Conference on Database and Expert Systems Applications (DEXA'97)
A. Hameurlain, A Min Tjoa (Eds.), Springer, Berlin, 1997, pp. 240-251

First Steps to a Formal Framework for Multilevel Database Modifications

Frank Buddrus[1], Heino Gärtner[2], and Sven-Eric Lautemann[1]

[1] Johann Wolfgang Goethe-Universität Frankfurt, Fachbereich Informatik, Datenbanken und Informationssysteme, Postfach 11 19 32, D-60054 Frankfurt am Main, Germany, {buddrus|lautemann}@informatik.uni-frankfurt.de
[2] Universität Bremen, Fachbereich Mathematik und Informatik, Arbeitsgruppe Datenbanksysteme, Postfach 33 04 40, D-28334 Bremen, Germany, gaertner@informatik.uni-bremen.de

Abstract. We propose a formal basis for operations which can be understood as implicitly used in many kinds of schema modifications. Approaches for view definition, schema evolution, and schema versioning all rely on operations which work either on instance, on schema, or on both levels. This paper discusses a basic set of these operations called *modification primitives* and describes their semantics on the basis of the Extended Entity Relationship (EER) Model in a Hoare-style notation. We focus on the structural part of the schema definition and outline our ideas for arbitrary manipulations of instances.

1 Introduction

Modern database applications call for flexible modeling concepts with a high degree of abstraction and the power to anticipate the requirements of an evolving environment. These aims have been addressed by research and developments in the area of database modeling languages and concepts [HK87, PM88]. Apart from the development of database modeling concepts, we see a parallel line of research efforts, which focus on the introduction of concepts for describing derived interfaces like views [AB91, MP96, SLT91, Sou95, Wie91], schema evolution [BKKK87, FMZ$^+$95, Lau97, Mon93, Odb95, PÖ95, SLH90, SZ87, Zic92] and (on object level) database versions [CJ90] in the respective model. An interesting point is, that there seems to be a great need for such database modifications independent of the data model.

In this paper we present operations, called *modification primitives*, which can serve as a basis for arbitrary schema and database modifications. The idea is to develop a set of these modification primitives which have the potential to be the building blocks of higher level schema and database derivation operations as they are used in the definition of views or in schema evolution. We chose the EER-Model [Gog94], which is an extension of the well-known Entity Relationship model [Che76, BCN92], because it is semantically well founded and contains all features which are widely accepted to be desirable for the structural part of an object-oriented database [ABD+89]. Figure 1 shows an example of an EER diagram. *Components* i.e. entity-valued attributes are denoted by arcs from attributes to entities. *Type constructions* are depicted as triangles. The construction named *is* in our example has *House* as input type and *Church* and *Castle* as output types.

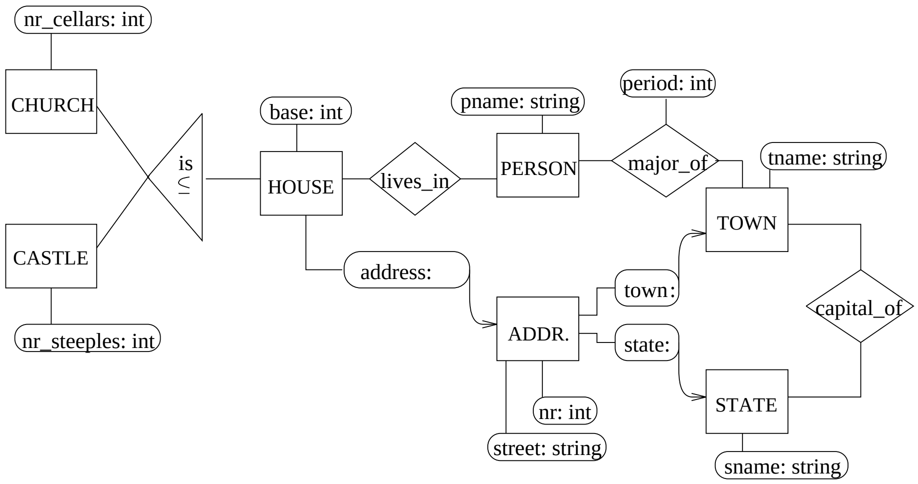

Figure 1. Sample EER diagram of a community.

