Treating Enhanced Entity Relationship Models in a Declarative Style*

Norbert Kehrer, Gustaf Neumann
Vienna University of Economics and Business Administration
Institute of Information Processing
Augasse 2–6
A–1090 Vienna, Austria

kehrer@wu-wien.ac.at, neumann@wu-wien.ac.at

Abstract

In this paper we present an approach to represent schema information, application data and integrity constraints in form of a logic program. An information system is specified by an enhanced entity relationship (EER) model which is transformed by means of a one-to-one mapping into a set of ground facts. The application data corresponding to the schema is represented by ground facts called observations. In order to check whether the application data conforms to the given schema, a set of general integrity rules is defined which expresses the dependencies (functional, inclusion and exclusion dependencies) implied by the EER model. In order to check whether the application EER model is a valid EER model a meta EER model is defined. Any application EER diagram appears as an instance of the meta EER diagram and can be represented using observations. This way the same set of integrity constraints can be used to check the conformance between the application data and the application EER diagram, the meta EER diagram and the application EER diagram. Since the representation of the meta EER diagram is an instance of the meta EER diagram, the validity of the meta EER diagram can be checked as well. The resulting logic program is composed of the application data, the application schema, the meta schema, a general set of constraints plus optionally additional application specific constraints and deduction rules.

1 Introduction

The ER approach [Che76] was introduced 15 years ago as a diagrammatical technique for unifying different database models. Several extensions to Chen's basic formalism were proposed to capture concepts like generalization [SS77] or categories [EWH85]. We will follow the enhanced entity relationship (EER) flavor as presented in [EN89]. In this paper we will not be concerned with certain EER constructs such as composite, derived or multi-valued attributes and predicate defined categories, or sub-/superclasses.

As the ER approach was seen primarily as a design tool mappings were developed for the implementation of ER models (eg. mapping to the relational model by [TYF86]). As a result the coupling between the ER models and its implementation was rather loose. In this paper we will follow a different approach where the ER model is viewed as an executable specification of an information system. An EER model is mapped by a simple one-to-one transformation from its graphical representation to a set of ground facts. This simple transformation can be performed either by hand or by a program we have developed which takes as its source the output of a public available graphical editor. In this paper we present a set of general integrity rules to check the application data against the schema. By using a meta EER diagram the schema of the EER model can also be checked.

In addition to the general EER specific integrity constraints, additional application specific constraints or further deduction rules may be added .

2 A Meta EER Diagram

The meta EER model in Figure 1 gives a short overview of the EER methodology and introduces the basic terminology. The diagram can be read as follows: The central EER concepts are *entity type* and *relationship type*. Both of these concepts are generalized to so called *modelled types*. Since a modelled type is either an entity type or a relationship type a disjoint generalization was used. A modelled type might be described by *attributes*. An attribute is identified by a *name* (identifying attribute) and characterized by its *type* (simple, identifying or multivalued). Composite attributes are constructed using the composite relationship type). Since the names of attributes are only unique per modelled type, attributes are modelled as weak entities with the modelled type as owner.

*Presented at the "*2nd Russian Conference on Logic Programming*", September 11-16, 1991, Leningrad, Proceedings in preparation.

