Reasoning about ER Models in a Deductive Environment*

Gustaf Neumann

University GH Essen

FB 5, Information Systems and Software Techniques

Altendorferstraße 97, Eingang B, D-45143 Essen, Germany

neumann@wi-inf.uni-essen.de

Fax: +49 (0201) 81003-73

Abstract. In this paper we present an approach to represent schema information, application data and integrity constraints as a logic program in the form of Datalog. The schema information is supplied as an enhanced entity relationship (EER) model which is transformed by an one-to-one mapping into a set of ground facts. The application data corresponding to the schema is also represented by ground facts in a single table. In order to check whether the application data conforms to the given schema, generic functional and inclusion dependencies are introduced, which can be mapped into concrete dependencies using the schema information. The concrete dependencies are used for checking the consistency between application data and the schema. The formulation of the constraints based on functional and inclusion dependencies led to a small extension of the EER model by allowing identifying attributes in relationship types. This extension leads to both simpler constraints and simpler EER application models.

Furthermore, we provide a meta EER model which can be used to check whether the application EER model is valid. Any application EER diagram

* Published in: Data & Knowledge Engineering, 19, p. 241-266, June, 1996.

is an instance of the meta EER diagram which can be specified using the proposed application data representation. The same integrity rules 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, and finally it can be used to check the meta EER model itself.

Keywords: Conceptual Modeling, Logic Programming, Expert Systems, Prototyping, SYLLOG.

1 Introduction

The entity relationship (ER) approach is undoubtedly a very popular tool for the communication between database designers and users of a database system. In our opinion this popularity is due to the fact that ER models have an easily explainable graphical representation, and that ER models can be mapped in a systematic way to a relational database schema [24]. Extensions of the basic formalism of Chen [3] were proposed by various authors to capture concepts like generalization [20] or categories [8]. In this paper we will follow mostly the enhanced entity relationship (EER) flavor as presented in [7]. We will not deal with certain EER constructs such as composite, derived or multi-valued attributes, categories and predicate defined sub-/superclasses.

We will present a prototyping environment based on a deductive database system for EER model designers in the form of an executable EER specification. The resulting system allows a database designer to model the schema of a database application together with some sample data, to experiment with various design approaches and to refine the EER model if necessary. We provide integrity constraints that help the database designer understand the consequences of various design approaches. The integrity constraints may be used to check the consistency of both, the application EER model (by checking it against the meta EER model) and the application data.

To accomplish this goal we develop a representation of EER models and a representation of the application data consisting only of a single table, which can be provided at design time. The representation of EER models is derived from a meta EER model.

The consistency rules for EER models are expressed by generic dependencies (mostly generic functional dependencies and generic inclusion dependencies) which are mapped by the deduction system into concrete dependencies using the schema information. The database designer can obtain these dependencies at any time during the design process to get a better understanding of the implications of the EER constructs used.

The approach to define the semantics of the EER model via functional and inclusion dependencies led to a natural extension of the EER model, namely allowing identifying attributes for relationship types. This extension is very similar to the treatment of weak entity types. By defining semantics for identifying attributes for relationship types, the special rules for handling weak entity types (that were necessary in the predecessor work [13]) became obsolete, the number of generic integrity constraints could be reduced. We will give the semantics for identifying attributes in relationship types later as well as examples where this extension leads to simpler EER models.

In this paper we will not address the mapping of EER models into relational schemata in order to obtain an efficient information system for a relational database system in production use. One approach for this step is to use techniques like the one presented in [24]. However, the derived functional and inclusion dependencies can be exploited for this purpose as well. As [18] shows, the derived dependencies can serve as a sufficient representation of the EER model and can be used in a normalization algorithm (such as Beeri and Bernstein [2]) as the only input to generate relational database schemata. The inclusion dependencies can be used to preserve the referential integrity implied by the EER model. Thus the dependencies are sufficient to represent the contents of EER models. By the use of a normalization algorithm additional dependencies can be added without the need of changing the mapping algorithm to relational schemata to incorporate the new dependencies.

In the earlier work [18] we showed that the derived dependencies (together with inference rules) can be used for schema inclusion (all dependencies from one model are implied by the dependencies of another model) and schema equivalence (bidirectional inclusion). However, this work was based on a Prolog implementation and covers inferences over dependencies, normalization algorithms etc. In this paper we use the declarative language SYLLOG [25, 14], where we do not have to address control flow and termination issues. The SYLLOG system is equipped with a powerful explanation component that provides positive and negative explanations of answers in the form of English sentences. It allows the EER modeler to ask questions like "*Why is the integrity violated?*" or "*Why is this dependency a consequence of my model?*".

In general, our approach does not rely on the SYLLOG system. With primarily syntactic differences the same logic formulation can be used in a variety of logic based systems. Predecessor work using Prolog is discussed in [12, 18] where in addition to specification skills programming skills are needed to achieve termination etc. We think that the same approach could also be implemented in other systems capable of dealing with negations in stratified Datalog programs such as LDL [19], RDL [11] or XSB [21]. The cited systems provide, however, no explanation facilities, which are very convenient especially for modeling purposes.

Our work was influenced by Dart and Zobel [6], who developed a first order specification of inference rules together with a set of integrity constraints for a graphical information systems specification language. In contrast to our work, they propose a new formalism called LOCS. We base our work on the well established and well known EER approach. Dart and Zobel [6] do not mention any attempt to check the well-formedness of the application schema using the same integrity constraints.

Finally, when we talk about prototyping in this paper we refer to prototyping on a conceptual level, i.e. prototyping of the semantics and constraints of the user data. This papers does not address other application development issues like user interfaces, transactions, locking, output, etc.

Section 2 describes our general framework, Section 3 addresses the representation of EER models and its data in SYLLOG, Section 4 presents the general integrity constraints, Section 5 covers integrity checking and explanations, Section 6 discusses the meta EER diagram with its applications, and Section 7 gives a conclusion.

2 Using a Deductive Database System for EER Modeling

Relational databases represent extensional knowledge (facts) in the form of tables. Deductive databases enhance relational databases by deduction rules, which are used to derive new facts from given facts. As a uniform representation of facts and rules Datalog clauses are used. A pure Datalog clause is a logical implication of the form "$A \leftarrow B_1 \wedge \ldots \wedge B_n$", where A and all B_i are positive (non-negated) literals without any function symbols. If the premise of the Datalog clause is empty ($n = 0$) it is referred to as a fact otherwise it is called a rule.

As mentioned in the introduction we use the deductive database system SYLLOG, which provides a near-English representation of facts and rules and a comfortable user interface. It extends pure Datalog by allowing negated literals in the premise of a rule as long as the database remains stratified [1]. SYLLOG is implemented using a backchain iteration procedure which gives clear semantics to stratified logic knowledge bases. The syntax of SYLLOG facts and rules will be described when needed.

In the deductive database system we are able to represent (a) the schema information, (b) application data together with (c) consistency rules and (d) application-specific deduction rules and constraints. The consistency rules that reason about the model and its data are a convenient way to define the meaning of EER concepts.

