ELEMENTARY ACTIONS ON AN EXTENDED ENTITY-RELATIONSHIP DATABASE

Gregor Engels
TU Braunschweig, Informatik
Postfach 33 29, D-3300 Braunschweig

ABSTRACT: Semantic data models have been widely studied for the conceptual specification of databases. However, most of these data models are restricted to the description of the static structure of a database. They do not provide means to specify the dynamic behaviour of a database.
This paper sketches a language for the specification of actions on databases which have been specified by an Extended Entity-Relationship (EER) schema. These actions are based on so-called elementary actions, which are automatically be derived from the EER schema. So, it can always be guaranteed that these schema dependent elementary actions preserve all inherent integrity constraints.
The semantics of the elementary actions is given in two steps: First, it is shown how the semantics of a database schema, i.e., a current database state, can be represented by an attributed graph. Then, the semantics of elementary actions is given by programmed graph replacements.

Keywords: conceptual database specification, operational semantics of database actions, programmed graph replacements

CONTENTS

0. INTRODUCTION

Semantic data models have been studied for the conceptual specification of database systems for certain application areas (cf. [HK 87]). However, most of these data models are restricted to the description of the static structure of a database. They do not provide means to specify the dynamic behaviour and especially database modifications.

Meanwhile, it has been widely accepted in the database community that the specification of databases should comprise the description of the static structure and the corresponding dynamic behaviour. In this sense, database specifications are comparable to abstract data type specifications.

Nevertheless, the classical, successfully used approach to specify database systems is to start with the specification of the static structure. According to the chosen data model, types of database objects and possible interrelations between instances of these object types are fixed in the first step of the database specification process. Afterwards, database actions are specified which model actions of the application area. Of course, these database actions should respect the constraints which arise from the specification of the static structure. So, database specifications can be regarded as *constructive data type specifications*, consisting of a data type construction part and a type dependent action specification part.

In the literature there exist several approaches to specify modifications of (system) states. The scope ranges from descriptive, non-deterministic specifications, for instance, by pre-/postconditions, up to procedural, deterministic ones. A great advantage of the last ones is that they are executable and, therefore, well-suited for a rapid prototyping of database specifications.

The issue of this paper is to present a language for the specification of actions on a database, the structure of which has been specified by an Extended Entity-Relationship (EER) schema. This data model together with a well-defined semantics have been developed at Braunschweig Technical University during the last years ([HNSE 87], [HoGo 88]). It is our objective to start with this definition of syntax and semantics of an EER schema, given in [HoGo 88], and to define syntax and semantics of actions on such an EER database.

The developed language has a procedural, operational style. This is motivated by the intended use of the action specification language within the database design environment CADDY ([EHHLE 89]). The environment CADDY offers an integrated set of tools to a database designer to support him/her during the specification and testing by rapid prototyping of a conceptual database schema. So, this executable

action specification language enables an immediate interpretation of actions and, therefore, supports the rapid prototyping facility of CADDY.

The main ideas, presented in this paper, are the following: We define how database states of an EER database can be represented by attributed graphs. Actions on a database can then be regarded as graph transformations. Because of the constraints given by the description of the static structure of a database, sequences of such graph transformations often have to be applied to a database state graph to yield a transition between correct database states. Therefore, the approach of *programmed graph grammars* is suited to describe such sequences of graph transformations. We show that such programmed graph replacements can be derived automatically from the description of the static structure of a database. These programmed graph replacements are called *elementary actions*. They describe the modification of a database object together with all update propagation operations (cf. [SSW 80]) to yield a consistent, correct database state after a modification.

There exist some approaches in the literature which propose graph grammars for database specifications; see for example [EK 80], [FV 83], or [Na 79], who gives a survey on some early approaches in the field. Programmed graph grammars have successfully been applied in the more general case of software specifications (e.g. [Na 87], [Gö 88]). In this case, programmed graph grammars are used to specify graph classes for a specific application by fixing all allowed graph modifications. Thereby, the structure of a graph is implicitly be determined. In contrast to this, in the approach in this paper, we start with a description of the structure of a graph and show how allowed modifications of such graphs can be derived automatically from this structure description.

The paper is organized as follows: In section 1, we summarize the definition of [HoGo 88] of syntax and semantics of an EER schema. In section 2, we describe how an EER schema and a corresponding database state can be represented as attributed graph. Section 3 shows how actions on an EER database can be automatically derived from the EER schema. The semantics of these actions is given by programmed graph replacements. Section 4 summarizes the ideas and results of this paper.

1. THE EXTENDED ENTITY-RELATIONSHIP MODEL

The extended Entity-Relationship model (EER model) was developed by the database group at Braunschweig Technical University during the last years [HNSE 87, HoGo 88]. This conceptual data model is an extension of the classical Entity-Relationship model [Ch 76]. Similar to other ER extensions [EWH 85, MMR 86, TYF 86, PRYS 89], it combines concepts known from semantic data models like TAXIS

[MyW 80], SDM [HaM 81], IFO [AbH 87], or IRIS [LyK 86] to a uniform conceptual data model.

Thereby, this model and the corresponding specification language offer features to a database designer which allow a problem-oriented, natural modelling of the information structure of a certain application task. The main important features of the EER model are

- the possibility to extend the set of allowed data types for attribute domains by new, application-dependent data types,
- components, i.e., object-valued attributes, to model structured entity types,
- the concept of type construction in order to support specialization and generalization, and
- several structural restrictions like the specification of key attributes.