The paper is structured as follows: Section 2 introduces the basic concepts for describing a database. The following four sections deal with the modification primitives for names, types, and structural properties. In Section 7 we sketch our ideas for integrating integrity constraints, view definitions, and other predicate based derivation concepts in our framework. In Section 8 we conclude by comparing our ideas — as far as they are presented in this paper — with related research and by discussing the shortcomings and advantages of our framework.

2 Basic Definitions

Before explaining the basic modification primitives we have to deal with some preliminaries. Our understanding of a schema is an adaptation of the Extended Entity Relationship (EER) Model [EGH+92] and our notation is based on [Gog94].

Definition 2.1 (Environment).
An Environment Z consists of a schema S and a database state DB.

$$Z = (S, DB)$$

Definition 2.2 (Schema).
A schema S consists of a structural part, integrity constraints, and a set of names

for the structural part.

$$S = (\text{STRUCT}, \text{CONSTRAINTS}, \text{NAME})$$

where CONSTRAINTS is a set of formulas without free variables and

$$\text{STRUCT} = \{\text{ENTTYPE}, \text{RELTYPE}, \text{ATTRIB}, \text{COMP}, \text{CONSTRUCT}\}$$

where ENTTYPE, RELTYPE, ATTRIB, COMP, CONSTRUCT are finite sets describing the identifiers used for the respective objects. NAME is a pairwise disjoint set of strings. Furthermore we define a bijective function $name : STRUCT \rightarrow \text{NAME}$.

A semantic function μ maps the structural components to semantic objects. A given ENTTYPE *Person* may in a certain state be mapped to persons p_1, p_2, and p_3 thus

$$\mu[\text{ENTTYPE}](\text{Person}) = \{p_1, p_2, p_3\}$$

For details see [Gog94]. Furthermore we define

$$\mu(X) = \bigcup_{x \in X} \mu[X](x)$$

with $X \in$ {ENTTYPE, RELTYPE, ATTRIB, COMP, CONSTRUCT}.

Definition 2.3 (Database state).
A database state is defined by the instances of all structural parts of the schema.

$$DB = (\mu(\text{ENTTYPE}), \mu(\text{RELTYPE}), \mu(\text{ATTRIB}), \mu(\text{COMP}), \mu(\text{CONSTRUCT}))$$

Definition 2.4 (Modification Primitive).
The modification primitive ω is the basic notion for modifying an environment. We distinguish the following operations:

- Add an element (α)
- Rename an element (β)
- Delete an element ($\bar{\pi}$)
- Retype an element (ρ)
- Select database elements (σ)

Thus

$$\omega : Z \rightarrow Z \text{ with } \omega \in \{\alpha, \beta, \bar{\pi}, \rho, \sigma\}$$

We state the semantics of our primitives in a Hoare-style notation [Bac86, GS94, LS87] but actually describe states by means of pre- and postconditions. Thus the description of our modification primitives will be of the following form:

$\{precondition\}$
  $\omega[\mathcal{I}](p_1, \ldots, p_n)$
$\{postcondition\}$

Where $\mathcal{I} \in$ STRUCT is an index, e.g. $\bar{\pi}$[ATTRIB] is the modification primitive to delete an attribute and p_i are the parameters to the operation. The precondition has to be valid in order to perform the operation ω. After the execution of ω we state that the postcondition is valid.

Definition 2.5 (Schema Invariants).
Schema invariants are sets of predicates that a consistent schema must fulfill.

We are a bit permissive in that point, because we do not explicitly define a complete set of schema invariants (see e.g. [BM93]). There are different ways to ensure schema invariants. For example, if the deletion of an entity type must not lead to attributes without a source entity schema invariants can be used:
$\{\forall a \in \text{ATTRIB asource}(a) \neq \bot\} \subseteq SI$
$\{SI, \ldots\}$
$\quad\bar{\pi}[\text{ENTTYPE}](e)$
$\{SI, \ldots\}$

Another way would be to ensure the predicate through the precondition of the respective modification primitive and thereby make the operation partial:
$\{\ldots, \forall a \in \text{ATTRIB asource}(a) \neq e, \ldots\}$
$\quad\bar{\pi}[\text{ENTTYPE}](e)$
$\{\ldots\}$

In the subsequent sections we describe the effects of the modification primitives.