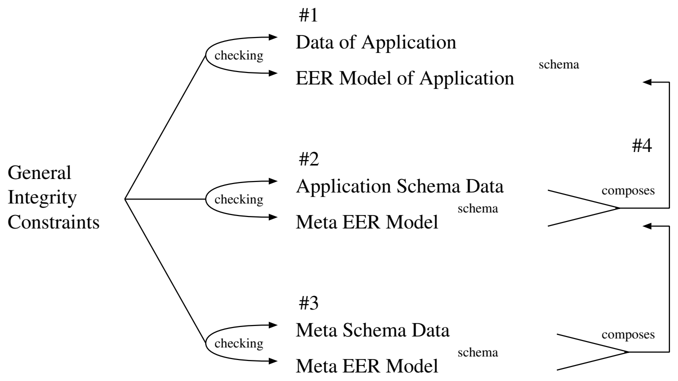


Fig. 1. Using General EER Constraints to Check Application and Schema Integrity

In our approach a given application EER diagram is mapped one-to-one to a set of facts. The application data will be given as facts as well. By applying general consistency rules we are able to check the conformance of data and schema (see Figure 1, #1). Since in our representation schema and data are kept in the same database, a meta EER diagram can be used to reason about the well-formedness of application EER diagrams by applying the general integrity constraints (Figure 1, #2). The application schemata are formulated as instances of the meta EER diagram. Similarly the well-formedness of the meta EER diagram is checked (Figure 1, #3). The resulting checked database has the form of a Datalog program which can be extended with deduction rules or additional constraints (which exceed the expressiveness of the EER methodology) as needed by the application. Such constraints will be discussed in Section 6. Additional deduction rules can be used for example to derive the schema information of the application EER model from the application schema data and the meta EER model (Figure 1, #4).

In addition to these general integrity constraints application-specific constraints which refer to schema or data concepts of a particular application can be formulated if necessary as an option.

3 Representing EER Diagrams and its Data in Datalog

The information contained in EER diagrams can be separated into two components:

1. An extensional part containing the names of the concepts used in the EER model, a certain 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 that are induced by the EER model.

The extensional part consists of the facts of the EER schema plus application data, the intensional part of EER specific rules and optionally additional application-specific constraints. The integrity rules will be used to check the conformance between the schema and the data. In order to check the well-formedness of a schema itself we will introduce a meta EER diagram.

The extensional part of an EER diagram is obtained by performing a one-to-one mapping from the diagram to a set of facts. We represent EER diagrams in terms of the links between basic EER concepts. These links are either roles, attributes, generalizations or weak entity types. Later, we will refer to the union of the roles and attributes as *descriptive elements (DE)*, and to the union of entity types and relationship types as *object types (OT)*.

3.1 The One-to-one Mapping of Basic EER Constructs

The Syllog sentences below describe the database schema for representing EER models. In general, a Syllog sentence is a line of words. Words starting with a prefix like *some-*, *the-*, *a-* etc. are logical variables. Words differing only in their prefix denote the same variable in Syllog.

One-to-one mapping of roles:

`Role` *a-name the-relationship-type*

the-entity-type `has` *a-cardinality a-participation*

where *a-cardinality* is either *"One"* or *"Many"* and *a-participation* is either *"Partial"* or *"Total"*. In EER diagrams roles are arcs connecting entity types and relationship types. All roles are labeled with their names and their cardinalities (1 for *"One"*, n or m for *"Many"*). Thick role arcs denote total participations, thin lines denote partial participations.

One-to-one mapping of attributes:

`Attribute` *the-attribute the-OT* `is` *eg-identifying*

where *eg-identifying* is either *"Simple"* or *"Identifying"*. *the-OT* is the name of an object type, i.e. an entity type or a relationship type. Graphically attributes are drawn as ellipses, identifying attributes are underlined.

One-to-one mapping of generalizations:

`Generalization` *a-gen-name* `supertype` *the-ET the-disjointness the-completeness*

`Subclass in generalization` *a-gen-name* `is` *an-ET*

where *the-disjointness* is either *"Overlapping"* or *"Disjoint"* and *the-completeness* is either *"Partial"* or *"Total"*. Generalizations are denoted in the diagrams as undirected arcs leading from a supertype to a small circle containing either a d (for disjoint subclasses) or an o (overlapping subclasses). The circle is connected with the supertype and the subtypes by arrows pointing from the subtype towards the supertype. A thick line connecting the supertype and the small circle indicates a total generalization. In cases where a supertype has a single subtype a partial overlapping generalization is assumed. In our representation a unique generalization name (*a-gen-name*) is used to represent the circle symbol. The supertype of a generalization is included in the predicate *Generalization*. Subtypes are specified in a separate table.

Identification of weak entity types:

`Owner of` *the-weak-type* `is` *an-ET an-ident-rel-name*

Weak entity types and identifying relationship types are drawn in a gray box using a thicker line style. For the representation it is sufficient to represent a weak entity type in form of a regular entity type together with the "`Owner of`" fact referencing to it. The identifying relationship should be mentioned only here and is assumed as a binary relationship without attributes.

3.2 Reasoning about the Structure of EER Schemata

Deriving Descriptive Elements: The facts described above can be queried from Syllog in a similar way as it could be done in a relational database system. It is straightforward to define rules based on these facts to obtain more problem adequate answers. For example generalization rules can be defined to deduce all descriptive elements. A Syllog rule consists of one or several Syllog sentences (premises) followed by a single line and one or several Syllog sentences (conclusions). For every binding of the logical variables in the premise for which the premises are true the conclusion with these bindings is a logical consequence. The conclusions of rules can be used in premises of other Syllog rules, or they can be used like all defined Syllog sentences in queries.

The following rules define that a descriptive element *DE* is either an attribute or a role, and that a role occurs in relationship types or in weak entity types..

`Attribute` *the-DE the-OT* `is` *eg-identifying*

`---------------------------------------`

```
DE eg-DE is a "attribute" of the-OT


Role the-DE the-OT references some-ET
--------------------------------------
DE eg-DE is a "role" of the-OT


Role the-role the-RST the-ET has the-cardinality the-participation
-------------------------------------------------------------------
Role some-role the-RST references some-ET


Owner of the-weak-ET is an-Owner an-ident-rel-name
--------------------------------------------------
Role an-ident-rel-name the-weak-ET references the-Owner
```

When these rules are defined we can use the newly defined sentences to query for example all the descriptive elements or all roles in the specified schema.

Notice the fourth rule which defines the name of the identifying relationship as a role of a weak entity type. This is an unusual approach that simplifies many of the integrity constraints significantly. A role is a reference to a tuple identifier of an entity type, and this is how it is used here as well. In the traditional approach the roles are only allowed in relationship types. Relationship types have two or more roles. In our approach a weak entity type is an object type with a single role.

Attribute Inheritance in Generalizations: In a generalization all attributes from a supertype are inherited by the subtypes. The one-to-one mapping simply states that a generalization exists but it does not state any kind of implications of that fact. In order to define attributed inheritance based on the one-to-one mapping it is in a first step necessary to define a predicate to derive the immediate subtype information from the base facts.

```
Generalization a-gen supertype the-Super the-disjointness the-completeness
Subclass in generalization a-gen is a-SubType
------------------------------------------------------------
eg-SubType is an immediate subtype of eg-Super in a-gen
```

In order to specify the actual attribute inheritance two rules implementing a transitive closure are used.

```
eg-Sub is an immediate subtype of eg-Super in a-gen
Attribute the-DE the-Super is eg-ident
-------------------------------------------------------------------
DE attribute the-DE the-Sub is inherited from eg-Super in some-gen

DE some-kind the-DE the-Super is inherited from eg-Super2 in some-gen
eg-Sub is an immediate subtype of eg-Super in a-gen2
--------------------------------------------------------------------
DE some-kind the-DE the-Sub is inherited from eg-Super2 in some-gen2
```

Finally it is straightforward to define rules which allow to refer with the same sentence to specified and inherited attributes:

```
DE attribute the-attribute the-OT is inherited from eg-Super in some-gen
----------------------------------------------------------------------
DE that-attribute is a "attribute" of that-OT
```

Similarly to the last rule the type of attribute can be obtained as well. All later rules reasoning about descriptive elements are based on these inheritance rules.

