15 GUMS — A General User Modeling Shell

Timothy W. Finin

Abstract

This chapter discusses the application of various kinds of default reasoning in system maintained models of users. In particular, we describe the general architecture of a domain independent system for building and maintaining *long term models* of *individual users*. The user modeling system is intended to provide a well defined set of services for an *application system* interacting with various users, and must build and maintain models of them. As the application system interacts with a user, it can acquire knowledge of him, and pass that knowledge on to the user model maintenance system for incorporation. We describe a prototype *general user modeling shell* (hereafter called *GUMS*) that we have implemented in Prolog. This system possesses some of the desirable characteristics we discuss.

1. Introduction — The Need for User Modeling

Systems that attempt to interact with people in an intelligent and cooperative manner need to know many things about the individuals with whom they are interacting. Such knowledge can be of several different varieties and can be represented and used in a number of different ways. Taken collectively, the information that a system has about its users is typically referred to as its *user model.* This is so even when it is distributed throughout many components of the system.

Examples that we have been involved with include systems that attempt to provide help and advice [FINI83, HOWE84, SHRA82], tutorial systems [SCHU86b], and natural language interfaces [WEBB84]. Each of these systems had to represent information about individual users. Most of the information was acquired incrementally through direct observation and/or interaction. These systems also had to infer additional facts about their users based on the directly acquired information. For example, the WIZARD help system [FINI83, SHRA82] had to represent which VMS operating system objects (e.g. commands, command qualifiers, concepts, etc.) a user was familiar with, and had to infer which other objects he was likely to be familiar with.

This chapter describes the general architecture for a domain independent system for building and maintaining *long term models* of *individual users.* The user modeling system is intended to provide a well defined set of services for an *application system* that interacts with various users, and needs to build and maintain models

of them. As the application system interacts with a user, it can acquire knowledge about him and pass that knowledge on to the user model maintenance system for incorporation. We describe a prototype *general user modeling shell* (hereafter called *GUMS*) implemented in Prolog. This system possesses some of the desirable characteristics we discuss.

The next section will discuss some of the general issues that arise in building user models. In particular, we will describe the range of possible issues that one might address in terms of four aspects: who is being modeled; what information is represented in the model; how the model is acquired and maintained, and how the model might be used. The third section provides an overview of the *GUMS* system and the kind of user modeling it supports. The fourth section describes the three kinds of default reasoning that *GUMS* employs: stereotypical reasoning, explicit default rules and failure as negation. Section five describes some of the details of the *GUMS* system, while the concluding section discusses some of the limitations of *GUMS* and avenues for future research.

2. What Kind of User Model?

The concept of incorporating user models into interactive systems has become common, but what has been meant by a user model has varied and is not always clear. In this section we will discuss, in general terms, what might be meant by the term *user model.* We will then characterize the approach to user modeling described in this work by answering the following questions: who is being modeled; what aspects of the user are being modeled; how is the model to be initially acquired and maintained; and why the model is needed.

The term "user model" has been used in many different contexts to describe knowledge that is used to support an interactive system. A survey of user modeling in support of natural language interfaces can be found in [KASS88b] and a discussion of user modeling for intelligent tutoring systems is provided by [KASS*]. In this section, we will look at a number of distinctions that will allow us to focus on an interesting and important class of "user models".

An initial definition for "user model" might be the following:

A user model is that knowledge about the user, either explicitly or implicitly encoded, which is used by the system to improve the interaction.

The definition is too weak, since it endows every interactive system with some kind of user model, usually of the implicit variety. In this chapter, we will focus our attention on those models where the information about the user is *explicitly* encoded. In particular, we are interested in user models that are designed along the lines of "knowledge bases". By this we have in mind the kinds of distinctions that are usually drawn to distinguish a *knowledge base* from a *data base.* In the context of user models, five features are important:

1. *Separate Knowledge Base* – Information about a user is collected in a separate module rather then distributed throughout the system.

2. *Explicit Representation* – The knowledge in the user model is encoded in a sufficiently expressive representation language. Such a representation language will typically provide a set of inferential services. That allows the knowledge of a user to be implicit, but automatically inferred when needed.
3. *Declarative rather than Procedural* – The knowledge in the user model is, as much as possible, encoded in a declarative rather than procedural manner.
4. *Support for Abstraction* – The modeling system provides ways to describe abstract as well as concrete entities. For example, the system might be able to discuss classes of users and their general properties as well as individuals.
5. *Multiple Use* – Since the user model is explicitly represented as a separate module, it can be used in several different ways (e.g. to support a dialog or to classify a new user). This requires that the knowledge be represented in a more general way that does not favor one use at the expense of another. It is highly desirable to express the knowledge in a way that allows it to be reasoned about as well as reasoned with. Similarly, the model ought to provide no barriers to its use by several different applications.

User models having these features fit nicely into current work in the broader field of *knowledge representation.* In fact, we could paraphrase Brian Smith's *knowledge representation hypothesis* [SMIB82] to get something like:

Any user model will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge the system has of the user and b) independent of such external semantical attribution, play a formal but causal and essential role in the behavior that manifests that knowledge.

In the remainder of this section we will explore the large space of user modeling possibilities by addressing the *who, what, how* and *why* of user modeling. Our own system will then be placed within this space.

2.1. Who Is Being Modeled?

Two aspects of who is being modeled are the degree of specialization and temporal extent. The degree of specialization involves whether we are modeling classes of users or individuals. Temporal extent refers to the persistence of the knowledge encoded in the user model – does it expire at the end of the current session or does it form a long-term knowledge base about the user.

