Qualified Answers That Reflect User Needs and Preferences

by

Terry Gaasterland[1] and Jorge Lobo[2]

Abstract

This paper introduces a formalism to describe the needs and preferences of database users. Because of the precise formulation of these concepts, we have found an automatic and *very simple* mechanism to incorporate user needss and preferences into the query answering process. In the formalism, the user provides a lattice of domain independent values that define preferences and needs and a set of domain specific *user constraints* qualified with lattice values. The constraints are automatically incorporated into a relational or deductive database through a series of syntactic transformations that produces an annotated deductive database. Query answering procedures for deductive databases are then used, with minor modifications, to obtain annotated answers to queries. Because preference declaration is separated from data representation and management, preferences can be easily altered without touching the database. Also, the query language allows users to ask for answers at different preference levels.

1 Introduction

Much work has been done to explore methods to handle user preferences in databases (see [CCL90, CD89]), human computer interaction (see [AWS92]), user models (see [McC88, KF88]), and artificial intelligence (see [AP86, Pol90, Par87]). Cooperative answering systems try to enable users to receive answers that they are actually seeking rather than literal answers to the posed questions (see [Mot90, GM88, GGMN92]). This paper presents a complementary approach that incorporates the handling of user preferences into the query answering procedure of a database. A declarative formalism for expressing user preferences and needs as a body of information separate from the database is defined. A query answering procedure then takes both the preferences and the data into account when providing answers.

The major advantages of the system described here are threefold. Preferences can be easily altered without touching the database. Users can ask for all answers either with or without the annotation of preference or for all answers that meet some level of preference. Preferences are captured by separate bodies of declarative information that can be changed

[1]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. EMAIL: gaasterland@mcs.anl.gov

[2]Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, IL 60680. EMAIL: jorge@eecs.uic.edu

independently. They are: (1) qualitative labels with an ordering expressed as an upper semi-lattice, (2) logical statements, and (3) a function for combining preferences.

The notions of *need* and *preference* are reflected through a lattice of values provided by the user. Lattice values are used together with logical statements to express preferences. As an illustration, consider a traveler, Kass, who wants to travel from Chicago to Amsterdam, preferably nonstop. If she has to make a stop, she would rather stop in Washington, where her boyfriend lives than in any other city. She absolutely does not want to stop in London. We can define a set of annotated user constraints that express Kass' restrictions:

nonstop_flight(A,B,Date,Flight):good.
direct_flight(A,B,Date,Flight):okay.
indirect_flight(A,B,Date,Flights):bad.
stopover(Flights,Airport):fine $\leftarrow$ *dc_airport(Airport).*
stopover(Flights,Airport):terrible $\leftarrow$ *london_airport(Airport).*

Consider the rule with the annotation *terrible*. The predicate *london_airport* in the body (to the right of the arrow) may be read as "Airport is located in London." The atom in the head (to the left of the arrow) may be read as "The flights in *Flights* involve a stopover in Airport." The entire constraint may be read as "A list of Flights that involves a stopover in Airport is terrible if the airport is a London airport." (See [Gaa92] for a discussion of natural language descriptions of constraints.) Furthermore, any answer that depends on a flight that stops over in a London airport should be annotated as terrible.

In this example, a set of five symbols {*terrible, bad, okay, good, fine*} reflects preference levels. When the following order is assigned to the symbols: *terrible* $<$ *bad, bad* $<$ *okay, okay* $<$ *good, okay* $<$ *fine*, then a higher rank indicates a higher preference. As will be described in Section 3, any upper semilattice of values may be used for ordering the symbols.

Now, when Kass asks the query "How can I travel to Amsterdam from Chicago on May 1?", expressed logically as, say, $\leftarrow$ *travel(chicago, amsterdam, (may,1,Time), TravelPlan),* the search space of the query should be modified with the constraints so that nonstop flights are noted as *good*; direct flights through Washington as *fine*; flights through any other city, except London noted as *okay*, and so on. Alternatively, she may want to ask for flights that are *fine* or better. Then all answers below this level must be discharged.

Suppose the lattice contains only two values, say *unacceptable* and *acceptable* with the order *unacceptable* $<$ *acceptable.* Let the user constraints on *direct_flight* and *nonstop_flight* be annotated with *acceptable* and the rest with *unacceptable.* In this case, the annotated user constraints reflect Kass' needs.

The method for handling user needs and preferences is summarized as follows: Once a user has provided a lattice of values and a set of user constraints annotated with the values, the constraints are automatically incorporated into a relational or deductive database through a series of syntactic transformations that produces an annotated deductive database. Query answering procedures for deductive databases are then used, with minor modifications, to obtain annotated answers to queries. In contrast with earlier work, the only burden on users is to express their preferences. The separation of preference declaration from data representation is achieved through the use of the theory of annotated logic programs, deductive databases, and the series of simple transformations that are invisible at the user level.

Preliminary background definitions are given in Section 2. Section 3 provides background on annotations and discusses the theoretical details of annotated deductive databases needed for user preferences and needs. It also provides a transformation of a normal logic program into an annotated logic program. Section 5 formally defines annotated user constraints and provides a transformation that incorporates a set of annotated user constraints into an annotated logic program. It also shows that answers obtained from the transformed program are properly annotated to reflect user preferences and needs as expressed in the annotated user constraints and the upper semilattice of values. Section 6 discusses briefly how a procedure can be defined by which the answers obtained for a query are annotated according to the constraints.

2 Background

Deductive databases are comprised of syntactic information and semantic information [GM78]. The syntactic information consists of the *intensional database* (IDB) and the *extensional database* (EDB). The IDB is a set of clauses, or rules, of the form $A \leftarrow L_1, \ldots, L_n$, $n > 0$, where A is an atom and each L_i is a literal. The EDB is a set of clauses, or facts, of the form $A \leftarrow$, where A is a ground atom.