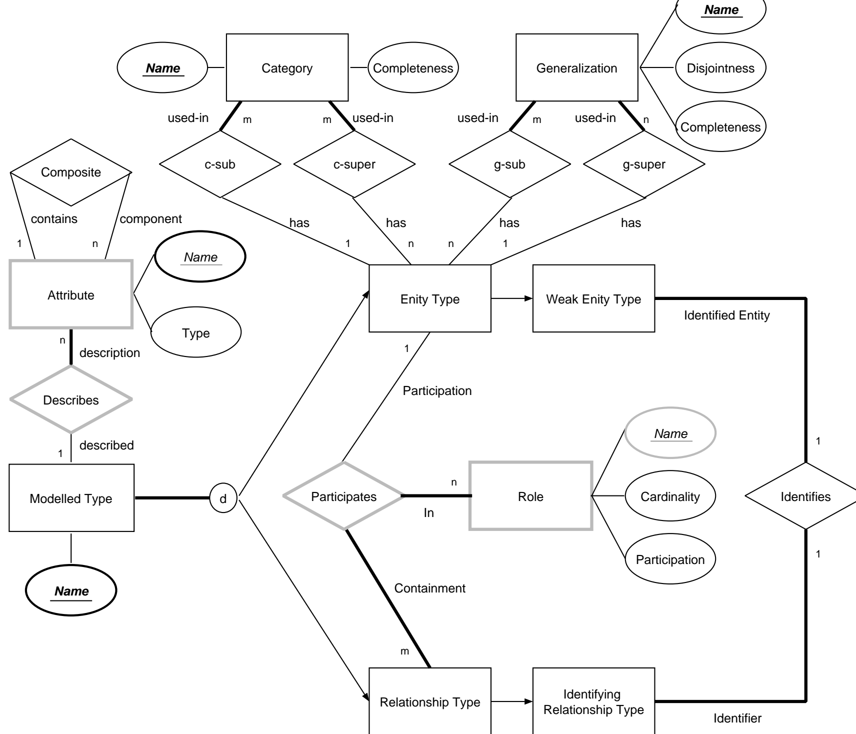

Figure 1: A Meta EER Diagram

Entity types and relationship types can be connected via *roles*, which are identified through a *name*, and which have a *cardinality* and a *participation* value. The role names are unique per relationship type, each occurrence of a relationship type participates in the *participates* relation (total participation). Each *weak entity type* (a subtype of entity type) is identified by an *identifying relationship type* (subtype of relationship type) and vice versa. The enhanced ER constructs *category* and *generalization* are used to define hierarchies of entity types. A generalization (identified by *name*, characterized by the attributes *disjointness* and *completeness*) has one entity type, a supertype and might have several (one or many) entity types as subtypes. A *category* on the contrary has one subtype and might have several supertypes.

3 The Mapping of the Schema and the Data of EER Diagrams into Logic

The information contained in an EER diagram can be separated into two components.

1. An extensional part containing the names of the concepts used in the EER model, a specifuc classification of these concepts (attribute, entity type, relationship type), the links between these basic concepts and the definition of certain properties of the concepts, and

2. an intensional part containing integrity constraints and deduction rules. In this paper we are concerned primarily with integrity constraints expressible within an EER model.

The extensional part of an application consists of the extensional part of the schema as specified by the EER diagram plus application data. The intensional part of the application is composed of the intensional part of the schema plus optionally additional application specific constraints over the data that cannot be expressed in an EER model (see section 5). The integrity rules will be used to check the conformance between the schema and the data. In order to check the wellformedness of a schema the meta EER diagram can be used.

The extensional part of an EER diagram can be obtained by performing a simple one-to-one mapping from the EER diagram to a set of facts. In our representation we represent an EER diagram in terms of the links between the basic EER concepts. These links are either roles, attributes, generalizations, or categories. In addition, a predicate is needed to identify weak entities. We are using in the following a Prolog like syntax for the logic formulas where constants start with lower case letters and where variables are capitalized. For better readability dashes are used to separate words within names.

1. One-to-one mapping of roles:

   ```
   role(Role-name, Rel-name, Ent-name, Cardinality, Participation)
   ```

   where *Cardinality* is either *one* or *many* and *Participation* is either *partial* or *total*.

2. One-to-one mapping of attributes:

```
attribute(Att-name, Mt-name, Type)
composite(Mt-name, Att-name, Att-name-component)
```

where *Type* is one of *simple*, *identifying* or *multivalued*. *Mt-name* stands for the name of a modelled type of the EER methodology, i.e. an entity type or a relationship type.

3. One-to-one mapping of generalizations:

```
generalization(Gen-name, Ent-name, Disjointness, Completeness)
g-sub(Gen-name, Ent-name)
```

where *Disjointness* is either *overlapping* or *disjoint* and *Completeness* is either *partial* or *total*.

4. One-to-one mapping of categories:

```
category(Cat-name, Ent-name, Completeness)
c-super(Cat-name, Ent-name)
```

where *Completeness* is either *partial* or *total*.

5. Identification of weak entity types:

```
identifies(Rel-name, Weak-ent-name)
```

The facts in items 1 to 5 keep the schema information in form of the one-to-one mapping from the diagram. All data of the application will be kept in the single predicate "*observation/4*":

6. Instances for the schema:

```
observation(Att-or-Role-name, Mt-name, Tupid, Value)
```

where *Att-or-Role-name* is the name of an attribute or a role, *Mt-name* is the name of a modelled type.