A user model can lie somewhere on a specialization scale that ranges from *generic* to *individual.* A generic user model assumes a homogeneous set of users – all individuals using the system are similar enough with respect to the application that they can be treated as the same type of user. Many natural language systems, for example, maintain a single generic user model in order to infer the user's goals, or to recognize a user's misconceptions [ALLE82b, CARB83, CARN83c, GERS81]. A somewhat more individuating point on this scale is to employ a set of fixed, generic models to represent different subclasses of users. For example, the UC

system [WILE86, CHIN86, CHIN*] classifies a user as belonging to one of four possible generic classes: *novice, beginner, intermediate, or expert.*

Modeling the very general beliefs held by large classes of users is extremely important when building intelligent systems. For example, in modeling a person's beliefs about a timeshared computer system we may want to include a rule like:

If a user U believes that machine M is running, then U will believe that it is possible for him to log onto M.

This sort of rule is required to support the kinds of cooperative interactions studied in [JOSH84a, JOSH84b], such as the following:

```
User:   Is linc.cis.upenn.edu up?
System: Yes, but you can't log on now.
        Preventative maintenance is being done until 11:00am.
```

Individual user models, on the other hand, contain information specific to a single user. A user modeling system that keeps individual models thus will have a separate model for each user of the system. A simple example of this kind of model is the commonly used convention of customizing the behavior of a utility such as an editor or mail system by allowing each user to have a *profile* file that states his preferences for various system parameters [RICH83].

A natural way to combine the system's knowledge about classes of users with its knowledge of individuals is through the use of *stereotypes* – generic models with specializations for individuals. A stereotype is a cluster of default facts about the user that tend to be related to each other. One can model an individual as belonging to a particular stereotype (e.g. *novice Unix User*) with a set of overriding facts (e.g. *knows how to use pipes*). Examples of systems that have used stereotypes include GRUNDY [RICH79b], and the Real-Estate Advisor [MORI85b].

2.2. What Is to Be Modeled?

The *contents* of a user model naturally vary from application to application. Following Kass and Finin [KASS*], we will classify this knowledge into four categories: goals and plans, capabilities, attitudes, and knowledge or belief.

A user's goals and plans are intimately related to one another. The user's *goal* is a state of affairs he wishes to achieve, while a *plan* is a sequence of actions or events that he expects to result in the realization of a goal. Furthermore, each step in a plan has its own subgoal to achieve, which may be realized by yet another subplan. One of the hallmarks of an intelligent, cooperative system is that it attempts to help the user achieve his underlying goals, even when not explicitly stated. The principle way for one agent to know another's underlying goal(s) is by recognizing his observable actions as constituting a (possibly partial) plan for achieving a domain-relevant goal. A great deal of work has been done on the problem of *plan recognition* in support of cooperative interactions. Examples in the natural language domain include Allen and Perrault [ALLE80] and Carberry [CARB83, CARB88, CARB*]. Intelligent Tutoring Systems have introduced the idea of keeping a catalog of "buggy" plans [BROW78, SLEE82a, JOHN84] as a means of recognizing mis-

conceptions or missing knowledge. The use of plan recognition is also relevant to the problem of providing help and advice, as in [FINI83, WILE86].

A second category of knowledge to be modeled has to do with the user's capabilities. Although some systems might have to reason about a user's physical capabilities, such as the ability to perform some action that the system may recommend, it is more typical for a system to model and reason about mental capabilities, such as the ability to understand a recommendation or explanation provided by the system. Examples of the latter often arise when a system is generating an *explanation*, as in Wallis and Shortliffe [WALL82] and Paris TAILOR system [PARI88]. Webber and Finin [WEBB84] have surveyed ways that an interactive system might reason about its user's capabilities to improve the interaction.

A third category of knowledge represents a user's bias, preferences and attitudes. GRUNDY [RICH79b] was an early user modeling system that used a model of a user's preferences to recommended books to read. The Real-Estate Advisor [MORI85b] and HAM-ANS [HOEP83b, MORI*] are other systems which have tried to model this. Swartout [SWAR83] and McKeown [MCKE85a] address the effects of the user's *perspective* or *point of view* on the explanations generated by a system.

The final category of information we will consider is *knowledge* and *belief*.[1] Modeling the user's knowledge involves a variety of things: domain knowledge, general world knowledge, and knowledge about other agents. Knowing what the user believes to be true about the application domain is especially useful for many types of systems. In generating responses, knowledge of the concepts and terms the user understands or is familiar with allows the system to produce responses incorporating those concepts and terms, while avoiding concepts the user might not understand. This is particularly true for intelligent help systems [FINI83, WILE86], which must provide clear, understandable explanations to be truly helpful. Providing definitions of data base items (such as the TEXT system does [MCKE85c]) has a similar requirement to express the definition at a level of detail and in terms the user understands.

A final form of user knowledge that is important for interactive systems is knowledge about other agents. As an interaction with a user progresses, not only will the system be building a model of the beliefs, goals, capabilities and attitudes of the user, the user will also be building a model of the system. Sidner and Israel [SIDN81b] make the point that when individuals communicate, the speaker will have an *intended meaning*, consisting of both a *propositional attitude* and the *propositional content* of utterance. The speaker expects the hearer to recognize the intended meaning, even though it is not explicitly stated. Thus a system must reason about what model the user has of the system when making an utterance, because this will affect what the system can conclude about what the user intends the system to understand by the user's statement. Kobsa [KOBS84] has studied some of the difficult representational problems involved with building recursive models of the beliefs of other agents.

---

[1] In the context of modeling other individuals, an agent does not have access to objective truth and hence cannot really distinguish whether a proposition is *known* or simply *believed to be true*. Thus the terms *knowledge* and *belief* will be used interchangeably.

2.3. How Is the Model to Be Acquired and Maintained?

The acquisition and maintenance of the user model are closely related topics. By acquisition, we mean the techniques for learning new facts about the user. Maintenance involves incorporating this new information into the existing model and squaring away any discrepancies or contradictions. We will briefly review some of the possibilities for these two related problems.