Let us illustrate the EER model by a very small example (cf. Fig. 1.1). It models the world of persons who might be married or not, and who live at a certain town. This is expressed by the entity types **PERSON** and **TOWN**, and the relationship **lives-at**. The partition of objects of type **PERSON** into married and unmarried persons is described by the type construction **part**, which has **PERSON** as input type and **MARRIED** resp. **NOT_MARRIED** as output types. Each object of a certain entity type is described by some attribute values. In case of key attributes (e.g. **Name** at **PERSON**, expressed by a dot in the diagram), the value of this attribute uniquely identifies an object in the set of all objects of this type. Attributes may also be object-valued, as it is in case of the attribute **Info** at **PERSON**. This enables the modelling of complex structured objects which contain subobjects, also called components.

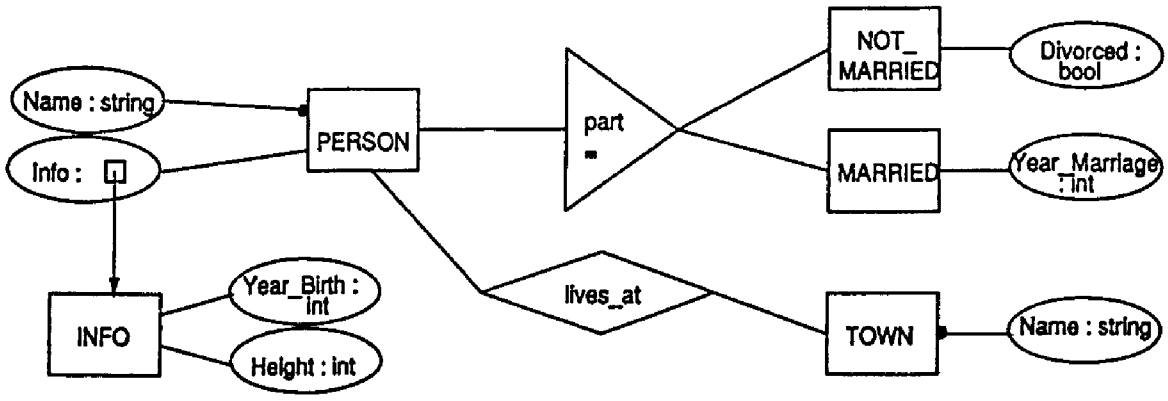

Figure 1.1: Example of an EER schema

An EER schema bases upon a data type signature DS = (DATA, OPNS), which consists of a set DATA of data type names and appropriate operations OPNS, and which has a fixed semantics. This data type signature DS has to be specified by a database designer in a previous step so that the data types can be used as attribute domains. In the example of Fig. 1.1, DS contains the specifications of int, string, and bool.

Due to [HoGo 88], syntax and semantics of an EER schema can then be defined as follows:

Def. 1.1:

The syntax of an extended Entity-Relationship schema **EER(DS)** over **DS** is given by

- the finite sets **ENTITY**, **RELSHIP**, **ATTRIB**, **COMP**, **T_CONSTR**, and
- the partial functions
  - **attr : (ENTITY ∪ RELSHIP) × ATTRIB -o→ DATA**
  - **key : ENTITY -o→ ATTRIB**
  - **comp : ENTITY × COMP -o→ ENTITY**
- the total functions
  - **particip : RELSHIP → ENTITY⁺**
  - **input, output : T_CONSTR → *F* (ENTITY)** (*F* ≙ powerset of finite subsets)
- the predicate
  - **is-partition ⊂ T_CONSTR**

The following conditions must hold:

(i) for all **e** ∈ domain(**key**): **key (e) = a** implies **(e, a)** ∈ domain **(attr)**

(ii) **comp** is injective
(i.e., that an entity type is component of at most one (other) entity type)

(iii) **output (c_1) ∩ output (c_2) = ∅** for two distinct c_1, c_2 ∈ **T_CONSTR**
(i.e., each constructed entity type is uniquely constructed)

(iv) It is not allowed that **connection⁺ (e, e)** holds for some **e** ∈ **ENTITY**, where **connection⁺** is the transitive closure of the relation **connection** defined by: if e_{in} ∈ **input(t)** and e_{out} ∈ **output(t)** for some **t** ∈ **T_CONSTR**, then **connection(** e_{in}, e_{out} **)** holds (i.e., type construction is cycle free).

(v) **comp(e, c) = e'** with **e, e'** ∈ **ENTITY**, **c** ∈ **COMP** implies ¬ ∃ **t** ∈ **T_CONSTR** with **e'** ∈ **input(t)** or **e'** ∈ **output(t)** (i.e., components are not allowed to be input or output type of a type construction)

So, the syntax of an EER schema consists of finite sets of names for entity types, relationship types, etc., and of functions and predicates which fix the contextfree structure of **EER(DS)**. Further contextsensitive rules are demanded by the conditions (i) - (v).

As already mentioned, the semantics of the data type signature **DS** is fixed. Let **μ[DATA]** denote the semantical domain of **DATA**. Let **IFISETI** denote the class of finite sets, and **IFUNI** the class of total functions.

