Default Reasoning and Stereotypes in User Modelling

TIM FININ
Unisys Corporation

ABSTRACT: This article discusses the application of various kinds of default reasoning in systems which must maintain a model of its users. In particular, we describe a general architecture of a domain-independent system for building and maintaining *long term models of individual users*. The user modelling system is intended to provide a well defined set of services for an *application system* which is interacting with various users and has a need to 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 modelling system* (hereafter called *GUMS$_1$*) which we have implemented in Prolog. This system satisfies some of the desirable characteristics we discuss.

KEYWORDS: Default Reasoning, User Modelling.

INTRODUCTION—THE NEED FOR USER MODELLING

Systems which 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 of its users is typically refered 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 which attempt to provide help and advice,[11,14,40] *tutorial systems*[39], and natural language interfaces.[48] Each of these systems has a need to represent information about individual users. Most of the information is acquired incrementally through direct observation and/or interaction. These systems also needed to infer additional facts about their users based on the directly acquired information. For example, the WIZARD help

International Journal of Expert Systems
Copyright © 1988 by JAI PRESS, INC.
Pp. 131–158. Volume 1, Number 2, ISSN 0894–9077
All rights of reproduction in any form reserved.

system[11,40] had to represent which VMS operating system objects (e.g., commands, command qualifiers, concepts, etc.) a user was familiar with and to infer which other objects he was likely to be familiar with.

This article describes a general architecture for a domain-independent system for building and maintaining *long term models of individual users*. The user modelling system is intended to provide a well-defined set of services for an *application system* which is interacting with various users and has a need to 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 modelling system* (hereafter called $GUMS_l$) which we have implemented in Prolog. This system satisfies 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 $GUMS_l$ system and the kind of user modelling it supports. The fourth section describes the the three kinds of default reasoning that $GUMS_l$ employs: stereotypical reasoning, explicit default rules and failure as negation. Section five describes some of the details of the $GUMS_l$ system. The concluding section discusses some of the limitations of $GUMS_l$ and avenues for future research.

WHAT KIND OF USER MODEL?

The concept of encorporating 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 modelling 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 how will it be used.

The term "user model" has been used in many different contexts to describe knowledge that is used to support a interactive system. A survey of user modelling in support of natural language interfaces can be found in,[19] and a discussion of user modelling for intelligent tutoring systems is provided by.[22] Kobsa and Whalster [24] have edited a book which brings together a collection of papers discussing user modelling in the context of man–machine dialogue systems. 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 paper, we will focus our

attention on those models in which 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 representation language which is sufficiently expressive. Such a representation language will typically provide a set of inferential services. This will allow some of 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 modelling 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 dialogue 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 which have 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*[44] 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 modelling possibilities by addressing the *who, what, how* and *why* of user modelling. Our own system will then be placed within this space.

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 modelling 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 which 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.[2,4,6,12] 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[49,7] classifies a user as belonging to one of four possible generic classes: *novice, beginner, intermediate*, or *expert*.

Modelling the very general beliefs held by large classes of users is extremely important to building intelligent systems. For example, in modelling 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.*

It is just this sort of rule which is required in order to support the kinds of cooperative interactions studied in,[16] and [17] 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 modelling 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 which states his preferences for various system parameters.

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,[37] and the Real-Estate Advisor.[30]

What is to be Modeled?

The *contents* of a user model naturally varies from application to application. Following Kass and Finin,[22] 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, and 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 achiev-

ing 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[1] and Carberry.[4,5] Intelligent Tutoring Systems have introduced the idea of keeping a catalog of "buggy" plans[3,42,15] as a means of recognizing misconceptions or missing knowledge. The use of plan recognition is also relevant to the problem of providing help and advice, as in [11] and [49].

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[47] and Paris's TAILOR system.[33] Webber and Finin[48] have surveyed ways that an interactive system might reason about its user's capabilities to improve the interaction.

A third category of knowledge represent's a user's bias, preferences and attitudes. GRUNDY[37] was an early user modelling system which used a model of a user's preferences to recommend books to read. The Real-Estate Advisor[30] and HAM-ANS[13,31] are other systems which have tried to model this. Swartout[45] and McKeown[28] 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] Modelling the user's knowledge involves a variety of things: domain knowledge, general world knowledge and knowledge of 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 especially true for intelligent help systems,[11,49] which must provide clear, understandable explanations to be truly helpful. Providing definitions of data base items (such as the TEXT system does[29]) 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[41] 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 the 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[23] has studied some of the difficult representational problems involved with building recursive models of the beliefs of other agents.

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 encorporating 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, began 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, and thus forms an opinion about the preferences of the user based on this initial list of attributes. As another example, each time the UMFE system[43] needs to know a new fact about its user, it simply asks.

Acquiring knowledge about the user implicitly is usually more difficult. This usually involves "eavesdropping" on the user–system interaction in order to observe the user's behavior, and from it to infer facts which go into the model. Various aspects of implicit acquisition have been explored, including the use of presuppositions,[18] misconception recognition,[26,27] stereotype selection,[37,38,30] and the use of default rules.[34,20,21]

Maintenance involves encorporating 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 which 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 representations and reasoning systems on which the modelling system is built. The 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*,[36] in which certain facts can be held in the absence of evidence to the contrary.

