scan very_useful
716446830 XUM:Scan command_window
716446832 XUM:Scan unix
716446858 XUM:Quit
716446858 Mon Sep 14 15:00:58 1992

716446911 Mon Sep 14 15:01:51 1992
716446911 XUM:Start
716446912 XUM:Rescan
716446916 XUM:Scan editors
716446917 XUM:Scan sam
716446919 XUM:Scan basics
716446921 XUM:Scan minimal
716446987 XUM:QVOpen
716447031 XUM:QVClose
716447184 XUM:Quit
716447184 Mon Sep 14 15:06:24 1992

716514082 Tue Sep 15 09:41:22 1992
716514082 XUM:Start
716514082 XUM:Rescan
716514084 XUM:Quit
716514084 Tue Sep 15 09:41:24 1992

716592423 Wed Sep 16 07:27:03 1992
716592423 XUM:Start
716592423 XUM:Rescan
716592424 XUM:Quit
Wed Sep 16 07:27:04 1992

717311983 Thu Sep 24 15:19:43 1992
717311983 XUM:Start
717311983 XUM:Rescan
717311996 XUM:Quit

719448572 Mon Oct 19 08:49:32 1992
719448572 XUM:Start
719448572 XUM:Rescan
719448578 XUM:Quit

719449302 Mon Oct 19 09:01:42 1992
719449302 XUM:Start
719449302 XUM:Rescan
719449303 XUM:Quit

720047953 Mon Oct 26 08:19:13 1992
720047953 XUM:Start
720047953 XUM:Rescan
720047956 XUM:Quit
720047956 Mon Oct 26 08:19:16 1992

References

G Abowd and R Beale, ‘‘Users, systems and interfaces: a unifying framework for interaction,’’ *HCI’91: People and Computers VI*, pp. 73-87, Cambridge University Press, Cambridge, 1991.

J R Anderson, *Rules of the mind,* Lawrence Erlbaum, NJ, 1993.

D Benyon, J Kay, and R C Thomas, ‘‘Building user models of editor usage,’’ in *Proceedings of the Third International Workshop on User Modeling - UM92*, ed. by E Andre, R Cohen, W Graf, B Kass, C Paris, and W Wahlster, pp. 113-132, IBFI (International Conference and Research Center for Computer Science), Schloss Dagstuhl, Wadern, Germany, August 9-13, 1992, 1992.

N S Borenstein, *Programming as if people mattered: friendly programs, software engineering, and other noble delusions,* Princeton University Press, Princeton, New Jersey, 1991.

G Brajnik, G Guida, and C Tasso, ‘‘User modeling in expert man-machine interfaces: a case study in intelligent information retrieval,’’ *IEEE Transactions on Systems, Man and Cybernetics*, vol. 20, no. 1, pp. 166-185, 1990.

G Brajnik and C Tasso, ‘‘A flexible tool for developing user modeling applications with nonmonotonic reasoning capabilities,’’ in *Proceedings of the Third International Workshop on User Modeling - UM92*, ed. by E Andre, R Cohen, W Graf, B Kass, C Paris, and W Wahlster, pp. 42-66, Deursches Forschungszentrum fur Kunstliche Intelligenz, 1992.

G Brajnik and C Tasso, ‘‘A shell for developing mon-monotonic user modeling systems,’’ *International Journal of Human-Computer Studies*, vol. 40, no. 1, pp. 36-62, 1994.

G Brajnik, C Tasso, and A Vaccher, ‘‘A shell for non-monotonic user modeling systems,’’ in *Proceedings of the International Joint Conference on Artificial Intelligence Workshop W.4: Agent Modeling for Intelligent Interaction*, ed. by J Kay and A Quilici, pp. 149-163, Sydney, Australia, 1990.

J S Brown and R R Burton, ‘‘Diagnostic models for procedural bugs in basic mathematical skills,’’ *Cognitive Science*, vol. 2, pp. 155-92, 1978.

J S Brown, R R Burton, and K M Larkin, ‘‘Representing and using procedural bugs for educational purposes,’’ *Proceedings of the 1977 Annual Conference ACM Seattle*, pp. 247-255, 1977.