3 Adding Elements

In this section we present the formal definitions for adding elements to an EER schema. The add primitive (α) is used to create new structures, i.e. entity types, relationship types, attributes, components, and compositions.

Entity Types

To achieve a small and orthogonal set of primitives, the creation of entity and relationship types does not allow the specification of attributes. Instead they have to be added later using α[ATTRIB].

$$\begin{aligned}
&\{SI, Z, ename \notin \text{NAME} = \{name_1, \ldots, name_m\}, E \notin \text{ENTTYPE} = \{E_1, \ldots, E_n\}\}\\
&\quad\alpha[\text{ENTTYPE}](E, ename)\\
&\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m, ename\}, name(E) = ename,\\
&\text{ENTTYPE} = \{E_1, \ldots, E_n, E\}\} \qquad (1)
\end{aligned}$$

The comparison of pre- and postcondition of formula (1) shows that E has been added to the set ENTTYPE of entity types, *ename* has been added to the set NAME of names and assigned to E by defining the value of $name(E)$.

Relationship Types

The roles of a relationship type R, i.e. the set of entity types involved in R must not be modified in our model, because of the following reason: if R does not contain any relationships $r \in R$ any modification can of course be done by simply deleting R and adding a new relationship type R' with the required roles, i.e. in this case no extra update primitive is required. If, on the other hand, relationships already exist for relationship type R, the addition of new roles to R would require an update of the stored relationships, i.e. the $\perp$ value would have to be set for the new entity in each $r \in R$ which is not allowed in the EER model by definition. A relationship must always include one entity for each of its roles. Therefore, all required roles of a relationship type R have to be specified when R is created. In the precondition of formula (2) operation LTS (ListToSet) is used to change the list of related entity types into a set.

$$\begin{aligned}
&\{SI, Z, rname \notin \mathrm{NAME} = \{name_1, \ldots, name_m\},\\
&R \notin \mathrm{RELTYPE} = \{R_1, \ldots, R_k\}, LTS(< E_1, \ldots, E_n >) \subseteq \mathrm{ENTTYPE}\}\\
&\quad \alpha[\mathrm{RELTYPE}](R, rname, < E_1, \ldots, E_n >)\\
&\{SI, Z, \mathrm{NAME} = \{name_1, \ldots, name_m, rname\}, name(R) = rname,\\
&\mathrm{RELTYPE} = \{R_1, \ldots, R_k, R\}\}
\end{aligned} \tag{2}$$

Attributes

$$\begin{aligned}
&\{SI, Z, aname \notin \mathrm{NAME} = \{name_1, \ldots, name_m\}, A \notin \mathrm{ATTRIB} = \{A_1, \ldots, A_n\},\\
&D \in DS, X \in \mathrm{ENTTYPE} \cup \mathrm{RELTYPE}\}\\
&\quad \alpha[\mathrm{ATTRIB}](A, aname, X, D)\\
&\{SI, Z, \mathrm{NAME} = \{name_1, \ldots, name_m, aname\}, name(A) = aname,\\
&\mathrm{ATTRIB} = \{A_1, \ldots, A_n, A\}, asource(A) = X, adest(A) = D,\\
&\mu[\mathrm{ATTRIB}](A)(X) = \perp\}
\end{aligned} \tag{3}$$

Components

Analogously to the above given arguments for relationship types, modification of the source and destination entity types of a component and of input and output entity types of a construction are not allowed. Therefore, those have to be specified in the creation of a component or construction type.

$$\begin{aligned}
&\{SI, Z, compname \notin \mathrm{NAME} = \{name_1, \ldots, name_m\},\\
&Comp \notin \mathrm{COMP} = \{Comp_1, \ldots, Comp_n\}, E_{source}, E_{dest} \in \mathrm{ENTTYPE},\\
&csource(Comp) = \perp, cdestination(Comp) = \perp\}\\
&\quad \alpha[\mathrm{COMP}](Comp, compname, E_{source}, E_{dest})\\
&\{SI, Z, \mathrm{NAME} = \{name_1, \ldots, name_m, compname\}, name(Comp) = compname,\\
&\mathrm{COMP} = \{Comp_1, \ldots, Comp_n, Comp\}, csource(Comp) = E_{source},\\
&cdestination(Comp) = E_{dest}\}
\end{aligned} \tag{4}$$

