A Declarative Conceptual Modelling Language: Description and Example Applications

Marco A. Casanova[1], Andrea S. Hemerly[1] and Antonio L. Furtado[1,2]

[1]Centro Científico Rio
IBM Brasil
Caixa Postal 4624
20.001, Rio de Janeiro, RJ - Brasil

[2]Departamento de Informática
Pontifícia Universidade Católica do RJ
R. Marquês de S. Vicente, 225
22.453, Rio de Janeiro, RJ - Brasil

ABSTRACT. A declarative conceptual modelling language, implemented as an extension to Prolog, is described. The language is based on an extended version of the entity-relationship (ER) model for the declaration of the information classes and the formulation of queries, and adopts an abstract data type (ADT) approach to define and execute application-oriented update operations.

The language is an integral part of a workbench that provides rapid prototyping at the conceptual level and that supports expert level features. Simple examples to illustrate the direct use of the workbench over a database / knowledge base application and the addition of expert level features are also included.

1. Introduction

This paper first describes a declarative conceptual modelling language that is part of a workbench to support the direct use of knowledge base/database applications, as well as to serve as a foundation for expert level features to be developed over these applications. Then it illustrates the direct use of the workbench over a database / knowledge base application and the addition of expert level features.

The language follows an extended version of the entity-relationship (ER) model for the declaration of the information classes and the formulation of queries, and adopts an abstract data type (ADT) approach to define and execute application-oriented update operations. The language is implemented as an extension to Prolog, following a declarative style, in the sense that every aspect of an application is declared with the help of facts and clauses, including the update operations.

In this ER/ADT information/operation model, individual entity instances retain their identity across the different classes to which they may belong (via the is_a hierarchy), with respect to their existence, attributes and participation in relationship instances. Also, in the spirit of abstract data types, update requests are limited to the utilization of application-oriented operations.

The workbench permits rapid prototyping of the ER design, that is, the workbench does not treat the ER design as a mere documentation of the application, but as an executable specification. The workbench provides a better basis

for expert level features, because their specification can take advantage of the richer ER/ADT semantics. The workbench also contains a transparent SQL interface and a query-the-user facility, described in [Fu2].

The direct use of the workbench is demonstrated over an example database. After presenting its specification, we describe the execution of queries and update operations. Next, it is shown how to add rules, as needed when extending databases to knowledge bases. Over the example thus expanded, we show how queries can be handled by expert level features, running under the workbench, so as to avoid *misconstruals*. Features like these are being experimented in prototypes developed as part of project **NICE** [CF,HCF], whose purpose is to investigate *cooperative* query processing methods to reduce the cost of developing "help desks" and similar advanced database interfaces. Cooperative query processing has been explored, for example, in [BJ,CCL,CD], through the use of richer conceptual models, and in [Mo], via the generalization of failed queries. A natural language database query system, which recognizes users' presuppositions about the application domain, is also described in [Ka]. The problem of detecting and responding to plan-generation misconstruals is investigated in [Qu]. A good survey of user model techniques can be found in [KW].

The paper is organized as follows. Section 2 presents the syntax for the structural aspects of the languages and discusses queries and updates. Section 3 illustrates the direct use of the workbench over an example, which is taken again in section 4 where expert level features that contribute to avoid misconstruals are examined. Section 5 contains the conclusion. Finally, Appendix A lists the complete specification of the example in section 3 and Appendix B gives a Prolog implementation of the algorithm to block misconstruals, described in section 4.

2. Description of the Language

2.1. Facts and Fact Frames

A *fact* denotes the existence of either an entity or a relationship instance, or captures that one such instance has a certain value for a given attribute. In the prototype, whenever the same attribute name is used in the definition of more than one entity or relationship class, it implies that the attribute will have the same domain. A *key* is an identifying attribute, in the sense that entity or relationship instances that have the same value for the key are indeed the same instance, regardless of the class to which such instances belong. In the current prototype, entity instances cannot have compound keys, i.e. keys consisting of more than one attribute. The key of a relationship instance, on the contrary, is in general compound, since it consists of the keys of the participating entity instances; an exception is the case of binary one-to-n relationship instances, whose key is that of the determining participant (i.e. the participating entity depicted on the "n side" in the ER diagram).

A *database* for a conceptual schema is a set of facts. The syntax for facts is:

<entity class>¢<key>
<entity class>¢<key>\<attribute>(<value>)
<relationship class>#<participants list>
<relationship class>#<participants list>\<attribute>(<value>)

where <participants list> is a list of pairs of the form <entity class>¢<key>.

To refer to more than one attribute of an entity or relationship instance, a frame construct can be used:

<entity class>¢<key> has <attribute frame>
<relationship class>#<participants list> has <attribute frame>

where <attribute frame> is a list of <attribute>:<value> pairs. If <attribute> is multivalued, then <value> will unify with one of the values the attribute currently has, and with the other values upon backtracking.

If this is not the appropriate behavior, a different construct can be used:

<entity class>¢<key> has_gr <attribute frame>
<relationship class>#<participants list> has_gr <attribute frame>

where <attribute frame> contains a pair, <attribute>:<value>, if the attribute is single valued, or a pair <attribute>:<list of values>, if the attribute is multivalued. In the latter case, <list of values> naturally is the list of values <attribute> currently has.