Direct answers to database queries, which are clauses of the form $\leftarrow B_1, \ldots, B_n$. are found by using SLD-resolution on the query, IDB, and EDB clause to produce a search tree. The root node of the search tree is the query clause; each node in the tree is produced by applying an IDB rule to the node above.

The semantic information in a deductive database usually consists of a set of *integrity constraints* (IC), of the form $A_1, \ldots, A_m \leftarrow C_1, \ldots, C_n$, where the A_is and C_is are atoms whose predicate appears in an EDB fact or the head of an IDB rule. An integrity constraint restricts the states that a database can take. For example, the integrity constraint *No person can be both male and female*, possibly written as $\leftarrow$ *person(X),male(X),female(X)*, restricts people in a database from having two genders. Integrity constraints are considered semantic information rather than syntactic information because the constraints on a database add no new deductive knowledge to the database.

If we consider semantic information to be information about the information in the database, *user constraints* are another form of semantic information. A user constraint expresses a state of the database that the user wishes to be *true* of the database. For example, if the user only wants to know about students who are enrolled in English 430, she may express a constraint like the following: *enrolled(X,english430)* $\leftarrow$ *student(X)*, indicating that *student(X)* will be considered *true* during a search only when the student X is enrolled in English 430; otherwise the user constraint is violated in the search.

Now we are prepared to turn to the investigation of how to use annotations to capture and adhere to users preferences over database domains.

3 Annotated Logic Programs and Databases

To enable a user to specify preferences and then receive answers according to those preferences, we must be able to do the following:

- Allow a user to specify a set of user constraints as logical statements in the language of the deductive database and rank the constraints according to preferences through the annotation.
- Integrate the annotated user constraints into the rules and facts of the deductive database to form a new annotated deductive database.
- Accept a query from a user and return a set of annotated answers.
- Allow the user to annotate queries and receive answers that satisfy the annotation.

First, we must precisely define annotation in logic programs. In our discussion of annotation, we follow closely the notation in Kifer and Subrahmanian [KS92]. An *annotated logic program* comprises a set of annotated clauses of the from:

$$A : \alpha \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n.$$

A and the Bs are atoms as usually defined in logic programs; α and the βs are *annotation terms.* $A : \alpha$ is the *head* of the annotated clause, and $B_1 : \beta_1, \ldots, B_n : \beta_n$ the *body.* The annotation terms are defined based upon an upper semi-lattice $\mathcal{T}$, a family of total continuous functions over $\mathcal{T}$ and an enumerable set of *annotation variables.* For our purpose we assume that $\mathcal{T}$ is a complete lattice and denote the lattice ordering by $\leq$, the least upper bound operator by $\sqcup$, the greatest lower bound by $\sqcap$, and the top and the bottom elements of lattice by $\top$ and $\bot$ respectively.

The lattice reflects the rankings of the user about the importance of states expressed in a user constraint. For example, consider the constraints from Section 1. They included a constraint about not stopping in London, a constraint about preferring direct flights over indirect flights, that is flights with a change of planes, and a constraint about preferring nonstop flights over direct flights. The user may assign the value *terrible* to the constraint about London and the value *bad* to the constraint about indirect flights, the value *okay* to the constraint about direct flights, and the value *good* to the constraint about nonstop flights. The bottom of the lattice is *terrible.* The lattice is completed with the top element *very good*, and the partial order is given by the transitive and reflexive closure of the following relation: *terrible* $<$ *bad*, *bad* $<$ *okay*, *okay* $<$ *fine*, *okay* $<$ *good*, *fine* $<$ *very good*, *good* $<$ *very good.*

Alternatively, the lattice may be interpreted to reflect the degree of confidence in the statement expressed in the constraint. Consider an example from the world of molecular biology in which two constraints say that hydrophobic amino acids appear on the inside of a protein molecule and that hydrophilic amino acids wind up on the outside. These states tend to be true about protein molecules, but they are not always true. So let us annotate the hydrophobic constraint with *usually* and the hydrophilic constraint with *almost always.* Now consider a program that generates alternative protein molecule structures. Generated

structures that have hydrophobic amino acids all on the inside and hydrophilic amino acids all on the outside will receive the annotation almost always; structures with all hydrophobic inside and one or more hydrophilic inside will receive the annotation *usually*. If *almost always* is considered to be of higher confidence than *usually*, the first set of structures will be preferred to the second set of structures.