Constructions

One of the general schema invariants SI specifies, that there must be no circles in the set of constructions. Predicate $path(E, E')$ tests if entity type E is an input of a construction that has entity type E' as output. Predicate $path^*(E, E')$ is the transitive closure of $path(E, E')$ and is required to check if a new construction can be added to an EER schema. This is forbidden by definition, if the new construction introduces a circle, i.e. if there exist two entity types E and E' such that $path^*(E, E')$ and $path^*(E', E)$. The formal definition of $path$ and $path^*$ looks like this:

$$\begin{aligned} path(E, E') &:= \exists Constr \in \mathrm{CONSTRUCT}: \\ &\quad E \in input(Constr) \wedge E' \in output(Constr) \\ path^*(E, E') &:= \exists E_1, \ldots, E_m \in \mathrm{ENTTYPE}: \\ &\quad E = E_1 \wedge E' = E_m \wedge \forall i \in \{1, \ldots, (m-1)\} : path(E_i, E_{(i+1)}) \end{aligned}$$

$$\begin{aligned} &\{SI, Z, constrname \notin \mathrm{NAME} = \{name_1, \ldots, name_m\}, \\ &Constr \notin \mathrm{CONSTRUCT} = \{Constr_1, \ldots, Constr_n\}, \\ &\{E_{inp\_1}, \ldots, E_{inp\_k}, E_{outp\_1}, \ldots, E_{outp\_l}\} \subseteq \mathrm{ENTTYPE}, \\ &input(Constr) = \perp, output(Constr) = \perp, \\ &\neg(\exists E \in \{E_{inp\_1}, \ldots, E_{inp\_k}\}, \exists E' \in \{E_{outp\_1}, \ldots, E_{outp\_l}\} : path^*(E', E))\} \\ &\quad \alpha[\mathrm{CONSTRUCT}](Constr, constrname, \{E_{inp\_1}, \ldots, E_{inp\_k}\}, \{E_{outp\_1}, \ldots, E_{outp\_l}\}) \\ &\{SI, Z, \mathrm{NAME} = \{name_1, \ldots, name_m, constrname\}, name(Constr) = constrname, \\ &\mathrm{CONSTRUCT} = \{Constr_1, \ldots, Constr_n, Constr\}, \\ &input(Constr) = \{E_{inp\_1}, \ldots, E_{inp\_k}\}, output(Constr) = \{E_{outp\_1}, \ldots, E_{outp\_l}\}\} \end{aligned} \tag{5}$$

4 Renaming Elements

The rename primitive (β) can be used to change the name of a structural element. This is done by modification of the function $name$ that maps STRUCT to NAME. The rename primitive works in the same way for each structural element X. It replaces the old name of X by the new one in the set of names and changes the value of $name(X)$ accordingly.

$$\begin{aligned} &\{SI, Z, newname \notin \mathrm{NAME} = \{name_1, \ldots, name_m, oldname\}, name(X) = oldname, \\ &X \in STRUCT\} \\ &\quad \beta(X, newname) \\ &\{SI, Z, \mathrm{NAME} = \{name_1, \ldots, name_m, newname\}, name(X) = newname\} \end{aligned} \tag{6}$$

5 Deleting Elements

If structural elements are removed from an environment these operations do not only affect the schema but also the database part of the environment. These changes will be reflected by the respective μ-functions in the pre- and postconditions.

Attributes

The removal of an attribute has a quite straightforward definition and is depicted in Figure 2 for an attribute of a relationship type.

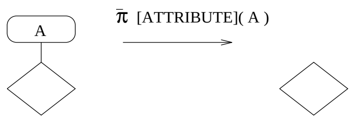

Figure 2. Deletion of attributes.