3.3 A Simple Application Schema and its Extensional Representation

The schema representation introduced above will now be used to represent a sample application EER diagram (Figure 2). It should be noted that this mapping is easy

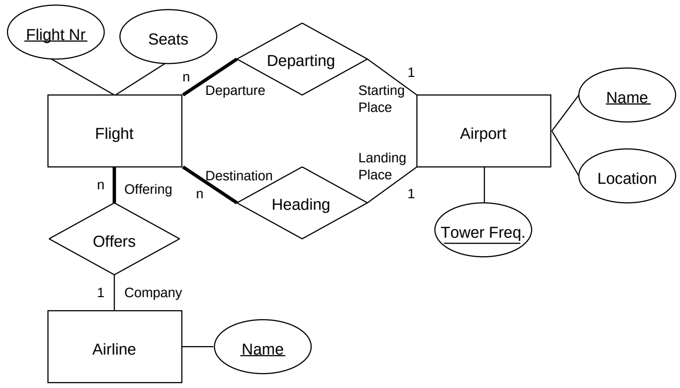


Fig. 2. An EER Diagram Specifying a Simple Airline Information System

enough to be done by a fairly simple transformation program. We have developed such a program that transforms EER diagrams drawn with the graphical editor TGIF [4] into a relational schema conforming to the schema representation. Since the EER diagram in Figure 2 contains only entity types, relationship types, and attributes it suffices to use *Role* and *Attribute* sentences.

In SYLLOG a table is written as a SYLLOG sentence, followed by two lines, followed by the values. All facts are ground (i.e. variable free). The following two tables are the complete representation of the sample diagram in Figure 2.

`Role` *the-name*	*the-relationship-type*	*the-entity-type* `has`	*some-cardinality*	*some-participation*
`Starting-Place`	`Departing`	`Airport`	`One`	`Partial`
`Departure`	`Departing`	`Flight`	`Many`	`Total`
`Landing-Place`	`Heading`	`Airport`	`One`	`Partial`
`Destination`	`Heading`	`Flight`	`Many`	`Total`
`Company`	`Offers`	`Flight`	`Many`	`Total`
`Offering`	`Offers`	`Flight`	`One`	`Partial`

`Attribute` *the-attribute*	*the-OT* `is`	*the-type-of-attribute*
Name	Airport	Identifying
Location	Airport	Simple
Tower-Freq	Airport	Identifying
Flight-Nr	Flight	Identifying
Seats	Flight	Simple
Name	Airline	Identifying

3.4 A Set of Instances for the Airline Schema

In order to make the EER diagram executable it is necessary to store the data for the diagram in the same SYLLOG knowledge base. To achieve maximum flexibility we decided to use a fairly atomistic representation schema based on so-called *Observations*. An observation is a fact determining that some attribute (or role) belonging to a certain entity type (or relationship type) has for a given object a certain value. Thus all instances of the EER schema are defined by a single predicate *Observation* with this four arguments.

`Observation for` *the-DE the-OT* `is` *the-tupid* `with` *some-value*

The first two arguments refer to the schema information, the third argument *tupid* stands for tuple identifier and is used to group the various *Observation*s to a certain tuple (aggregation). The *tuple-identifier* uniquely determines the object in the database. The tuple identifier is a concept comparable to the surrogate in [5].

Note that a representation based on *Observations* allows us to cope with null values (no observation available) or with multi-valued attributes (several observations with identical first three arguments and different fourth arguments) in a simple way. Rumbaugh et.al. [22] describe independently from our work a very similar approach for storing object models in a relational database system.

Observation for *the-DE*	*the-OT* is	*the-tupid* with	*some-value*
Name	Airline	AL1	Aua
Name	Airline	AL2	Malev
Name	Airport	AP1	Wien-Schwechat
Location	Airport	AP1	Schwechat
Tower-Freq	Airport	AP1	123
Name	Airport	AP2	Jfk
Location	Airport	AP2	New-York-City
Tower-Freq	Airport	AP2	222
Flight-Nr	Flight	F1	123
Seats	Flight	F1	130
Flight-Nr	Flight	F2	234
Seats	Flight	F2	170
Starting-Place	Departing	D1	AP2
Departure	Departing	D1	F1
Landing-Place	Heading	H1	AP1
Destination	Heading	H1	F1

One might argue that this representation is very atomistic and hard to use. Nevertheless the representation allows us to access the application data in a very general way (for example from the integrity rules) and simplifies schema modifications significantly (e.g. introducing a new attribute).

In order to provide a more traditional view of the data specified in the *Observation* facts one might provide rules. The following two SYLLOG rules show how to make the data more accessible and how to specify additional rules which are not expressible in the EER model (recursively defined transitive closure).

Observation for Flight-Nr Flight is	*some-tuple-flight*	with	*some-flight*
Observation for Seats Flight is	*some-tuple-flight*	with	*some-seats*
Observation for Offering Offers is	*some-tuple-offers*	with	*some-tuple-flight*
Observation for Company Offers is	*some-tuple-offers*	with	*some-tuple-airline*
Observation for Name Airline is	*some-tuple-airline*	with	*some-airline*
Observation for Departure Departing is	*a-tuple-departing*	with	*some-tuple-flight*
Observation for Starting-PlaceDeparting is	*a-tuple-departing*	with	*some-airport-a*
Observation for Destination Heading is	*a-tuple-heading*	with	*some-tuple-flight*
Observation for Landing-Place Heading is	*a-tuple-heading*	with	*some-airport-b*
Observation for Name Airport is	*some-airport-b*	with	*the-place-b*
Observation for Name Airport is	*some-airport-a*	with	*the-place-a*

---

some-flight is a flight of *an-airline* from *a-place-a* to *a-place-b* with *some-seats* seats

one can fly from *some-place-a* to *some-place-b*

one can fly from *some-place-a* to *some-place-b*

one can fly from *some-place-b* to *some-place-c*

---

one can fly from *some-place-a* to *some-place-c*

Rules like the first one can be found by applying a procedure as suggested in [24], [23], or [7] to transform an EER model into a relational schema. These procedures are oriented towards generating a small number of relations together with their attributes. These relations are used as conclusions of SYLLOG rules where the attributes are stated as variables. The premises of the rules are formed by grouping together the *Observations* that are needed to specify the variables in the conclusion. Both the generation of a relational schema and the grouping of the *Observations* can be done automatically, so the advantages of the representation of the data as *Observations* and of the possibility to easily access the data can be achieved without additional effort.

The transformation into relations with a high number of attributes can entail some disadvantages: If a particular *Observation* is missing which is needed to form such a many-attribute relation, either the whole relation tuple will be omitted or a special representation for null values would be needed. This null value problem occurs if an attribute is missing or when a partial n-to-1 relationship type is represented as additional attributes of the table corresponding to an entity type. An simple solution for missing attributes would be to use another integrity constraint that forbids missing values. This constraint would be very similar to the constraint that each role in a relation must be specified, which is discussed in the next section. A solution for the partial relation problem would be to map such relationship types to separate tables.