For the lattices, we will initially consider two set of continuous functions. 1) For each $i \geq 1$, we will have an i-ary function $\sqcup_i$, the natural extension of $\sqcup$ to i arguments. 2) For each $i \geq 1$, we will have an i-ary function $\sqcap_i$, the natural extension of $\sqcap$ to i arguments. Whenever it is not ambiguous, we will use $\sqcup$ and $\sqcap$ instead of $\sqcup_i$ and $\sqcap_i$. Hence an annotation term can be recursively defined as follows: 1) Any element of $\mathcal{T}$ is an annotation term. 2) Any annotation variable is an annotation term. 3) If $a_1, \ldots, a_i$ are annotation terms then $\sqcup_i(a_1, \ldots, a_i)$ and $\sqcap(a_1, \ldots, a_i)$ are *complex* annotation terms. Nothing else is an annotation term. If A is an atom and α an annotation term then $A : \alpha$ is called an *annotated atom*. If α is a constant $A : \alpha$ is called *c-annotated*; if α is a variable *v-annotated*. In annotated logic programs, complex terms appear only in the head of the program clauses; the annotations in the bodies are either annotation variables or constants. Let $\mathcal{L}$ be the language of an annotated logic program. The Herbrand base $\mathcal{L}$, $HB_{\mathcal{L}}$, is the set of all ground (non-annotated) atoms. The semantics of annotated logic programs is defined in terms of annotated interpretations. An annotated interpretation $I_{\mathcal{A}}$, is a binary relation subset of $HB_{\mathcal{L}} \times \mathcal{T}$ such that if a pair (A, α) is in $I_{\mathcal{A}}$ then for every $\beta \leq \alpha$, (A, β) is also in $I_{\mathcal{A}}$. We identify an annotated interpretation $I_{\mathcal{A}}$ with the set of annotated ground atoms $\{A : \alpha | (A, \alpha) \in I_{\mathcal{A}}\}$. We can now define *satisfaction*. An annotated interpretation $I_{\mathcal{A}}$ satisfies a ground annotated atom $A : \alpha$, $I_{\mathcal{A}} \models A : \alpha$ iff $A : \alpha \in I_{\mathcal{A}}$.[3] $I_{\mathcal{A}}$ satisfies a non-ground annotated atom $A : \alpha$ iff it satisfies each ground instance of the annotated atom. $I_{\mathcal{A}}$ satisfies an annotated program clause iff for each ground instance of the clause where each annotated atom in the body of the ground clause is satisfied by $I_{\mathcal{A}}$ the head of the clause is also satisfied. Then an *annotated model* of an annotated program $\Pi_{\mathcal{A}}$ is an interpretation that satisfies each clause in the program.

There is a natural partial order that can be defined between annotated interpretations $I_{\mathcal{A}}$ and $J_{\mathcal{A}}$. We say that $I_{\mathcal{A}} \preceq J_{\mathcal{A}}$ iff for each (A, α) in $I_{\mathcal{A}}$ there exists (A, β) in $J_{\mathcal{A}}$ such that $\alpha \leq \beta$.[4] The next step is to extend the T_P operator of van Emden and Kowalski [vEK76] to compute the least annotated model of an annotated program $\Pi_{\mathcal{A}}$.

Definition 3.1 Let $\Pi_{\mathcal{A}}$ be an annotated program. A monotonic operator from annotated interpretations to annotated interpretations, $T_{\Pi_{\mathcal{A}}}$, is defined as:

$$T_{\Pi_{\mathcal{A}}}(I_{\mathcal{A}}) = \{A : \alpha \mid \alpha \leq \alpha' \text{ and } A : \alpha' \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n \text{ is a ground instance of a clause in } \Pi_{\mathcal{A}} \text{ and } \forall i,\ 1 \leq i \leq n,\ (B_i, \beta_i) \in I_{\mathcal{A}}\}$$

■

We can iteratively find the least annotated minimal model by finding the least ordinal δ such that $T^{\delta}_{\Pi_{\mathcal{A}}}(\emptyset) = T^{\delta+1}_{\Pi_{\mathcal{A}}}(\emptyset)$. In other words, the least annotated model of $\Pi_{\mathcal{A}}$ coincides

[3] Slightly abusing the definition of grounding, we assume that complex terms in the annotations are evaluated to values in $\mathcal{T}$.

[4] Note that this relation is equivalent to saying that $I_{\mathcal{A}} \subseteq J_{\mathcal{A}}$.

with the least fixpoint of the operator $T_{\Pi_{\mathcal{A}}}$.

We want to allow negation in the rules and facts of a deductive database. This can be easily done by extending the concepts of negation in normal logic programs to annotated programs. Such an extension based on the stable model semantics can be found in appendix A.

4 Transforming from LP to ALP

We want to allow users to give annotated user constraints for any logic program or deductive database. Thus, we must unify the language of user constraints and logic programs. First, we define how to transform any normal logic program into an annotated logic program. Then we show how the model semantics of the normal logic program is translated into the model semantics of the annotated logic program.

The transformation of a logic program is as follows:

Transformation 1 Let π be a (normal) program clause:

$$A \leftarrow B_1, \ldots, B_n, \ \mathbf{not} C_n, \ldots, \ \mathbf{not} C_m.$$

The annotated transformation of π is the (normal) annotated clause:

$$A : \sqcap\{v_1, \ldots, v_n\} \leftarrow B_1 : v_1, \ldots, B_n : v_n, \ \mathbf{not} C_n : \bot, \ldots, \ \mathbf{not} C_m : \bot$$

where the v_i are n distinct annotation variables. We assume that $\sqcap\{\} = \top$.

Let Π be a normal logic program. The annotated transformation, $\Pi_{\mathcal{A}}$, of Π is the set of annotated transformed clauses from Π. □

The annotation of an atom in an annotated interpretation reflects the confidence or preference in the validity of that atom. Hence, any atom that is true in a given interpretation I must be true at any level of preference or confidence in the corresponding annotated interpretation. Thus, the transformation of an interpretation into an annotated interpretation is defined as follows:

Transformation 2 Associate with each interpretation I an annotated interpretation $I_{\mathcal{A}}$ such that $I_{\mathcal{A}} = \{A : \alpha | A \in I, \text{ for every } \alpha \in \mathcal{T}\}$. □

From these definitions, it is easy to show that the following lemma holds.

Lemma 1 Let Π be a normal logic program and $\Pi_{\mathcal{A}}$ its annotated transformation. Then M is a stable model of Π iff $M_{\mathcal{A}}$ is an annotated stable model of $\Pi_{\mathcal{A}}$.

In this section, we have reviewed the notion of annotated logic programs, and we have precisely defined the annotation framework needed for handling user preferences and needs.