Tupid is an tuple identifier that uniquely determines a modelled type in the database. It is also used to group the various *Observation*s to a certain tuple (aggregation). The tuple identifier is a concept comparable to the surrogate in [Cod79]. Note that a representation based on *Observations* allows us to cope with null values (no observation available) or with multivalued attributes (several observations with identical first three arguments and different fourth arguments) in a simple and uniform manner.

A typical application consists of schema information (one-to-one mapping) and corresponding data in form of observations:

(1) $\mathit{schema}_{\mathit{application\ EER}} + \mathit{observations}^{\mathit{describing\ application\ data}}_{\mathit{application\ EER}} = \mathit{application\ program}$

By using the integrity rules introduced in the next section it is possible to check the data against the schema. In order to check the wellformedness of the application EER diagram the meta EER diagram can be used where the application EER diagram is given in form of observations for the meta EER diagram:

(2) $\mathit{schema}_{\mathit{meta\ EER}} + \mathit{observations}^{\mathit{describing\ application\ EER}}_{\mathit{meta\ EER}} = \mathit{schema}_{\mathit{application\ EER}}$

(3) $\mathit{schema}_{\mathit{meta\ EER}} + \mathit{observations}^{\mathit{describing\ application\ EER}}_{\mathit{meta\ EER}} + \mathit{observations}^{\mathit{describing\ application\ data}}_{\mathit{application\ EER}} = \mathit{application\ program}$

In order to check the wellformedness of the meta EER diagram the meta EER diagram itself can be expressed in terms of observations:

(4) $\mathit{schema}_{\mathit{meta\ EER}} + \mathit{observations}^{\mathit{describing\ meta\ EER}}_{\mathit{meta\ EER}} = \mathit{schema}_{\mathit{meta\ EER}}$

(5) $\mathit{observations}^{\mathit{describing\ meta\ EER}}_{\mathit{meta\ EER}} + \mathit{observations}^{\mathit{describing\ application\ EER}}_{\mathit{meta\ EER}} + \mathit{observations}^{\mathit{describing\ application\ data}}_{\mathit{application\ EER}}$
$= \mathit{application\ program}$

Item (5) shows that in principle the whole application could be specified only in terms of observations plus a single set of EER specific integrity rules. However, when the system is maintained "*manually*", it appears to be very hard to distinguish the various abstraction layers and to comprehend the observations. To reduce this disadvantage a simple set of rules can be given which deduces the representation of the one-to-one mapping from a set of observations. The integrity constraints in the next section assume the schema to be in the representation of the one-to-one mapping plus a single set of observations (like the items (1), (2) and (4)).

4 General Integrity Constraints of the EER Model

4.1 Functional Dependencies

Marking attributes as identifying and the specification of cardinalities of 1 in relationship types in an EER model are possible ways to express functional dependencies on the modelled data.

A functional dependency (FD) is a constraint on a relation R which states that the values of a tuple on one set of attributes X uniquely determine the values on another set Y of attributes. It is written as $X \Rightarrow Y$ and is formally defined by the following implication [GV89]:

$$t_1(X) = t_2(X) \rightarrow t_1(Y) = t_2(Y)$$

t_1 and t_2 are two different tuples of R. If the values on the set of attributes X are the same in t_1 and t_2 then the values on the attribute set Y have to be the same, too. In other words, a FD $X \Rightarrow Y$ is violated if there exist two tuples which have the same values in X and different values in Y. This can be expressed by the following rule:

```
not-determine(value(Att-LHS,Mt-LHS), value(Att-RHS,Mt-RHS)) :-
      two(value(Att-RHS,Mt-RHS), T1,T2),
      not two(value(Att-LHS,Mt-LHS), T1,T2).
```

This rule defines the violation of a functional dependency of the type *value(attribute-LHS)* $\Rightarrow$ *value(attribute-RHS)* (where the *attribute*s are atomic attributes). Since in our representation both the values and the tuple identifiers are accessible in the same way, we could express dependencies of the form *value(attribute-LHS)* $\Rightarrow$ *tupid(attribute-RHS)* or *tupid(attribute-LHS)* $\Rightarrow$ *value(attribute-RHS)* or *tupid(attribute-LHS)* $\Rightarrow$ *tupid(attribute-RHS)* with the same ease. We could generalize the rule as follows:

```
not-determine(LHS, RHS) :-
      two(RHS, T1,T2),
      not two(LHS, T1,T2).
```