4 General Integrity Constraints of the EER Model

In this section we will present a set of general integrity constraints which can be used to check whether the instances of an EER diagram conform to the restrictions entailed by this EER diagram. We describe the different types of integrity constraints and show how integrity checking can be implemented using stratified Datalog knowledge bases in SYLLOG. It is assumed that the EER diagram is represented as a set of facts for the predicates resulting from the one-to-one mapping of the meta EER diagram (*Attribute*, *Role*, etc.) and that the instances of the EER diagram are given as ground facts using the predicate *Observation*.

As both are available – namely, the information about the EER model and the instances of the model – we are able to check the integrity of a database with one general set of integrity constraints. Unlike other approaches [24, 16, 7] which generate their own set of integrity constraints for each EER model, we only have one set of integrity constraints which can be used for any EER model.

An integrity constraint can be formulated in SYLLOG as a deduction rule with the conclusion stating that the constraint is violated. The premise of the rule states the conditions of the violation and combines predicates referring to EER schema information and the *Observation* predicate containing the instances. To make the rules shorter and more readable we have introduced auxiliary predicates.

4.1 Functional Dependencies

Identifying attributes and cardinality ratios of 1 are means to express uniqueness in EER models. We will see that functional dependencies are a very general and powerful instrument to define such constraints exactly.

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 of attributes Y. FDs are specified in the form $X \Rightarrow Y$ and are formally defined by the following implication [15, 10]:

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

t_1 and t_2 are two different tuples of R. If the values of the set of attributes X are the same in t_1 and t_2 then the values of the attribute set Y have to be the same. A FD is violated if there exist two tuples which have the same values in X and different values in Y. Note that X and Y are sets of attributes, where each attribute is not necessarily the same as an attribute in the EER model. Later we will refer to these attributes as *dependency attributes* to avoid confusion. Since names of descriptive elements (DE) in the ER model are only unique for each object type (OT), a dependency attribute is defined as a pair $DE : OT$, where both constituents are ground values. In order to refer to the tuple identifier we use the special DE *tupid*, which can be seen as a pseudo attribute in the underlying relation. The dependency attributes on the left hand side of the dependency are frequently called LHS, the ones on the right hand side RHS.

A rule for checking a FD with an atomic LHS and RHS can be expressed in SYLLOG as follows:

```
two observations with different values in [the-DE-RHS the-OT-RHS] are the-t1 the-t2
not: two observations with different values in [the-DE-LHS the-OT-LHS] are the-t1 the-t2
-----------------------------------------------------------------------------------------
ATOMIC [ the-DE-LHS the-OT-LHS ] does not determine value of [the-DE-RHS the-OT-RHS]
```

This rule succeeds whenever the FD is violated, i.e. when two tuples *t1* and *t2* with different values in the attributes of the RHS exist, and when for the same tuples no different values on the LHS exist. Since in our representation both the values and the tuple identifiers are accessible in the same way, we could express dependencies of the form

$$\textit{DE-LHS:OT-LHS} \Rightarrow \textit{tupid:OT-RHS}$$
$$\textit{tupid:OT-LHS} \Rightarrow \textit{DE-RHS:OT-RHS}$$
$$\textit{tupid:OT-LHS} \Rightarrow \textit{tupid:OT-RHS}$$

with the same ease.

In cases where the left hand side of a functional dependency is not atomic (like: $AB \rightarrow C$), the representation in a pure Datalog language is more complicated since the arguments have to be function symbol free and no list of attributes can be specified. It is necessary to reformulate sentences of the form "*the dependency is violated if for any different RHS all LHS are equal*" into "*the dependency is violated if for any different RHS no element of the LHS is different*". The second formulation uses negation such that it is only necessary to look at a single tuple at a time. Now the attributes on the LHS can be consequences of a separate rule which has the only purpose to output the attributes of the LHS of the specified dependency. For each different non-atomic LHS it is necessary to use such a specialized rule which is referenced by the integrity rule. This pure Datalog approach was used in [13].

We will now switch to another approach which is based on a built-in predicate in SYLLOG that allows to concatenate and split strings. This predicate allows to collect various data items in a single argument, so we can use a single rule to check all functional dependencies.

`dependencies for` *some-generic-fd* `are` *some-LHS* `=>` *some-RHS*

the-LHS `do not determine value of` *the-RHS*

`violation text for` *some-generic-fd* `is` *eg-text*

```
-----------------------------------------------------------
fd some-generic-fd is violated : eg-text
```

The dependency attributes in *some-LHS* or *some-RHS* can now be single attributes or a set of attributes which is kept sorted to achieve a unique representation. Note that by switching to this approach there is a clear separation between the generation of the dependencies (first premise), the general testing of functional dependencies (second premise), and the generation of a meaningful notification text in case of a violation. The general rule for testing functional dependencies now has the following form:

```
two observations with different values in the-RHS-attr are the-t1 the-t2
not : two observations with different values in the-LHS-attr are the-t1 the-t2
--------------------------------------------------------------------------
the-LHS-attr do not determine value of the-RHS-attr

value of some-dep-attr in eg-t1 is the-values1
value of some-dep-attr in eg-t2 is the-values2
not : the-values1 equal the-values2
---------------------------------------------------------------------
two observations with different values in some-dep-attr are the-t1 the-t2
```

Note that only the testing rule needs actual data from the *Observations*. Thus the separation of rules generating the dependencies and the actual testing rules allow to reason about dependencies of the schema at a time when no actual data is given.

4.2 Generic Functional Dependencies for EER

A generic dependency is a deduction rule, the consequences of which are in the form of concrete dependencies. In this section we present generic functional dependencies of EER, the next section discusses generic inclusion dependencies. In [13, 18] the notation of the generic dependencies was given in a language different from the implementation language. In this paper we will stick to the SYLLOG notation which in our opinion is sufficiently concise and readable, and still directly executable.

E-1: Tuple identifier determines single-valued attributes: Chen defines an attribute as a function which maps an entity set or a relationship set into a value set [3]. For our representation this means that the value of each single-valued descriptive element of the object type OT is determined by the tuple identifier of OT:

```
DE eg-DE is a eg-Attribute-or-Role of the-OT
pair ( tupid , eg-OT ) forms dependency attribute the-LHS
pair ( the-DE , eg-OT ) forms dependency attribute the-RHS
-------------------------------------------------------------
dependencies for "E-1" are the-LHS => the-RHS
```

The first premise of this rule returns every descriptive element of every object type defined in the system. This sentence is defined in terms of attributes and roles and handles inheritance as well. We will discuss inheritance in more detail in a later section. The second and third premise pack names of descriptive elements and object types into dependency attributes as discussed above.

E-2: All descriptive elements determine tuple identifier: Since entity types and relationship types refer to sets of values [3] the tuple containing all descriptive elements must be unique.

```
all descriptive elements of some-OT are the-LHS
pair ( tupid , eg-OT ) forms dependency attribute the-RHS
----------------------------------------------------------
dependencies for "E-2" are the-LHS => the-RHS
```

This is an example of a FD where the left hand side is not atomic, but consists of all descriptive elements of an object type. The first premise returns for every defined OT the set of descriptive elements in terms of dependency attributes *the-LHS*, the second premise forms for every of these object types the RHS of the dependency in form of the tuple identifier.

For weak entity types the set of descriptive elements must contain the tuple identifier of the owner entity type. As explained in Section 3.2 the identifying relationship of a weak entity type is treated here as a role of the weak entity type and is therefore included in the descriptive elements. No special treatment for weak entity types is needed.