Def. 1.2:

The semantics of an extended Entity-Relationship schema **EER(DS)** over DS is given by

- a function **μ[ENTITY] : ENTITY → IFISETI**

- a function μ[RELSHIP] : RELSHIP $\rightarrow$ |FISET|
  such that particip(r) = $\langle e_1, ..., e_n \rangle$ for r $\in$ RELSHIP implies
  μ[RELSHIP](r) $\subset$ μ[ENTITY](e_1) $\times$ ... $\times$ μ[ENTITY](e_n)
- a function μ[ATTRIB] : ATTRIB $\rightarrow$ |FUN| such that
  attr(e, a) = d implies μ[ATTRIB](a) : μ[ENTITY](e) $\rightarrow$ μ[DATA](d) resp.
  attr(r, a) = d implies μ[ATTRIB](a) : μ[RELSHIP](r) $\rightarrow$ μ[DATA](d)
- a function μ[COMP] : COMP $\rightarrow$ |FUN| such that
  comp(e, c) = e' implies bijective functions
  μ[COMP](c) : μ[ENTITY](e) $\rightarrow$ μ[ENTITY](e')
- a function μ[T_CONSTR] : T_CONSTR $\rightarrow$ |FUN| such that
  input(t) = { $i_1, ..., i_n$ }, output(t) = { $o_1, ..., o_m$ } implies injective functions
  $\mu[T\_CONSTR](t) : \bigcup_{j=1}^{m} \mu[ENTITY](o_j) \rightarrow \bigcup_{k=1}^{n} \mu[ENTITY](i_k)$

The following conditions must hold:

(i) ("disjoint sets of instances")
for two distinct e, e' $\in$ ENTITY: μ[ENTITY](e) $\cap$ μ[ENTITY](e') = $\emptyset$

(ii) ("key attributes")
for all e $\in$ ENTITY, a $\in$ ATTRIB with key(e) = a and attr(e, a) = d
μ[ATTRIB](a) : μ[ENTITY](e) $\rightarrow$ μ[DATA](d) is injective

(iii) ("partition")
for all t $\in$ is_partition: | input(t) | = 1 and μ[T_CONSTR](t) is bijective

The definition of the semantics of an EER schema assigns to the schema a fixed database state. This database state consists of sets of instances of entity resp. relationship types. The association of current attribute or component values to entity resp. relationship instances is described by functions. Analogously, the rearrangement of instances of entity types by a type construction is fixed by corresponding functions.

The definition contains several restricting conditions, which have to be fulfilled in a current database state. These restrictive conditions are termed "schema inherent integrity constraints" in the database literature, because they are expressed by syntactical means within the schema. For example, type constructions can be used as partition operator (condition (iii)). This means that, in contrast to the general case, all instances of the input entity types have to be contained in the set of instances of output types.

2. GRAPH REPRESENTATION OF DATABASE STATES

Section 1 summarized the definitions of syntax and semantics of an EER schema as they were given in [HoGo 88]. The main idea was to represent an EER schema

and a corresponding database state by a set of functions and predicates. As it is our intention to describe state transitions by graph replacements, at first we need a graph representation of a database state. Therefore, we define how syntax and semantics of an EER schema can equivalently be represented by an attributed graph.

Def. 2.1:

A directed, labelled graph G = (Nodes, Nodelabels, nodelab, Edgelabels, Edges) is defined by

- a finite set of Nodes,
- a finite set of Nodelabels,
- a labelling function nodelab : Nodes → Nodelabels,
- a finite set of Edgelabels,
- a set of labelled Edges ⊂ Nodes × Edgelabels × Nodes.

The syntax of an EER schema can then be represented by a directed, labelled graph, where all identifiers for entity or relationship types, attributes, etc. occur as node labels. Labelled edges represent the functions and predicates of the schema and describe the contextfree and contextsensitive interrelations within an EER schema. Representation 1 defines how the constituents of the Schema_Graph are constructed for a given EER schema.

Representation 1:

Schema_Graph := (S_Nodes, S_Nodelabels, S_nodelab, S_Edgelabels, S_Edges)
with
S_Nodelabels := ENTITY ∪ RELSHIP ∪ ATTRIB ∪ COMP ∪ T_CONSTR ∪ DATA
S_nodelab : S_Nodes → S_Nodelabels bijective function
S_Edgelabels := { E_Key, E_Attrib, E_Comp, E_Input, E_Output, E_Attr_Type, E_Comp_Type,
E_Partition } ∪ { E_Part_1, ..., E_Part_n },
where n := max { | particip(r) | | r ∈ RELSHIP }

The set S_Edges is constructed as follows:

For all e ∈ ENTITY ∪ RELSHIP, e' ∈ ENTITY, r ∈ RELSHIP, a ∈ ATTRIB, c ∈ COMP,
t ∈ T_CONSTR, d ∈ DATA holds

- (e, E_Key, a) ∈ S_Edges *iff.* (e, a) ∈ domain(attr) and key(e) = a
- (e, E_Attrib, a) ∈ S_Edges *iff.* (e, a) ∈ domain(attr) and key(e) ≠ a
- (a, E_Attr_Type, d) ∈ S_Edges *iff.* ((e, a) ∈ domain(attr) and attr(e, a) = d)
- (e, E_Comp, c) ∈ S_Edges *iff.* (e, c) ∈ domain(comp)
- (c, E_Comp_Type, e') ∈ S_Edges *iff.* ((e, c) ∈ domain(comp) and comp(e, c) = e')
- for i = 1, ..., | particip(r) | :
  (r, E_Part_i, e) ∈ S_Edges *iff.* (particip(r) = $\langle e_1, ..., e_n \rangle$ and e_i = e)
- (t, E_Partition, e) ∈ S_Edges *iff.* e ∈ input(t) and t ∈ is_partition
- (t, E_Input, e) ∈ S_Edges *iff.* e ∈ input(t) and not (t ∈ is_partition)
- (t, E_Output, e) ∈ S_Edges *iff.* e ∈ output(t)

Figure 2.1 shows the representation of the example of an EER schema of Fig. 1.1 as **Schema_Graph**.