However, conflicts with schema invariants can arise, if for example entity types without attributes are disallowed. This might also be the case, if the respective attribute is referenced in a constraint.

$$\begin{aligned}
&\{SI, Z, [\text{NAME} = \{name_1, \ldots, name_m, aname\}, name(A) = aname,\\
&\text{ATTRIB} = \{A_1, \ldots, A_n, A\},\\
&\mu(\text{ATTRIB}) = \mu[\text{ATTRIB}](A_1) \cup \cdots \cup \mu[\text{ATTRIB}](A_n) \cup \mu[\text{ATTRIB}](A)\}\\
&\quad \bar{\pi}[\text{ATTRIB}](A)\\
&\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m\}, name(A) = \bot, \text{ATTRIB} = \{A_1, \ldots, A_n\},\\
&\mu(\text{ATTRIB}) = \mu[\text{ATTRIB}](A_1) \cup \cdots \cup \mu[\text{ATTRIB}](A_n)\}
\end{aligned} \tag{7}$$

Relationship Types

Figure 3 shows the deletion of a relationship type. We assume, that the relationship type has no attributes.

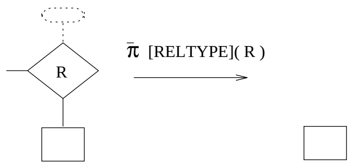

Figure 3. Deletion of relationship types.

$$\begin{aligned}
&\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m, rname\}, name(R) = rname,\\
&\text{RELTYPE} = \{R_1, \ldots, R_n, R\}, \neg\exists A \in \text{ATTRIB} : R \in asource(A)\}\\
&\quad \bar{\pi}[\text{RELTYPE}](R)\\
&\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m\}, name(R) = \bot, \text{RELTYPE} = \{R_1, \ldots, R_n\},\\
&\mu(R) = \bot, participants(R) = \bot\}
\end{aligned} \tag{8}$$

Entity Types

We assume, that the entity type has neither attributes, nor components, and is not member of any construction or relationship type.

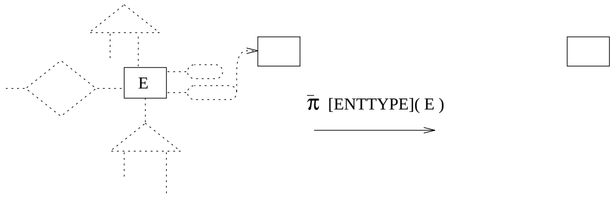

Figure 4. Deletion of entity types.

$$\begin{array}{l}
\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m, ename\}, name(E) = ename, \\
\text{ENTTYPE} = \{E_1, \ldots, E_n, E\}, \neg\exists A \in \text{ATTRIB} : E \in asource(A), \\
\neg\exists Comp \in \text{COMP} : E \in csource(Comp), \\
\{E, set(E), list(E), bag(E)\} \cap cdestination(\text{COMP}) = \emptyset, \\
\forall r \in \text{RELTYPE} : E \notin participants(r), \forall c \in \text{CONSTRUCT} : E \notin input(c) \cup output(c)\} \\
\quad \bar{\pi}[\text{ENTTYPE}](E) \\
\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m\}, name(E) = \perp, \\
\text{ENTTYPE} = \{E_1, \ldots, E_n\}, \mu(E) = \perp\}
\end{array} \tag{9}$$

Components

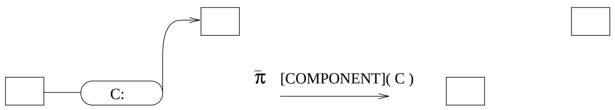

Figure 5. Deletion of components.

$$\begin{array}{l}
\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m, compname\}, name(Comp) = compname, \\
\text{COMP} = \{Comp_1, \ldots, Comp_n, Comp\}\} \\
\quad \bar{\pi}[\text{COMP}](Comp) \\
\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m\}, name(Comp) = \perp, \\
\text{COMP} = \{Comp_1, \ldots, Comp_n\}, \\
csource(Comp) = \perp, cdestination(Comp) = \perp\}
\end{array} \tag{10}$$

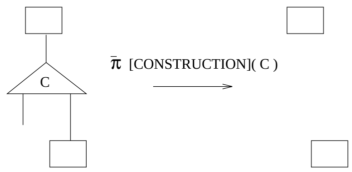

Figure 6. Deletion of constructions.