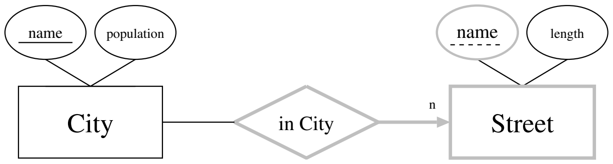
E-3: Identifying attribute together with all roles determine tuple identifier: For each identifying attribute *att* of every object type of the EER schema there exists a functional dependency where *att* together with all roles determines the tuple identifier.

```
the opt identifying attribute and all roles of some-OT are the-LHS
pair ( tupid , eg-OT ) forms dependency attribute the-RHS
----------------------------------------------------------
dependencies for "E-3" are the-LHS => the-RHS
```

This generic dependency handles the following cases:

1. For object types without roles (regular entity types) this rule means that every identifying attribute determines the tuple identifier. This definition corresponds to the definition of an identifying attribute as an attribute whose values can be used to identify an entity uniquely [7], because in our approach an entity is represented by its tuple identifier.
2. For object types with a single role (weak entity types) every identifying attribute determines its tuple identifier together with the role (tupid of owner). This conforms to the definition in [3] which states that the identifying attribute of weak entity types does not determine the weak entity, but together with the owner entities it does. By treating the identifying relationship a role in the weak entity type, we need no special treatment of weak entity types here. An example of

a weak entity type of the concrete dependencies derived by SYLLOG is given in Figure 3.



E-1: tupid:City $\Rightarrow$ name:City
E-1: tupid:City $\Rightarrow$ population:City
E-1: tupid:Street $\Rightarrow$ in-city:Street
E-1: tupid:Street $\Rightarrow$ length:Street
E-1: tupid:Street $\Rightarrow$ name:Street
E-2: name:Street,length:Street,in-city:Street $\Rightarrow$ tupid:Street
E-2: population:City,name:City $\Rightarrow$ tupid:City
E-3: name:City $\Rightarrow$ tupid:City
E-3: name:Street,in-city:Street $\Rightarrow$ tupid:Street

Fig. 3. An EER fragment with a Weak Entity Type and its Concrete Dependencies Derived by E-1 to E-3

3. For object types with multiple roles (relationship types) this rule allows an extension of the EER model as presented in [7], since it gives a meaning to a relationship type with an identifying attribute (the identifying attribute together with all roles determines the tuple identifier of the relationship type). Note that an identifying attribute of a relationship type is treated exactly like a role by the functional dependencies.
4. For every object type which has roles and no identifying attribute the rule states that all roles together determine the tuple identifier.

E-4: 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 the set of all other roles of rel [24]:

`Role` *the-OneRole the-RST some-entity-type* `has One` *some-participation*

```
the opt identifying attribute and all roles of eg-RST ( ne the-OneRole ) are the-LHS
pair ( the-OneRole , eg-RST ) forms dependency attribute the-RHS
-------------------------------------------------------------------
dependencies for "E-4" are the-LHS => the-RHS
```

The first premise holds for all roles of relationship types with cardinality *"One"*. The second premise collects optional identifying attributes (similar to the generic dependency E-3) and all other roles in the LHS dependency attributes. The *OneRole* is used in the RHS of the dependency.

Note that all the constraints for relationship types are independent of the degree of the relationship type.

4.3 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. An inclusion dependency (ID) of the form $A \subseteq B$ is violated iff there exists a value of the dependency attribute A which is not a value of the dependency attribute B.

In SYLLOG IDs can be expressed by the following two rules, where the first rule obtains the defined inclusion dependencies, calls the test and produces a short diagnostic message in case of a violation. The second rule actually tests whether there are values for LHS which are not included in RHS.

```
dependencies for some-generic-id are some-LHS << some-RHS
the-LHS is not included in the-RHS
violation text for some-generic-id is eg-text
---------------------------------------------------------
id some-generic-id is violated : eg-text
```

`value of` *the-LHS* `is` *some-value*

`not : value of` *the-RHS* `is` *the-value*

```
---------------------------------
```

the-LHS `is not included in` *the-RHS*

Note that this structure is very similar to the rules for FD testing. There is again a clear separation between the rules to derive dependencies and the rules performing the actual testing.

4.4 Generic Inclusion Dependencies for EER

E-5: Participating entities must be included in entity type: The values of a role must be tuple identifiers in the referenced entity types:

`Role` *some-role the-OT* `references` *some-ET*

`pair (` *some-role* `,` *the-OT* `) forms dependency attribute` *the-LHS*

`pair ( tupid ,` *the-ET* `) forms dependency attribute` *the-RHS*

```
--------------------------------------------------------
```

`dependencies for "E-5" are` *the-LHS* `<<` *the-RHS*

The first premise uses the sentence defined in Section 3.2. Therefore the rule covers as well the inclusion dependency from identifying relationship types and weak entity types.

Note that in our representation we use the artificial tuple identifiers as contents of role fields. These tuple identifiers should be replaced by the corresponding contents of an identifying attribute in the final application. The elimination of tuple identifiers is only relevant at the time when relational schemata are produced, which is beyond the scope of this paper. [18] discusses the elimination of tuple identifiers in detail. By using the tuple identifiers in roles we can postpone the key determination, the design decision which identifying attribute (or which attribute combination) should be used as a key in the relational schema to a later step. Conceptually this is not relevant.

E-6: Totally participating entity types must be included in roles: 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:

```
Role some-role the-RST some-ET has some-card Total
pair ( tupid , the-ET ) forms dependency attribute the-LHS
pair ( some-role , the-RST ) forms dependency attribute the-RHS
----------------------------------------------------------
dependencies for "E-6" are the-LHS << the-RHS
```

E-7: Generalizations: In both partial or total generalizations the instances of a subtype are included in the supertype. This dependency can be formulated via tuple identifiers or via its descriptive elements. The second one is the more interesting case (used as well by [17]) and is presented below.

```
Subclass in generalization a-gen is an-ET
all descriptive elements of subtype some-ET in a-gen are the-LHS
all descriptive elements of a supertype in a-gen are the-RHS
----------------------------------------------------------
dependencies for "E-7" are the-LHS << the-RHS
```

This generic inclusion dependency states that the values for all inherited attributes must be included in the supertypes. For the generalization in Figure 4 the following concrete inclusion dependencies are derived from E-7 (the names of the object types are abbreviated for typesetting reasons):

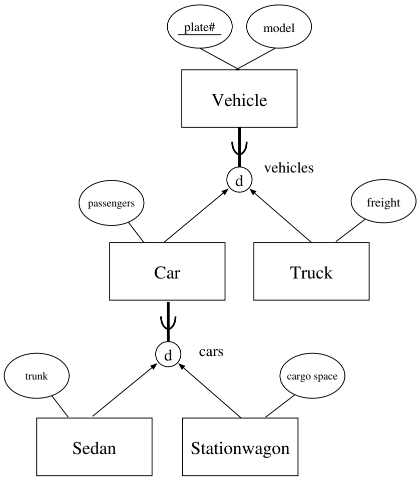


Fig. 4. A Nested Generalization

$$\text{plate\#:C,model:C} \subseteq \text{plate\#:V,model:V}$$

$$\text{plate\#:Se,passengers:Se,model:Se} \subseteq \text{plate\#:V,passengers:C,model:V}$$

$$\text{plate\#:St,passengers:St,model:St} \subseteq \text{plate\#:V,passengers:C,model:V}$$

$$\text{plate\#:T,model:T} \subseteq \text{plate\#:V,model:V}$$

4.5 Additional Constraints

Aside from the very general and unconditional dependency rules in the last section additional constraints could be formulated that express some more concepts of the EER model or that help to discover problems in the representation.