J S Brown and K Van Lehn, ‘‘Repair theory: a generative theory of bugs in procedural skills,’’ *Cognitive Science*, vol. 4, pp. 379-426, l980.

D Browne, P Totterdell, and M Norman, in *Adaptive user interfaces*, p. 227pp, Academic Press, San Diego, California, 1990.

P Brusilovsky, ‘‘Student as user: Towards an adaptive interface for an intelligent learning environment,’’ in *Proceedings of the AI-ED’93, World Conference on Artificial Intelligence in Education*, ed. by P Brna, S Ohlsson, and H Pain, pp. 386-393, Charlottesville, 1993.

P Brusilovsky and E Schwarz, ‘‘User as student: towards an adaptive interface for advanced web-based applications,’’ in *User Modeling, Proceedings of the Sixth International Conference UM97*, ed. by A Jameson, C Paris, and C Tasso, Springer, Chia Laguna, Sardinia, Italy, 1997.

B G Buchanan and E H Shortliffe, in *Rule-based expert systems: the MYCIN experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, Reading, Mass, 1984.

S Bull, ‘‘See yourself write: a simple student model to make students think,’’ in *User Modeling, Proceedings of the Sixth International Conference UM97*, ed. by A Jameson, C Paris, and C Tasso, pp. 315 - 326, Springer, Chia Laguna, Sardinia, Italy, 1997.

S Bull and P Brna, ‘‘What does Susan know that Paul doesn’t? (and vice-versa): contributing to each other’s student model,’’ *International Conference on Artificial Intelligence in Education*, pp. 568 - 570, IOS Press, 1997.

S Bull, P Brna, and H Pain, ‘‘Extending the scope of the student model,’’ *User Modeling and User-Adapted Interaction*, vol. 5, no. 1, pp. 44 - 65, 1995.

S Bull and E Broady, ‘‘Spontaneous peer tutoring from sharing student models,’’ *International Conference on Artificial Intelligence in Education*, pp. 143 - 150, IOS Press, 1997.

S Bull and H Pain, ‘‘Did I say what I think I said, and do you agree with me? : inspecting and questioning the Student Model,’’ *Proceedings of World Conference on Artificial Intelligence in Education*, pp. 501 - 508, AACE, Washington DC, USA, 1995.

S Bull, H Pain, and P Brna, ‘‘Student modelling in an intelligent computer assisted language learning system: the issues of language transfer and learning strategies,’’ *Proceedings of the International Conference on Computers in Education*, pp. 121-126, Tapei, Tawain, 1993.

S Bull and M Smith, ‘‘Using targeted negotiation to support students learning,’’ *Proceedings of International Conference on Computers in Education*, pp. 173 - 181, Singapore, 1995.

A Bullock and O Stallybrass, *The Fontana Dictionary of Modern Thought*, 1977.

R R Burton and J S Brown, ‘‘An investigation of computer coaching for informal learning activities,’’ *International Journal of Man-Machine Studies*, vol. 11, pp. 5-24, 1979.

G Butler, *Unix coach*, Honours Thesis, Basser Dept of Computer Science, University of Sydney, 1992.

B Carr and I Goldstein, ‘‘Overlays: a theory of modelling for computer aided instruction,’’ AI-Memo 406, MIT, Cambridge, MA, 1977.

J M Carroll, *The Nurnberg Funnel: designing minimalist instruction for practical computer skill*, MIT Press, Cambridge, Mass, 1990.

D Chin, ‘‘KNOME: modeling what the user knows in UC,’’ in *User models in dialog systems*, ed. by A Kobsa and W Wahlster, pp. 74-107, Springer-Verlag, Berlin, 1989.

D N Chin, ‘‘User Modelling in UC, the UNIX Consultant,’’ *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pp. 24-28, ACM Press, 1986.

W J Clancey, in *Knowledge-based tutoring: the GUIDON program*, MIT Press, Cambridge, Mass, 1987.

P Cohen, *Heuristic reasoning about uncertainty: An artificial intelligence approach*, Pitman., 1985.