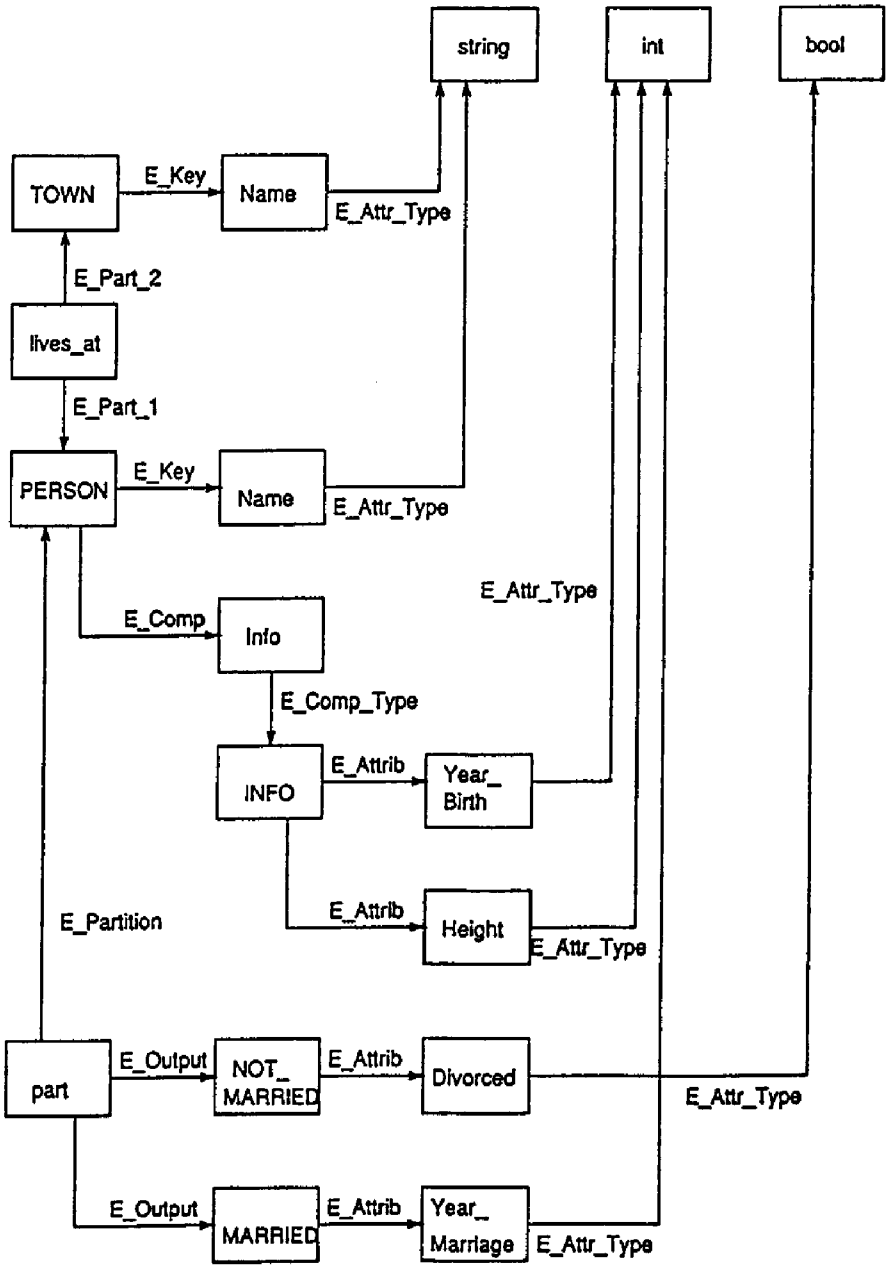

Figure 2.1: Example of a **Schema_Graph**

For the representation of a database state, we need node attributes additionally: Let **Node_Attrib** be a finite set of node attributes. Each node attribute **n_a** $\in$ **Node_Attrib** has a domain of values, termed **n_a_Values**. Let **Node_Attrib_Values** be $\bigcup_{\substack{n\_a \in \\ Node\_Attrib}} n\_a\_Values$.

Def. 2.2:

An attributed graph **AG = (Nodes, Nodelabels, nodelab, Edgelabels, Edges, node_attr, attr_value)** over **Node_Attrib** and **Node_Attrib_Values** is defined by

- (Nodes, Nodelabels, nodelab, Edgelabels, Edges) a directed, labelled graph
- node_attr: Nodes -o→ Node_Attrib a partial function, which assigns a node an attribute
- attr_value : Nodes -o→ Node_Attrib_Values a partial function, which assigns a node an attribute value.

The following condition must hold:

domain(node_attr) = domain(attr_value) and
node_attr(n) = n_a implies attr_value (n) ∈ n_a_Values

In our case, node attribute values are elements of the semantic domain of data types. So, Node_Attrib := DATA and d ∈ Node_Attrib implies d_Values := μ[DATA](d)

The graph, termed DB_Graph, to represent the semantics of an EER schema, i.e., a database state, then is an extension of the Schema_Graph by the following nodes and edges:

Representation 2:

DB_Graph := (DB_Nodes, DB_Nodelabels, DB_nodelab, DB_Edgelabels, DB_Edges, DB_node_attr, DB_attr_value)

with

- S_Nodes ⊂ DB_Nodes
- DB_Nodelabels := S_Nodelabels ∪ { ENT_INST, REL_INST }
- $DB\_nodelab|_{S\_Nodes} := S\_nodelab$
- DB_Edgelabels := S_Edgelabels ∪ { E_Inst, E_Comp_Val, E_Origin }

and

(a) for all e ∈ ENTITY, for all $\underline{e_i}$ ∈ μ[ENTITY](e) :

(a1) $\underline{e_i}$ ∈ DB_Nodes and DB_nodelab($\underline{e_i}$) := ENT_INST

(a2) (e, E_Inst, $\underline{e_i}$) ∈ DB_Edges

/* the subgraph with root e is duplicated for each $\underline{e_i}$ */

(a3) for all a ∈ ATTRIB with (e, a) ∈ domain(attr):