2.2. Classes of Facts

The conceptual schema of a database at the ER/ADT level is specified through clauses that define the entity and relationship classes that exist and the structure of the is_a hierarchy. Relationships of arbitrary arity are allowed and binary one-to-n relationships are singled out. The syntax of the clauses to declare the conceptual schema is:

entity(<entity class>,<key>)
is_a(<entity class>,<entity class>)
relationship(<relationship class>,<participant classes>)
one_to_n(<relationship class>,<determining participant class>)
attribute(<entity class>,<attribute>)
attribute(<relationship class>,<attribute>)
domain(<attribute>,<value variable>,<validity check>,<cardinality>)

where <participant classes> is a list of <entity class> elements, <validity check> is an expression involving <value variable> to define the possible values that can be associated with <attribute>, and <cardinality> is either "single" or "multi", to distinguish between single and multivalued attributes.

Attributes and participation in relationships are inherited along the is_a hierarchy. The current prototype does not provide mechanisms to avoid ambiguities in case of inheritance from more than one parent class, or when a class inherits an attribute also defined in the class.

2.3. Data Structure Declaration and Mapping

Facts are stored in relational data structures, which can take the form of ground unit clauses of Prolog predicates or tuples of SQL tables. In both cases, the relational schema is declared by clauses of the form:

```
structure(<structure name>, <attribute list>)
```

where <structure name> is either a predicate symbol or the name of an SQL table. The names of the structures to be handled as SQL tables should be indicated in a clause:

```
sql_structures(<list of structure names>)
```

To ease the mapping between ER/ADT and relational schemas, the current prototype requires that the names of the columns of SQL tables be the same as the names of the corresponding attributes. On the other hand, the names of the data structures (predicates or tables) are arbitrary. The mapping between the two schemas is established by clauses with the following format:

```
ent_structure(<entity class>, <name of data structure>)
rep_ent_structure(<entity class>, <name of data structure>)
rel_structure(<relationship class>, <name of data structure>)
rep_rel_structure(<relationship class>, <name of data structure>)
ext_ent_structure(<entity class>, <relationships>, <name of data structure>)
```

where <relationships> is a list of <relationship class>.

The motivation for these different clauses comes from the way we design relational structures to accommodate the entity-relationship facts. Exactly one data structure, designated respectively by an "ent_structure" or "rel_structure" clause, must correspond to each entity or relationship class, storing the key attributes together with all the single-valued attributes. For each multivalued attribute, there must be a data structure, indicated in a "rep_ent_structure" or "rep_rel_structure", containing only the key and the attribute involved. Finally, whenever an entity class E participates in one or more one-to-n relationship classes, the data structure of E is extended to also represent the relationships. In such cases, an "ext_ent_structure" clause (instead of an "ent_structure" clause) is used to designate the data structure. A detailed description of the design method is found in [TCF].

2.4. Operations over Facts

In the spirit of abstract data types, the only way to update facts is through predefined application-oriented operations. Following a convenient STRIPS-like scheme [FN,LA], each operation O is specified by a set of clauses, which indicate the facts that are added and deleted by O (i.e., the effects of O) and the preconditions for the execution of O, in terms of logical expressions involving facts that should or should not hold. The syntax of the clauses to specify operations is:

```
<operation>(<name of operation>, <parameter list>)
added(<fact>,<operation>) <- <antecedent>
deleted(<fact>,<operation>) <- <antecedent>
precond(<operation>,<expression involving facts>) <- <antecedent>
```

where <parameter list> consists of the names of the domains to which the parameter values must belong. An "operation" clause provides the *signature* of an operation, and the designer must ensure its consistency with the other clauses referring to the operation. In the "added", "deleted" and "precond" clauses, the <antecedent>, which is a Prolog expression, is often omitted. When present, it provides additional criteria to check whether the clause is applicable and contributes to the instantiation of variables appearing in the head of the clause. Notice that the Prolog expression may in particular refer to other such clauses and to database facts. Of special interest is the case of the antecedent expression of a "precond" clause of an operation O referring to "added" and "deleted" clauses of O; in such cases, the "precond" clause may indeed express a post-condition rather than a precondition, since it is allowed to look at the effects that the execution of O would have.

Preconditions are used to enforce integrity constraints dynamically, in the sense that they restrict the application of the defined operations to guarantee that they can only lead to valid states.

In adherence to the original ADT principles, operations do not "belong" to classes, as happens with strict object-oriented systems. Instances of several classes may be affected by an operation that refers to them through its parameters. As a consequence, inheritance of operations along the is_a hierarchy is provided in a trivial way. To see why this is true, assume the existence of an instance i of an entity class E, such that E is_a F. Assume further that an operation O includes as one of its parameters a reference to an instance of class F. Then, since we require that instances of an entity class must also exist as instances of all classes located above it in the is_a hierarchy, we conclude that O is applicable to i simply because i is also an instance of F.

As a related point that can be illustrated by further elaborating the above example, consider the specification of an operation O' this time referring to instances of E. Suppose that we want the effects of O' to subsume the effects of O, in the sense that O' has all the effects of O plus some others. The indication of subsumed effects can be succinctly done by including either or both of the following clauses in the definition of O':

```
added(F, O') <- added(F, O)
deleted(F, O') <- deleted(F, O)
```

the same provision being possible for preconditions, through the inclusion of "precond" clauses of an analogous format.