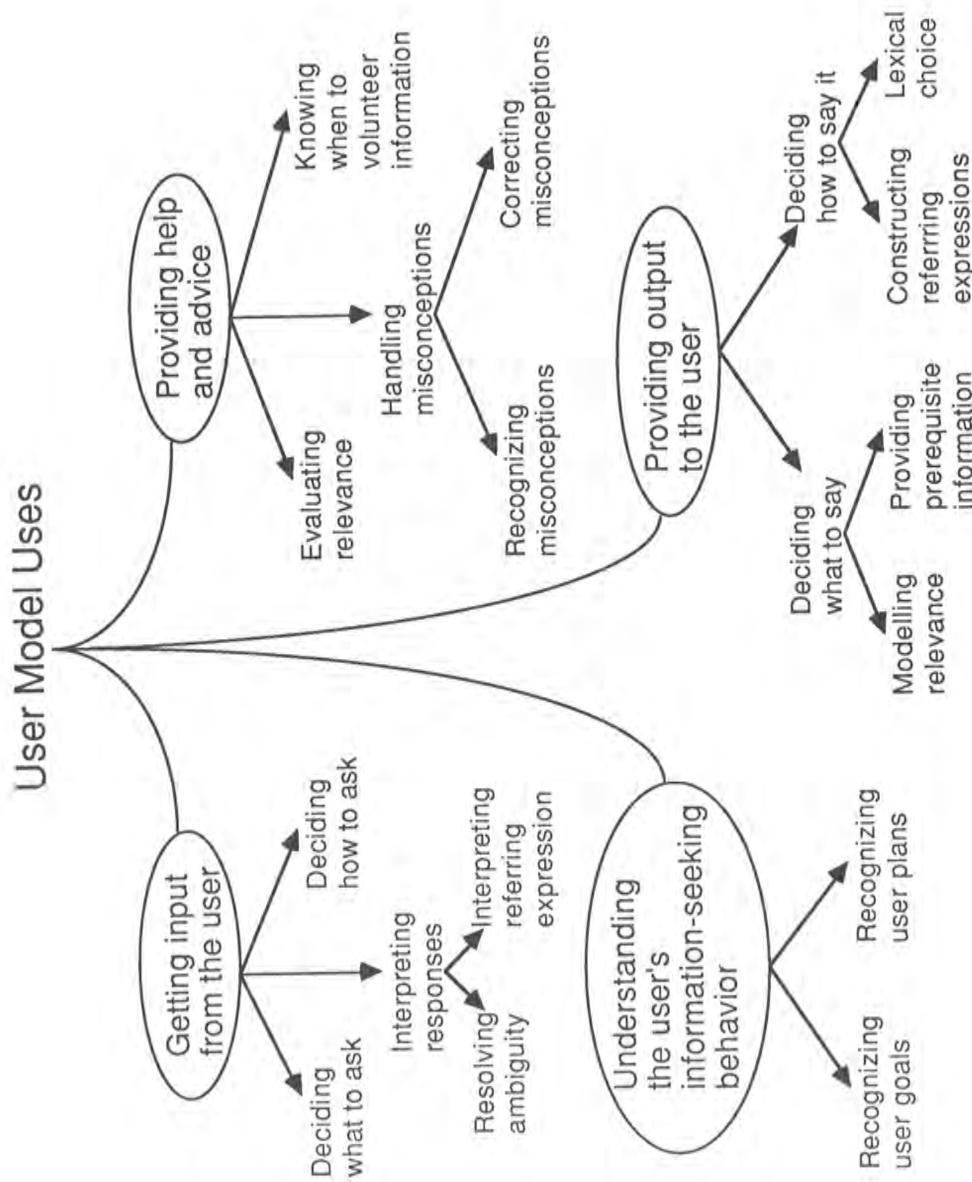
How is the Model to be Used?

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 paper.

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.

A GENERAL USER MODELLING SYSTEMS

The previous section described a large conceptual space of user modelling 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 modelling might be described as *User Model as Deductive Database*. The $GUMS_l$ system allows one to construct models of individuals which appear to be a collection of facts which 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 is being modelled? $GUMS_l$ is designed for building *long term* models of *individual users*. We want to represent the knowledge and beliefs of individuals and to do so in a way that results in a persistent record which can grow and change as necessary. These users are associated with a *stereotype* from which additional facts derive.

What is being modelled? Our interest is in providing a domain-independent user modelling tool. The content of the user model will vary greatly from application 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,[48] a CAI system for second language learning may need to model a user's knowledge of grammatical rules,[39] an intelligent database query system may want to model which fields of a data base relation a user is interested in,[32] and an expert system may need to model a user's domain goals.[35] However, our paradigm of user model as deductive database makes modelling certain things much easier than others. It is easy to construct models which can be 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 $GUMS_l$ is to let the application be responsible for acquisition, and to provide a set of services for maintenance. The application must select the initial stereotype 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 encorporating 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 user modelling facility, the uses to which the knowledge in the model is put is 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 $GUMS_l$.

The System Organization

Our goal is to provide a general user modelling utility organized along the lines shown in Figure 2. The user modelling system provides a service to an application program which interacts directly with a user. Note that the user modelling system does not have access to the interaction between the application and the user. Everything it learns of the user must come directly from the application. The *GUMS*, system has a separate knowledge base for each application it serves. An application 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 which 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 stereotypical norm.