In the next section, we shall define annotated user constraints and show how to integrate a set of annotated user constraints into an annotated logic program.

5 Annotated User Constraints

User constraints express statements of the form *"if a condition C is true then I would like to assume the formula $\mathcal{F}$ to be false."* This statement can naively be translated into the implication $C \rightarrow \neg\mathcal{F}$. However, the simple addition of such implications to the theory can create inconsistencies. The inconsistencies arise because the intention of the user is more than just adding the implication to the program. The user wants the new information to prevail over previous data in the system. This behavior may be obtained if we treat the implication $C \rightarrow \neg\mathcal{F}$ as an exception on the truth value of $\mathcal{F}$ in the way that Kowalski and Sadri introduce exceptions into logic programs [KS90]. Although exceptions avoid inconsistencies, there are some properties of user constraints that cannot be captured with Kowalski and Sadri's definition of exception. As a result, we must extend the theory beyond that of Kowalski and Sadri.

To illustrate, consider an employee of a company in Chicago who is planning a business trip to Paris. She would like to get flight information from a database, but before doing so, she would like to inform the database that she *prefers* direct flights. Before posing the query to the database, she could introduce the constraint *ignore non-direct flights.* If there are no direct flights from Paris to Chicago, the answer to the query according to the constraint would then be empty. The new information in the constraint takes precedence over the existing information.

Assuming that the employee must take the trip anyway, it would be better to return answers that include non-direct flights and to let the employee know that they are a less than ideal solution. One possibility is to let the employee modify the constraint and ask the query again. Suppose instead, we allow the employee to annotate the constraint with a value that indicates the low priority of non-direct flights, as in *non-direct flights:bad,* Any answers that violate these constraint would receive the annotation. Instead of eliminating indirect flights, annotations enable the employee to give them low priority among all possible answers. Then the employee has her preferences respected when the query is asked: she finds direct flights if they exist and finds any flight if no direct flight exists. Formally,

Definition 5.1 A *user constraint* v is an annotated normal clause of the form:

$$A : \alpha \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n,\ \mathbf{not}C_n : \bot, \ldots,\ \mathbf{not}C_m : \bot.$$

where $A : \alpha$ is a c-annotated atom and the $B_i : \beta_i$ are c- or v-annotated atoms. We say that the constraint v is in *homogeneous form* if the atom A has the form $p(X_1, \ldots, X_k)$, where the X_i are k distinct variables. The atom $p(X_1, \ldots, X_k)$ is called an *homogeneous* atom and the predicate symbol p is called a *constrained* predicate symbol. ∎

In this paper we assume that all the constraints are in homogeneous form. This is not a restriction if the equality predicate can be considered part of the language. However, we will not introduce the equality predicate in this paper since it will complicate the description of the annotated logic program semantics unnecessarily. The extension of the semantics to cover equalities is straightforward.

The user constraint v can be interpreted as saying that if the antecedent of the implication, $(B_1 : \beta_1, \ldots, B_n : \beta_n,\ \mathbf{not}C_n : \bot, \ldots,\ \mathbf{not}C_m : \bot)$, is true then *at most* $A : c$ can be

accepted to be true. Formally,

Definition 5.2 Let the annotated clause

$$A : c \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n,\ \mathbf{not}C_n : \bot, \ldots,\ \mathbf{not}C_m : \bot$$

be a user constraint v and $I_{\mathcal{A}}$ an annotated interpretation. $I_{\mathcal{A}}$ satisfies v iff for any ground instance $A' : c \leftarrow B'_1 : \beta'_1, \ldots, B'_n : \beta_n,\ \mathbf{not}C'_n : \bot, \ldots,\ \mathbf{not}C'_m : \bot$ of v such that $(B'_1 : \beta'_1, \ldots, B'_n : \beta_n,\ \mathbf{not}C'_n : \bot, \ldots,\ \mathbf{not}C'_m : \bot)$ is satisfied in $I_{\mathcal{A}}$, $I_{\mathcal{A}}$ satisfies $A' : e$ only when $e \leq c$. ■

As a simplification to the user interface, we allow users to pair an annotation with the head of each constraint as follows:

$$A : c \leftarrow B_1, \ldots, B_n,\ \mathbf{not}C_n, \ldots,\ \mathbf{not}C_m.$$

To transform the annotation/constraint pair into a fully annotated user constraint, each B_i receives a unique annotation variable β_i, and each $\mathbf{not}C_j$ receives the annotation $\bot$. Users who wish to be more sophisticated can define the annotation of the head atom A with a complex annotation term constructed from the β_is according to the annotation definitions in Section 3.

Transformation 1 establishes a translation of normal logic programs into annotated normal logic programs. Now we introduce a transformation that incorporates a set of user constraints, $\mathcal{U}$, into an annotated program, $\Pi_{\mathcal{A}}$.

Transformation 3 For any clause π of the form:

$$A : \alpha \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n,\ \mathbf{not}C_n : \bot, \ldots,\ \mathbf{not}C_m : \bot$$

in $\Pi_{\mathcal{A}}$ with a constrained predicate symbol in the head, replace π with the annotated clauses:

$$A : \alpha \sqcap v \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n, (p'(\vec{X}) : v)\theta,\ \mathbf{not}C_n : \bot, \ldots,\ \mathbf{not}C_m : \bot$$

$$A : \alpha \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n,\ \mathbf{not}(p'(\vec{X}) : \bot)\theta,\ \mathbf{not}C_n : \bot, \ldots,\ \mathbf{not}C_m : \bot$$

to obtain $\Pi_{\mathcal{A},\mathcal{U}}$ such that

1. $p(\vec{X})$ is the head of an homogeneous constraint in $\mathcal{U}$ renamed apart from π.
2. $p(\vec{X})$ and A unify with mgu θ.
3. v is a new annotation variable not appearing in π.