Constructions

$$\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m, constrname\}, name(Constr) = constrname,$$
$$\text{CONSTRUCT} = \{Constr_1, \ldots, Constr_n, Constr\}\}$$
$$\bar{\pi}[\text{CONSTRUCT}](Constr)$$
$$\{SI, Z, \text{NAME} = \{name_1, \ldots, name_m\}, name(Constr) = \perp,$$
$$\text{CONSTRUCT} = \{Constr_1, \ldots, Constr_n\},$$
$$input(Constr) = \perp, output(Constr) = \perp\} \quad (11)$$

6 Retyping Elements

The retype primitive (ρ) is used to change the type of attributes and components. In order to perform a reasonable update to the database part of the environment this primitive has to respect type convertibilities. For attributes we can specify convertibility in the form of tables. The type of a component is given by the entity of its destination. We assume one component c being type convertible to *c'* if a (transitive) input/output relation exists along constructions from the type of *c'* to the type of c.

Attributes

Given a type d and a type convertible type d', conversions can be applied using function *convert:* $DS \rightarrow DS$ according to Table 1. The first table describes convertibility of basic data types, the second one convertibility of sorts of data types. Type convertibility is read from left to right (row entry to column entry). Here $\times$ indicates that retyping is allowed, — indicates that retyping is not applicable.

	int	float	real	string
int	×	×	×	×
float	×	×	×	×
real	×	×	×	×
string	—	—	—	×

	d'	set$_{d'}$ {}	bag$_{d'}$ {{}}	list$_{d'}$ <>
d	×	×	×	×
set$_d$ {}	—	×	×	×
bag$_d$ {{}}	—	×	×	×
list$_d$ <>	—	×	×	×

Table 1. Type convertibility for simple and sorts of data types.

Type convertible attributes (indicated by the function *convertible:* $DS \times DS \rightarrow$ *boolean* as given by Table 1) can be retyped. Performing a retype primitive on an attribute the database part of the environment is modified by converting all data of the attribute concerned.

$$
\begin{aligned}
&\{SI, Z, A \in \text{ATTRIB}, adest(A) = D, D, D' \in DS, convertible(D, D'),\\
&\mu[\text{ATTRIB}](A) = \{(e_1, v_1), \ldots, (e_n, v_n)\}\}\\
&\quad \rho[\text{ATTRIB}](A, D')\\
&\{SI, Z, A \in \text{ATTRIB}, adest(A) = D' \in DS,\\
&\mu[\text{ATTRIB}](A) = \{(e_1, convert_{D,D'}(v_1)), \ldots, (e_n, convert_{D,D'}(v_n))\}\}
\end{aligned} \tag{12}
$$

Components

Components that are type convertible can be retyped. Performing a retype primitive on a component the database part of the environment is modified in the way that each data of the component concerned is changed to its input type regarding (transitive) construction relations with the new type.

$$
\begin{aligned}
&\{SI, Z, Comp \in \text{COMP}, cdestination(Comp) = E_{dest}, \{E_{dest}, E'_{dest}\} \subseteq \text{ENTTYPE},\\
&\mu[\text{COMP}](Comp) = \{(e_1, e_{d1}), \ldots, (e_n, e_{dn})\},\\
&\exists Constr \in \text{CONSTRUCT} : E'_{dest} \in input(Constr), E_{dest} \in output(Constr),\\
&\{(e_{d'1}, e_{d1}), \ldots, (e_{d'n}, e_{dn})\} \subseteq \mu[\text{CONSTRUCT}](Constr)\}\\
&\quad \rho[\text{COMP}](Comp, E'_{dest})\\
&\{SI, Z, Comp \in \text{COMP}, cdestination(Comp) = E'_{dest}, \{E_{dest}, E'_{dest}\} \subseteq \text{ENTTYPE},\\
&\mu[\text{COMP}](Comp) = \{(e_1, e_{d'1}), \ldots, (e_n, e_{d'n})\}\}
\end{aligned} \tag{13}
$$