The knowledge in a user model can be acquired either *explicitly* or *implicitly*. Explicitly acquired knowledge might come directly from the system designer in the form of built in knowledge of stereotypes, from the application system the user model is serving, or from the user himself. GRUNDY, for example, begins with a new user by interviewing him for certain information. From this initial data, GRUNDY makes judgments about which stereotypes most accurately fit the user, thus forming an opinion about the preferences of the user based on this initial list of attributes. As another example, each time the UMFE system [SLEE85] needs to know a new fact about its user, it simply asks.

Acquiring knowledge about the user implicitly is usually more difficult. Implicit acquisition involves "eavesdropping" on the user-system interaction in order to observe the user's behavior and from it to infer facts that go into the model. Various aspects of implicit acquisition have been explored, including the use of presuppositions [KAPS82], misconception recognition [MCCO85a, MCCO*], stereotype selection [RICH79b, RICH*, MORI85b], and the use of default rules [PERR88, KASS87a, KASS87b].

Maintenance involves incorporating new knowledge about an individual user into an existing model. The new, incoming information might be consistent or inconsistent with the current model. If it is consistent, maintenance involves triggering inferences to add additional facts to the model. If the new knowledge is inconsistent with the current model, then the inconsistency must be resolved in some manner. The possibilities depend on the underlying representation and reasoning systems on which the modeling system is built. Two major approaches are *evidential reasoning*, which allows the model to hold that a certain fact is true with a certain degree of belief, and *default reasoning* [REIT80], in which certain facts can be held in the absence of evidence to the contrary.

2.4. Why Is the Model There?

The knowledge about a user that a model provides can be used in a number of ways in an intelligent, interactive system. The particular opportunities depend, of course, on the application. Figure 1 presents a general taxonomy of possible uses. At the top level, user models can be used to support (1) the task of recognizing and interpreting the information seeking behavior of a user, (2) providing the user with help and advice, (3) eliciting information from the user and (4) providing information to him. Situations where user models are used for many of these purposes can be seen in the examples presented throughout this book.

The model can be accessed in two primary ways: facts can be added, deleted or updated from the model, and facts can be looked up or inferred. A forward chaining

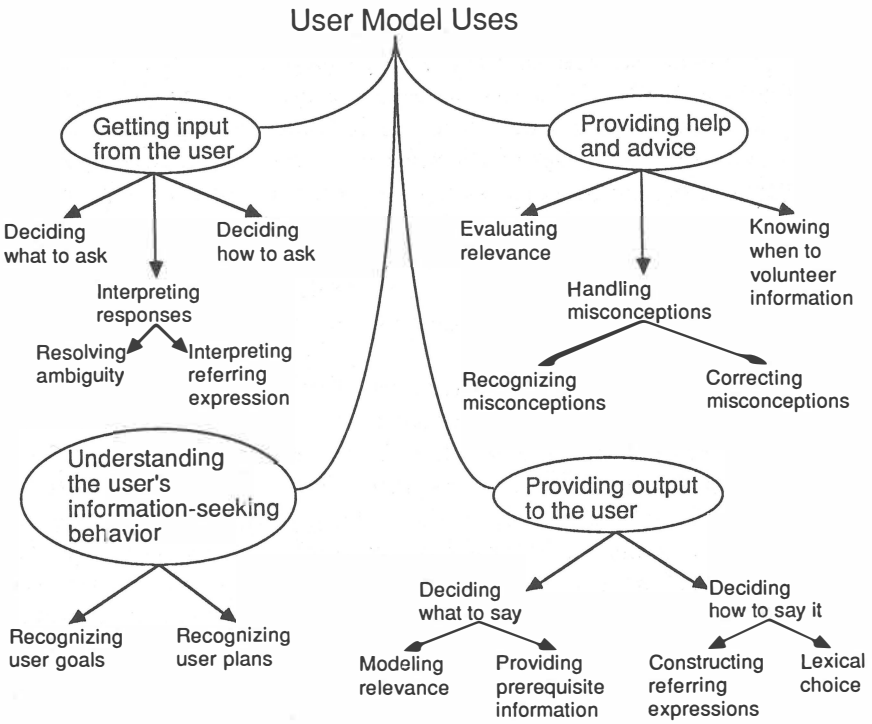

Figure 1. Uses for knowledge of the user

component, together with a truth maintenance system, can be used to update the default assumptions and keep the model consistent.

3. A General User Modeling System

The previous section described a large conceptual space of user modeling possibilities in terms of four dimensions. This section will place our own work in this space and give an overview of its major characteristics. Our paradigm for user modeling might be described as *User Model as Deductive Database*. The *GUMS* system allows one to construct models of individuals that appear to be a collection of facts that can be asserted, retracted or queried. Some of these facts are deduced through the use of inference rules associated with stereotypes to which the user belongs.

Who is being modeled? The *GUMS* system is designed for building **long term** models of **individual users**. We want to represent the knowledge and beliefs of individuals, and do so in a way that results in a persistent record which can grow and change as necessary. These users are associated with one or more *stereotypes* from which additional facts derive.

What is being modeled? Our interest is in providing a domain-independent user modeling tool. The contents of the user model will vary greatly from appli-

cation to application and is left for the model builder to decide. For example, a natural language system may need to know what language terms a user is likely to be familiar with [WEBB84], a CAI system for second language learning may need to model a user's knowledge of grammatical rules [SCHU86b], an intelligent database query system may want to model which fields of a data base relation a user is interested in [MOTR86], and an expert system may need to model a user's domain goals [POLL85]. However, our paradigm of user model as deductive database makes modeling certain things much easier than others. It is easy to construct models that are expressed as a set of relations. In particular, we do not provide any general facilities for building deep models of an agent's reasoning strategies or procedural knowledge.