E-8: Total Generalizations: A generalization may be total or partial. A total generalization specifies that every entity in the superclass must be a member of some subclass in the specialization [7]. In other words, the constraint of a total generalization with supertype *super* is violated if there exists a tuple identifier t in *super* which does not exist in any of its subclasses.

Let us assume the set of tuple identifiers in *super* is T^{super} and the set of tuple identifiers of the immediate subclasses $1 \cdots n$ is $T^{sub_1} \cdots T^{sub_n}$. It is possible to state this constraint as

$$\forall t \in T^{super}(t \in T^{sub_1} \vee \cdots \vee t \in T^{sub_n})$$

or as an equivalence of the form:

$$\bigcup_{n}^{i=1} T^{sub_i} = T^{super}$$

Both formulations cannot be stated as simple inclusion dependencies that hold unconditionally. The first formulation leads to a disjunctive inclusion dependency (in the form of: one of the following IDs must hold), the second formulation refers to tuple identifiers of *different* object types (we required a ground value for the object type). It is straightforward to implement both of these formulations with a special rule in SYLLOG.

E-9: All descriptive elements must be specified in a tuple: In some applications it is reasonable to state that no null values are allowed. Null values are represented in the *Observation* facts by omission, when only one of two different descriptive elements in a given tuple has a value associated. Note that it is a consequence of a definition of null values by omission that tuples having only null values cannot be represented.

```
DE eg-DE1 is a eg-Attribute-or-Role of the-OT
DE eg-DE2 is a eg-Attribute-or-Role of the-OT
not : eg-DE1 equals eg-DE2
there is a value for the-DE1 the-OT in some-tupid
not : there is a value for the-DE2 the-OT in the-tupid
violation text for E-9 is eg-text
---------------------------------------------
nvd E-9 is violated : eg-text
```

If an EER application wants to permit null values for attributes it is sufficient to bind the variable *eg-Attribute-or-Role* to *role* e.g. as first premise of the rule above.

The following **naming constraints** restrict the names of object types and tuple identifiers in observations.

E-10: Naming of object types and descriptive elements: This rule checks whether the names of descriptive elements and object types used in the *Observation* facts are specified in the schema. This rule differs from earlier dependencies, since this is an inclusion dependency between data and schema whereas earlier inclusion dependencies refer only to data items.

An observation containing a DE of an object type OT does not conform to the schema if DE or OT has not been specified in the corresponding schema.

```
Observation for a-DE a-OT is a-tuple-id with a-value
not : DE some-DE some-OT is defined in the schema
-----------------------------------------------------------------
schema conformity violated by tuple the-tuple-id for the-DE the-OT
```

E-11: Unique Tuple Identifiers: All top level object types of the schema must have unique tuple identifiers. Top level object types (TOT) are either object types not occurring in any hierarchy (generalization or category) or the top generalization in a hierarchy. All tuples of subclasses are also members of the superclass.

This constraint can be expressed as a FD stating that the tuple identifier of these object types determines the name of the object type:

$$\forall TOT : \mathit{tupid}{:}TOT \Rightarrow \mathit{name}(TOT)$$

Note that this dependency differs from the generic FDs since it is a dependency between a tuple identifier and a *name* of an object type. This dependency is somewhat home-cooked since it depends on the *Observation* representation, where the names of the object types are available. After a transformation of the EER model into a relational schema this dependency will not be necessary anymore.

E-12: Disjoint Generalizations: A disjoint generalization states that for every disjoint generalization D no entity must exist in more than one subclass in D. In Figure 4 *"Car"* and *"Truck"* are subclasses in the generalization *vehicles*; the supertype in *vehicles* is *"Vehicle"*. This constraint can be formulated as a FD of the form:

$$\forall D : \textit{tupid:subclass(D)} \Rightarrow \textit{name}(D)$$

Again, this is a dependency referring to the name of an object class.

5 Integrity Checking in Syllog

The checking of the integrity constraints is performed by querying the constraint violations which were defined above. If there is a positive answer the database is inconsistent. The following rule can be used to test all dependencies and succeeds on violations.

```
some-kind-of-dependency some-dependency-id is violated : eg-text
--------------------------------------------------------------
database is inconsistent due to some-dependency-id : eg-text
```

It is certainly possible to restrict the query to some specific *dependency-id*. In order to examine the reason for the inconsistency one can use Syllog's explanation facility where one or more derivation trees are presented in a hypertext fashion (see next section).

The integrity rules can be extended to include the tuple identifiers of the violated *Observations*. In such cases the answers to queries contain the observations which violate the constraint, and a set of observations which represents a consistent database can be generated. Such a consistent subset is comprised of the answers to the predicate *Consistent observation* which may be defined as follows:

```
Observation for the-DE the-OT is the-t the-value
not : database is inconsistent due to some-dependency-id for the-DE the-OT tuple eg-t
-------------------------------------------------------------------------------------
Consistent observation the-DE the-OT the-t the-value
```

The integrity of an inconsistent database can be recovered by deducing the consistent observations and replacing the set of *Observation* facts by the set of consistent observations. This new set of observations is not necessarily a consistent database, because due to the deletion of the inconsistent observations other integrity constraints (e.g. inclusion dependencies) might now be violated. So the process of integrity checking and generation of consistent observations has to be repeated until no more inconsistent observations can be detected (fixed point).

This iterative process of recovering the integrity of a database is necessary, because the integrity constraints are based upon the predicate *Observation* and therefore use the stored facts regardless of the recognition of some of these facts as inconsistent observations by other integrity constraints. One possible solution would be to use the predicate *Consistent observation* instead of the basic *Observation* predicate in the integrity constraints, but then the problem arises that the knowledge base becomes unstratified and cannot be used with traditional inference mechanisms.

5.1 Generating Meaningful Explanations

If the database is inconsistent, it is important to know the reasons for the integrity violations in order to update the database appropriately. In order to obtain detailed diagnostics of the violation SYLLOG can generate explanations of its answers. This feature can be used efficiently to detect the reasons for integrity constraints violations.

A query for inconsistencies in the database might return the following answer table, stating that the system found three violations of the constraints, using rules *E-2*, *E-3* and *E-6*.

```
database is inconsistent due to some-generic-dependency : eg-text
==================================================================
      E-2         the set of all descriptive elements determines tupid
      E-3         the optional identifying attribute and all roles determine tupid
      E-6         totally participating entity types are included in role types
```

By selecting for example the second violation (*E-3*), an explanation can be generated, which has essentially the same form as the defined rules, except that most of the variables in the explanation are now ground.

```
Yes, that is true
Because ...

fd E-3 is violated : the optional identifying attribute and all roles determine tupid
-----------------------------------------------------------------------------------
database is inconsistent due to E-3 : the optional identifying attribute and all roles deter...

dependencies for E-3 are name:Airport => tupid:Airport
name:Airport do not determine value of tupid:Airport
violation text for E-3 is the optional identifying attribute and all roles determine tupid
---------------------------------------------------------------------------------
fd E-3 is violated : the optional identifying attribute and all roles determine tupid
```

E-3 is violated because there is a concrete functional dependency of the form

$$name{:}Airport \Rightarrow tupid{:}Airport$$

but the LHS of this dependency does not determine the RHS. As the following part of the explanation shows, there are two different airports *AP1* and *AP2* with equal names (RHS attributes), namely *"Wien-Schwechat"*.

```
two observations with different values in tupid:Airport are AP1 AP2
not : two observations with different values in name:Airport are AP1 AP2
------------------------------------------------------------------------
name:Airport do not determine value of tupid:Airport

value of name:Airport in AP1 is Wien-Schwechat
value of name:Airport in AP2 is Wien-Schwechat
Wien-Schwechat equal Wien-Schwechat
------------------------------------------------------------------------
not : two observations with different values in name:Airport are AP1 AP2
```

This is only a part of the explanation which contains as well information how and why the concrete FD was derived, etc. The explanation facility of the SYLLOG system uses a hypertext widget similar to what is used in popular WWW browsers so that the user can easily click on premises in the explanation text in order to see the explanation for this particular sentence.