In this clause *LHS* and *RHS* stand for *value(Att,Mt)* or *tupid(Mt)* (*tupid(Mt)* is a shorthand for *tupid(_,Mt)*). The *two* predicate is defined below. It uses the predicate *obs*, which is identical to *observeration* except for weak entities (see later).

```
two(value(Role,Type), Tuple0,Tuple1, V0,V1) :- obs(Role,Type,Tuple0,V0), obs(Role,Type,Tuple1,V1), not V0 == V1.
two(tupid(Role,Type), Tuple0,Tuple1, V0,V1) :- obs(Role,Type,Tuple0,V0), obs(Role,Type,Tuple1,V1), not Tuple0 == Tuple1.
```

In cases where the left hand side of a functional dependency is not atomic we use a clause to define the left hand side attributes and proceed as follows:

```
not-determine(FullLhs, RHS) :-
      extract_goal(FullLhs,Forall,LHS),
      two(RHS, T1,T2),
      not ( (Forall, two(LHS, T1,T2)) ).
```

This way the formulation for the violation *"for any two different RHS, all elements of the LHS must be equal"* is reformulated as *"... no element of the LHS is allowed to be different"*.

Identifying attribute determines tuple identifier

For each identifying attribute *Att* of a modelled type of the EER schema there exists a functional dependency between *Att* and the tuple identifier of the form:

value(identifying-att,modelled-type) $\Rightarrow$ *tupid(modelled-type)*

This corresponds to the definition of an identifying attribute as an attribute whose values can be used to identify an entity uniquely, because in our approach an entity is represented by its tuple identifier.

```
violated(value(Att,Mt) => tupid(Mt)) :-
    attribute(Att, Mt, identifying),
    not-determine(value(Att,Mt), tupid(Mt)).
```

It has to be noted that the identifying attribute of weak entity types does not determine the weak entity [Che76], but together with the owner entities it does. This could be expressed informally as

value(identifying-att + tupids-of-owners,weak-ent) $\Rightarrow$ *tupid(weak-ent)*

where + is used as a constructor. The concatenation of value and tuple identifiers of owners is achieved in the implementation within the *obs* predicate used by *two* and *not-determine*.

Tuple identifier determines singlevalued attributes

A (singlevalued) attribute is a function which maps from an entity set or a relationship set into a value set. For our representation this means that the value of each singlevalued attribute of the modelled type *Mt* is determined by the tuple identifier of *Mt*:

$$tupid(modelled\text{-}type) \Rightarrow value(any\text{-}attribute,modelled\text{-}type)$$

We need not check the constraint that the identifying attribute determines the values of the other attributes, because it follows from the two previous constraints:

$$value(identifying\text{-}att,modelled\text{-}type) \Rightarrow tupid(modelled\text{-}type) \wedge$$
$$tupid(modelled\text{-}type) \Rightarrow value(any\text{-}attribute,modelled\text{-}type) \rightarrow$$
$$value(identifying\text{-}att,modelled\text{-}type) \Rightarrow value(any\text{-}attribute,modelled\text{-}type)$$

Entities participating in a relationship type with cardinality *One*

Each role *R* of a relationship type *Rel* in which an entity type participates with cardinality *One* is determined by all other roles of *Rel* together [TYF86]:

$$value(other\ roles,rel\text{-}type) \Rightarrow value(one\text{-}role,rel\text{-}type)$$

This constraint is independent of the degree of the relationship type.

```
violated(value(other-roles,Rel) => value(Role1,Rel)) :-
    role(Role1,Rel,_,1,_),
    not-determine(other_role(Role1,Rel,Role2) ^ value(Role2,Rel), value(Role1,Rel)).
```

This is an example of a FD where the left hand side is not atomic, but consists of all the roles of a relationship type. The goal in front of ˆ computes each role of the LHS of the functional dependency.

All roles determine tuple identifier

All roles of a relationship type *r* together determine the tuple identifier of *r*:

$$value(all\ roles,rel\text{-}type) \Rightarrow tupid(rel\text{-}type)$$

This constraint can be expressed as:

```
violated(value(all-roles,Rel) => tupid(Rel)) :-
    relation(Rel),
    not-determine(role(Role,Rel,_,_,_) ^ value(Role,Rel), tupid(Rel)).
```

As above, this is an example of a FD with a not atomic left hand side.

4.2 Inclusion Dependencies

The use of relationship types, generalizations, and specializations in EER models indicates that entity or relationship sets are subsets of some other entity or relationship set. The property of being a subset of another set is covered by inclusion dependencies.