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

Stereotypes

A user model is most useful in a situation where the application does not have complete information about the knowledge and beliefs of its users. This leaves us with the problem of how to model a user, given we have only a limited amount of knowledge about him. Our approach involves using several forms of default reasoning techniques: stereotypes, explicit default rules, and failure as negation. We assume that the *GUMS*, system will be used in an application which incrementally gains knowledge about its users throughout the interaction. But the mere ability to gain knowledge about the user is not enough. We cannot wait until we have full knowledge about a user to reason about him. Fortunately we can very often make a generalization about users or classes of users. We call a such a generalization a stereotype.

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.

Figure 2: A General Architecture for a User Modelling Utility

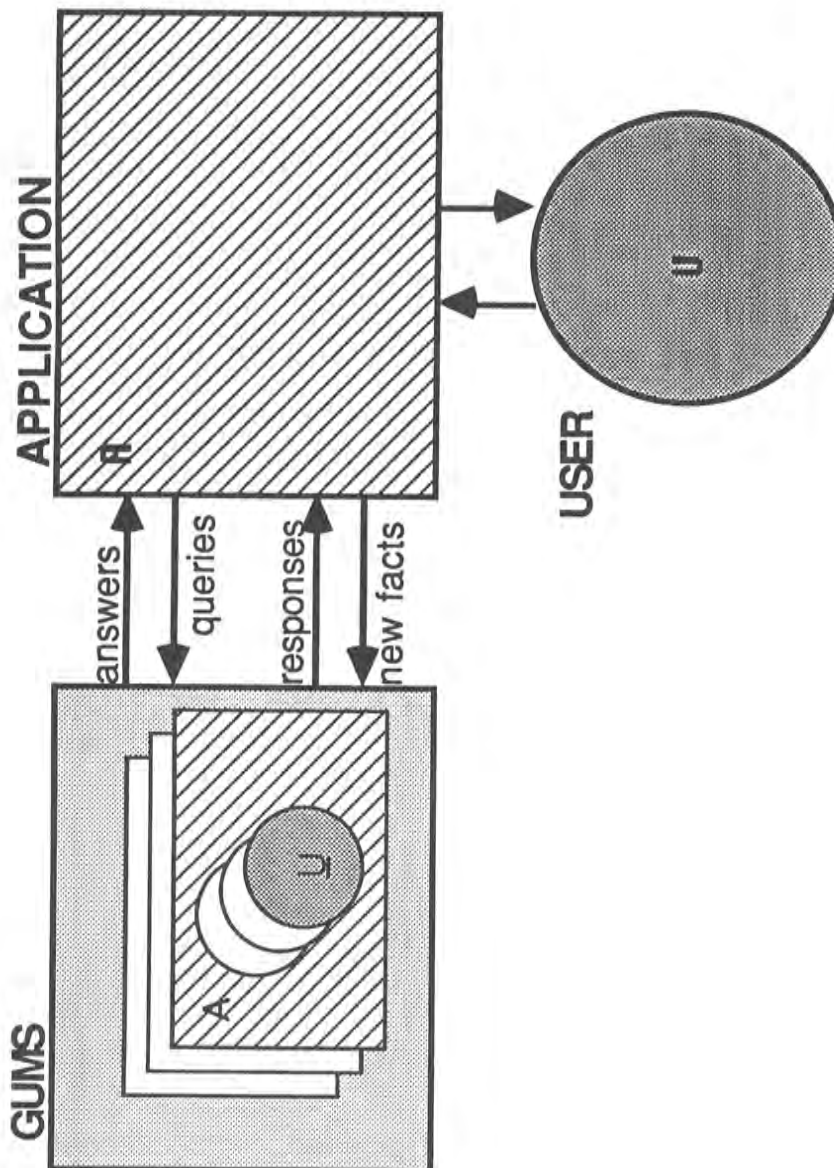
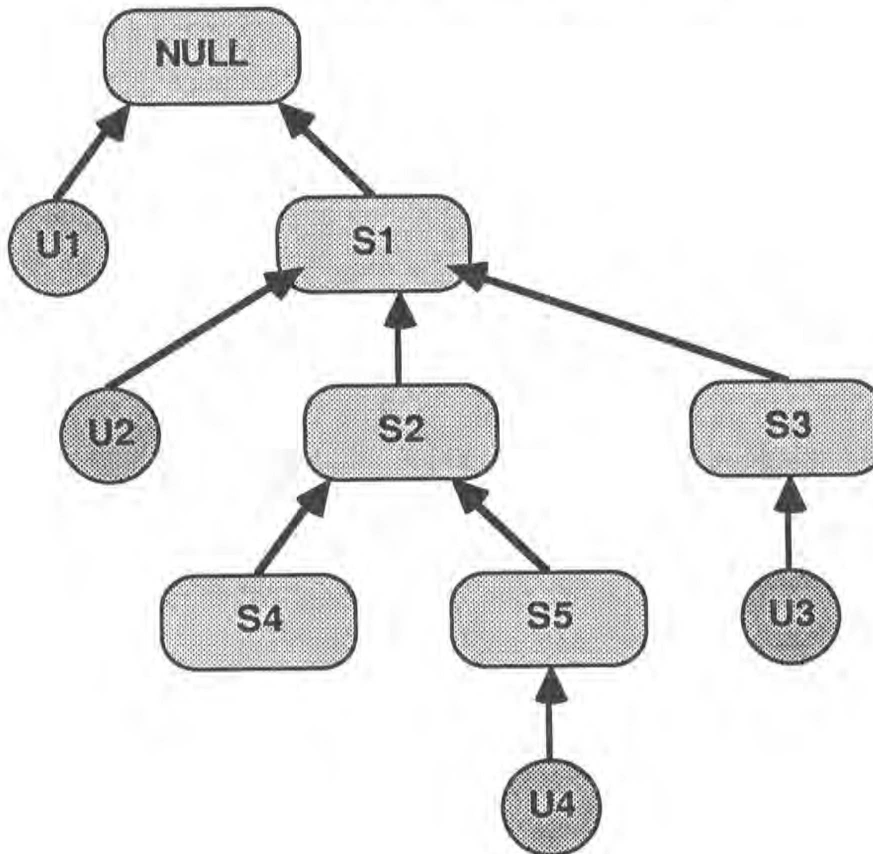