How is the model acquired and maintained? Our approach in *GUMS* is to let the application be responsible for acquisition and to provide a set of services for maintenance. The application must select the initial stereotypes for the user and add new facts about the user as it learns them. How the application discovers these facts (i.e. explicitly or implicitly) is up to the application. Maintaining the model involves incorporating these new facts, checking that they are consistent with previously learned or inferred facts, and resolving any discrepancies and contradictions. This can be accomplished by undoing contradictory default assumptions and/or shifting the user to a new stereotype. The details of how this is done will be described later.

Why is the model there? Given our goal of producing a general purpose, domain independent use modeling facility, the uses to which the knowledge in the model is put are outside of our design. We will, however, give a more syntactic account of how the model is to be used in terms of how the application interacts with *GUMS*.

3.1. The System Organization

Our goal is to provide a general user modeling utility organized along the lines shown in Figure 2. The user modeling system provides a service to an application program that interacts directly with a user. Note that the user modeling system does not have access to the interaction between the application and the user. Everything it learns about the user must come directly from the application. The *GUMS* system has a separate knowledge base for each application it serves. Each knowledge base consists of two parts: (1) a collection of *stereotypes* organized into a taxonomy, and (2) a collection of models for the individuals. The individual models are installed in the stereotype hierarchy as leaves. Figure 3 shows such a hierarchy. As will be seen later, a stereotype is a collection of facts and rules that are applicable for any person who is seen as belonging to that stereotype. These facts and rules can be either *definite* or *default*, allowing for an individual to vary from the stereotypic norm.

The application program gathers information about the user through its interaction, choosing to store some of this information in the user model. Thus, one service the user modeling facility provides is accepting (and storing) new informa-

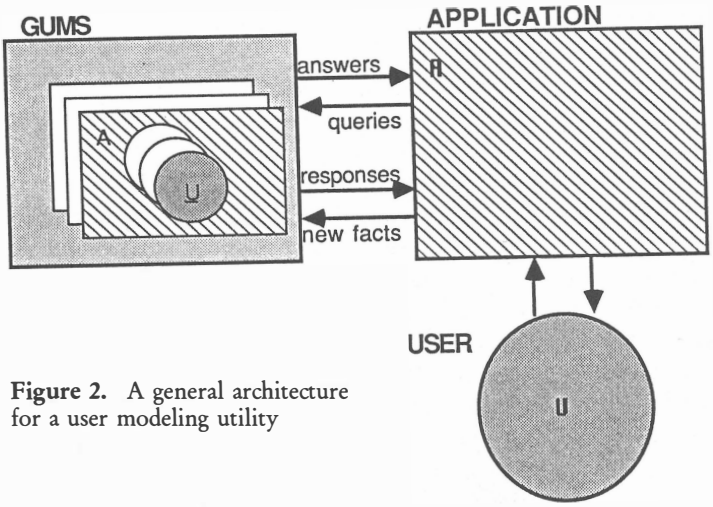

Figure 2. A general architecture for a user modeling utility

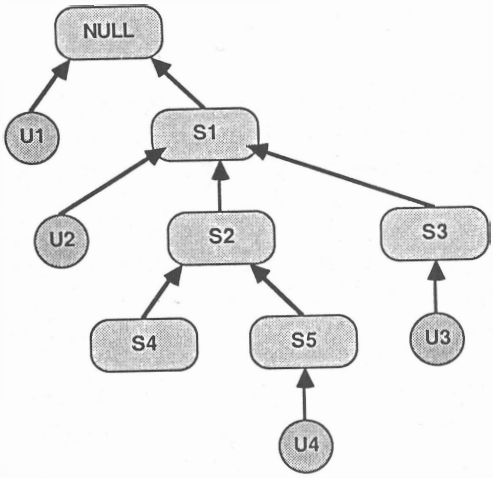

Figure 3. A hierarchy of stereotypes and individuals

tion about the user. This information may trigger an inferential process which could have a number of outcomes:

- The user modeling system may detect an inconsistency and so inform the application
- The user model may infer a new fact about the user which triggers a demon causing some action (e.g. informing the application).
- The user model may need to update some previously inferred default information about the user.

Another kind of service the user model must provide is answering queries posed by the application. The application may need to look up or deduce certain information about its current user.

3.2. Default Reasoning in GUMS

User modeling is most useful in situations in which a system must draw a number of plausible conclusions about a user from a small base of definite knowledge about him. Thus, default reasoning of some kind or another is at the heart of most user modeling systems. Our approach in the *GUMS* system is to use several forms of default reasoning techniques: stereotypes, explicit default rules, and failure as negation. These three types of default reasoning are closely related and, in general, formalizable in terms of a single type of default reasoning. We have found it convenient, however, to separate the default reasoning in *GUMS* for several reasons.

In the *GUMS* system the different kinds of default reasoning are used to capture generalizations of different *grain size* and form a *hierarchy* with respect to the strength of their conclusions. Stereotypes are used to capture generalizations of large classes of users. Within a stereotype, explicit default rules are used to express the stereotypic norms which are allowed to vary across individuals. Failure as negation is used as a general technique for gathering weak evidence for beliefs about a user when no stronger evidence exists.

3.2.1. Stereotypes and Individuals

In *GUMS*, a stereotype consists of a set of facts and rules that are believed to apply to a class of users. Thus a stereotype gives us a form of default reasoning in which each rule and fact in the stereotype is taken to hold in the absence of evidence that the user does not belong in the stereotype.

Stereotypes can be organized in hierarchies in which one stereotype subsumes another if it can be thought to be more general. A stereotype S_1 is said to be more general than a stereotype S_2 if everything which is true about S_1 is necessarily true about S_2. Looking at this from another vantage point, a stereotype inherits all the facts and rules from every stereotype that it is subsumed by. For example, in the context of a *programmer's apprentice* application, we might have stereotypes corresponding to different classes of programmer, as is suggested by the hierarchy in Figure 4.