In addition to the precond, added and deleted clauses belonging to a specific application, there may be present a number of general (i.e. application-

independent) clauses of these types distinguished by the prefix "sys". The current version of the prototype contains "sys:precond" clauses establishing that:

P1. an instance of an entity-class E such that E is_a F can be added only if the instance exists in class F

P2. a value of an attribute of an entity or relationship instance can only be added if the instance exists

P3. a relationship instance can be added only if all participating entity instances exist

Clauses of type "sys:deleted" are also included, establishing that:

D1. if an instance of an entity-class E is deleted, then it is also deleted from all entity-classes F such that F is_a_ E (letting "is_a_" be the transitive closure of "is_a")

D2. if an instance of an entity or relationship class is deleted then all its attributes are also deleted

D3. if an instance of an entity-class is deleted then all relationship instances where it participates are deleted

These general clauses are based on assumptions that are often adopted with the entity-relationship model. Broadly speaking, they preserve integrity constraints inherent in the model. The "sys:precond" clauses *restrict* additions, whereas the "sys:deleted" clauses *propagate* deletions. The presence of these "sys" clauses reduces the number of clauses that an application designer has to introduce for each operation. On the other hand, the designer can make a "sys:precond" clause vacuous for a specific operation O by simply providing an appropriate "precond", "added" or "deleted" clause in the definition of O. For example, pre-condition P2 becomes vacuous, if an operation O that is allowed to add a value for an attribute of an instance, also adds the instance itself. Similarly, the propagation of deletions can be changed into blocking for an operation O by attaching a "precond" clause to O that enforces the blocking of the operation. For example, the designer may include a "precond" clause preventing the deletion of an entity instance, if a certain attribute of the entity is still defined, or if the instance still participates in some instance of a specified relationship class.

A few "sys:added" and "sys:deleted" clauses were included to handle certain situations where null values are involved. Although these clauses are meaningful at the conceptual level, since nulls are used here to express undefined values, we must point out that their presence is mainly justified to ensure the correct mapping of the ER facts into the relational structures. In our STRIPS-based method to define operations, a "deleted" clause is the way to indicate that an operation O causes, as one of its effects, a single-valued attribute A of an entity or relationship instance to become undefined. A "sys:added" clause complements the deletion of the current value of A, by assigning to it the null value. Conversely, the addition of a value to a currently undefined attribute is complemented by the removal of its null value, through a "sys:deleted" clause. Note that, in the present prototype, to replace a non-null value of a single-valued attribute by another non-null value, both a "deleted" and an "added" clause must be provided. One-to-n

relationships are treated in about the same way as single-valued attributes. The removal of an one-to-n relationship instance, which of course entails the removal of all its attributes, is complemented through "sys:added" clauses to indicate (by inserting nulls) that the participant on the "one side" and the single-valued relationship attributes have become undefined. Notice that, if this participant is replaced by another one, rather than removed, the current relationship attributes are equally removed. Finally, a "sys:deleted" clause provides the deletion of a null denoting an undefined participant when a valid participant is added.

We have still two more "sys:precond" clauses to mention. They implement our strategy (proposed in [VF]) to handle operations in case some of its effects already hold. These clauses prevent the execution if one or more facts that the operation should add are already present in the database or if facts to be deleted are absent. We find that this "all or nothing" strategy is compatible with the notion of *database transactions*, where several commands are involved and there is no commitment with respect to database updates if any failure occurs.

2.5. Query and Update Requests

Over an ER/ADT database, a user can formulate *query requests* and *update requests* as Prolog goals. For queries, a goal would consist of a Prolog expression involving one or more facts with the syntax described in section 2.1.

If a query refers to an attribute of an entity or relationship instance and, although the instance exists, the value of the attribute is currently undefined, the query fails as would be expected. However, we decided that the prototype should allow queries on undefined attributes declared as single-valued to succeed in the special case where the query mentions the "null" value explicitly.

The frame construct is convenient in the formulation of queries if more than one attribute is mentioned in connection to the same entity or relationship instance. Frames can be used in flexible ways. If a term corresponding to a frame is indicated by a variable, the execution of the goal will instantiate the variable to a list involving all attributes of the given entity or relationship instance which have non-null values in the database. If the user is only interested in a few specific attributes, he may indicate the frame explicitly as a list containing the desired attributes in any order he chooses, paired with variables to be instantiated with the corresponding values; in this case, for attributes whose value is not defined the respective variables will remain uninstantiated. Powerful operations have been introduced for frames, especially unification and generalization [Ful]. Moreover, a query with frames has a better performance than a conceptually equivalent query where attributes of the same instance are indicated separately, since by working on entire frames the prototype is able to collapse database accesses so that each access retrieves all values requested that happen to be kept in the same underlying data structure.

Query requests can also involve schema information. All types of declarative clauses described in sections 2.2 and 2.4 (and even section 2.3, if one needs to reach a lower level) can appear in goal expressions.

Update requests are effected by goal expressions containing calls to the defined operations. Although, syntactically, these calls are direct, they are actually intercepted by a meta-predicate "exec_op" which checks the values of the parameters that are not variables or "null"s, tests the preconditions and, in case of success, applies additions and deletions to the appropriate data structures to reflect what the added and deleted clauses specify.

At the beginning of a session, where query and update requests will be posed, two preparatory goals must be executed:

```
<- enable_structures().
<- enable_operations().
```

the effect of the former being that the "sql_fact" predicate of the PSQL tool is applied (as described in [Fu2]) to all structures in the "sql_structures" clause, whereas the effect of the latter is to add to the workspace clauses of the form:

```
<operation template> <- exec_op(<operation template>)
```

where <operation template> consists of the operation name followed by a parenthesized sequence of variables denoting the formal parameters of the operation. The ability to enter calls to operations directly, that we mentioned earlier in this section, results from the presence of these clauses.

3. An Example of Direct Utilization of the Workbench

This section briefly describes an application and illustrates the power of the query language. Appendix A contains the complete description of the example as it runs under the Prolog prototype.