Figure 3: A Hierarchy of Stereotypes and Individuals

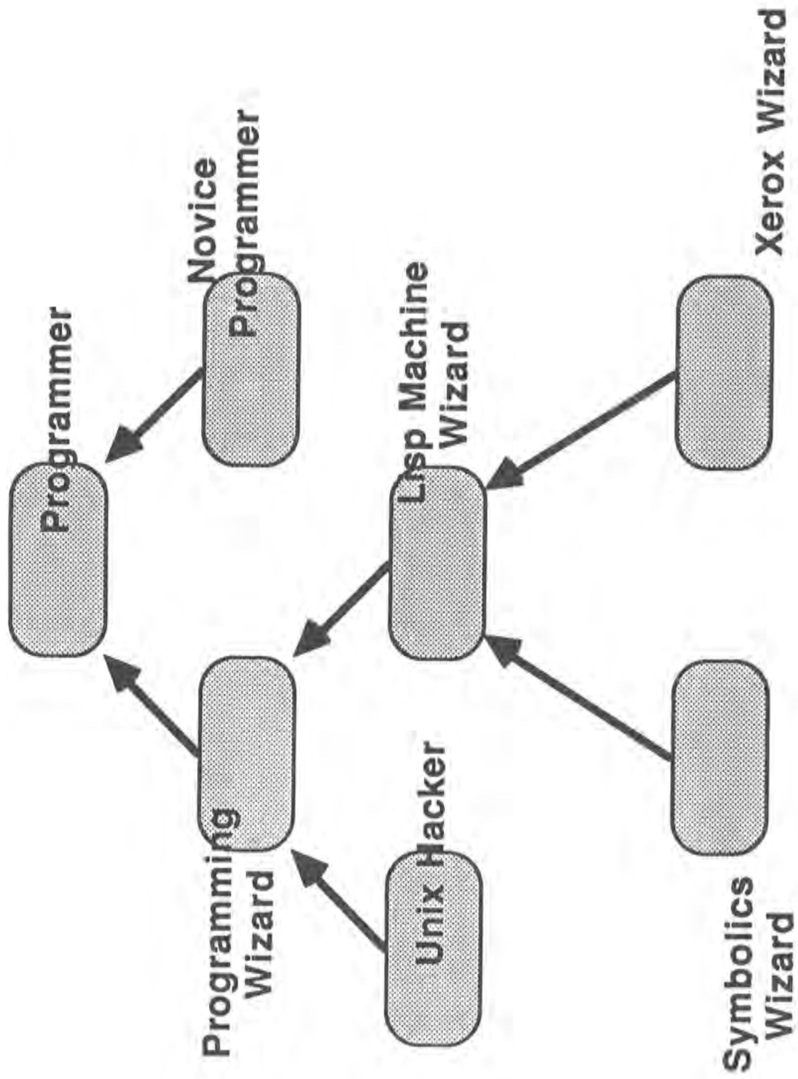


Stereotypes can be organized in hierarchies in which one stereotype subsumes another if it can be thought to be more general. A stereotype S_1 is said to be more general than a stereotype S_2 if everything which is true about S_1 is necessarily true about S_2. Looking at this from another vantage point, 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.

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 *programmer's apprentice*, for example, we may wish to describe a particular user as a *Symbolics Wizard* 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, a set (possibly empty) of subsumed stereotypes, and a set (again, possibly empty) of individuals currently

Figure 4: A Hierarchy of Stereotypes



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 descendant stereotypes. The default facts and rules, on the one 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.

As an example, consider modelling 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[2] 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.

Individuals

The knowledge structure for an individual is simpler than that for a stereotype. An individual has exactly one stereotype with which it 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 knowldge base. If a contradiction arises, then the individual must be reclassified as belonging to another stereotype.