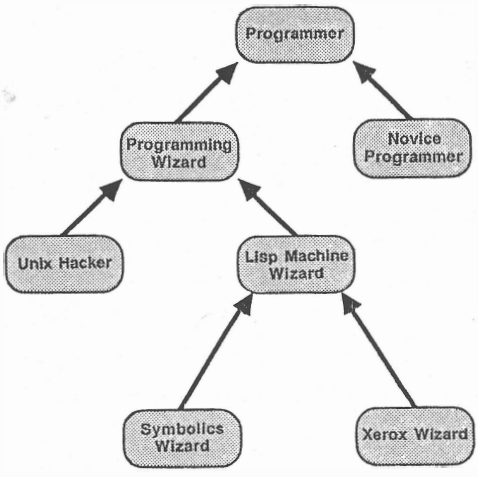

Figure 4. A hierarchy of stereotypes

In general, we will want a stereotype to have any number of immediate ancestors, allowing us to compose a new stereotype out of several existing ones. In the context of a *programmers apprentice*, for example, we may wish to describe a particular user as a *SymbolicsWizard* and a *UnixNovice* and a *ScribeUser*. Thus, the stereotype system should form a general lattice. Our current system, however, constrains the system to a tree.

A stereotype is comprised of several kinds of knowledge, as depicted in Figure 5. Each stereotype can have a single subsuming stereotype, and a set (possibly empty) of subsumed stereotypes and a set (again, possibly empty) of individuals currently believed to be modeled by the stereotype. The real contents of the stereotype, however, consists of two databases of facts and rules, one definite and the other default. The definite facts and rules constitute a *definition* of sorts for the stereotype, in that they determine what is necessarily true of any individual classified under this stereotype. If our knowledge of an individual contradicts this information, then he cannot be a member of this stereotype or any of its descendent stereotypes. The default facts and rules, on the other hand, define our initial beliefs about any individual who is classified in this stereotype. They can be overridden by specific information that we later learn about the individual. The final component of a stereotype is a collection of meta-knowledge about the predicates used in the definite and default knowledge bases. This information is used in the processing of negative queries and is described later.

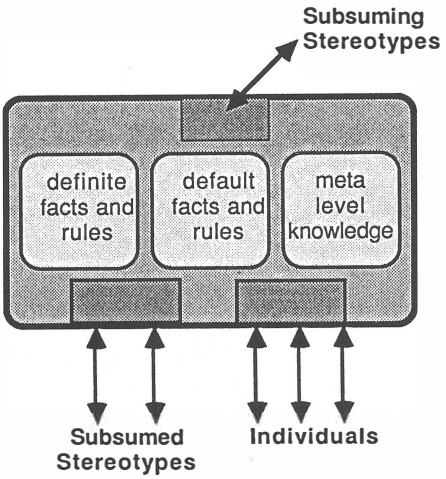

Figure 5. A stereotype

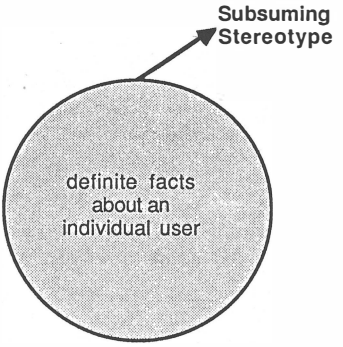

Figure 6. An individual

As an example, consider modeling the knowledge a programmer might have. We can be sure that a programmer will know what a *file* is, but we can only guess that a programmer will know what a *file directory* is. If we have categorized a given user under the *programmer stereotype* and discover (perhaps through direct

interaction with him) that he is not familiar with the concept of a *file* then we can conclude that we had improperly chosen a stereotype and must choose a new one. But if we got the information that he did not know what a *file directory* was, this would not rule out the possibility of his being a programmer.

The knowledge structure for an individual is simpler than that for a stereotype. An individual has exactly one stereotype (in the current version) with which he is currently associated, and a collection of definite, ground facts which are true, as is shown in Figure 6. The facts known about an individual override any default facts known or deducible from the stereotype associated with the individual. They must, however, be consistent with the stereotype's definite knowledge base. If a contradiction arises, then the individual must be reclassified as belonging to another stereotype.

There are a number of ways we might go about trying to reclassify an individual when it is discovered that the current stereotype is no longer valid. In general, we can distinguish *domain independent* and *domain dependent* techniques. Domain dependent techniques are particularly useful when we have some understanding of the reasons why a user might be mis-classified. For example, the user may be learning new domain knowledge while he is using the system and this new knowledge may cause him to "grow out of" the current stereotype. In such a situation, we may wish to directly encode a "growth path" of stereotypes. Another domain dependent technique is to treat certain user attributes as triggers which, when true, evoke particular stereotypes associated with them [RICH*].

Domain independent techniques for selecting a new stereotype in the face of conflicting information involve (conceptually) two stages: generating candidates for the new stereotype and selecting from among them. A very general and powerful technique would be to use a classification strategy similar to that used in the KL-ONE family of representation languages [BRAC85]. Such a strategy would consider all of the possible stereotypes and find the set of *most specific* subsuming stereotypes.

In the *GUMS* system, we have used a very simple domain independent strategy for finding a new stereotype when the known facts about an individual contradict some facts associated with the user's current stereotype. The ancestors of the current stereotype are searched in order of specificity until one is found in which there is no contradiction. Thus, we are guaranteed to find a stereotype which subsumes the user, although it will in general not be the most specific one possible.

Although we have adopted this strategy because of its relative simplicity, both in terms of complexity and inherent computational costs, it does correspond to some of our intuitions on how people adopt new stereotypes. The intuition is that the more we learn about an individual, the less we depend on stereotypic reasoning to fill in our beliefs about him. This is mirrored in the *GUMS* system in its stereotype shifting strategy. As the system learns more about a user, it will discover contradictions with its current stereotype model. When faced with a contradiction, it moves up the hierarchy until it finds a stereotype in which the contradiction is resolved. Moving up the hierarchy corresponds to a lessening of stereotypic reasoning, since the higher models are necessarily less complete.