6 Using a Meta EER Diagram to Reason about the Well-formedness of EER Models

So far, we are only able to check whether some given data conforms to the integrity constraints of a given application EER diagram. In the next step we will check whether the EER diagram is a valid EER diagram. This task is performed by using the meta EER diagram of Figure 5 and additional integrity constraints which are not expressible in the meta EER diagram.

The meta EER diagram in Figure 5 can be read as follows: The central EER concepts are *entity type* and *relationship type*. Both of these concepts are generalized to *object types*. Since an object type is either an entity type or a relationship type the generalization is disjoint. An object type might be described by *attributes*, which are identified by a *name* and characterized by a *type* (simple or identifying). Since

Fig. 5. A Meta EER Diagram

the names of attributes are only unique per object type, *attribute* is defined as a weak entity type with *object type* as owner.

Entity types and relationship types can be connected via *roles*. Roles are identified using a *name* and have a *cardinality* and a *participation* value. As one of the consequences of the generic FDs the name of the role together with the participating relationship type determines the participating entity type. It is not allowed that two roles of the same relationship type with the same names connect to two different entity types. Each occurrence of a relationship type has to participate in the *role* relation (total participation). Figure 6 shows an alternative formulation of the role constraints without using an identifying attribute in relationship types [13].

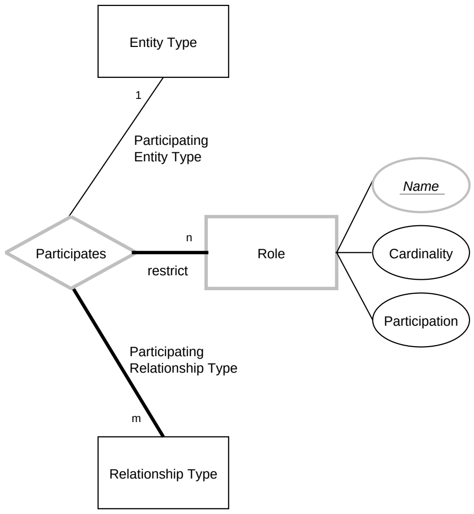


Fig. 6. Partial Meta EER Diagram without Identifying Attribute in Relationship Type

This more complicated construct uses a weak entity type connected to a ternary relationship type.

Each *weak entity type* is a role of entity type in the *owns* relationship type. Again this relationship type has an identifying attribute *name*. As a consequence of the generic FDs the weak entity type and the name of the *owns* relationship together determine the owning entity type.

Generalizations are used to define hierarchies of entity types. A generalization is identified by a *name* and characterized by the attributes *disjointness* and *completeness*. A generalization must have one entity type as a supertype and might have several (one or many) entity types as subtypes.

The meta EER diagram may be used to represent application EER diagrams. After performing the same one-to-one mapping from the meta EER diagram to the schema facts (as shown above for the airline EER diagram in Figure 2), the schema representation of an application EER diagram can be specified in form of *observations* for the meta diagram. Now it is possible to check whether the application EER diagram is a valid EER diagram.

If more than one schema of an EER model is specified the developer has to take care that the names of the schemata do not interfere. [18] uses a module prefix to distinguish different models.

It is interesting to note that applying a synthesis-based normalization algorithm such as [2] on the implied FDs together with an optimization technique for total participations in *:1 relationship types [18] results in a relational schema for representing EER diagrams, which is actually the schema of the one-to-one mapping presented in Section 3.1.

Since the integrity constraints presented above depend on the schema representation of the one-to-one mapping, it is necessary to lift the observation facts to the schema representation of the application EER in order to test the schema and application data at the same time. This mapping can be achieved by deduction rules like the following:

$$\frac{\begin{array}{lll}
\texttt{Observation for Name Attribute is} & \mathit{some\text{-}att} & \mathit{some\text{-}attribute\text{-}name} \\
\texttt{Observation for Type Attribute is} & \mathit{some\text{-}att} & \mathit{the\text{-}type\text{-}of\text{-}attribute} \\
\texttt{Observation for Description Describes is} & \mathit{some\text{-}rel} & \mathit{some\text{-}att} \\
\texttt{Observation for Described Describes is} & \mathit{some\text{-}rel} & \mathit{some\text{-}OT} \\
\texttt{Observation for Name } \mathit{an\text{-}et\text{-}or\text{-}rt} \texttt{ is} & \mathit{some\text{-}OT} & \mathit{some\text{-}OT\text{-}name}
\end{array}}{\texttt{Attribute } \mathit{some\text{-}attribute\text{-}name}\ \mathit{some\text{-}OT\text{-}name} \texttt{ is } \mathit{the\text{-}type\text{-}of\text{-}attribute}}$$

Now it is possible to use (a) the schema information of the meta EER diagram plus (b) observations of the application EER model with (c) deduction rules like the one above plus application data (as well in form of observations), to check (1) the well-formedness of the application EER diagram (the airline schema) and (2) the conformance of the application data relative to the application EER model. This is possible only by applying the general EER integrity constraints presented in Section 4.

One can, however, check more than is specified in the 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* has to be either *total* or *partial* (or that the type of an *attribute* must be either *simple* or *identifying*). Another constraint is that the names of roles and attributes of a relation must be disjoint. Examples of EER constructs that should be forbidden are: *entity-types* that are *owner* of itself, *subtypes* being own *supertypes*, two supertypes in a generalization that do not have a common root.

Two other constraints are that each relationship type must have at least two roles attached and that the names of attributes and roles per relationship type are disjoint. This are examples of application-specific constraints of the meta EER diagram.

`Role` *some-role-name1 the-rel some-entity1* `has` *some-c1 some-p1*

`Role` *some-role-name2 the-rel some-entity2* `has` *some-c2 some-p2*

`not :` *some-role-name-1* `equals` *some-role-name2*

`----------------------------------------------------------`

`database is inconsistent due to ASC-1 :` *"relationship has less than 2 roles"*

As demonstrated above one can easily derive the schema information of the application EER diagram from observations of the meta EER diagram. Since the meta EER diagram itself is an EER diagram it can be represented and checked in the same way.

In order to check the consistency between the application schema and its data it is only necessary to store the observation information and to use deduction rules like above for the schema information. Thus, none of the instances of the one-to-one mapping introduced in an earlier section must be available in form of facts, since these sentences might be formulated as deduction rules based on observations. One could even rewrite the integrity constraints to access the observation information of the schema directly which would make the deduction rules for *Attribute* and the like unnecessary. This point, however, stresses more the flexibility of the approach; a system stored only in observations would make SYLLOG's explanations longer and would also make the system harder to understand.

If the meta EER diagram is specified in terms of instances of itself (using *Observation*), the general and application-specific integrity rules introduced in the previous section can be applied to check the well-formedness of the meta EER model as well.