J A Collins, J E Greer, V S Kumar, G I McCalla, P Meagher, and R Tkatch, ‘‘Inspectable user models for just-in-time workplace training,’’ in *User Modeling, Proceedings of the Sixth International Conference UM97*, ed. by A Jameson, C Paris, and C Tasso, pp. 327-337, Springer, Chia Laguna, Sardinia, Italy, 1997.

R Cook, *Viewable individual user models for a text editor,* Honours Thesis, Basser Dept of Computer Science, University of Sydney, 1991.

R Cook and J Kay, ‘‘The justified user model: a viewable, explained user model,’’ in *Proceedings of the Fourth International Conference on User Modeling UM94*, ed. by A Kobsa and D Litman, pp. 145-150, MITRE, UM Inc, Hyannis, Massechusetts, USA, 1994.

R Cook and J Kay, *Tools for viewing um user models*, SSRG Report 93/3/50.1, Dept of Computer Science, University of Sydney, Australia, 1993.

R Cook, J Kay, G Ryan, and R C Thomas, ‘‘A toolkit for appraising the long term usability of a text editor,’’ *Software Quality Journal*, vol. 4, pp. 131-154, Chapman and Hall, London, 1995.

A T Corbett and J Anderson, ‘‘Knowledge tracing: modeling the acquisition of procedural knowledge,’’ *User Modeling and User-Adapted Interaction*, vol. 4, pp. 253 - 278, 1995.

K Crawford and J Kay, ‘‘Metacognitive processes and learning with intelligent educational systems,’’ in *Perspectives on Cognitive Science*, ed. by P Slezak, T Caelli, and R Clark, pp. 63-77, Ablex, 1993.

K Crawford and J Kay, *Shaping learning approaches with intelligent learning systems*, 3, pp. 1472-6, Proceedings of the International Conference for Technology in Education, France, 1992.

A Csinger, *User models for intent-based authoring*, PhD thesis, University of British Columbia, 1995.

P Dillenbourg and J Self, *A framework for learner modelling,* AI Report, Department of Computing, Lancaster University, 1990.

A Dix, J Finlay, G Abowd, and R Beale, *Human-computer interaction,* Prentice Hall, 1993.

A Dix and C Runciman, ‘‘Abstract models of interactive systems,’’ in *People and Computers : Designing the Interface*, ed. by P Johnson and S Cook, pp. 13-22, Cambridge University Press, 1985.

J Doyle, ‘‘A truth maintenance system,’’ *Artificial intelligence*, vol. 12, no. 3, pp. 231-171, 1979.

R O Duda, P E Hart, and N Nilsson, ‘‘Subjective bayesian methods for rule-based inference systems,’’ Technical Note 124, Artificial Intelligence Centre, SRI International, Menlo Park, Ca, 1976.

K D Eason, ‘‘Towards the Experimental Study of Usability,’’ *Behaviour and Information Technology*, vol. 3, pp. 133-143, 1984.

M Elsom-Cook, ‘‘Student modelling in intelligent tutoring systems,’’ *Artificial Intelligence Review*, vol. 7, no. 3-4, pp. 227-240, 1993.

S Elzer, J Chu-Carroll, and S Carberry, ‘‘Recognizing and utilizing user preferences in collaborative consultation dialogues,’’ *Proceedings of the Fourth International Conference on User Modeling UM94*, pp. 19-24, MITRE, User Modeling Inc, Hyannis, Massechusetts, USA, 1994.

T W Finin, ‘‘GUMS - a general user modeling shell,’’ in *User models in dialog systems*, ed. by A Kobsa and W Wahlster, pp. 411-431, Springer-Verlag, Berlin, 1989.

J Fink, ‘‘A flexible and open architecture for the user modeling shell system BGP-MS,’’ *Doctorial symposium, Proceedings of UM96, Fifth International Conference on User Modeling*, pp. 237-239, User Modeling Inc, Kailua-Kona, Hawaii, 1996.

J Fink and J Hohle, ‘‘Mechanisms for flexible representations and use of knowledge in user modeling shell systems,’’ in *User Modeling, Proceedings of the Sixth International Conference UM97*, ed. by A Jameson, C Paris, and C Tasso, pp. 403-414, Springer, Chia Laguna, Sardinia, Italy, 1997.