$a_{\underline{e_i}}$ ∈ DB_Nodes and ($\underline{e_i}$, E_Attrib, $a_{\underline{e_i}}$) ∈ DB_Edges

DB_nodelab($a_{\underline{e_i}}$) := a

DB_node_attr($a_{\underline{e_i}}$) := attr(e, a)

DB_attr_value($a_{\underline{e_i}}$) := μ[ATTRIB](a)($\underline{e_i}$)

(a4) for all c ∈ COMP with (e, c) ∈ domain(comp)

$c_{\underline{e_i}}$ ∈ DB_Nodes and ($\underline{e_i}$, E_Comp, $c_{\underline{e_i}}$) ∈ DB_Edges

DB_nodelab($c_{\underline{e_i}}$) := c

($c_{\underline{e_i}}$, E_Comp_Val, μ[COMP](c)($\underline{e_i}$)) ∈ DB_Edges

(b) for all r ∈ RELSHIP, for all $\underline{r_i}$ ∈ μ[RELSHIP](r) with $\underline{r_i} = \langle \underline{e_{i_1}}, \ldots, \underline{e_{i_n}} \rangle$

(b1) $\underline{r_i}$ ∈ DB_Nodes, and DB_nodelab($\underline{r_i}$) := REL_INST

(b2) (r, E_Inst, $\underline{r_i}$) ∈ DB_Edges

(b3) for all a ∈ ATTRIB with (r, a) ∈ domain(attr):

$a_{\underline{r_i}}$ ∈ DB_Nodes and ($\underline{r_i}$, E_Attrib, $a_{\underline{r_i}}$) ∈ DB_Edges

DB_nodelab(a_{r_i}) := a

DB_node_attr(a_{r_i}) := attr(r, a)

DB_attr_value(a_{r_i}) := μ[ATTRIB](a)(r_i)

(b4) (r_i, E_Part_j, e_{i_j}) ∈ DB_Edges for j = 1, ..., n

(c) for all t ∈ T_CONSTR, for all e ∈ domain(μ[T_CONSTR](t)):

(e, E_Origin, μ[T_CONSTR](t)(e)) ∈ DB_Edges

Figure 2.2 gives an example for a DB_Graph, which represents a (very small) database state.