3.2.2. Default Reasoning with Rules

Within a stereotype, we need to describe some knowledge as being necessarily true with respect to (properly classified) members of the stereotype and other knowledge as *default* information. This default information is to be taken as true in the absence of contradictory information. In the *GUMS* system, each rule or fact in the stereotype is either *definite* (i.e. necessarily true) or *default*. The **certain/1** predicate is used to introduce a definite fact or rule and the **default/1** predicate to indicate a default fact or rule, as in:

certain (P). a definite fact: P is true.

certain (P if Q). a definite rule: P is true if Q is definitely true, and P is assumed to be true if Q is only assumed to be true.

default (P). a default fact: P is assumed to be true unless it is known to be false.

default (P if Q). a default rule: P is assumed to be true if Q is true or assumed to be true and there is no definite evidence to the contrary.

Figure 7 depicts graphically the relationships between clauses with *certain* and with *default* information.

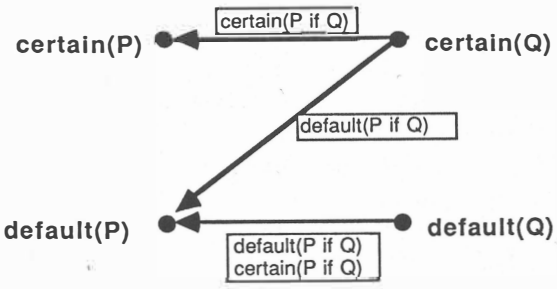

Figure 7. The relationship between certain and default clauses

As an example, consider a situation in which we need to model a person's familiarity with certain terms. This is a common situation in systems that produce text as explanations or in response to queries where there is a wide variation in the users' familiarity with the domain. For example, we might use the rules and facts:

(a) default(understandsTerm(ram)).
(b) default(understandsTerm(rom) if understandsTerm(ram)).
(c) certain(understandsTerm(pc) if understandsTerm(ibmpc)).
(d) certain(~understandsTerm (cpu)).

to represent the following assertions, all of which are considered as pertaining to a particular user with respect to the stereotype containing the rules:

(a) Assume the user understands the term *ram* unless we know otherwise.

(b) Assume the user understands the term *rom* if we know or believe he understands the term *ram*, unless we know otherwise.

(c) This user understands the term *pc* if he understands the term *ibmpc*.

(d) This user does not understand the term *cpu*.

3.2.3. Negation As Failure

GUMS also treats negation as failure in some cases as a default rule. In general, logical deduction is performed using an open world assumption. That is, the failure to prove a proposition is not taken as evidence that it is not true. Many logic programming languages, such a Prolog, encourage the interpretation of unprovability as logical negation. Two approaches have been put forward to justify the negation as failure rule – making the *closed world assumption* and *completing the database*. In the closed world approach [REIT78], we assume that anything not inferable from the database is by necessity false. Making this meta-level assumption introduces a potential problem in that we do not know what the corresponding object-level assumptions are. The second approach is based upon the concept of a completed database [CLKL78]. A completed database is the database constructed by rewriting the set of clauses defining each predicate to an **if and only if** definition that is called the completion of the predicate. The purpose of the completed definition is to indicate that the clauses that define a predicate define **every** possible instance of that predicate.

Any approach to negation as failure requires that a negated goal be *ground* (i.e. contain no uninstantiated variables) before execution.[2] Thus we must have some way of insuring that every negated literal will be bound. In *GUMS* we have used a simple variable typing scheme to achieve this, as will be discussed later.

We have used a variant of the completed database approach to show that a predicate within the scope of a negation is closed. Making the meta-level assertion *declare (closed(P))* signifies that the definition of the predicate *P* is actually an *if and only if* definition. This same technique was used by Kowalski [KOWA79] to indicate completion. We further assume that a predicate is not completed unless it is declared closed. For example, consider the following clauses:

```
declare(closed(p)).
p if q
r if q
```

Since the predicate *p* is complete, proving that *q* is false allows us to deduce that *p* is false. Since the predicate *r* is not complete, proving that *q* is false does not allow us to conclude that *r* is false, although it does provide weak evidence that *r* may be false.

Thus in *GUMS* we have the ability to express that a default should be taken from the lack of certain information (i.e. negation as failure), as well as from the presence of certain information (i.e. default rules). For example, we can have a

[2] A slightly less restrictive rule could allow a partially instantiated negated goal to run but would produce the wrong answer if any variable was bound.

default rule for the programmer stereotype that can conclude knowledge about linkers from knowledge about compilers, as in:

```
default(knows(linkers) if knows(compilers))
```

We can also have a rule that will take the lack of knowledge about compilers as an indication that the user probably knows about interpreters, as in:

```
certain(knows(interpreters) if ~ knows(compilers))
```

This system also allows explicit negative facts, both definite and default. When negation is proved in reference to a negative fact then negation is not considered a default case. Similarly, negation as failure is not considered a default when the predicate being negated is closed. Such distinctions are possible because the *GUMS* interpreter is based on a four value logic.

The distinction between truth or falsity by default (i.e. assumption) and truth or falsity by logical implication is an important one to this system. The central predicate of the system is the two argument predicate *show*, which relates a goal expressed as a literal to a truth value. Thus *show (Goal, Val)* returns in the variable *Val* the "strongest" belief in the literal *Goal*. The variable *Val* can be instantiated to *true, false, assume (true)*, or *assume (false)*. The meanings of these values are as follows:

true	definitely true according to the current database.
assume (true)	true by assumption (i.e. true by default)
assume (false)	false by assumption
false	definitely not true according to the current database.

These values represent truth values for a given user with respect to a given stereotype. If the stereotype is not appropriate, then even definite values may have to change.