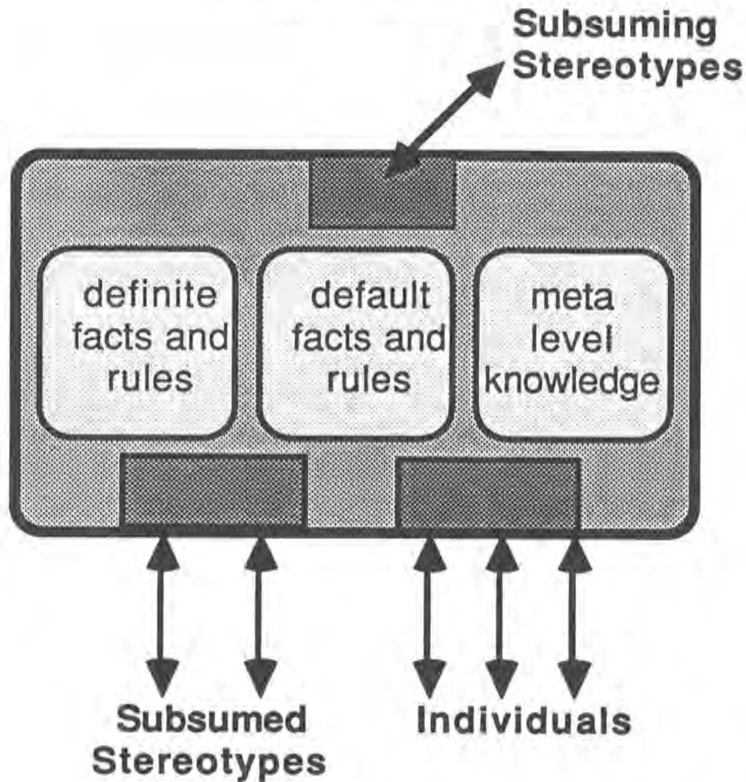
Default Reasoning with Rules

In *GUMS*, we use the **certain/1** predicate 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.

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 which need

Figure 5: A Stereotype



to produce text as explanations or in response to queries and in which there is a wide variation in the users' familiarity with the domain. We might use the following rules and facts:

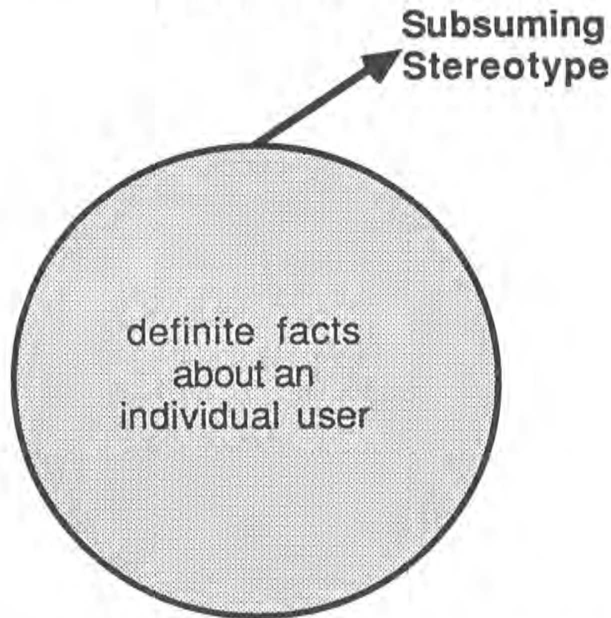
(a) default(understands Term(ram)),
(b) default(understands Term(rom) if understands Term(ram)),
(c) certain(understands Term(pc) if understands Term(ibmpc)),
(d) certain(~understands Term(cpu)),

to represent these 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 *ran* 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*.

GUMS, also treats negation as failure, and in some cases as a default rule. In general, logic is interpreted using an open world assumption. That is, the failure to be able 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 un-

Figure 6: An Individual



provability as logical negation. Two approaches have been forwarded to justify the negation as failure rule. One approach is the closed world assumption.[9] In this case we assume that anything not inferable from the database is by necessity false. One problem with this assumption is that this is a metalevel assumption and we do not know what the equivalent object level assumptions are. A second approach originated by Clark is based upon the concept of a completed database.[8] 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 before execution. (Actually 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.) 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. A predicate is closed if and only if it is defined by an *if and only if* statement and every other predicate in the definition of this predicate is closed. We allow a metalevel statement *completed(P)* that is used to signify that by predicate *P* we really intend the *if and only if* definition associated with *P*. This same technique was used by Kowalski[25] to indicate completion. We assume that a predicate is not completed unless it is declared to be so. So if *P* is not explicitly closed, ***not P*** is decided by default.

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 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 and default facts. 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 *G* 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.

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.

Having a four value logic allows us to distinguish conclusions made from purely logical information from those dependent on default information. Four value logic also allows a simple type of introspective reasoning that may be useful for modelling 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 *T* if the strongest truth value that can be derived for *G* is *T*.

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 believe 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 $GUMS_l$ 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 uses 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 will be instantiated to every pattern allowed by the datatype given the full power of unification. To specify the type information, one must specify argument types for a predicate, subtype information and type instance information. For example, the following says 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 (canProgram (person,programmingLanguage)).

subtype (programmingLanguage, functionalLanguage).

inst (functionalLanguage, scheme).
```

To see why this is necessary, 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, than 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 $GUMS_l$ 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
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 $GUMS_l$ results in the attempt to satisfy a stream of goals which includes the ground goal **~knows(lint)**.

THE CURRENT *GUMS*$_1$ SYSTEM

The *GUMS*$_l$ system is currently implemented in cProlog. At the heart of the system is a backward chaining metainterpreter which implements the default reasoning engine. This metainterpreter is shown in the Appendix.

Our current system has several limitations. One problem is that it does not extract all of the available information from a new fact learned of the user. If we assert that a predicate is *closed*, we are saying that the set of (certain) rules for the predicate form 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 expresses as:

```
certain (knows (io_redirection) if
        knows (pipes),
        knows (files))
default (~knows (pipes))
default (~knows (files))
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)
?- 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*$_l$ 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 or existing default information effect 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 until 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[10] is another alternative that we will investigate.

The *GUMS*$_1$ 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 *Fact* or any legal instance of it contradicts the current belief state then the user model adopts successively higher stereotypes in the hierarchy until one is found in which all of the added facts are consistent. If no stereotype is successful then no stereotype is used, 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 if necessary and creates a new user who then is 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 users information and clears the workspace for a new user. *Status* is instantiated by the system on an error.