7 Manipulating Instances

In this section we sketch our ideas for incorporating instance manipulations in our framework. There are basically two main issues to consider: (i) Restricting current instances, and (ii) generating new instances. This distinction resembles the separation of object generating and object preserving queries [MP96], however in our context object preservation is only feasible for the first case (i) because we want the $\mu(X)$ to be disjoint. The current instances can thus be restricted by an operation $\sigma_{add\_constraint}$:

$$
\begin{aligned}
&\{SI, Z, \varphi \text{ is a constraint}\}\\
&\quad \sigma_{add\_constraint}(\varphi(X))\\
&\{SI, Z, \text{CONSTRAINTS} = \text{CONSTRAINTS} \cup \varphi, \mu(X) = \{\mu(X) \mid \varphi\}\}
\end{aligned} \tag{14}
$$

The basic idea for other instance manipulations is to provide a mechanism σ_{trans} to encapsulate a query $\mathcal{Q}$ and define a type compatible iterator $\mathcal{I}$, which maps the results of the query $r(\mathcal{Q})$ to a target structure $\mathcal{T}$.

$$\begin{array}{l}\{SI, Z, \sigma_{trans} \text{ is type compatible}\} \\ \quad \sigma_{trans}(\mathcal{I}, \mathcal{Q}, \mathcal{T}) \\ \{SI, Z, \mu(\mathcal{T}) = \mu(\mathcal{T}) \cup \mathcal{I}(\mathcal{T}, r(\mathcal{Q}))\}\end{array} \tag{15}$$

Consider for example the creation of entity type "MAJOR" which will hold all persons who are or have once been majors of a town with attributes "mname:string" and "townname:string". This type can easily be created according to (3) and (1). With the above introduced primitive σ_{trans} we can now populate our new entity type:

α[ENTTYPE](major, "MAJOR");
α[ATTRIB](mname, "mname", major, string);
α[ATTRIB](townname, "townname", major, string);
$\sigma_{trans}(\mathcal{I}, \mathcal{Q}, \mathcal{T})$

with

$$\begin{array}{l}\mathcal{I} = \forall i \in r(\mathcal{Q}) \text{ new}[\mathcal{T}](\text{mname} \leftarrow i.1, \text{townname} \leftarrow i.2) \\ \mathcal{T} = \text{major} \\ \mathcal{Q} = -[\text{pname}(p),\ \text{tname}(t) \mid (p : \text{PERSON}) \wedge (t : \text{TOWN}) \wedge \text{major\_of}(p, t)] -\end{array}$$

This example is of course only a rough sketch of our ideas to encapsulate queries. Especially the iterator has to be adapted, when query languages other than the one presented in [Gog94] are used.

8 Conclusion and Future Work

systems, multi database systems, schema evolution, views

In many application domains, schema updates have to be performed frequently to reflect changes in the application environment. This is especially true as databases are very long lived. Schema modifications are also an important means for multidatabase systems, interoperability, and federated database systems. We have presented first steps to a formal approach to the specification of basic schema and database updates for the EER model.

The intension of our approach is to provide a solid base for concepts or procedures which rely on schema and database modifications. This is however not completely the case, so far. An important issue which needs to be incorporated in our framework is a way to propagate updates from a derived to an old environment, which would allow for instance updatable views. Another important point would be to extend the interface to the query language to support arbitrary complex query results. Finally, dynamic and temporal constraints need to be considered.

However, the approach presented in this paper provides an integrated way to handle schema and database updates and has therefore the potential to describe arbitrary complex schema and database modifications. Furthermore, a formal model is required to ensure a complete description and correct implementation of a system's behaviour regarding updates to schemas and databases.

We plan to extend the presented approach to complex schema updates each of which is composed of several of the basic updates presented here. The formal description supports the evaluation of such compositions, which are required for instance in software engineering environments. Moreover, a formal model will then allow an analysis and comparison of the semantics of schema update primitives of different systems.

References

AB91. Serge Abiteboul and Anthony Bonner. Objects and Views. In *Proc. Intl. Conf. on Management of Data*, pages 238–247. ACM SIGMOD, May 1991.

ABD+89. Malcolm Atkinson, François Bancilhon, David DeWitt, David Dittrich, David Meier, and Stanley Zdonik. The Object-Oriented Database System Manifesto. In *Proc. of the 1st Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 40–57, Kyoto, Japan, December 1989.

Bac86. Roland C. Backhouse. *Program Construction and Verification*. Prentice Hall, London, 1986.