The relative strength of the four truth values is shown by the partial ordering in Figure 8. Evidence that a fact is either *true* or *false* is the strongest and outweighs any evidence that the fact is assumed to be true or false. The *GUMS* system is "optimistic" in that it takes direct evidence that a fact is assumed to be true to be stronger than evidence that a fact is assumed to be false. This choice has been made to reflect the fact that, in our current applications, the unprovability of a proposition is weak evidence that it is false. Note that this multi-valued logic is simpler than those based on the notion of a *bilattice* discussed by Ginsberg [GINS86].

Having a four value logic allows us to distinguish conclusions made from purely logical information from those dependent on default information. The four value logic also allows a simple type of introspective reasoning that may be useful for modeling the beliefs of the user. We currently use a default rule to represent an uncertain belief about what the user knows or believes, but we could imagine a situation where we would like to model uncertainties that the user has in his beliefs or knowledge. One such predicate is an embedded *show* predicate. The embedded *show* relation holds between a goal G and a truth value X if the strongest truth value that can be derived for G is X.

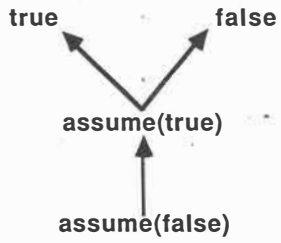

Figure 8. The relative strength of the four truth values

As an example, consider a situation in which we need to express the fact that a user will use a operating system command that he believes might erase a file only if he is certain that he knows how to use that command. We might encode this using the *show* predicate as follows:

```
certain(okay_to_use(Command) if
        can_erase_files(Command),
        show(know(Command),true)).
```

The embedded show predicate forces the *know (Command)* goal to be a definite fact in order for the conclusion to hold.

Another predicate, *assumed (Pred)*, will evaluate the truth of *Pred* and "*strengthen*" the result. The *strengthen* operation maps assumed values into definite values (e.g. **assume(true)** becomes **true, assume(false)** becomes **false** and **true** and **false** remain unchanged). The *assumed* predicate is used to express a certain belief from an uncertain knowledge or belief. For example we might want to express a rule that a user will always want to use a screen editor if he believes one may be available.

```
certain (willUse(screenEditor) if
        assumed(available(screenEditor))).
```

The interpreter that *GUMS* is based on is a metalevel interpreter written in Prolog. The interpreter must generate and compare many possible answers to each subquery, because of the multiple value logic and the presence of explicit negative information. Strong answers to a query (i.e. *true* and *false*) are sought first, followed by weak answers (i.e. *assume(true)* and *assume(false)*). Because strong answers have precedence over weak ones, it is not necessary to remove weak information that contradicts strong information.

Another feature of this system is that we can specify the **types** of arguments to predicates. This type information is used to allow the system to handle non-ground goals. In our system, a **type** provides a way to enumerate a complete set of possible values subsumed by that type. When the top-level *show* predicate is given a partially instantiated goal to solve, it uses the type information to generate a stream of consistent, fully instantiated goals. These ground goals are then tried sequentially.

That goals must be fully instantiated follows from the fact that negation as failure is built into the evaluation algorithm. Complex terms are instantiated to every pattern allowed by the datatype given the full power of unification. To specify the

type information, one must declare the data types, the subtype relations between the types, the instances of these types and, for each predicate, the required type of each argument. For example, the following assertions declare that the **canProgram** predicate ranges over instances of *person* and *programmingLanguage*, that the type *functionalLanguage* is a sub-type of *programmingLanguage* and that the value *scheme* is an instance of the type *functionalLanguage*:

```
declare(schema(canProgramm(person,programmingLanguage))).
declare(subtype(programmingLanguage,functionalLanguage)).
declare(instance(functionalLanguage,scheme)).
```

To see why it is necessary to fully instantiate goals, consider the following problem. We would like to model a programmer's knowledge of tools available on a system. This might be done in order to support an intelligent help system, for example. We might want to encode the following general rules:

C Programmers tend to use all of the relevant tools that are available to them.
If a C programmer does not use some relevant tool, then this is probably because he is not aware of it.
A person is probably a C programmer if they have ever used the C compiler.
Every C tool is relevant to a C programmer.
We have complete knowledge of every tool a user has used.
Lint is a C tool.

which could be expressed in *GUMS* as follows

```
default(knows(Tool) if relevant(Tool), cprogrammer).
default(~knows(Tool) if ~used(Tool), relevant(Tool)).
default(cprogrammer if used(cc)).
certain(relevant(Tool) if cTool(Tool), cprogrammer).
declare(closed(used)).
ctool(lint).
```

Suppose we want to discover a tool of which an individual user is not aware. To answer this, we would like to pose the query *~knows(X)* where X is an uninstantiated variable. Let's assume that this is to be done with respect to an individual knowledge base that includes the following definite facts:

```
used(cc).
used(emacs).
used(ls).
```

The normal interpretation of variables in Prolog goals is that they are *existentially* quantified. The normal interpretation of negated goals is failure as negation, i.e. unprovability. When these two are combined, as in the goal **~used(X)**, the result is that the negation "flips" the quantifier, resulting in a goal which is the equivalent of **$\forall$x not(used(x)).** Moreover, satisfying a negated goal will never result in any variables in the goal being instantiated. Thus, in the example above, we would be unable to discover any tools which this user did not know about since the negated goal **~used(X)** must fail because it is possible to prove **used(emacs)**.

It is possible to "fix" this particular example by reordering the subgoals in the second rule, to get:

```
default(~knows(Tool) if relevant(Tool), ~used(Tool)).
```

This will ensure that the variable *Tool* is instantiated before the negated subgoal is reached. This solution, and the more general one of delaying the evaluation of a negated goal until it is ground, will not always work. Our strategy is to force all goals to be completely ground and thereby avoid this problem.[3] Posing the goal **~knows(X)** to *GUMS* results in the attempt to satisfy a stream of goals which includes the ground goal **~knows(lint)**.