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 modelling 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.

CONCLUSIONS

Many interactive systems have a strong need to maintain models of individual users. We have presented a simple architecture for a general user modelling utility which is based on the ideas of a default logic. This approach provides a simple system which can maintain a database of known information about users as well as use rules and facts which are associated with a stereotype which is believed to be appropriate for this user. The stereotype can contain definite facts and define 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 planning to use it to support the modelling needs of several projects. We are also exploring a more powerful approach to user modelling based on the notion of a *truth maintenance system*.

NOTES

1. In the context of modelling 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. perhaps through direct interaction with him
3. This is similar to the treatment of negation used by Walker.[46]

REFERENCES

(1) Allen, James F., & C. Raymond Perrault. "Analyzing Intention in Utterances." *Artificial Intelligence* (1980) *15*: 143–178.

(2) Allen, James F., Frisch, Alan M., & Diane J. Litman. "ARGOT: The Rochester Dialogue System." *Proceedings of the 2nd National Conference on Artificial Intelligence* (1982), pp. 66–70.

(3) Brown, J.S., & R.R. Burton. "Diagnostic Models for Procedural Bugs in Basic Mathematical Skills." *Cognitive Science* (1978) *2*: 155–192.

(4) Carberry, Sandra. "Tracking User Goals in an Information Seeking Environment." *Proceedings of the 3rd National Conference on Artificial Intelligence* (1983), pp. 59–63.

(5) Carberry, Sandra. "Modeling the User's Plans and Goals." *Computational Linguistics Special Issue on User Modelling* (1988).

(6) Carbonell, Jaime G., Boggs, W. Mark, Mauldin, Michael L., & Peter G. Anick. "The XCALIBUR Project: A Natural Language Interface to Expert Systems." *8th International Conference on Artificial Intelligence* (1983), pp. 653–656.

(7) Chin, David. "User Modelling in UC, the Unix Consultant." *Proceedings of the CHI–86 Conference* (Boston, April 1986), pp. 24–28.

(8) Clark, Keith L. "Negation as Failure." In *Logic and Databases*, edited by J. Minker and H. Gallaire (New York: Plenum Press, 1978).
(9) Reiter, R. "Closed World Databases." In *Logic and Databases*, edited by J. Minker and H. Gallaire (New York: Plenum Press, 1978), pp. 149–177.
(10) DeKleer, J. "An Assumption Based Truth Maintenance System." *Proceedings of IJCAI–85, IJCAI* (August 1985).
(11) Finin, T.W. "Help and Advice in Task Oriented Systems." *Proc. 7th Int'l. Joint Conf. on Art. Intelligence, IJCAI* (August 1982).
(12) Gershman, A. "Finding Out What the User Wants—Steps Toward an Automated Yellow Pages Assistant." *7th International Conference on Artificial Intelligence* (1981), pp. 423–425.
(13) Hoeppner, Wolfgang, Christaller, Thomas, Marburger, Heinz, Morik, Katharina, Nebel, Bernhard, O'Leary, Mike, and Wolfgang Wahlster. "Beyond Domain Independence: Experience with the Development of a German Language Access System to Highly Diverse Background Systems." *8th International Conference on Artificial Intelligence* (1983), pp. 588–594.
(14) Howe, A. and T. Finin. "Using Spreading Activation to Identify Relevant Help." *Proceeding of the 1984 Canadian Society for Computational Studies of Intelligence, CSCSI* (1984). Also available as Technical Report MS-CIS–84–01, Computer and Information Science, U. of Pennsylvania.
(15) Johnson, W. Lewis, and Elliot Soloway. "Intension-Based Diagnosis of Programming Errors." *Proceedings of the 4th National Conference on Artificial Intelligence* (1984), pp. 162–168.
(16) Joshi, A., Webber, B., & R. Weischedel. "Preventing False Inferences." *Proceedings of COLING–84* (Stanford CA, July 1984).
(17) Joshi, A., Webber, B., & R. Weischedel. "Living Up to Expectations: Computing Expert Responses." *Proceedings of AAAI–84* (Austin TX, August 1984).
(18) Kaplan, S.J. "Cooperative Responses from a Portable Natural Language Database Query System". *Artificial Intelligence* (1982) 19, 2: 165–188.
(19) Kass, Robert, and Tim Finin. "Modelling the User in Natural Language Systems." *Computational Linguistics, Special Issue on User Modelling* (in press, 1988).
(20) Kass, Robert. "Implicit Acquisition of User Models in Cooperative Advisory Systems." MS-CIS–87–05, Department of Computer and Information Science, University of Pennsylvania, 1987.
(21) Kass, Robert, and Tim Finin. "Rules for the Implicit Acquisition of Knowledge About the User." *Proceedings of the 6th National Conference on Artificial Intelligence* (1987). Also available as technical report number MS-CIS–87–10, from the Department of Computer Science, University of Pennsylvania.
(22) Kass, Robert. "The Role of User Modelling in Intelligent Tutoring Systems." In *User Models in Dialog Systems*, edited by Alfred Kobsa and Wolfgang Wahlster (New York: Springer-Verlag, 1988).
(23) Kobsa, Alfred. "Three Steps in Constructing Mutual Belief Models from User Assertions." *Proceedings of the 6th European Conference on Artificial Intelligence* (1984), pp. 423–427.
(24) Kobsa, Alfred, and Wolfgang Wahlster. *User Models in Dialog Systems*. (New York: Springer-Verlag, 1988).
(25) Kowalski, Robert. *Logic for Problem Solving*. (New York: North-Holland, 1979).
(26) McCoy, Kathleen F. "Correcting Object-Related Misconceptions." MS-CIS–85–57, Department of Computer and Information Science, University of Pennsylvania, 1985.
(27) McCoy, Kathleen F. "Reasoning on a User Model to Respond to Misconceptions."