Inclusion dependencies (ID) specify that each member of some set *A* must also be a member of a set *B*. An inclusion dependency $A \subseteq B$ is violated iff there is an occurrence (value or tuple identifier) of *A* which is not an occurrence of *B*. The following rule expresses this content:

```
not-included(LHS,RHS) :-
    extract_goal(LHS,Goal,L),
    one(L,V),
    not (Goal,one(RHS,V)).
```

Like the rule for FD violations this rule may be used to check inclusion dependencies between tuple identifiers and attribute or role values in any combination. The predicate *one* is defined similar to the predicate *two* and returns either a tuple identifier or a value depending on its first argument.

Participating entities included in entity type

The values of a role of a relationship type must be tuple identifiers of the entity type participating in that role:

$$value(role,rel\text{-}type) \subseteq tupid(entity\text{-}type)$$

This ID is checked by the following rule:

```
violated(value(Role,Rel) << tupid(Ent)) :-
     role(Role,Rel,Ent,_,_),
     not-included(value(Role,Rel),tupid(Ent)).
```

Totally participating entity types

For entity types which participate totally in a relationship type the previous ID has to hold in the other direction, too. Each tuple identifier of an entity type e must be a value of a role in which e participates totally:

$$\textit{tupid(entity)} \subseteq \textit{value(role,rel-type)}$$

Generalizations

A generalization may be total or partial. A total generalization specifies the constraint that every entity in the superclass must be a member of some subclass in the specialization [EN89]. For our representation this means that in a total generalization with supertype *super* there must be at least one subclass *sub* for each tuple identifier T of *super*, where T is included in *sub*:

$$\textit{tupid(supertype-in-total-gen)} \subseteq \textit{tupid(at-least-one-subtype)}$$

This constraint is valid only for total generalizations. There is also an inclusion dependency *tupid(subtype)* $\subseteq$ *tupid(supertype)* for both partial and total generalizations. In our representation we guarantee through the use of deduction rules that the tuple identifiers of supertypes are also tuple identifiers of the subtypes. Therefore this ID needs not to be checked.

Categories

Similar to generalizations we have to assure that for total categories tuple identifiers of the superclasses are also tuple identifiers of the subclass in a category and that the attributes are inherited. This mechanism may not be applied to partial categories, because not every entity of a supertype has to be member of the subclass. Instead the members of the subclass in partial categories have to be stated explicitly. Therefore we will have to check if the tuple identifiers and attribute values of a subclass in a partial category occur in one of the superclasses specified for the category, which is expressed by the inclusion dependencies:

$$\textit{tupid(subclass)} \subseteq \textit{tupid(some-superclass)}$$
$$\textit{value(att,subclass)} \subseteq \textit{value(att,some-superclass)}$$

The mechanism of attribute inheritance will be described in more detail in a later section.

All roles in a relationship must be specified

In each relationship instance the associated entities have to be specified. This constraint is violated if there are two different roles *role1* and *role2* in a relationship type *rel* and the set of tuple identifiers of *rel* which have a value for *role1* is a proper subset of the tuple identifiers of *rel* having a value for *role2*. A is a proper subset of B if $A \subseteq B$ and not $B \subseteq A$. So the constraint for the violation using the $\subseteq$ operator may be written as:

$$\textit{tupid(role1,rel-type)} \subseteq \textit{tupid(role2,rel-type)} \wedge \neg\, \textit{tupid(role2,rel-type)} \subseteq \textit{tupid(role1,rel-type)}$$

4.3 Exclusion Dependencies

An exclusion dependency (ED) is the constraint indicating that no member of a set A is a member of a set B (empty intersection). An ED $A \rightleftharpoons B$ is violated iff a tuple identifier of A is also a tuple identifier of B. We define this constraint violation only for tuple identifiers and not for attribute values or roles because this is not expressible in the EER methodology. This constraint can be formulated simply as:

```
not-excluded(LHS,RHS) :-
    one(LHS,V),
    one(RHS,V).
```

Disjoint subclasses

A disjointness constraint on a generalization specifies that the subclasses in the generalization must be disjoint [EN89]. This constraint can be expressed by mutual exclusion dependencies between all the subclasses. Let *sub1* and *sub2* be two different subclasses of a disjoint generalization. If a tuple identifier of *sub1* is also a tuple identifier of *sub2* the ED of disjoint subclasses is violated:

$$\textit{tupid(disjoint-subclass-1)} \rightleftharpoons \textit{tupid(disjoint-subclass-2)}$$