3.1. Conceptual Level Specification of the Application

The conceptual level specification defines entity classes that correspond to employees, trainees, departments, projects and clients, where trainees are a subclass of employees. It also defines relationship classes capturing that employees work in departments and participate in projects, and that clients sponsor departments in view of specific projects. Furthermore, the specification contains integrity constraints requiring that an employee can work in only one department and that he can only participate in sponsored projects of his department.

The mapping between the conceptual level specification and the relational data structure level specification has the following properties: it keeps the data on employees and on departments in SQL tables; it embeds the "works" one-to-n relationship in the "emp" table, together with the attributes of employees; and it

maintains the attribute "task" of relationship "participates", which is multivalued, in a separate table.

The application has operations to install a department indicating the city where its headquarters will be, to hire employees to work in a department, to hire trainees, to separately designate the job that an employee will have in his department, to raise an employee's salary, to fire an employee, to propose a project, to associate in a sponsorship contract a client and a department with respect to a project, to assign employees to projects, to add more tasks to assigned employees, to give final approval to a project, and a few others.

Some features in the definition of operations deserve comments (we refer the reader to Appendix A). The assign operation has a precondition saying that an employee E can be assigned to a project P only if E works in a department that sponsors P. The salary raise operation affects only the salary of an employee, by adding the indicated amount (to reflect this update, only one field of the appropriate "emp" tuple is changed). When a project is initially proposed, it is marked as pending, a condition that can be later removed by an execution of the approve operation issued by the Projects Control Department, say (this removal is implemented by setting to "null" the second field of the corresponding "pr" clause. The definition of operation to hire trainees includes an "added" clause concisely declaring that the operation adds all facts added by the operation that hires employees.

3.2. Sample Executions of the Operations

Suppose that the database is initially empty and that the following operations are executed:

```
G1.  <- install('D1','NY').
G2.  <- hire('McCoy',100,'D1').
G3.  <- designate('McCoy','chair').
G4.  <- propose('Alpha').
G5.  <- associate('Spock Ltd.','D1','Alpha',1991,'c123').
G6.  <- hire_tr('Savik',80,'D1','graduate').
G7.  <- assign('Savik','Alpha','record-keeping').
G8.  <- add_task('Savik','Alpha','communications').
```

From the definition of the operations in Appendix A, the reader may find what facts will start to hold or cease to hold when these goals are executed, and how the data structures will be updated. In particular, the reader may appreciate the consequences of the application-independent clauses (prefixed with "sys") that establish general preconditions and effects of operations. For instance, if the goal "`<-fire('Savik')`" is executed, the direct effect is that Savik ceases to exist as an employee, but the "sys" clauses will also make her cease to exist as a trainee, and all facts related to attributes of this entity instance in both entity classes, as well as of its participation in relationships, will be also removed.

By way of an example, we follow the execution of G6. Recall from Appendix A the definition of "hire" and "hire-tr":

```
H1.  operation(hire, [name,sal,dname]).
H2.  added(emp¢N, hire(N,S,D)).
H3.  added(emp¢N\sal(S), hire(N,S,D)).
H4.  added(works#[emp¢N,dept¢D], hire(N,S,D)).

H5.  operation(hire_tr,[name,sal,dname,level]).
H6.  added(F, hire_tr(N,S,D,L)) <- added(F, hire(N,S,D)).
H7.  added(trainee¢N, hire_tr(N,S,D,L)).
H8.  added(trainee¢N\level(L), hire_tr(N,S,D,L)).
```

The execution of goal G6, "<- hire_tr('Savik',80,'D1','graduate')", directly creates the following new facts, via H7 and H8:

```
F1.  added(trainee¢'Savik', hire_tr('Savik',80,'D1','graduate')).
F2.  added(trainee¢'Savik'\level('graduate'),
          hire_tr('Savik',80,'D1','graduate')).
```

and, indirectly, the following new facts, via H6 and H2, H3 and H4:

```
F3.  added(emp¢'Savik', hire('Savik',80,'D1')).
F4.  added(emp¢'Savik'\sal(80), hire('Savik',80,'D1')).
F5.  added(works#[emp¢'Savik',dept¢'D1'], hire('Savik',80,'D1')).
```

The conceptual information expressed by F1 through F5 is in fact stored, via the mapping clauses, as the following two ground unit clauses (but recall that it is in part physically stored as SQL tuples):

```
R1.  emp('Savik',80,'D1',null).
R2.  tr('Savik','graduate').
```

The complete database at the end of the execution of the operations in G1 through G8 also contains the clauses:

```
R3.  dept('D1','NY').
R4.  emp('McCoy',100,'D1','chair').
R5.  pr('Alpha',true).
R6.  cln('Spock Ltd.','new').
R7.  spon('Spock Ltd.','D1','Alpha',1991,'c123').
R8.  part('Savik','Alpha').
R9.  tsk('Savik','Alpha','record-keeping').
R10. tsk('Savik','Alpha','communications').
```