Unpublished paper from UM86, the International Workshop on User Modelling, Maria Laach, West Germany.

(28) McKeown, Kathleen R. "Discourse Strategies for Generating Natural-Language Text." *Artificial Intelligence* (1985) 27: 1–41.

(29) McKeown, Kathleen R. "Tailoring Explanations for the User." *9th International Conference on Artificial Intelligence* (1985), pp. 794–798.

(30) Morik, Katharina, and Claus-Rainer Rollinger. "The Real-Estate Agent—Modeling Users by Uncertain Reasoning." *AI Magazine* (1985) 6: 44–52.

(31) Morik, Katharina, "Modeling the User's Wants." Unpublished paper from UM86, the International Workshop on User Modelling, Maria Laach, West Germany.

(32) Motro, A. "Query Generalization: A Method for interpreting Null Answers." In *Expert Database Systems*, edited by Larry Kerschberg (Menlo Park, CA: Benjamin/Cummings, 1985).

(33) Paris, Cecile L. "Tailoring Object Descriptions." *Computational Linguistics Special Issue on User Modelling* (1988).

(34) Perrault, C. Raymond. "An Application of Default Logic to Speech Act Theory." *Proceedings of the 6th National Conference on Artificial Intelligence* (1987).

(35) Pollack, M. "Information Sought and Information Provided." *Proceedings of CHI'85, Assoc. for Computing Machinery (ACM)* (San Francisco, CA, April 1985), pp. 155–160.

(36) Reiter, Ray. "A Logic for Default Reasoning." *Artificial Intelligence* (1980) 13, 1: 81–132.

(37) Rich, Elaine. "User Modelling Via Stereotypes." *Cognitive Science* (1979) 3: 329–354.

(38) Rich, Elaine. "Stereotypes and User Modelling." In *User Models in Dialog Systems*, edited by Alfred Kobsa and Wolfgang Wahlster (New York, Springer-Verlag, 1988).

(39) Schuster, E., and T. Finin. "VP2: The Role of User Modelling in Correcting Errors in Second Language Learning." *Proc. Conference on Artificial Intelligence and the Simulation of Behavior, AISB* (1985).

(40) Shrager, J., and T. Finin. "An Expert System that Volunteers Advice." *Proc. Second Annual National Conference in Artificial Intelligence, AAAI* (August 1982).

(41) Sidner, C.L., and D.J. Israel. "Recognizing Intended Meaning and Speakers' Plans." *Proc. 7-IJCAI* (Vancouver B.C., Canada, August 1981), pp. 203–208.

(42) Sleeman, D. "Assessing Aspects of Competence in Basic Algebra." In *Intelligent Tutoring Systems*, edited by D. Sleeman and J.S. Brown (New York: Academic Press, 1982), pp. 185–200.

(43) Sleeman, D.H. "UMFE: A User Modelling Front End SubSystem." *International Journal of Man-Machine Studies* (1985) 23: 71–88.

(44) Smith, Brian. *Reflection and Semantics in a Procedural Language*. Ph.D. Thesis, MIT, Cambridge, MA, 1982. Also available as Technical Report MIT/LCS/TR–272.

(45) Swartout, William R. "XPLAIN: A System for Creating and Explaining Expert Consulting Programs." *Artificial Intelligence* (1983) 21: 285–325.

(46) Walker, Adrian. *Knowledge Systems and Prolog*. (Reading, MA: Addison-Wesley, 1987).

(47) Wallis, J.W., and E.H. Shortliffe. "Explanatory Power for Medical Reasoning Expert Systems: Studies in the Representation of Causal Relationships for Clinical Consultations." STAN-CS–82–923, Department of Computer Science, Stanford University, 1982.

(48) Webber, B., and T. Finin. In "Response: Next Steps in Natural Language Interaction." In *Artificial Intelligence Applications for Business*, edited by W. Reitman (Norwood, NJ: Ablex Publ. Co., 1984).

(49) Wilensky, R., Mayfield, J., Albert, A., Chin, D.N., Cox, C., Luria, M., Martin J., and D. Wu. "UC-A Progress Report." UCB/CSD 87/303, Department of EECS, University of California, Berkeley, 1986.