BCN92. Carlo Batini, Stefano Ceri, and Shamkant Navathe. *Conceptual Database Design – an Entity-Relationship Approach*. Benjamin/Cummings, Redwood, 1992.

BKKK87. Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 311–322, San Francisco, CA, May 1987. ACM Press.

BM93. Elisa Bertino and Lorenzo Martino. *Object-Oriented Database Systems - Concepts and Architectures*. Addison-Wesley, Wokingham, 1993.

Che76. P. P. Chen. The entity relationship model - toward an unified view of data. *ACM Trans. on Database Systems*, 1(1):9, March 1976. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

CJ90. Wojciech Cellary and Geneviève Jomier. Consistency of Versions in Object-Oriented Databases. In *Proc. of the 16th Int'l Conf. on Very Large Databases (VLDB)*, pages 432–441, Brisbane, Australia, 1990. Morgan Kaufmann.

EGH+92. Gregor Engels, Martin Gogolla, Uwe Hohenstein, Klaus Hülsmann, Perdita Löhr-Richter, Gunter Saake, and Hans-Dieter Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering*, 9:157–204, 1992.

FMZ+95. Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran, and Joëlle Madec. Schema and Database Evolution in the O_2 Object Database

System. In *Proc. of the 21st Int'l Conf. on Very Large Databases (VLDB)*, pages 170–181, Zurich, Switzerland, September 1995. Morgan Kaufmann.

Gog94. Martin Gogolla. An extended entity-relationship model fundamentals and pragmatics. *Lecture Notes in Computer Science*, 767, 1994.

GS94. David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1994.

HK87. Richard Hull and Roger King. Semantic Database Modelling: Survey, Applications and Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

Lau97. Sven-Eric Lautemann. A Propagation Mechanism for Populated Schema Versions. In *Proc. of the 13th Int'l Conf. on Data Engineering (ICDE)*, pages 67–78, Birmingham, U.K., April 1997. IEEE, IEEE Press.

LS87. Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification (Second edition)*. John Wiley and Sons, New York, N.Y., 1987.

Mon93. Simon Monk. *A Model for Schema Evolution in Object-Oriented Database Systems.* PhD thesis, Lancaster University, February 1993. 101 pages.

MP96. Renate Motschnig-Pitrik. Requirements and comparison of view mechanisms for object-oriented databases. *Information Systems*, 21(3):229–252, 1996.

Odb95. Erik Odberg. *MultiPerspectives: Object Evolution and Schema Modification Management for Object-Oriented Databases.* PhD thesis, Norwegian Institute of Technology, February 1995. 408 pages.

PM88. Joan Peckham and Fred Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3):153–189, September 1988.

PÖ95. Randal J. Peters and M. Tamer Özsu. Axiomatization of dynamic Schema Evolution in Objectbases. In *Proc. of the 11th Int'l Conf. on Data Engineering (ICDE)*, Taipei, Taiwan, March 1995. IEEE Press.

SLH90. Barbara Staudt Lerner and A. Nico Habermann. Beyond Schema Evolution to Database Reorganization. In *Proc. of the 5th Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 67–76, Ottawa, Canada, October 1990. ACM Press.

SLT91. Marc H. Scholl, Christian Laasch, and Markus Tresch. Updateable Views in Object-Oriented Databases. In *Proc. of the 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 189–207, Munich, Germany, December 1991. Springer-Verlag. Lecture Notes in Computer Science No. 566.

Sou95. Cássio Souza dos Santos. Design and implementation of object-oriented views. *Lecture Notes in Computer Science*, 978, 1995.

SZ87. Andrea H. Skarra and Stanley B. Zdonik. Type Evolution in an Object-Oriented Database. In *Research Directions in Object-Oriented Programming*, pages 393–415. MIT Press, 1987.

Wie91. Gio Wiederhold. Views, Objects, and Databases. In *On Object-Oriented Database Systems*, pages 29–43. Springer, 1991.

Zic92. Roberto Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In François Bancilhon, Claude Delobel, and Paris Kanellakis, editors, *Building an Object-Oriented Database System – The Story of O_2*, pages 146–182. Morgan Kaufmann, San Mateo, California, 1992.