G Fisher and D Ackerman, ‘‘The importance of models in making complex systems comprehensible,’’ in *Mental models and Human-computer Interaction 2*, ed. by M J Tauber, pp. 23-33, Elsevier, 1991.

J Galliers, ‘‘Autonomous belief revision and communication,’’ in *Belief Revision*, ed. by Gardenfors, Cambridge University Press, 1992.

N Parandeh Gheibi and J Kay, ‘‘Supporting a coaching system with viewable learner models,’’ in *Proceedings of the International Conference for Computers Computer Technologies in Education*, ed. by V Petrushin and A Dovgiallo, pp. 140-141, Kiev, Ukraine, 1993.

D Goldberg, D Nichols, B M Oki, and D Terry, ‘‘Using collaborative filtering to weave an information tapestry,’’ *Communications of the ACM*, vol. 35, no. 12, pp. 61-70, 1992.

I P Goldstein, ‘‘The genetic graph: a representation for the evolution of procedural knowledge,’’ in *Intelligent tutoring systems*, ed. by D Sleeman and J S Brown, Academic Press, New York, l982.

J E Greer and G I McCalla, ‘‘A computational framework for granularity and its application to educational diagnosis,’’ *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 477-482, Detroit, 1989.

G Harman, *Change in view: principles of reasoning*, MIT Press, Cambridge, Mass, 1986.

A Henderson and M Kyng, ‘‘From customisable systems to intelligent agents,’’ in *Readings in Human-Computer Interaction*, ed. by R M Baeker, J Grudin, W Buxton, and S Greenberg, pp. 783-792, Morgan Kaufman, 1995.

K Hook, J Karlgren, A Waern, N Dahlback, C G Jansson, K Karlgren, and B Lemaire, ‘‘A glass box approach to adaptive hypermedia,’’ *User Modeling and User-Adapted Interaction*, vol. 6, no. 2-3, pp. 157-184, 1996.

E Horvitz, ‘‘Agents with beliefs: reflections on bayesian methods for user modeling,’’ *User Modeling, Proceedings of the Sixth International Conference UM97*, pp. 441-2, Springer-Verlag, New York, 1997.

X Huang, ‘‘Modelling a student’s inconsistent beliefs and attention,’’ in *Student Modelling: The Key to Individualized Knowledge-Based Instruction*, ed. by J E Greer and G I McCalla, pp. 267-280, Springer-Verlag, 1994.

X Huang, G I McCalla, J E Greer, and E Neufeld, ‘‘Revising deductive knowledge and stereotypical knowledge ina student model,’’ *User Modeling and User-Adapted Interaction*, vol. 1, no. 1, pp. 87-116, 1991.

M Ikeda and R Mizoguchi, ‘‘FITS: a framework for ITS - a computational model of tutoring,’’ *Journal of Artificial Intelligence in Education*, vol. 5, no. 3, pp. 319-348, 1994.

A Jameson, C Paris, and C Tasso, ‘‘Reader’s Guide,’’ in *User Modeling, Proceedings of the Sixth International Conference UM97*, Springer-Verlag, New York, 1997.

W L Johnson and E Soloway, ‘‘PROUST,’’ *Byte*, vol. 10, no. 4, 1985.

P W Jordan, S W Draper, K K MacFarlane, and S McNulty, ‘‘Guessability, Learnability, and Experienced User Performance,’’ in *HCI ’91 People and Computers VI: Usability Now!*, ed. by D Diaper and N Hammond, pp. 237-245, Cambridge University Press, 1991.

T Kamba, K Bharat, and M Albers, *The Krakatoa Chronicle: an interactive personalized newspaper on the web*, pp. 159-170, 1995.

R Kass, ‘‘Building a user model implicitly from a cooperative advisory dialog,’’ *User Modeling and User-Adapted Interaction*, vol. 1, no. 3, pp. 203-258, 1991.

J Kay, ‘‘An explicit approach to acquiring models of student knowledge,’’ in *Proceedings of ARCE, Advanced Research on Computers and Education*, ed. by R Lewis and S Otsuki, pp. 263-268, 1990a.