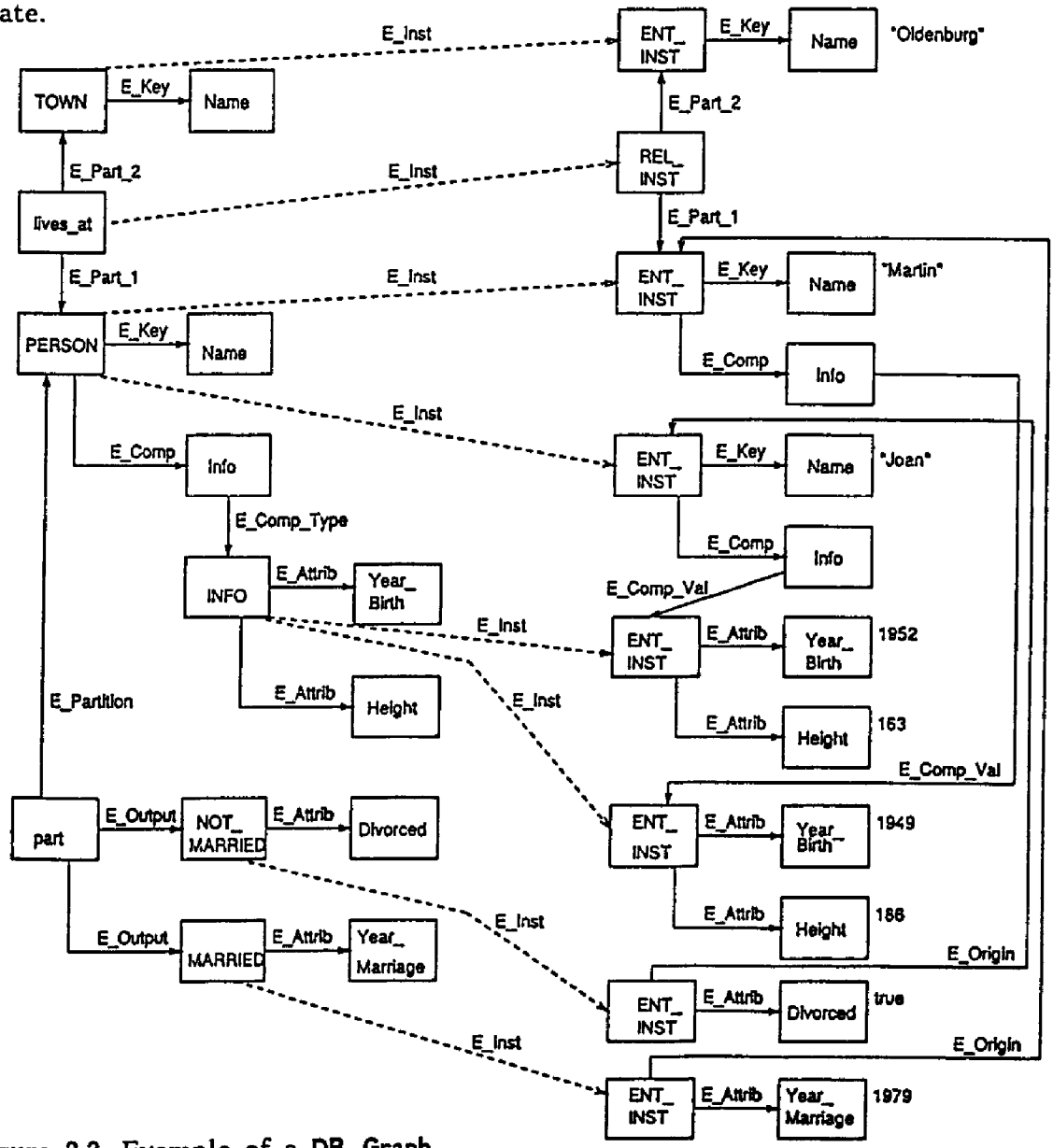

Figure 2.2: Example of a DB_Graph

3. ACTIONS ON AN EXTENDED ENTITY-RELATIONSHIP DATABASE

Up to now, we have described how the static structure of a database can be modelled. Using the concepts of our EER model, an EER schema fixes the possible structure

of objects and their interrelations in a database. But, the specification of the static structure of a database is only a first step in modelling a database. A database is not a dead, unchangeable read-only memory of objects, but an alive, often changing storage, where objects can be inserted, deleted, or updated. Of course, all these modifications have to regard the structural restrictions specified by the EER schema.

In the literature, there exist several approaches to specify those infinite sets of allowed modifications of a current system state by a finite description. The scope ranges from descriptive, non-deterministic specifications, for instance, by pre-/post-conditions, up to procedural, deterministic ones. Here, we present an operational, procedural approach, as it is our intention to get an executable description of database actions. This supports rapid prototyping and testing of action specifications by a database designer during the conceptual database design phase.

The set of specifications of database actions can be subdivided into the three types:

basic actions - *elementary* actions - *complex* actions

Basic actions describe the modification of exactly one database object. After the execution of such a basic action, the new database state may be not a correct one. This means that this local modification caused a violation of the database structure, as it is demanded by the EER schema. In this case, additional basic actions, known as *update propagations* [SSW 80], are necessary to yield a new correct database state. Minimal sequences of basic actions starting and resulting in a correct database state are described by *elementary actions*. Elementary actions can be composed to *complex actions* by the use of language constructs, e.g. control structures, known from procedural programming languages like Modula-2.

It is not the topic of this paper to present the language for the description of complex actions in more detail (see [Wo 89]). Here, we concentrate on the set of basic and elementary actions.

3.1 BASIC ACTIONS

Basic actions describe the modification of exactly one database object. Such a modification can be

(i) the insertion or deletion of an instance of an entity or relationship type together with all attribute values,

(ii) the addition or removal of a component of an instance of an entity type,

(iii) the insertion or deletion of the membership of a database object in a certain type construction, or

(iv) the update of attribute values of existing database objects.

For our example, the signature of some of these basic actions have the following form:

(i) basic_insert_PERSON (name : string) : PERSON
basic_delete_PERSON (p : PERSON)
basic_insert_lives_at (p : PERSON; t : TOWN) : lives_at
(ii) basic_add_comp_INFO (p : PERSON; year_of_birth : int; height : int) : INFO
basic_remove_comp_INFO (p : PERSON)
(iii) basic_insert_cons_PERSON_MARRIED(p : PERSON; year_of_marriage : int) : MARRIED
basic_delete_cons_MARRIED (m : MARRIED)
(iv) basic_update_INFO.Height (i : INFO; height : int)

All insert actions are functions which yield as result a modified database state, and, additionally, a reference to the inserted instance of an entity or relationship type. All attributes of the entity or relationship type occur as formal parameters of the basic insert action.
All delete actions as well as the insertion of a relationship instance only require references to instances as actual parameters to denote the database objects which are relevant for the execution of this action.

All these basic actions are implicitly given for each entity or relationship type of the specified EER schema. This means that the syntax, i.e. the signature, and, as we will see, also the semantics of basic actions can automatically be derived from the EER schema.

We have shown in section 2 how database states can be represented by attributed graphs. As the execution of a basic database action modifies the current database state, this execution can be viewed as the application of an appropriate graph transformation rule.
For example, the semantics of the action basic_insert_PERSON can be described by the following graph transformation rule:

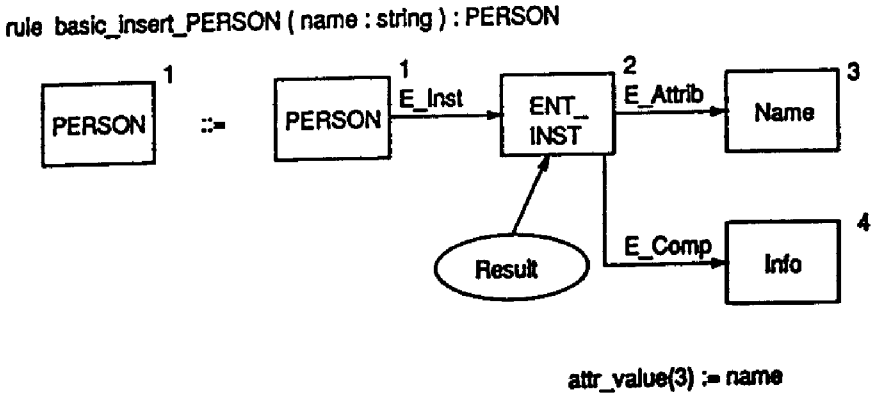

Figure 3.1

Each graph transformation rule consists of the following four components: two graphs, termed the *left-hand* and *right-hand* side of the rule, an *embedding transformation*, and a sequence of *attribute assignments*. These components control the application of a rule in several steps. The first step is to identify a subgraph in the current DB_Graph (cf. representation 2 in section 2), which is isomorphic to the left-hand side. The corresponding subgraph is removed in the second step of the application of a graph replacement rule, and replaced by a graph which is isomorphic to the right-hand side. In the third step, the newly inserted graph has to be connected appropriately to the DB_Graph by additional edges according to the given embedding transformation. Here, we only need the identical embedding. Therefore, it is not contained in Figure 3.1 (for further details see [ELS 87]). In the last step, the attribute values of the inserted nodes have to be set according to the specified attribute assignments.