Finally we add to $\Pi_{\mathcal{A},\mathcal{U}}$ all the user constraints in $\mathcal{U}$ replacing each predicate symbol p in the head of the constraints with the new predicate symbol p'. □

The first property we can show is that the user constraints are satisfied by the newly transformed program.

Theorem 1 Let $M_{\mathcal{A}}$ be an annotated stable model of $\Pi_{\mathcal{A},\mathcal{U}}$. Then for any constraint $v \in \mathcal{U}$, $\Pi_{M_{\mathcal{A}}}$ satisfies v. □

Although Theorem 1 shows that Transformation 3 is correct it does not provide any criteria to select a transformation. For example, we can merely annotate all the head atoms in $\Pi_{\mathcal{A}}$ with $\perp$ to produce $\Pi_{\mathcal{A},\mathcal{U}}$, but this transformation unnecessarily constrains the rules in $\Pi_{\mathcal{A}}$. We would like to show that the restrictions imposed in the program are somehow minimal. This minimality can be expressed in terms of the partial relation $\preceq$ that exists between the annotated interpretations. Before we present the theory showing this minimality we need to define a new relation between annotated interpretations. We say that two annotated interpretations $I_{\mathcal{A}}$ and $I'_{\mathcal{A}}$ are *similar* with respect to a set of ground atoms $\mathcal{G}$ iff the projection of these interpretations over their first argument intersected with $\mathcal{G}$ produces the same set. That is, for a ground atom $A \in \mathcal{G}$ there exists α such that $(A, \alpha) \in I_{\mathcal{A}}$ iff there exists β such that $(A, \beta) \in I'_{\mathcal{A}}$.

The following theorem shows that $\Pi_{\mathcal{A},\mathcal{U}}$ is the best program that complies with the user needs expressed through the set of user constraints $\mathcal{U}$.

Theorem 2 Let $\mathcal{L}$ be the language of $\Pi_{\mathcal{A}}$.

1. For every annotated stable model $M_{\mathcal{A}}$ of $\Pi_{\mathcal{A}}$ there exists a similar annotated stable model $M_{\mathcal{A},\mathcal{U}}$ of $\Pi_{\mathcal{A},\mathcal{U}}$ w.r.t. $HB_{\mathcal{L}}$ such that $M_{\mathcal{A},\mathcal{U}} \bigcap HB_{\mathcal{L}} \preceq M_{\mathcal{A}}$ and there is no annotated interpretation $I_{\mathcal{A}}$ that satisfies $\mathcal{U}$ and $M_{\mathcal{A},\mathcal{U}} \bigcap HB_{\mathcal{L}} \preceq I_{\mathcal{A}} \preceq M_{\mathcal{A}}$.
2. For every annotated stable model $M_{\mathcal{A},\mathcal{U}}$ of $\Pi_{\mathcal{A},\mathcal{U}}$ w.r.t. $HB_{\mathcal{L}}$ there exists a similar annotated stable model $M_{\mathcal{A}}$ of $\Pi_{\mathcal{A}}$ such that $M_{\mathcal{A},\mathcal{U}} \bigcap HB_{\mathcal{L}} \preceq M_{\mathcal{A}}$ and there is no annotated interpretation $I_{\mathcal{A}}$ that satisfies $\mathcal{U}$ and $M_{\mathcal{A},\mathcal{U}} \bigcap HB_{\mathcal{L}} \prec I_{\mathcal{A}} \preceq M_{\mathcal{A}}$. □

6 Answering Queries via Annotations

Sections 3 and 5 lay the theoretical foundation for meeting users preferences and needs through annotated user constraints. Now let us examine an example that illustrates the power of the approach.

Consider our user, Kass, from Section 1. Suppose that Kass is using a simple deductive database with the following rules:

travel(A,B,Date,Plan) ← *fly(A,B,Date,Plan).*
fly(A,B,Date,[Flight]) ← *nonstop_flight(A,B,Date,Flight).*
fly(A,B,Date,[Flight]) ← *direct_flight(A,B,Date,Flight).*
fly(A,B,Date,Flights) ← *indirect_flight(A,B,Date,Flights).*
nonstop_flight(A,B,Date,F) ← *flight(F,A,B,Date),* **not** *stopover(F,X).*
direct_flight(A,B,Date,F) ← *flight(F,A,B,Date), stopover(F,X).*
indirect_flight(A,B,Date1,[F|Flights]) ← *flight(F,A,X,Date1), fly(X,B,Date2,Flights).*
stopover(Flights,Place) ← *member(F,Flights), stopover(F,Place).*

Recall Kass' user constraints from Section 1:

nonstop_flight(A,B,Date,Flight):good.
direct_flight(A,B,Date,Flight):okay.
indirect_flight(A,B,Date,Flights):bad.
stopover(Flights,Airport):fine $\leftarrow$ *dc_airport(Airport).*
stopover(Flights,Airport):terrible $\leftarrow$ *london_airport(Airport).*

As defined in Section 5, the last two user constraints transform into the following fully annotated user constraints in which β is an annotation variable:

stopover(Flights,Airport):fine $\leftarrow$ *dc_airport(Airport):*β.
stopover(Flights,Airport):terrible $\leftarrow$ *london_airport(Airport):*β.

When the deductive database is transformed into an annotated logic program using Transformation 2 and then into a new annotated logic program using Kass' user constraints and Transformation 3, the annotated logic program in Figure 1 results.