Unique Tuple Identifiers

Two different modelled types *mt1* and *mt2* may not contain the same tuple identifier unless *mt1* is a subtype of *mt2* or *mt2* is a subtype of *mt*. The goal *subtype(X,Y)* checks if one entity type is a subtype (via a subclass or category) of another entity type.

```
violated(tupid(Mt1) === tupid(Mt2)) :-
      not-excluded(tupid(Mt1),tupid(Mt2)),
      Mt1 /== Mt2,
      not subtype(Mt1,Mt2),
      not subtype(Mt2,Mt1).
```

4.4 Schema Conformity

The conformity of the database with the schema – i.e. the attributes, roles, and modelled types appearing in *Observation* facts must be specified in the schema – cannot be checked by one of the above dependencies, because the rules only refer to data integrity whereas the schema conformity may be viewed as an inclusion dependency between data and schema representation. Therefore we use a separate consistency rule for this dependency.

An observation containing an attribute or role *ra* of a modelled type *mt* does not conform to the schema if *ra* or *mt* have not been specified in the schema correspondingly.

```
violated(not-in-schema(Ra,Mt)) :-
    observation(Ra, Mt, T, V),   not attribute-or-role(Ra, Mt).
```

4.5 Type Hierarchy

The concepts of the generalization and category allow the construction of a hierarchy of entity types. In our representation we use a mechanism for the inheritance of attributes in that hierarchy and for the inclusion of tuple identifiers of one entity type in other entity types. It is built upon the following rules:

- In a generalization the tuple identifiers and attributes which were stated in an *Observation* for a subtype become tuple identifiers and attributes of the supertype. The entity – represented by the tuple identifier – belongs to both types. The name of the supertype is an alias for the name of the subtype. Therefore we call this process "aliasing".
- In a total category the attributes specified for a superclass are inherited by the subclass, and the tuple identifiers of the superclass become tuple identifiers of the subclass (aliasing).

Nonetheless, additional *Observations* may be specified for the superclass in a generalization and for a subclass of a category. Therefore the corresponding inclusion dependencies, which we described earlier, have to be checked.

The inheritance of attributes and the aliasing of tuple identifiers are performed by deduction rules for a predicate called *Observation-In-Hierarchy*. This predicate has the same arguments as *Observation* and covers all *Observations* plus the ones that result from the inheritance mechanism. Actually the integrity constraint definitions are based upon this predicate except in the cases where *Observation* is used explicitly.

5 Application specific Integrity Constraints

For certain applications, it will be necessary to specify integrity constraints which are not expressible within an EER diagram. Let us look at the meta EER diagram again to identify such application specific constraints. In the meta EER diagram there are essentially two types of constraints missing: domain restrictions and exclusion of certain (recursive) definitions. An example of a missing domain restriction would be to specify that the *participation* of a *role* is either *total* or *partial* (or that the *role* in which a weak entity type participates in an *identifying Relationship* must be *total*, the names of roles and attributes of a relation must be disjoint). An example of an EER construction that should be forbidden would be if an entity-type is *owner* of itself (or if a *subtype* is its own *supertype*, two supertypes in a generalization do not have a common root). Another constraint would be that each relationship type must have at least two roles attached. We call such integrity constraints "*application specific*" (as opposed to the general constraints valid for all EER models), since they are only needed for testing the schema (i.e. the application meta-EER).

6 Using the Integrity Constraints and Stratification Problems

When the consistency rules of the last section are used the observation facts are checked with respect to the schema information. Within this test the rules are assumed to be correct, some of the facts might be invalid or missing (constraint satisfaction problem). A straightforward implementation of the integrity constraints (eg. in Prolog) will

lead to the naive and exhaustive approach where all constraints are applied on all observations each time the set of observations is modified. This is no practical solution for applications with realistic sizes and further investigations are necessary to reduce the search space.

In addition the integrity constraints might be used to enforce integrity and to trigger from an update of the facts additional, so called "*induced*" updates (see for example [BDM87]).