3.3. Sample Queries

Considering the database state reached through the executions of operations given in the preceding section, it is easy to see that the sample queries below will produce the result indicated (notice that queries (3) and (5) use the frame construct):

```
(1) query:     who works in department D1?
```

```
    in Prolog: <- forall(works#[emp¢N,dept¢'D1'], write(N)).
    answer:    'McCoy', 'Savik'

(2) query:     to what entity classes does Savik belong?
    in Prolog: <- forall(E:'Savik', write(E)).
    answer:    emp, trainee

(3) query:     give all attributes available on Savik, as trainee.
    in Prolog: <- trainee¢'Savik' has F & write(F).
    answer:    [level:graduate, sal:80]

(4) query:     is there some employee whose job is still undefined?
    in Prolog: <- works#[emp¢N,dept¢D]\job(null) & write(N-D).
    answer:    'Savik' - 'D1'

(5) query:     which tasks have been assigned to Savik in project Alpha?
    in Prolog: <- participates#[emp¢'Savik',proj¢'Alpha'] has_gr F
                  & write(F).
    answer:    [task: ['communications','record-keeping']]

(6) query:     is it true that project Alpha is sponsored for 1991?
    in Prolog: <- sponsors¢[client¢*,dept¢*,proj¢'Alpha']\year(1991)
                  & write(yes).
    answer:    yes

(7) query:     has project Alpha been approved already?
    in Prolog: <- (¬ proj¢'Alpha'\pending(*) & write(yes)
                     | prst('still pending') & nl).
    answer:    still pending
```

3.4. Adding a Knowledge Base Rule

Until now we have only considered a factual database in the present example. Knowledge bases would, in addition, include *rules*. To provide an example, to be further explored in connection with the expert level features of the next section, we introduce a rule establishing that a project is "ongoing", in the sense that its execution is under way, if it is being sponsored for the current year and it is no longer pending. Besides the rule, we assume some way to indicate the current year, which could be an access to the system's internal clock or a unit clause. The Prolog declarations follow. As a step towards a pseudo-natural language notation, "ongoing" is introduced as a prefix operator, obviating the need for the special symbols used at the ER/ADT level:

```
op("ongoing",prefix,50).

current(1991).
```

```
ongoing P <-
  current(Y) &
  sponsors#[client¢*,dept¢*,proj¢P]\year(Y) &
  ¬proj¢P\pending(true).
```

Given the state of the database captured in clauses R1 through R10, the query request

```
<- ongoing 'Alpha'.
```

will fail, since the project is indeed currently sponsored but it is still pending.

4. An Example of Expert Level Features: Avoiding Misconstruals

In this section we illustrate how the user interface provided by the workbench can be enhanced by the superimposition of expert level features.

The purpose of the features to be presented is to intercept query requests and provide more than literal answers to what is asked. More specifically, answers will in some situations be expanded in order to avoid invalid user inferences, or *misconstruals* [We], as explained in section 4.1. To detect that an answer can lead a particular user to a misconstrual, one must have available *models* of the individual users (or classes of users).

In [HCF] we have outlined a formal approach to user modelling that is fully compatible with the logic programming paradigm. Based on this approach, we propose an algorithm to prevent a broad class of misconstruals, in the context of queries only, described in section 4.2. Section 4.3 traces two queries that may induce misconstruals, over the example introduced in section 3.4.

4.1. Misconstruals and User Modelling

When interacting with a database, a user is typically tempted to infer further information from that explicitly obtained from previous queries. However, his inferences are not necessarily valid, because his model of the world is often incomplete or even faulty. For example, after consulting the database, an auditor may find that a project, P, has gained the support of a client for the current year, and unadvisedly infer that its execution will start at once, when the actual beginning of the activities still depends on the approval of the Projects Control Department. A more **cooperative** database system would have informed the auditor that the client's sponsorship has indeed been granted, assumed as the original question, and would have added that the beginning of activities has been delayed. To achieve cooperativeness, the system would naturally have some model of typical auditors.

To address the problem of invalid user inferences, we consider a *cooperative interface* that passes additional information to the user when it discovers that he has gathered enough data to infer information that contradicts the database. The

interface essentially simulates user's inferences and compares the result with what can be derived from the database.

We assume that, in the context of a given session, the user remembers his past interactions with the deductive database and that he can use the information thus obtained in his (real world) inferences. The results of past interactions in the current session are kept in a *log*, which will indicate that certain facts hold and that certain other facts do not hold in the database.

For simplicity, we consider that the interface knows exactly the class of users accessing the database at a given time, which isolates our problem from that of classifying users. Thus, from now on, when we refer to the user, we mean any user in this class. The *user model* is a theory, designed together with the database, that abstracts out the rules that the user adopts to reason about the domain of discourse in question. We stress that we use the term "user model" to mean a model of how the user reasons, which is somewhat different from the use of the term in the literature.

We model the user's inferences during a session by the deductions from the user model and the positive facts that the current log indicates to hold. In particular, we assume that the user reasons about negated facts only through *negation as finite failure* [Ll]. This intuitively means that, in a given session, we model the inference of a negated fact $\neg A$, by the failure, in a finite number of steps, to find a proof for A from the user model and the facts that the log indicates to hold.

For a detailed discussion of the theoretical aspects involved, see [HCF].

4.2. Algorithm to Avoid Misconstruals

The cooperative interface we propose uses an algorithm to process users' queries so as to avoid misconstruals. It does not take into account update operations, however. This section informally outlines the algorithm for the propositional case only, whereas Appendix B contains the Prolog implementation for the full first-order case.

The algorithm consists of two mutually recursive parts, that we call *"query"* and *"propagate"*. In the "query" part, given a query request A formulated by a user U, the algorithm first checks if A follows from what the user already knows, i.e. from the model of U extended with the knowledge placed in the user log during the process. If this fails, query A is posed to the database (provided that the authorization requirements are fulfilled). If this also fails, the process stops. If A follows from the database, A is added to the log, and the "propagate" part is entered to look at possible misconstruals induced by A.