J Kay, ‘‘An explicit approach to acquiring models of student knowledge,’’ in *Advanced Research on Computers and Education*, ed. by R Lewis and S Otsuki, pp. 263-268, Elsevier, North Holland, 1991.

J Kay, ‘‘Interactive student modelling using concept mapping,’’ *Proceedings of the First Aust Artificial Intelligence Congress*, 1986.

J Kay, ‘‘um: a user modelling toolkit,’’ *Second International User Modelling Workshop*, p. 11, Hawaii, 1990b.

J Kay, ‘‘The um toolkit for cooperative user modelling,’’ *User Modeling and User-Adapted Interaction*, vol. 4, no. 3, Kluwer, 1995.

J Kay and R J Kummerfeld, ‘‘Customization and Delivery of Multimedia Information,’’ in *Proceedings of Multicomm 94, Vancouver*, pp. 141-150, 1994.

J Kay and R J Kummerfeld, ‘‘User Models for Customized Hypertext,’’ in *Advances in hypertext for the World Wide Web*, ed. by J Mayfield and C Nicholas, pp. 47-69, Springer Verlag, 1997.

J Kay and R C Thomas, ‘‘Studying long term system use,’’ *Communications of the ACM*, vol. 4, no. 2, pp. 131-154, ACM, 1995.

J Kay and R C Thomas, ‘‘Visualisation of entrenched user preferences,’’ *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pp. 285-286, ACM Press, Addison Wesley, Vancouver, BC Canada, 1996.

G P Kearsley, *Artificial Intelligence & Instruction - Applications and Methods*, Addison-Wesley, 1987.

B W Kernighan and P J Plauger, *Software Tools*, Addison-Wesley, Reading, Mass, 1976.

J de Kleer, ‘‘An assumption-based truth-maintenance system,’’ *Artificial Intelligence*, vol. 28, pp. 127-162, 1986.

J de Kleer, ‘‘An assumption-based TMS,’’ in *Readings in non-monotonic reasoning*, ed. by M L Ginsberg, pp. 280-297, Morgan Kaufman, 1987.

A Kobsa, ‘‘User modeling in dialog systems: potentials and hazards,’’ *AI and Society*, vol. 4, pp. 214-231, Springer-Verlag, 1990a.

A Kobsa, ‘‘Modeling the user’s conceptual knowledge in BGP-MS, a user modeling shell system,’’ *Computational Intelligence*, vol. 6, no. 4, pp. 193-208, 1990b.

A Kobsa, ‘‘Towards inferences in BGP-MS: combining modal logic and partition hierarchies for user modeling,’’ in *Proceedings of the Third International Workshop on User Modeling - UM92*, ed. by E Andre, R Cohen, W Graf, B Kass, C Paris, and W Wahlster, pp. 35-41, Deursches Forschungszentrum fur Kunstliche Intelligenz, 1992.

A Kobsa, ‘‘A standard for the performatives in the communication between applications and user modeling systems,’’ *ftp://ftp.informatik.uni-essen.de/pub/UMUAI/others/rfc.ps*, 1996.

A Kobsa, T Kuhme, and U Malinowski, ‘‘User modeling: recent work, prospects and hazards,’’ in *Adaptive user interfaces - principles and practice*, ed. by M Schneider-Hufschmidt, pp. 111-128, North Holland, 1993.

A Kobsa and W Pohl, ‘‘The user modeling shell system BGP-MS,’’ *User Modeling and User-Adapted Interaction*, vol. 4, no. 2, pp. 59-106, Kluwer, 1995.

A Kobsa and W Wahlster, in *User models in dialog systems*, Springer-Verlag, Berlin, 1989.

Y Kono, M Ikeda, and R Mizoguchi, ‘‘To contradict is human: student modelling of inconsistency,’’ in *Intelligent tutoring systems*, ed. by C Frasson, G Gauthier, and G McCalla, pp. 451-458, Springer-Verlag, 1992.

Y Kono, M Ikeda, and R Mizoguchi, ‘‘THEMIS: a nonmonotonic inductive student modeling system,’’ *Journal of Artificial Intelligence in Education*, vol. 5, no. 3, pp. 371-413, 1994.