7 Conclusion and Future Work

In this paper we presented a set of general integrity constraints for EER models which are implemented using the logic-based language SYLLOG. The system can help the developer of a database system to understand the consequences of the EER constructs used by displaying the constraints for all or certain constructs, it is able to check these constraints against some test data, and it is able to generate explanations, which is very important especially in the case of integrity violations. From our experience the system is very *fast*[1] on today's PC class machines, even though we concentrated our efforts on a clear, declarative representation of the rules and did not consider procedural efficiency in the formulation.

[1] Adrian's suggestion

Integrity checking is certainly a naïve approach to exploit the integrity information, since it might be too costly for reasonably sized databases. Performance improvements could be achieved by only testing the relevant rules on each update.

As pointed out earlier it would be desirable not only to check the whole database, but rather to compute the set of consistent tuples or observations, ignoring the invalid (or incomplete) information during the computations of an application. As a consequence either a layering of the integrity constraints must be introduced (e.g. first compute the set of consistent observations using only integrity-constraint-1, apply integrity-constraint-2 on its results, and so on), or to specify the integrity constraints recursively, which leads to non stratified knowledge bases. The disadvantage of the first approach is that the layering of the integrity constraints might be very hard (for n integrity rules exist $n!$ different layerings), the disadvantage of the second approach is that most implemented deduction methods rely on stratified programs.

As mentioned earlier a problem of a different nature are name clashes when several EER models are kept in a single knowledge base (e.g. a meta EER model and several application EER models). With the presented rules the user has to take care that the names of the object types do not interfere. To eliminate this problem a model identifier has to be introduced which is essentially passed to all sentences in the knowledge base.

Although our approach has several shortcomings, it might lead to better understanding of EER modeling and of integrity checking in general. We think our system is a very powerful prototyping system and case designer, in which traditional relational databases could easily be integrated.

Acknowledgments

Many thanks to everyone who made it possible for me to finish this paper. First of all thanks to my wife Lore who patiently read various versions of this paper and made many helpful suggestions. Secondly, many thanks to Adrian Walker and all the effort he put into the Syllog project. Without him this paper would not have been possible. Finally many thanks to all my colleagues (in particular to Norbert Kehrer) and the students who worked with me on this project in the past.

References

[1] C. Apt, H. Blair and A. Walker, Towards a Theory of Declarative Knowledge, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, 1987) 89–148.

[2] C. Beeri and P.A. Bernstein, Computational Problems Related to the Design of Normal Form Relational Schemas, *ACM Transactions on Database Systems, 4:1* (1979) 30–59.

[3] P.P. Chen, The Entity-Relationship Model – Toward a Unified View of Data, *ACM Transactions on Database Systems, 1:1* (1976) 9–36.

[4] W.C. Cheng, *Tgif 2.6 - A Xlib based drawing facility under X11* (available via anonymous ftp from ftp.x.org, since May 1991).

[5] E. Codd, Extending the Database Relational Model to Capture More Meaning, *Transactions on Database Systems, 4:4* (1979) 397–434.

[6] P.W. Dart and J. Zobel, Conceptual Schemas Applied to Deductive Database Systems, *Information Systems, Vol. 13* (1988) 273–287.

[7] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems* (Benjamin/ Cummings, Redwood City, 1989).

[8] R. Elmasri, J. Weeldreyer and A. Hevner, The Category Concept: An Extension to the Entity-Relationship Model, *International Journal on Data and Knowledge Engineering, 1:1* (1985) 75–116.

[9] H. Gallaire, J. Minker and J.-M. Nicolas, Logic and Databases: A Deductive Approach, *ACM Computing Surveys 16:2* (1984) 153–185.

[10] G. Gardarin and P. Valduriez, *Relational Databases and Knowledge Bases* (Addison-Wesley, Reading, 1989).

[11] G. Kiernan, C. de Maindreville and E. Simon, Making Deductive Databases a Practical Technology: A Step Forward, in: *Proceedings of the ACM SIGMOD, International Conference on the Management of Data* (ACM, Atlantic City, 1990) 237–246.

[12] N. Kehrer and G. Neumann, Treating Enhanced Entity Relationship Models in a Declarative Style, in: A. Voronkov ed., *Logic Programming – First Russian Conference on Logic Programming, Irkutsk, Russia, September 1990 – Second Russian Confer-*

ence on Logic Programming, St. Petersburg, Russia, September 1991 – Proceedings, Lecture Notes in Artificial Intelligence (Springer-Verlag, Berlin, 1992) 263–270.

[13] N. Kehrer and G. Neumann, An EER Prototyping Environment and its Implementation in a Datalog Language, in: *Proceedings of 11th International Conference on the Entity-Relationship Approach*, Lecture Notes in Computer Science 645 (Springer-Verlag, Berlin, 1992) 243–261.

[14] T. Krueger, G. Kurian, A. Nair, G. Neumann, U. Neumerkel, S. Nusser, P. Reintjes, A. Taylor, D. Tzoar and A. Walker, People Oriented Software Technology, and its Use in Environmental Reporting, in: N. Revell, A. Min Tjoa, eds., *Proc. 6th International Conference on Database and Expert Systems Applications*, Lecture Notes in Computer Science 978 (Springer-Verlag, Berlin, 1995) 136–145.

[15] D. Maier, *The Theory of Relational Databases* (Computer Science Press, Rockville, 1983).

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

[17] V.M. Markowitz and A. Shoshani, Representing Extended Entity-Relationship Structures in Relational Databases, *ACM Transactions on Database Systems, 17:3* (1992) 423–464.

[18] G. Neumann, *Datenmodellierung mit deduktiven Techniken* (Physica Verlag, Heidelberg, 1994).

[19] S. Naqvi and S. Tsur, *A Logic Language for Data and Knowledge Bases* (Computer Science Press, New York, 1989).

[20] J. Smith and D. Smith, Database Abstractions: Aggregation and Generalization, *ACM Transactions on Database Systems, 2:2* (1977) 105–133.

[21] K.F. Sagonas, T. Swift and D.S. Warren, XSB as an Efficient Deductive Database Engine, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (ACM, Minneapolis, 1994) 442–453.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented*

Modeling and Design (Prentice Hall, Englewood Cliffs, 1991).

[23] T.J. Teorey, *Database Modeling and Design: The Entity-Relationship Approach* (Morgan Kaufmann, San Mateo, 1990).

[24] T.J. Teorey, D. Yang and J.P. Fry, A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model, *ACM Computing Surveys 18:2* (1986) 197–222.

[25] A. Walker (ed.), M. McCord, J.F. Sowa and W.G. Wilson, *Knowledge Systems and Prolog* (2nd Edition, Addison-Wesley, Reading, 1990).

Bio Sketch

Gustaf Neumann was appointed Professor of Information Systems and Software Techniques at the University of Essen, Germany, in 1995. A native of Vienna, Austria, he graduated from the Vienna University of Economics and Business Administration (WU), Austria, in 1983 and holds a Ph.D. from the same university. He joined the faculty of WU in 1983 as Assistant Professor at the MIS department and served as head of the research group for "*Logic Programming and Intelligent Information Systems*".

Before joining the University of Essen, Gustaf Neumann was a visiting scientist at IBM's T.J. Watson Research Center in Yorktown Heights, NY, from 1985-1986 and 1993-1995.

In 1987 he was awarded the Heinz-Zemanek award of the Austrian Association of Computer Science (OCG) for best dissertation ("*Metainterpreter Directed Compilation of Logic Programs into Prolog*").

Professor Neumann has published books and papers in the areas of program transformation, data modeling, and information systems technology. He is the author of several widely used programs that are freely available, such as the TEX-dvi converter dvi2xx and the graphical frontend package Wafe.