Now we are ready to consider Kass' query about traveling from Chicago to Amsterdam on May 1, $\leftarrow$ *travel(chicago, amsterdam, (may,1,Time), TravelPlan).*

Without loss of generality, we assume that queries have only one atom and correspond to a rule defining the query whose head contains the query variables as follows:

$\leftarrow$ *query(Time,TravelPlan).*
query(Time,TravelPlan) $\leftarrow$ *travel(chicago, amsterdam, (may,1,Time), TravelPlan).*

Since the query is to an annotated logic program, it must be annotated. Annotations on the query are handled as follows:

- When users ask queries without annotation, that indicates that they are not interested in the annotation values. Even so, an annotation variable is attached prior to search so that the query is compatible with the program. As follows from Sections 3 and 5 all answers are returned. Substitutions for the annotation variable are not returned to the user.
- If the user asks the query together with an annotation variable β, all answers are returned, and each answer has an annotation value associated with it.
- If the user asks the query together with an annotation value c, all answers at that value or above in the lattice are returned. As follows from Sections 3 and 5, the answers are annotated with the value given by the user.[5]

The process of answering a query is very similar to any SLD-resolution style proof procedure. Let us examine each case in turn with Kass' query.

When the query is presented to the program without an annotation variable, it receives a temporary variable, β, and expands to three alternative queries *Q1*, *Q2*, and *Q3* as follows:

Q: $\leftarrow$ *query(Time,TravelPlan):*β.
Q': $\leftarrow$ *travel(chicago, amsterdam, (may,1,Time), TravelPlan):*β.

[5]The query can also be thought of as the conjunction $(Q : \beta) \wedge (\beta \geq c)$, where the constant c specifies the least acceptable value for the answers and the variable β receives an annotation value for each answers. Space prevents us from expanding on this variation here.

$travel(A,B,Date,Plan):\beta \leftarrow fly(A,B,Date,Plan):\beta.$

$fly(A,B,Date,Flight):\beta \leftarrow nonstop\_flight(A,B,Date,Flight):\beta.$

$fly(A,B,Date,Flight):\beta \leftarrow direct\_flight(A,B,Date,Flight):\beta.$

$fly(A,B,Date,Flights):\beta \leftarrow indirect\_flight(A,B,Date,Flights):\beta.$

$nonstop\_flight(A,B,Date,F):\sqcap\{\beta,N\} \leftarrow$
$flight(F,A,B,Date):\beta,$
$nonstop\_flight'(A,B,Date,F):N,$
$\textbf{not}\ stopover(F,X):\bot.$

$nonstop\_flight(A,B,Date,F):\beta \leftarrow$
$flight(F,A,B,Date):\beta,$
$\textbf{not}\ nonstop\_flight'(A,B,Date,F):\bot,$
$\textbf{not}\ stopover(F,X):\bot.$

$direct\_flight(A,B,Date,F):\sqcap\{\beta_1,\beta_2,N\} \leftarrow$
$flight(F,A,B,Date):\beta_1,$
$stopover(F,X):\beta_2,$
$direct\_flight'(A,B,Date,F):N.$

$direct\_flight(A,B,Date,F):\sqcap\{\beta_1,\beta_2\} \leftarrow$
$flight(F,A,B,Date):\beta_1,$
$stopover(F,X):\beta_2,$
$\textbf{not}\ direct\_flight'(A,B,Date,F):\bot.$

$indirect\_flight(A,B,Date,[F|Flights]):\sqcap\{\beta_1,\beta_2,N\} \leftarrow$
$flight(F,A,X,Date):\beta_1,$
$fly(X,B,Date2,Flights):\beta_2,$
$indirect\_flight'(A,B,Date,[F|Flights]):N.$

$indirect\_flight(A,B,Date,[F|Flights]):\sqcap\{\beta_1,\beta_2\} \leftarrow$
$flight(F,A,X,Date):\beta_1,$
$fly(X,B,Date2,Flights):\beta_2,$
$\textbf{not}\ indirect\_flight'(A,B,Date,[F|Flights]):\bot.$

$stopover(Flights,Place):\sqcap\{\beta_1,\beta_2,N\} \leftarrow$
$member(F,Flights):\beta_1,$
$stopover(F,Place):\beta_2,$
$stopover'(Flights,Place):N.$

$stopover(Flights,Place):\sqcap\{\beta_1,\beta_2\} \leftarrow$
$member(F,Flights):\beta_1,$
$stopover(F,Place):\beta_2,$
$\textbf{not}\ stopover'(Flights,Place):\bot.$

$nonstop\_flight'(A,B,Date,Flight):good.$

$direct\_flight'(A,B,Date,Flight):okay.$

$indirect\_flight'(A,B,Date,Flights):bad.$

$stopover'(Flights,Airport):fine \leftarrow dc\_airport(Airport):\beta.$

$stopover'(Flights,Airport):terrible \leftarrow london\_airport(Airport):\beta.$

Figure 1: Transformed Constrained Annotated Logic Program

Q': ← *fly(chicago,amsterdam,(may,1,Time),TravelPlan):β*
Q1: ← *nonstop_flight(chicago,amsterdam,(may,1,Time),TravelPlan):β.*
Q2: ← *direct_flight(chicago,amsterdam,(may,1,Time),TravelPlan):β.*
Q3: ← *indirect_flight(chicago,amsterdam,(may,1,Time),TravelPlan):β.*

For *Q1 and Q2*, the variable β receives the substitution $\{\sqcap\{\beta', N\}/\beta\}$, where β' is a renamed variable in the rules for *nonstop_flight*, and *direct_flight*. For Q1, the variable β receives the substitution $\{\sqcap\{\beta'_1, \beta'_2, N'\}/\beta\}$, where β'_1, β'_2, and N' are renamed variables in the rules for *indirect_flight*. Expanding *Q1* produces the following two alternatives:

Q1-1: ← *flight(TravelPlan,chicago,amsterdam,(may,1,Time)):β',*
nonstop_flight'(chicago,amsterdam,(may,1,Time),TravelPlan):N,
not *stopover(TravelPlan,X):⊥.*

Q1-2: ← *flight(TravelPlan,chicago,amsterdam,(may,1,Time)):β',*
not *nonstop_flight'(chicago,amsterdam,(may,1,Time),TravelPlan):⊥,*
not *stopover(TravelPlan,X):⊥.*

The first choice *Q1-1* resolves with the renamed user constraint to unify *N* with the value *good*. Assume that the *flight* atom resolves with some fact that has ⊤ as its annotation with the substitution {*101/TravelPlan,17:15/Time*}. Also assumes that *stopover(101,X)* fails for all *X*. Then one answer substitution for the query through *Q1-1* is {*101/TravelPlan, 17:15/Time, good/β*}. Following our definition of how to treat un-annotated queries, the value *good* for β is not returned to Kass.

When the query is presented to the program with an annotation variable, the annotation value is included with each answer. Suppose that the database contains the following facts:

flight(chicago,amsterdam,101,(may,1,17:15)):⊤.
flight(chicago,dc,102,(may,1,14:15)):⊤.
flight(dc,amsterdam,102,(may,1,20:30)):⊤.
flight(chicago,amsterdam,102,(may,1,14:15)):⊤.
flight(chicago,london,103,(may,1,18:00)):⊤.
flight(london,amsterdam,104,(may,2,08:30)):⊤.
dc_airport(national):⊤.
london_airport(hethrow):⊤.
stopover(102,dc):⊤.

Then the set of answer substitutions for the query would be the following:

{ { *101/TravelPlan,17:15/Time,good/β* },
{ *102/TravelPlan,14:15/Time,fine/β* },
{ *[103,104]/TravelPlan,18:00/Time,terrible/β* } }

If the query were annotated with the value *fine*, only the second answer substitution would be returned. If it were asked with the annotation *okay*, the first and second answer substitutions would be returned but not the third.

Adaptations of bottom-up procedures can also be done in a manner that is similar to the modifications to the top-down procedure. In addition, if the query specifies an annotated constant, the annotation indicates a selection that should be made before projections or joins. Thus, this annotation should be pushed down in the deduction tree before starting the bottom-up evaluation.

Observe that incorporating annotated user constraints into relational databases requires

two steps: (1) adding one argument to some of the relations to store the annotations and (2) adding a procedure to compute operations over the lattice. These extensions can be done automatically without the intervention of the database designer.

7 Conclusions

We have shown how annotated logic programs can be used to model user needs and preferences. The user expresses preferences through a domain independent lattice of values. Domain specific needs and preferences are then expressed through annotated user constraints, that is, logical statements that are qualified with the values in the lattice. The work of Kifer and Subrahmanian [KS92] provides a theoretical basis for annotating logical clauses with values. Because of the precise formulation of our formalism we have found an automatic and *very simple* mechanism to incorporate user needs and preferences into query processing. We have provided a method to transform a logic program or deductive database without annotations together with a set of annotated user constraints into an annotated logic program. The user may then ask a query and receive answers that are qualified, or annotated, with values from the lattice.

We have discussed two variations on the querying process: asking a query and receiving all answers, each with an annotation value; and asking a query together with an annotation value and receiving all answers with that value or higher. In Section 6, we showed how query answering procedures for deductive databases can be adapted through minor modifications to return answers with annotations for databases produced by Transformations 1, 2, and 3 given in Sections 4 and 5. We have an implementation of the transformations and of the modified query answering procedure. [6]

The example presented in Section 6 is simple but illustrative of how user constraints can be applied. The real advantage of the methodology becomes apparent when there are a large number of user constraints. Molecular biological databases provide an example of a domain rich in user constraints.

The transformations described in this paper are not limited to the incorporation of annotated user constraints into deductive databases. We see that they will also be useful for combining multiple deductive databases, each of which expresses an expert's view of the world, into a single database. If each original deductive database is annotated with values that reflect both experts' names and levels of confidence, then the query answering methods described in this paper would produce answers whose annotations reflect the expert positions. Related future directions for this work are to investigate the expressibility of annotated user constraints as a user description paradigm and to explore how to use them to combine different users' judgements about preferred answers.

In summary, using the method described in this paper, a set of annotated user constraints, a lattice of preference values, can be used with a relational or deductive database to return qualified answers that reflect user preferences and needs.

[6] In contrast to general annotated logic programs, our procedures are much simpler and closer to traditional query evaluation procedures because we do not need to adhere to *ideals* (see [KS92] for details on ideals).

Acknowledgments

Terry Gaasterland was supported for this work by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38. The NSF partially supported Jorge Lobo for this work under grant #IRI-9210220. We thank Zoran Budimlik for taking the time to implement our ideas.

A Negation

We want to allow negation in the rules and facts of a database over which a user has expressed needs and preferences. Thus, we must show how to deal with negation in annotated logic programs. We will use the stable model semantics to interpret negation in normal logic programs[7]. The stable model semantics characterizes the meaning of a normal program by a set of minimal models called *stable models*, which are defined using the Gelfond-Lifschitz transformation. This transformation is defined as follows.