D Kupper, ‘‘Plan processing in user models,’’ in *Doctorial symposium, User Modeling, Proceedings of the Sixth International Conference UM97*, ed. by A Jameson, C Paris, and C Tasso, pp. 447-448, Springer, Chia Laguna, Sardinia, Italy, 1997.

R W Lawler and M Yazdani, in *Journal of Artificial Intelligence in Education*, Ablex, 1987.

M J Lawson, ‘‘Being executive about meta-cognition,’’ *Cognitive Strategies and Educational Performance*, Academic Press, Orlando, 1984.

S Maass, ‘‘Why systems transparency?,’’ in *The psychology of computer use*, ed. by T R G Green, S J Payne, and G C Van der Veer, pp. 19-28, Academic Press, 1983.

R Mack, ‘‘Understanding and learning text-editing skills: observations on the role of the new user expectations,’’ in *Cognition, computing and cooperation*, ed. by S P Robertson, W Zachary, and J B Black, pp. 304-337, Ablex, Norwood, New Jersery, 1990.

U Malinowski, T Kuhme, D Dieterich, and M Schneider-Hufschmidt, ‘‘A Taxonomy of Adaptive User Interfaces,’’ in *People and Computers VII*, ed. by A Monk, D Diaper, and M D Harrison, pp. 391-414, Cambridge University Press, 1992.

H Mandl and A Lesgold, *Learning Issues for Intelligent Tutoring Systems*, Springer-Verlag, New York, 1988.

J P Martins and S C Shapiro, ‘‘A model for belief revision,’’ *Artificial Intelligence*, vol. 35, no. 1, pp. 25-79, 1988.

G I McCalla and J E Greer, in *Student Modelling: The Key to Individualized Knowledge-Based Instruction*, Springer-Verlag NATO ASI Series, 1994a.

G I McCalla and J E Greer, ‘‘Granularity-based reasoning and belief revision in student models,’’ in *Student Modelling: The Key to Individualized Knowledge-Based Instruction*, ed. by J E Greer and G I McCalla, pp. 39-62, Springer-Verlag, 1994b.

C Miller and K Swift, in *The handbook of nonsexist writing for writers, editors and speakers*, Lippincott and Crowell, New York, 1980.

M Milosavljevic, ‘‘Augmenting the user’s knowledge via comparison,’’ in *User Modeling, Proceedings of the Sixth International Conference UM97*, ed. by A Jameson, C Paris, and C Tasso, pp. 119-130, Springer, Chia Laguna, Sardinia, Italy, 1997.

R Mizoguchi and M Ikeda, ‘‘A generic framework for ITS and its evaluation,’’ in *Advanced Research on Computers in Education*, ed. by R Lewis and S Otsuki, pp. 63-72, North Holland, 1991.

K J Mock and V R Vemuri, ‘‘Adaptive user models for intelligent information filtering,’’ in *http://www.glue.umd.edu/unee/medlab/filter/gwic.ps*.

M Morita and Y Shinoda, ‘‘Information filtering based on user behaviour analysis and best match retrieval,’’ in *Proceedings of the Seventeenth ACM/SIGIR Conf on Research and Development in*, ed. by W B Croft and C van Rijsbergen, pp. 272-281, 1994.

A Motro and P Smets, *Uncertainty management in information systems,* Kluwer, Boston/London/Dordrecht, 1997.

M Murphy, ‘‘Learner modelling for intelligent CALL,’’ in *User Modeling, Proceedings of the Sixth International Conference UM97*, ed. by A Jameson, C Paris, and C Tasso, pp. 301 - 312, Springer, Chia Laguna, Sardinia, Italy, 1997.

W R Murray, ‘‘An endorsement-based approach to student modelling for planner-contolled tutors,’’ *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1100-1106, Morgan Kaufman, 1991.

L R Neal, ‘‘Cognition-sensitive design and user modelling for syntax-directed editors,’’ in *Proceedings of the 2nd International Conference on Human-Computer Interaction*, ed. by G Salveny, vol. 11, pp. 99-102, Honolulu, 1987.

L R Neal, ‘‘The role of user models in systems design,’’ TR-18-89, Center for research in computing technology, Harvard University, 1989.