Our approach to graph replacements bases on [Na 79], and on an extended version described in [ELS 87] and [En 86], where the reader can find a complete description.

Sometimes, it is useful to restrict the applicability of a graph replacement rule to a specific subgraph in the host graph. Therefore, we extend the approach of [ELS 87] by the introduction of *node-valued parameters* and *node-valued functions*. This means that the application of a graph replacement rule may yield as result a certain node of the host graph. This is expressed in the right-hand side by a "Result"-node (cf. Figure 3.1). Furthermore, node-valued parameters are allowed. For example, the rule for the addition of a component of type INFO has as parameter the node p of an instance of a person:

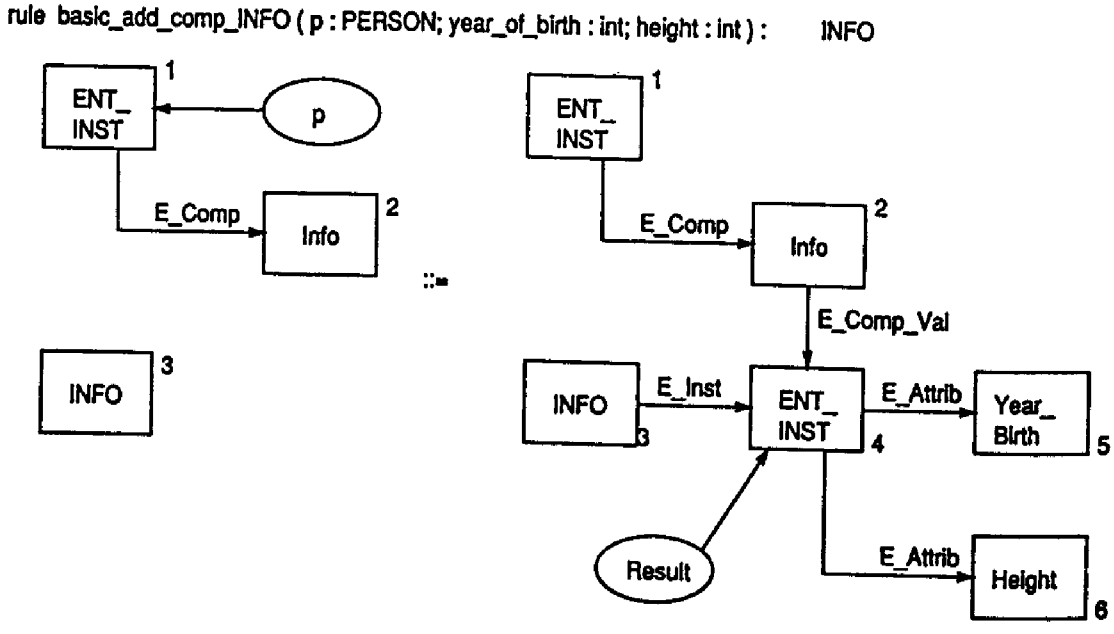

Figure 3.2

As an example for a basic delete action, we give the rule for the removal of the component INFO from a person. The current person is denoted by the node-valued parameter p.

rule basic_remove_comp_INFO (p : PERSON)

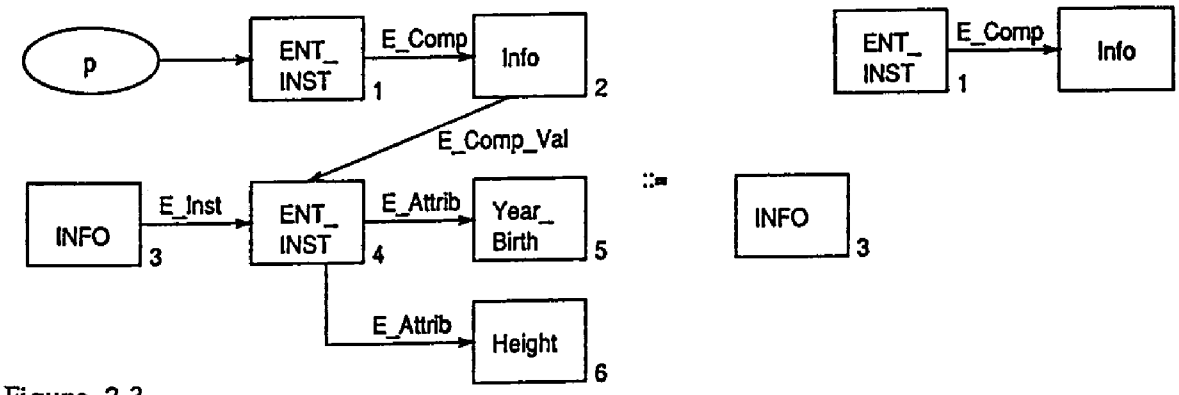

Figure 3.3

It is obvious that all these graph replacement rules for the description of basic actions on a database can automatically be derived from the specified EER schema and the corresponding graph representation. In this sense, a database designer, who has specified a database schema, has also implicitly specified all schema dependent basic actions. But, as we have already mentioned, the execution of a basic action may yield an incorrect database state. Therefore, basic actions have to be composed to elementary actions.