Another interesting problem is to reason about inconsistencies and to try to derive consistent subsets of the set of observations even if some observations are invalid. A convenient formulation of the predicate "*consistent-observation*" is typically a non-stratified problem. To illustrate this problem we will use a small example using a single entity type *a* with two attributes *i* (for an identifying attribute) and *o* (for another non-identifying attribute) and two constraints:

```
    observation(a,i,t1,1).
    observation(a,o,t1,x1).
    observation(a,i,t2,2).
    observation(a,o,t2,x2).
    observation(a,i,t3,1).
    observation(a,o,t3,x3).

[1]  fdv(E,i,T,V) :- obs(E,i,T,V), obs(E,i,T2,V), not (T==T2).
[2]  nmv(E,A,T,V) :- obs(E,A,T,V), other-att(A,A1), not obs(E,A,T,V1).

[3]  consistent-observation(E,A,T,V) :-
         observation(E,A,T,V), not fdv(E,A,T,V), not nmv(E,A,T,V).

[4a] obs(T,A,V) :- observation(E,A,T,V).
[4b] obs(T,A,V) :- consistent-observation(E,A,T,V).
```

The consistency rule *fdv* stands for functional dependency violated, *nmv* means no-missing-violated (all attributes must be specified). For simplicity *other-att* is defined here as:

```
[5] other-att(i,o).
[6] other-att(o,i).
```

The clauses [1,2,3,4a,5,6] represent the stratified approach where the integrity constraints are based directly on the observations. It can be easily seen that *fdv* holds for *t1* and *t3* and that *nmv* never holds. If the integrity constraints are based upon *consistent-observation* instead of *observation* by using [4b] instead of [4a], the program is non-stratified, since there is now a "*recursion going through a negation*". The intended semantics is that now the observations of the *o*-attributes are also removed from the set of *consistent-observations*. The set of consistent observations that is true regardless of the order in which the consistency rules are applied consists only the observations containing *t2*.

For this example the same results could be obtained using a layered stratified program, where the first consistency rule *fdv* would be based on the given observations resulting in a set *obs1*. The set *obs1* could be used as a basis for the second consistency rule *nmv* leading to the final consistent set. Different results will be obtained when the order of the consistency rules is changed. The second disadvantage of the layering approach using stratified programs is that in the general case, where *N* consistency rules are given, *N!* different layerings are possible, which will result in different models. The intended "*save subset*" of consistent observations should contain only the intersection of these models.

References

[ABW87] C. Apt, H. Blair, A. Walker: "*Towards a Theory of Declarative Knowledge*", in Minker (ed.): "*Foundations of Deductive Databases and Logic Programming*", Morgan Kaufmann, Los Altos 1987.

[BDM87] F. Bry, H. Decker, R. Manthey: "*A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases*", Proceedings of EDBT 88, May 1988, Venice.

[Che76] P. Chen: "*The Entity Relationship Model – Toward a Unified View of Data*", Transaction of Database Systems, 1:1, March 1976.

[Che91] W.C. Cheng: "*Tgif 2.6 - A Xlib based drawing facility under X11*", available via anonymous ftp from export.lcs.mit.edu, May 1991.

[Cod79] E. Codd: "*Extending the Database Relational Model to Capture More Meaning*", Transactions of Database Systems, 4:4, December 1979.

[DZ88] P.W. Dart, J. Zobel: "*Conceptual Schemas Applied to Deductive Database Systems*", Information Systems, Vol. 13, pp. 273–287, 1988.

[EN89] R. Elmasri, S.B. Navathe: "*Fundamentals of Database Systems*", Benjamin/Cummings, Redwood City, Calif. 1989.

[EWH85] R. Elmasri, J. Weeldreyer, A. Hevner: "*The Category Concept: An Extension to the Entity-Relationship Model*", International Journal on Data and Knowledge Engineering, 1:1, May 1985.

[GV89] G. Gardarin, P. Valduriez: "*Relational Databases and Knowledge Bases*", Addison-Wesley, Reading 1989.

[MS89] V. M. Markowitz, A. Shoshani: "*On the Correctness of Representing Extended Entity-Relationship Structures in the Relational Model*", in J. Clifford, B. Lindsay, D. Maier: "*Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*", ACM, New York 1989, pp. 430–439.

[TYF86] T.J. Teorey, D. Yang, J.P. Fry: "*A logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model*", ACM Computing Surveys 18, 2, June 1986.

[Teo90] T.J. Teorey: "*Database Modeling and Design: The Entity-Relationship Approach*", Morgan Kaufmann, San Mateo, Calif. 1990.

[SS77] J. Smith, D. Smith: "*Database Abstractions: Aggregation and Generalization*", Transactions of Database Systems, 2:2, June 1977.