N Negroponte, *Being Digital,* Hodder and Stoughton, 1995.

William Newman and Michael Lamming, *Interactive system design,* Addison-Wesley, 1995.

J Nielsen, *Usability Engineering*, Academic Press, Harcourt Brace, 1993.

J Nielsen, ‘‘Estimating the number of subjects needed for a thinking aloud test,’’ *International Journal of Human-Computer Studies*, vol. 41, no. 1-6, pp. 385-397, 1994.

R E Nisbett and T D Wilson, ‘‘Telling more than we can know: verbal reports on mental processes,’’ *Psychological Review*, vol. 84, pp. 231 - 259, 1977.

D A Norman, ‘‘Some observations on mental models,’’ in *Mental models*, ed. by D Gentner and A L Stevens, Lawrence Erlbaum, 1983.

D A Norman, ‘‘Cognitive engineering,’’ in *User centered system design*, ed. by D A Norman and S W Draper, pp. 31-65, Lawrence Erlbaum, 1986.

J Orwant, ‘‘Apprising the user of user models: Doppelganger’s interface,’’ *Proceedings of the Fourth International Conference on User Modeling*, pp. 151 - 156, MITRE, User Modeling Inc, Hyannis, Massechusetts, USA, 1994.

J Orwant, ‘‘The Doppelganger user modeling system,’’ in *Proceedings of the International Joint Conference on Artificial Intelligence Workshop W.4: Agent Modeling for Intelligent Interaction*, ed. by J Kay and A Quilici, pp. 164-168, Sydney, Australia, 1990.

J Orwant, *Doppelganger goes to school: machine learning for user modeling*, MIT MS Thesis, MIT Media Laboratory, 1993.

J Orwant, ‘‘Heterogenous learning in the Doppelganger user modeling system,’’ *User Modeling and User-Adapted Interaction*, vol. 4, no. 2, pp. 59-106, Kluwer, 1995.

H Pain, S Bull, and P Brna, ‘‘A student model for its own sake,’’ *Proceedings of European Conference on Artificial Intelligence in Education*, pp. 191 - 198, Lisbon, 1996.

A Paiva, ‘‘Towards a consensus on the communication between user modeling agents and application agents,’’ *Workshop on Standardisation of User Modeling Shells in the International Conference on User Modeling (UM96)*, Kailua-Kona, Hawaii, 1996.

A Paiva and J Self, ‘‘TAGUS - a user and learner modeling workbench,’’ *User Modeling and User-Adapted Interaction*, vol. 4, no. 3, pp. 197-228, Kluwer, 1995.

A Paiva, J Self, and R Hartley, ‘‘Externalising learner models,’’ *Proceedings of World Conference on Artificial Intelligence in Education*, pp. 509 - 516, AACE, Washington DC, 1995.

C Paris, M R Wick, and W B Thompson, ‘‘The line of reasoning versus the line of explanation,’’ *Proceedings of the AAAI Workshop on explanation*, pp. 4-7, 1988.

C L Paris, ‘‘The user of explicit user models in a generation system for tailoring answers to the user’s level of expertise,’’ in *User models in dialog systems*, ed. by A Kobsa and W Wahlster, pp. 200-232, Springer-Verlag, Berlin, 1989.

R Pike, ‘‘The Text Editor sam,’’ *Software Practice and Experience*, vol. 17, pp. 813-845, 1987.

M C Polson and J J Richardson, *Foundations of Intelligent Tutoring Systems*, Erlbaum, 1988.

Jenny Preece, Yvonne Rogers, H Sharp, D Benyon, S Holland, and T Carey, *Human-computer interaction*, Addison-Wesley, 1994.

J Psotka, L D Massey, and S A Mutter, *Intelligent Tutoring Systems: Lessons Learned*, Erlbaum, Hillsdale, N J, 1988.

P Resnick and H Varian, ‘‘Recommender Systems, Special issue,’’ *Communications of the ACM*, vol. 40, no. 30, 1997.

E Rich, ‘‘User modeling via stereotypes,’’ *Cognitive Science*, vol. 3, pp. 355-66, 1979.