More specifically, "propagate" takes in turn each conditional clause Y <- B in the model of U such that A is one of the facts conjoined in B and processes the clause as follows. If Y follows from what the user already knows and also from the database, then Y is not by itself a misconstrual, but it may indirectly cause one, which

is checked by calling "propagate" recursively to examine the consequences of Y. If Y follows from what the user already knows, but it does not follow from the database, then Y is a misconstrual. To avoid that the user infer Y, the algorithm first looks for some negated fact Z in the body B of the clause in question such that Z follows from the database. To check Z against the database, "query" is called recursively. If one such negated fact is found, it is added to the log, effectively inhibiting henceforward the erroneous deduction of Y. Otherwise, on returning to the execution of "propagate", a clause is added to the log to explicitly block the deduction of Y.

If the algorithm is executed again for the same query A, the query will be answered from the user's model plus log and no further action is needed. Also, queries involving the misconstruals thus identified will correctly end in failure.

The Prolog implementation of the algorithm was designed for the full first-order case and it is prepared to handle the propagation of variable bindings, as suggested in [HCF]. It also distinguishes the clauses belonging to the user's model (and to his log) from the system's clauses by adding to the former a prefix which is typically the user's identification (*userid*).

4.3. Informal Description of two Examples

We give in this section two examples, both related to the rule introduced in section 3.4, to indicate how the interface operates. The examples are introduced informally, with the pertinent Prolog expressions shown in figures 1, 2 and 3. The first example illustrates how to avoid misconstruals that arise when the user incorrectly invokes negation as finite failure for the lack of information, whereas the second example has to do with a type of misconstrual that arises when the user has inadequate rules.

Consider that the deductive database has a rule saying that a project is ongoing if duly sponsored by a client and if it is not pending. Suppose that project alpha is sponsored for the current year and that it is still pending (i.e. the Projects Control Department has not yet issued its approval). That is, let D be the following deductive database (see Figure 1 for the formal definition):

D.1. Project *p* is ongoing, if *p* is sponsored and it is not pending
D.2. Project Alpha is sponsored
D.3. Project Alpha is pending

```
/* rules */

op("ongoing",prefix,50).

ongoing P <-
  current(Y) &
  sponsors#[client¢*,dept¢*,proj¢P]\year(Y) &
  ¬proj¢P\pending(true).

/* facts */

current(1991).

dept('D1','NY').
pr('Alpha',true).
cln('Spock Ltd.','new').
spon('Spock Ltd.','D1','Alpha',1991,'c123').
```

Figure 1: Deductive Database

Suppose that the user believes in the same rule as the database, that is, that a project is ongoing if duly sponsored by a client and if it is not pending. This is equivalent to assuming a user model U that contains only one rule (see Figure 2 for the formal definitions):

U.1. Project *p* is ongoing, if *p* is sponsored and it is not pending

Suppose now that the user starts the dialog with the query:

Q. Is project Alpha sponsored?

The answer to Q therefore is YES. That is, at this point the user knows:

A_1. Project Alpha is sponsored

If no extra information is passed to the user in the log, after the first query he will know fact A_1, from which he would wrongly infer fact A_2:

A_2. Project Alpha is ongoing

However, the interface will anticipate and avoid this misconstrual as follows. By applying the algorithm of the preceding section to simulate a deduction *R* of A_2 from the user model U and A_1, it will detect that *R* cannot be accepted because it is possible to infer the negation of fact A_3:

A_3. Project p is pending

from U and A_1, by negation as finite failure, whereas it is not possible to infer the negation of A_3 from the database (since the database in fact includes A_3). Hence, the interface will include A_3 in the log to avoid the user's misconstrual. Indeed, the user can no longer infer A_2 using the complete information he obtained from the database (that is, A_1 and A_3).

The answer combined with the extra information in the log is roughly equivalent to the following English sentence:

Project Alpha is sponsored, but it is not ongoing because it is pending

```
/* knowledge when session starts */

u: (ongoing P) <-
   current(Y) &
   u: (sponsors#[client¢*,dept¢*,proj¢P]\year(Y))  &
   ¬ u: (proj¢P\pending(true)) .

/* query posed */

<- query(sponsors#[client¢*,dept¢*,proj¢'Alpha']\year(1991), a).

/* knowledge added to log */

u:(sponsors#[client¢'Spock Ltd.', dept¢'D1', proj¢'Alpha']\year(1991)).
u:(proj¢'Alpha'\pending(true)).
```

Figure 2: User U

We stress that the misconstrual we just illustrated was caused by an incorrect use of negation as finite failure and that it could be blocked by including an additional fact in the log. Our next example illustrates a second type of misconstrual that arises when the user has inadequate rules.

Suppose now that the user believes that a project is always ongoing, if it is sponsored. That is, let the user model now be V (see Figure 3 for the formal definitions):

V.1. Project *p* is ongoing, if *p* is sponsored

Assume the same deductive database D (including rule D.1 exactly as before). Then, the answer to Q remains unchanged, from which the user can again wrongly infer fact A_2. The interface will again detect that A_2 does not follow from the database. The interface cannot block this misconstrual, however, by inserting additional facts in the log because the user's perception of the domain of discourse differs from that captured by the rules of the database. The interface will then act differently and include in the log an indication that A_2 is not deductible from the database.

The user can still infer A_2 from the answer to his query. However, his inference will not be consistent with the current log, since the log indicates that A_2 does not hold.