Definition A.1 [GL88] Let Π be a normal logic program and let I be an interpretation.

$$\Pi^I = \{(A \leftarrow B_1, \ldots, B_n) | \ (A \leftarrow B_1, \ldots, B_n, \ \mathbf{not} D_1, \ldots, \ \mathbf{not} D_m) \text{ is a ground instance of a clause in } \Pi \text{ and } \forall i, 1 \leq i \leq m, I \not\models D_i\}$$

Π^I is the Gelfond-Lifschitz transformation of Π with respect to I. ∎

The result of the Gelfond-Lifschitz transformation is a negation-free (possibly infinite) definite program. Stable models for logic programs may now be defined as follows.

Definition A.2 Let Π be a normal program. M is a stable model of Π iff M is the unique minimal model of Π^M. ∎

To capture the semantics of stable models into the framework of annotated logic programs we will take the view of negation that Kifer and Subrahmanian refer to as *ontological negation*. Given an annotated interpretation $I_{\mathcal{A}}$, we say that $I_{\mathcal{A}} \models \mathbf{not} A : \alpha$ iff $I_{\mathcal{A}} \not\models A : \alpha$, that is, $(A, \alpha) \notin I_{\mathcal{A}}$.

Definition A.3 (cf. [GL88]) Let $\Pi_{\mathcal{A}}$ be an annotated normal logic program and let $I_{\mathcal{A}}$ be an annotated interpretation.

$$\Pi_{\mathcal{A}}{}^{I_{\mathcal{A}}} = \{(A : \alpha \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n) | (A : \alpha \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n, \ \mathbf{not} D_1 : \delta_1, \ldots, \ \mathbf{not} D_m : \delta_m) \text{ is a ground instance of a clause in } \Pi_{\mathcal{A}} \text{ and } \forall i, \ 1 \leq i \leq m, \ I_{\mathcal{A}} \models \mathbf{not} D_i : \delta_i\}$$

$\Pi_{\mathcal{A}}{}^{I_{\mathcal{A}}}$ is the annotated Gelfond-Lifschitz transformation of $\Pi_{\mathcal{A}}$ with respect to $I_{\mathcal{A}}$. ∎

[7] Similar extensions can be done using other semantics such as the wellfounded semantics [VRS88].

Definition A.4 (cf. [GL88])

Let $\Pi_{\mathcal{A}}$ be an annotated normal program. $M_{\mathcal{A}}$ is an annotated stable model of $\Pi_{\mathcal{A}}$ iff $M_{\mathcal{A}}$ is the least annotated minimal model of ${\Pi_{\mathcal{A}}}^{M_{\mathcal{A}}}$. ∎

References

[AP86] J. F. Allen and C. R. Perrault. Analyzing intention in utterances. In Barbara J. Grosz, Karen Sparck Jones, and Bonnie Lynn Weber, editors, *Readings in Natural Language Processing*, pages 441–458. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1986.

[AWS92] C. Ahlberg, C. Williamson, and B. Shneiderman. Dynamic queries for information exploration: An implementation and evaluation. In *Proceedings of the ACM CHI '92*, pages 619–626, California, 1992.

[CCL90] W. W. Chu, Qiming Chen, and Rei-Chi Lee. Cooperative Query Answering via Type Abstraction Hierarchy. In *Draft Proceedings of the International Working Conference on Cooperative Knowledge Based Systems*, pages 67–68, University of Keele, England, October 3-5, 1990.

[CD89] F. Cuppens and R. Demolombe. How to Recognize Interesting Topics to Provide Cooperative Answering. *Information Systems*, 14(2):163–173, 1989.

[Gaa92] T. Gaasterland. *Generating Cooperative Answers for Deductive Databases.* PhD thesis, University of Maryland, Department of Computer Science, College Park, December, 1992.

[GGMN92] Terry Gaasterland, Parke Godfrey, Jack Minker, and Lev Novik. A Cooperative Answering System. In Andrei Voronkov, editor, *Proceedings of the Logic Programming and Automated Reasoning Conference*, pages 101–120, volume 2, St. Petersburg, Russia, July 1992.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K.A. Bowen, editors, *Proc. 5^{th} International Conference and Symposium on Logic Programming*, pages 1070–1080, Seattle, Washington, August 15-19 1988.

[GM78] H. Gallaire and J. Minker, editors. *Logic and Databases.* Plenum Press, New York, April 1978.

[GM88] A. Gal and J. Minker. Informative and Cooperative Answers in Databases Using Integrity Constraints. In V. Dahl and P. Saint-Dizier, editors, *Natural Language Understanding and Logic Programming*, pages 277–300. North Holland, 1988.

[KF88] R. Kass and T. Finin. Modeling the user in natural language systems. *Computational Linguistics*, 14(3):5–22, September 1988.

[KS90] R. Kowalski and F. Sadri. Logic Programming with Exceptions. In *Proceedings of the International Conference on Logic Programming*, Jerusalem, Israel, 1990.

[KS92] Michael Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its appli cations. *Journal of Logic Programming*, 1992.

[McC88] K. McCoy. Reasoning on a highlighted user model to respond to misconceptions. *Computational Linguistics*, 14:52–63, September 1988.

[Mot90] A. Motro. FLEX: A Tolerant and Cooperative User Interface to Database. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):231–245, June 1990.

[Par87] C. Paris. Combining discourse strategies to generate descriptions to users along a naive/expert spectrum. In *Proceedings of IJCAI*, pages 626–632, Milan, Italy, 1987 August 23-28, 1987.

[Pol90] M. E. Pollack. Plans as complex mental attitudes. In M.E. Pollack P.R. Cohen, J. Morgan, editor, *Intentions in Communication*, pages 77–103. MIT Press, 1990.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *J.ACM*, 23(4):733–742, 1976.

[VRS88] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proc. 7^{th} Symposium on Principles of Database Systems*, pages 221–230, 1988.