4. The Current GUMS System

The *GUMS* system is currently implemented in CProlog. At the heart of the system is a backward chaining meta-interpreter that implements the default reasoning engine.

Our current system has several limitations. One problem is that it does not extract all of the available information from a new fact learned about the user. If we assert that a predicate is *closed*, we are saying that the set of (certain) rules for the predicate forms a **definition**, i.e. a necessary and sufficient description. In our current system, however, the information still only flows in one direction. For example, suppose that we would like to encode the rule that a user knows about *I/O redirection* if and only if they know about *files* and about *pipes*. Further, let's suppose that the default is that a person in this stereotype does not know about *files* or *pipes*. This can be expressed as:

```
certain(knows(io_redirection) if
        knows(pipes),
        knows(files)).
default(~knows(pipes)).
default(~knows(files))
declare(closed(knows(io_redirection))).
```

If we learn that a particular user **does** know about *I/O redirection* then it should follow that he necessarily knows about both *files* and *pipes*. Adding the assertion

```
certain(knows(io_redirection))
```

however, will make no additional changes in the data base. The values of **knows(pipes)** and **knows(files)** will not change. A sample run after this change might be:

```
?- show(knows(io_redirection),Val).
        Val = true
?- show(knows(pipes),Val).
        Val = assume(false)
```

[3] This is similar to the treatment of negation used by Walker [WALK87].

```
?- show(knows(files),Val).
        Val = assume(false).
```

The reason for this problem is that the current interpreter was designed to be able to incorporate new information without actually using a full truth maintenance system. Before a fact *F* with truth value *V* is to be added to the data base, *GUMS* checks to see if an inconsistent truth value *V'* can be derived for *F*. If one can be, then a new stereotype is sought in which the contradiction goes away. New knowledge that does not force an obvious inconsistency within the database is added as is. Neither redundant information nor existing default information affect the correctness of the interpreter. Subtler inconsistencies are possible, of course.

Another limitation of the current system its inefficiency. The use of default rules requires us to continue to search for solutions for a goal until a strong one is found or all solutions have been checked. These two limitations may be addressable by redesigning the system to be based on a forward chaining truth maintenance system. The question is whether the relative efficiency of forward chaining will offset the relative inefficiency of truth maintenance. The use of an assumption based truth maintenance system [DEKL86] is another alternative that we will investigate.

5. The GUMS Command Language

Our current implementation provides the following commands to the application:

show(Query,Val) succeeds with *Val* as the strongest truth value for the goal *Query*. A Query is a partially or fully instantiated positive or negative literal. *Val* is returned and is the value of the current belief state. If *Query* is partially instantiated then it will return more answers upon backtracking if possible. In general, one answer will be provided for every legal ground substitution that agrees with current type declarations.

add(Fact,Status) sets belief in *Fact* to true if it is consistent with the current set of beliefs about the user. As described earlier, consistency is checked by trying to deduce the negation of the Fact. If *Fact*, or any legal instance of it, is found to be contradictory, *GUMS* adopts successively higher stereotypes until one is found in which all of the added facts are consistent. If no stereotype is successful then no stereotype is used and all answers will be based entirely on added facts. *Fact* must be partially or fully instantiated and can be either a positive or negative literal. *Status* must be uninstantiated and will be bound to a message describing the result of the addition (e.g. one of several error messages, *ok*, the name of a new stereotype, etc.).

create_user(UserName,Stereotype,File,Status) stores the current user model if necessary and creates a new model representing the user who then becomes the current user. *UserName* is instantiated to the desired name. *Stereotype* is the logical name of the stereotype that the system should assume to hold. *File* is the name of the file in which information pertaining to the user will be stored. *Status* is instantiated by the system and returns error messages. A user must be created in order for the system to be able to answer queries.

store_current(Status) stores the current user's information and clears the workspace for a new user. *Status* is instantiated by the system when an error occurs.

restore_user(User,Status) restores a previous user after saving the current user if necessary. *User* is the name of the user. *Status* is instantiated by the system to pass error messages.

done stores the system state of the user modeling system, saving the current user if necessary. This command should be the last command issued and needs to be issued at the end of every session.

6. Conclusions

Many interactive systems have a strong necessity to maintain models of individual users. We have presented a simple architecture for a general user modeling utility based on the ideas of default logic. This approach provides a simple system that can maintain a database of known information about users as well as use rules and facts associated with a stereotype which is believed to be appropriate for this user. The stereotype can contain definite facts and rules of inference as well as default information and rules. The rules can be used to derive new information, both definite and assumed, from the currently believed information about the user.

We believe that this kind of system will prove useful to a wide range of applications. We have implemented an initial version in Prolog and are beginning to use it to support the modeling needs of several projects. There are a number of interesting issues that need to be explored in extending the *GUMS* system, including the use of different reasoning systems such as assumption based TMS's and the development of better strategies for assigning or re-assigning an individual to a stereotype.

7. References

[ALLE80], [ALLE82b], [BRAC85], [BROW78], [CARB83], [CARB88], [CARB*], [CARN83c], [CHIN86], [CHIN*], [CLKL78], [DEKL86], [FINI83], [GERS81], [GINS86], [HOEP83b], [HOWE84], [JOHN84], [JOSH84a], [JOSH84b], [KAPS82], [KASS87a], [KASS87b], [KASS88b], [KASS*], [KOBS84], [KOWA79], [MCCO85a], [MCCO*], [MCKE85a], [MCKE85c], [MORI85b], [MORI*], [MOTR86], [PARI88], [PERR88], [POLL85], [REIT78], [REIT80], [RICH79b], [RICH83], [RICH*], [SCHU86b], [SHRA82], [SIDN81b], [SLEE82a], [SLEE85], [SMIB82], [SWAR83], [WALK87], [WALL82], [WEBB84], [WILE86]