The final answer will then be equivalent to the sentence:

Project Alpha is sponsored but it is not ongoing.

```
/* knowledge when session starts */

v: (ongoing P) <-
  current(Y) &
  v: (sponsors#[client¢*,dept¢*,proj¢P]\year(Y)).

/* query posed */

<- query(sponsors#[client¢*,dept¢*,proj¢'Beta']\year(1991), b).

/* knowledge added to log */

v:(sponsors#[client¢'Spock Ltd.', dept¢'D1', proj¢'Alpha']\year(1991)).
v:failed(ongoing 'Alpha').
```

Figure 3: User V

5. Conclusion

We described a declarative conceptual modelling language that is an integral part of a workbench that provides rapid prototyping at the conceptual level and that supports expert level features. We also provided simple examples to illustrate the direct use of the workbench over a database / knowledge base application, as well as the exploration of its use in connection with expert level features aiming at providing cooperative interfaces to information systems.

The prototype of the workbench is at an early stage of development, but it already implements all features of the language here described. It can be extended in several ways, either as a consequence of enriching the ER/ADT model, or to focus on the optimization of the algorithms and their implementation, among other points.

References

[AP] J. F. Allen and C. R. Perrault, "Analyzing intentions in utterances", *Artificial Intelligence* 15:3 (1980), 143-178.

[BJ] L. Bolc and M. Jarke (eds.), *Cooperative Interfaces to Information Systems*, Springer-Verlag (1986).

[CCL] W. Chu, Q. Chen and R-C. Lee, "Cooperative Query Answering via Type Abstraction Hierarchy", Proc. Int. Working Conference on Cooperating Knowledge based Systems, Univ. Keele, UK (1990).

[CD] F. Cuppens and R. Demolombe, "Cooperative answering: a methodology to provide intelligent access to databases", Proc. of the Second International Conference on Expert Database Systems, L. Kerschberg (ed.), Benjamin/Cummings (1989), 621-643.

[CF] M. A. Casanova and A. L. Furtado, "An Information System Environment based on Plan Generation", Proc. Int. Working Conference on Cooperating Knowledge based Systems, Keele, UK (1990).

[FN] R. E. Fikes and N. J. Nilsson - "STRIPS: a new approach to the application of theorem proving to problem solving" - Artificial Intelligence 2 (1971) 189-208.

[Fu1] A. L. Furtado - "Exploring the extensibility of IBM Prolog" - technical report CCR-124 - Rio Scientific Center of IBM Brasil (1991).

[Fu2] A. L. Furtado - "Two integrated tools for IBM Prolog: query-the-user & transparent use of SQL" - technical report CCR-126 - Rio Scientific Center of IBM Brasil (1991).

[HCF] A. S. Hemerly, M. A. Casanova and A. L. Furtado, "Cooperative behaviour through request modification", Proc. 10th Int'l. Conf. on the Entity-Relationship Approach, San Mateo, CA, USA (1991) 607-621.

[Ka] S. J. Kaplan, "Cooperative Responses from a Portable Natural Language Query System", *Artificial Intelligence* 19:2 (1982), 165-187.

[KW] A. Kobsa and W. Wahlster (eds.), *User Models in Dialog Systems*, Springer-Verlag (1989).

[LA] D. J. Litman and J. F. Allen - "A plan recognition model for subdialogues in conversations" - Cognitive Science 11 (1987) 163-200.

[Ll] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag (1987).

[Mo] A. Motro, "Query generalization: a technique for handling query failure", Proc. First International Workshop on Expert Database Systems (1984), 314-325.

[Qu] A. Quilici, "Detecting and Responding to Plan-Oriented Misconceptions", in *User Models in Dialog Systems*, A. Kobsa and W. Wahlster (eds.), Springer-Verlag (1989).

[TCF] L. Tucherman, M. A. Casanova and A. L. Furtado - "The CHRIS consultant - a tool for database design and rapid prototyping" - Information Systems 15:2 (1990).

[VF] P. A. S. Veloso and A. L. Furtado - "Towards simpler and yet complete formal specifications" - in "Information systems: theoretical and formal aspects" - A. Sernadas, J. Bubenko and A. Olive (eds.) - North-Holland Pub. Co. (1985) 175-189.

[We] B. L. Webber, "Questions, answers and responses: interacting with knowledge base systems", in *On knowledge base management systems*, M.L. Brodie and J. Mylopoulos (eds.) - Springer (1986).