APPENDIX—THE DEMO PREDICATE

This appendix defines the *demo* predicate which implements the heart of the *GUMS*[l] interpreter. The relation

show (Goal, Value)

holds if the truth value of proposition *Goal* can be shown to be *Value* for a particular ground instance of *Goal*. The *show* predicate first makes sure that *Goal* is a ground instance via a call to the *bindVars* predicate and then invokes the meta-evaluator *demo*. The relation

demo (Goal, Value, Level)

requires that *Goal* be a fully instantiated term and *Level* be an integer that represents the level of recursion within the demo predicate. The relation holds if the "strongest" truth value for *Goal* is *Value*.

```
:- op(950,fy,'~').
:- op(1150,xfy,'if').

% show that the truth-value of P is V by (1) binding any variables
% in P and (2) invoking the demo metainterpreter
show(P,V) :- bindVars(P), demo(P,V,O).

% truth values
demo(P,P,_) :- truthValue(P),!.

% reflection . . .
demo(demo(P,V1),V,D) :-
    !,
    nonvar(V1),
    demo (P,V1,D) -> V=true; V=false.

% disjunction . . .
demo ((P,Q),V,D) :- !,
    demo(P,V1,D),
    demo(Q,V2,D),
    upperbound(V1,V2,V).
```

```
% conjunction . . .
demo((P,Q),V,D) :- !,
    demo(P,V1,D),
    demo(Q,V2,D),
    lowerbound(V1,V2,V).

% negation . . .
demo(~P,V,D) :- !,
    demo(P,V1,D),
    negate(V1,V,P).

% assumption . . .
demo(assumed(P),V,D) :- !,
    demo(P,V1,D),
    strengthen(V1,V).

% call demo1 with deeper depth and then cut.
demo(P,V,Depth) :-
    Deeper is Depth+1,
    demo1(P,V,Deeper),
    retractall(temp(_,Deeper)),
    !.

% definite facts . . .
demo1(P,true,_) :- certain(P).
demo1(P,false,_) :- certain(~P).

% find a definite rule that yields TRUE or FALSE.
demo1(P,V,D) :-
    forsome(certain P if Q),(demo(Q,V,D),demoNote(V,D))).

demo1(P,V,D) :-
    forsome(certain(~P if Q),
        (demo(Q,V1,D),
         negate(V1,V,P),
         demoNote(V,D))).

% stop if the best so far was ASSUME(TRUE).
demo1(P,assume(true),D) :-
    retract(temp(assume(true),D)).

% default positive facts.
demo(P,assume(true),_) :- default(P).

% try default rules until one gives a positive value.
```

```
demo1(P,assume(true),D) :-
    forsome(default(P if Q),(demo(Q,V,D),positive(V))).

% default negative facts.
demo(P,assume(false),_) :- default(~P).

% default negative rules.
demo1(P,assume(false),D) :-
    forsome(default(~P if Q),(demo(Q,V,D),positive(V))).

% if P is closed, then its false.
demo1(P,false,_) :- closed (P),!.

% the default answer.
demo1(P,assume(false),_).

% demoNote(X,D) succeeds if X is TRUE or FALSE,
% otherwise it fails after updating temp(A,_)
% to be the strongest value known so far.

demoNote(V,_) :- known(V).
demoNote(V,D) :-
    not(temp(_,D)),
    !,
    assert(temp(V,D)),
    fail.
demoNote(assume(true),D) :-
    retract(temp(_,D)),
    !,
    assert(temp(assume(true),D)),
    fail.
```

% Relations on Truth Values

```
positive(X) :- X = true ; X = assume(true).

known(X) :- X = true ; X = false.

higher(true,_).
higher(assume(true),assume(false)).
higher(_,false).

upperbound(X,Y,Z) :- higher(X,Y) -> Z=X ; Z=Y.

lowerbound(X,Y,Z) :- higher(X,Y) -> Z=Y ; Z=X.
```

```
strengthen(assume(X),X).
strengthen(true,true).
strengthen(false,false).

% negation is relative to a predicate.
negate(true,false,_).
negate(assume(true),assume(false),_).
negate(assume(false),assume(true),_).
negate(false,true,P) :- closed(P).
negate(false,assume(true),P) :- not(closed(P)).

truthValue(true).
truthValue(false).
truthValue(assume(X)) :- truthValue(X).
```

% The Type System

```
% isSubtype(T1,T2) iff type T1 has an ancestor type T2.
isSubtype(T1,T2) :- subtype(T1,T2).
isSubtype(T1,T2) :-
    subtype(T1,T),
    isSubtype(T,T2).

% true if instance I is descendant from type T.
isInstance(I,T) :- inst(I,T).
isInstance(I,T) :-
    isSubtype(T1,T),
    isInstance(I,T1).

% true if T is a type.
isType(T) :- inst(_,T).
isType(T) :- subtype(T,_).
isType(T) :- subtype(_,T).
```

% Grounding Terms

```
% bindVars(P) ensures that all variables in P are bound or it fails.
bindVars(P) :- var(P),!,fail.
bindVars(P) :- atomic(P),!.
bindVars(P) :-
    schema(P,PS),
    P =.. [_|Args],
    PS =.. [_|Types],
    bindArgs(Args,Types).
```

```
bindArgs([ ],[ ]).
bindArgs([Arg|Args],[Type|Types]) :-
     bindArg(Arg,Type),
     bindArgs(Args,Types).

bindArg(Arg,Type) :-
     var(Arg),
     isInstance(Arg,Type).
bindArg(Arg,_) :- bindVars(Arg).

% scheme(P,S) is true if S is the schema for P, eg
% schema(give(john,X,Y),give(person,person,thing)).

% find a declared schema.
schema(P,S) :-
     functor(P,F,N),
     functor(S,F,N),
     declare(S),
     !.

% use the default schema F(thing,thing, . . . ).
schema(P,S) :-
     functor(P,F,N),
     functor(S,F,N),
     for(I,1,N,arg(I,S,thing)),
     !.
```

Manuscript received September 1986; Revised July 1987; Accepted July 1987.

Address correspondence to Timothy W. Finin, Paoli Research Center, Unisys Corporation, P.O. Box 517, Paoli, PA. 19301.