E Rich, ‘‘Users are individuals: individualizing user models,’’ *International Journal of Man-Machine Studies*, vol. 18, pp. 199-214, 1983.

E Rich, ‘‘Stereotypes and user modeling,’’ in *User models in dialog systems*, ed. by A Kobsa and W Wahlster, pp. 35-51, Springer-Verlag, Berlin, 1989.

M B Rosson, ‘‘Patterns of Experience in Text Editing,’’ in *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, ed. by A Janda, pp. 171-175, North-Holland, 1984.

J Self, ‘‘Bypassing the Intractable Problem of Student Modelling: Invited paper,’’ in *Proceedings of the 1st International Conference on Intelligent Tutoring Systems*, pp. 18 - 24, Montreal, 1988.

J Self, ‘‘Formal approaches to student modelling,’’ in *Student Modelling: The Key to Individualized Knowledge-Based Instruction*, ed. by J Greer and G McCalla, pp. 295 - 352, Springer-Verlag, 1994.

U Shardanand and P Maes, ‘‘Social Information Filtering: Algorithms for Automating ’Word of Mouth’,’’ *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, Denver, CO, 1995.

E H Shortliffe and D G Buchanan, ‘‘A model of inexact reasoning in medicine,’’ *Mathematical Biosciences*, vol. 23, pp. 351-379, 1975.

D Sleeman, ‘‘UMFE: a user modelling front-end subsystem,’’ *International Journal of Man-Machine Studies*, vol. 23, pp. 71-88, 1985.

E Soloway, K Ehrlich, J Bonar, and J Greenspan, ‘‘What do novices know about programming,’’ in *Directions in Human-Computer Interactions*, ed. by B Shneiderman and A Badre, Ablex, 1982.

R W Southwick, ‘‘Explaining reasoning: an overview of explanation in knowledge-based systems,’’ *The Knowledge Engineering Review*, vol. 6, no. 1, pp. 1-19, 1991.

L Strachan, J Anderson, M Sneesby, and M Evans, ‘‘Pragmatic user modelling in a commercial software system,’’ *User Modeling, Proceedings of the Sixth International Conference UM97*, pp. 189-200, Springer, Chia Laguna, Sardinia, Italy, 1997.

L Suchman, in *Plans and situated actions*, Cambridge University Press, Cambridge, UK, 1987.

P Szolovits and S G Pauker, ‘‘Categorical and probabilistic reasoning in medical diagnosis,’’ *Artificial Intelligence*, vol. 11, pp. 115-144, 1978.

P Tamir, ‘‘What do learning theories and research have to say to practitioners in science education?,’’ in *Research and Development in Higher Education*, ed. by J Lublin, vol. 7, pp. 162-166, 1984.

Harold Thimbleby, *User interface design,* Addison-Wesley, 1990.

R C Thomas, *Long term human computer interaction: an exploratory perspective,* Springer-Verlag, 1998.

H Vergara, ‘‘PROTUM - A Prolog based tool for user modeling,’’ Bericht 55/94 (WIS Memo 10), Konstanz University, 1994.

S Volet and J Lawrence, ‘‘Goals in the adaptive learning of university students.,’’ in *Learning and Instruction*, ed. by H Mandl, E De Corte, N Bennett, and H F Friedrich, pp. 497-516, Pergamon, 1990.

W Wahlster and A Kobsa, ‘‘Dialogue-based user models,’’ *Proceedings of the IEEE*, vol. 74, no. 7, pp. 948-960, 1986.

E Wenger, *Artificial Intelligence and tutoring systems - computational and cognitive approaches to the communication of knowledge*, Morgan Kaufmann, Los Altos, 1987.

S Yussen, ‘‘The role of metacognition in contemporary theories of cognitive development,’’ in *Metacognition, Cognition and Human Performance*, ed. by D Forrest-Pressley, vol. 1, Academic Press, Orlando, 1985.

L A Zadeh, ‘‘Fuzzy sets as a basis for a theory of possibility,’’ in *Fuzzy sets and systems*, North-Holland, Amsterdam, 1978.

I Zukerman and R McConachy, ‘‘Generating concise discourse that addresses a user’s inferences,’’ *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1202-1207, Chambery, France, 1993.