APPENDIX A

```
% EXAMPLE DATABASE / KNOWLEDGE BASE


% declaring entity classes and attributes

entity(emp,name).
entity(trainee,name).
entity(dept,dname).
entity(proj,pname).
entity(client,cname).

trainee is_a emp.

attribute(emp,sal).
attribute(trainee,level).
attribute(dept,city).
attribute(proj,pending).
attribute(client,status).

relationship(works, [emp,dept]).
relationship(participates, [emp,proj]).
relationship(sponsors, [client,dept,proj]).

one_to_n(works, emp).

attribute(works,job).
attribute(sponsors,contract).
attribute(sponsors,year).
attribute(participates,task).

domain(name,V,is_string(V),single).
domain(dname,V,is_string(V),single).
domain(pname,V,is_string(V),single).
domain(cname,V,is_string(V),single).
domain(sal,V,is_numb(V),single).
domain(level,V,
       V == 'graduate' | V == 'undergraduate',single).
domain(city,V,is_string(V),single).
domain(pending,V,V == true,single).
domain(status,V,is_string(V),single).
domain(job,V,is_string(V),single).
domain(contract,V,
       stconc('c',N,V) & st_to_at(N,M) & is_int(M),single).
domain(year,V,in_range(V,1900,2000),single).
domain(task,V,is_string(V),multi).
```

```
% declaring data structures

structure(emp, [name,sal,dname,job]).
structure(dept, [dname,city]).
structure(pr, [pname,pending]).
structure(tr, [name,level]).
structure(acc, [name,account]).
structure(part, [name,pname]).
structure(tsk, [name,pname,task]).
structure(cln, [cname,status]).
structure(spon, [cname,dname,pname,year,contract]).

sql_structures([emp,dept]).


% mapping between classes and data structures

ext_ent_structure(emp, [works], emp).
ent_structure(trainee, tr).
ent_structure(dept, dept).
ent_structure(proj, pr).
ent_structure(client, cln).

rel_structure(participates, part).
rel_structure(sponsors, spon).

rep_rel_structure(participates, tsk).


% defining operations

operation(install, [dname,city]).
added(dept¢D, install(D,C)).
added(dept¢D\city(C), install(D,C)).

operation(propose, [pname]).
added(proj¢P, propose(P)).
added(proj¢P\pending(true), propose(P)).

operation(approve, [pname]).
deleted(proj¢P\pending(true), approve(P)).

operation(hire, [name,sal,dname]).
added(emp¢N, hire(N,S,D)).
added(emp¢N\sal(S), hire(N,S,D)).
added(works#[emp¢N,dept¢D], hire(N,S,D)).
```

```
operation(hire_tr,[name,sal,dname,level]).
added(F, hire_tr(N,S,D,L)) <- added(F, hire(N,S,D)).
added(trainee¢N, hire_tr(N,S,D,L)).
added(trainee¢N\level(L), hire_tr(N,S,D,L)).

operation(raise,[name,increment]).
added(emp¢N\sal(S), raise(N,I)) <-
  emp¢N\sal(S0) & S := S0 + I .
deleted(emp¢N\sal(S0), raise(N,I)) <-
  emp¢N\sal(S0).

operation(designate, [name,job]).
added(works#[emp¢N,dept¢*]\job(J), designate(N,J)).
deleted(works#[emp¢N,dept¢*]\job(J), designate(N,K)) <-
  works#[emp¢N,dept¢*]\job(J).

operation(fire, [name]).
deleted(emp¢X, fire(X)).

operation(assign, [name,pname,task]).
precond(assign(N,P,T),
  works#[emp¢N,dept¢D] &
  sponsors#[client¢*,dept¢D,proj¢P]).
added(participates#[emp¢N,proj¢P], assign(N,P,T)).
added(participates#[emp¢N,proj¢P]\task(T), assign(N,P,T)).

operation(add_task, [name,pname,task]).
added(participates#[emp¢N,proj¢P]\task(T), add_task(N,P,T)).

operation(associate, [cname,dname,pname,year,contract]).
added(client¢C,associate(C,D,P,Y,Cn)) <- ¬client¢C.
added(client¢C\status('new'),associate(C,D,P,Y,Cn)) <-
  ¬client¢C\status(*).
added(sponsors#[client¢C,dept¢D,proj¢P],
      associate(C,D,P,Y,Cn)).
added(sponsors#[client¢C,dept¢D,proj¢P]\year(Y),
      associate(C,D,P,Y,Cn)).
added(sponsors#[client¢C,dept¢D,proj¢P]\contract(Cn),
      associate(C,D,P,Y,Cn)).


% example of a knowledge-base rule

op("ongoing",prefix,50).

current(1991).
```

```
ongoing P <-
  current(Y) &
  sponsors#[client¢*,dept¢*,proj¢P]\year(Y) &
  ¬proj¢P\pending(true).
```

APPENDIX B

```
/* ALGORITHM TO AVOID MISCONSTRUALS */

query(X,U) <-
  U : X & /.

query(X,U) <-
  authorized(X) &
  X &
  nl &
  write(sys : X - succeeds) &
  log(U : X) &
  propagate(X,U).

propagate(X,U) <-
  forall( conditional_clause(U : Y <- B) &
          in_conjunction(U : X,B) &
          (delax(gb(*)) | true)  & addax(gb(nil)) &  U : Y,
          (gb(C) & inv(C,C1) & certU(C1,U) &
             (certD(C1,Z)   -> ( write(sys : Z - fails) &
                                 log_mis(Z,U));
                               propagate(Y,U)) & fail )).

/* certification test for the deductive database */

certD([],X)     <- / & fail.
certD([¬V!C],X) <- ¬V-> (/ & certD(C,X)); X=¬V .
certD([ V!C],X) <-  V-> (/ & certD(C,X)); X=V .

/* certification test for the log - an optimization */

certU([],U)     <- /.
certU([¬V!C],U) <- / & certU(C,U).
certU([V!C],U)  <- / & (U:failed(V)-> (/ & fail); certU(C,U)).

log_mis(¬X,U) <- addax(U:X,log,1) & write (¬X - blocked).
log_mis( X,U) <- addax(U:failed(X),log,1) & write (X - blocked).

log(X) <- addax(X,log) & write (X - added).
```

```
y_(X) <- gb(C) & delax(gb(C)) & addax(gb([X!C])).
n_(X) <- gb(C) & delax(gb(C)) & addax(gb([¬X!C])).

authorized(X).

/* utilities */

conditional_clause(X) <-
  ax(*,X).

in_conjunction(X,X) <- ¬X =.. ["&"!*] & /().
in_conjunction(X,X & *).
in_conjunction(X,* & R) <- in_conjunction(X,R).

inv(L,L1)<-inv(L,[],L1).
inv([],L,L)<-cut().
inv([H!L],LL,L1)<- cut() & inv(L,[H!LL],L1).
```