3.2 ELEMENTARY ACTIONS

Let us illustrate such an elementary action by the example of the insertion of a married person (cf. Fig. 3.4).

```
elem action insert_MARRIED ( name : string;
                             year_of_birth : int; height : int; year_of_marriage : int );
objects p : PERSON; i : INFO; m : MARRIED;
begin
  if not PERSON_exists ( name ) then
    p := basic_insert_PERSON ( name );
    i := basic_add_comp_INFO ( p, year_of_birth, height );
    m := basic_insert_cons_PERSON_MARRIED ( p, year_of_marriage );
  end
end
```

Figure 3.4: Elementary action for the insertion of a married person

Each person is uniquely identified by the key attribute name. So, at first, it is checked whether a person with this name already exists. The semantics of this test can be

given by a subgraph test (cf. Fig. 3.5). Afterwards, a sequence of three basic actions has to be executed, to insert a new instance of type **PERSON**, to add as component an instance of type **INFO**, and to add a new instance of type **MARRIED** to express the partition. Because of the generation of these three new instances, the schema inherent integrity constraints of Def. 1.2 are fulfilled. For example, the generation of a new instance of type **INFO** within the action **basic_add_comp_INFO** guarantees that **μ[COMP](Info)** is bijective. Therefore, the resulting database state is a correct database state in the sense of Def. 1.2.

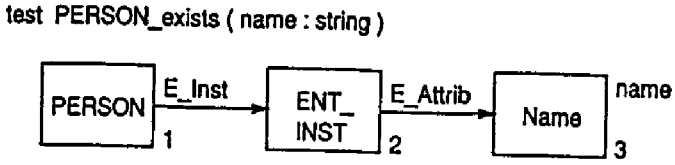

Figure 3.5: Example of a subgraph test

As each call of a basic action can be viewed as the application of a graph replacement rule and each boolean expression as the execution of a subgraph test, the whole elementary action describes a programmed graph replacement in the sense of [ELS 87], yielding an operational semantics for elementary actions.

In case of the deletion of a person, similar update propagation actions have to be executed to yield a new correct database state (cf. Figure. 3.6). Here, it has to be checked whether this person is a married or not-married person, and the existing one has to be deleted. In addition to basic modifying operations, there are basic read operations like **fetch_PERSON**, which yield as result a reference (or in the graph representation a node) to a database object. Furthermore, a person may participate in the relationship **lives_at**, where all occurrences have to be deleted, too.

```
elem action delete_PERSON ( name : string );
objects   p : PERSON;  m : MARRIED; n : NOT_MARRIED;
          la_set : set ( lives_at);
begin
  if PERSON_exists ( name ) then
    p := fetch_PERSON ( name );
    if MARRIED_exists ( p ) then
      m := fetch_PERSON_as_MARRIED ( name );
      basic_delete_cons_MARRIED ( m );
    else
      n := fetch_PERSON_as_NOT_MARRIED ( name );
      basic_delete_cons_NOT_MARRIED ( n );
    end;
```

```
        la_set := fetch_relship_lives_at_PERSON ( p );
        forall la in la_set do
          basic_delete_lives_at ( la )
        end;
        basic_remove_comp_INFO ( p );
        basic_delete_PERSON ( p )
      end
    end;
```

Figure 3.6: Elementary action for the deletion of a person

In both cases, only a subdiagram (or subgraph) of the current EER schema, termed *propagation subgraph*, is involved in the corresponding elementary action. For example, in the case of the insertion of an instance of an entity type e, instances of all entity types in the chain of partitions starting at a non-constructed entity type and ending at this entity type e have to be inserted. Furthermore, for each newly inserted instance all attribute values have to be set, and also all dependent components have to be inserted.

While the general proceeding for the insertion of a new instance is always the same, the shape of concrete elementary actions totally depend on the current database schema. Therefore, it is possible to build a generator for elementary actions, which automatically derives elementary actions from a given database schema. Because of our representation of database schemes by attributed graphs, this generator is a set of special graph algorithms. Each algorithm describes a traversal of the propagation subgraph of the Schema_Graph, which corresponds to the elementary action currently to be generated.

Figure 3.7 sketches the algorithm to generate an elementary insert action. This algorithm can be subdivided into four parts:

- In part 1, the chain of type constructions is computed, starting at the current entity type, indicated by ent_name, and ending at a non-constructed entity type.
- In part 2, the frame of an elementary insert action is generated (cf. Fig. 3.4), and the call of a basic insert action for an instance of the first entity type in the type construction chain is generated.
- In part 3, all components of this entity type are collected by an access (AG_...) to the attributed graph Schema_Graph, and appropriate basic actions to add components are generated. This generation may be recursive, if the component contains further components.
- In part 4, all entity types along the chain of type constructions are handled.

```
procedure  Gen_elem_insert_entity ( ent_name : string );
var var_name : string;
    ent_chain, components : list ( nodes );
    basic_ent, input_ent, output_ent : node;
begin
   /* part 1 */
   compute_partition_chain ( ent_name, ent_chain );
   /* part 2 */
   basic_ent := head ( ent_chain );
   Gen_frame_elem_insert_entity ( ent_name, basic_ent );
   Gen_basic_insert_entity ( basic_ent, var_name );
   /* part 3 */
   components := AG_get_target_node_list ( basic_ent, E_Comp );
   forall comp in components do
      Gen_add_component ( comp, var_name )
   end;
   /* part 4 */
   input_ent := basic_ent;
   while not empty ( ent_chain ) do
      output_ent := head ( ent_chain );
      Gen_handle_partition ( input_ent, output_ent, var_name );
      input_ent := output_ent
   end
end
```

Figure 3.7

A detailed description of the generators for elementary actions can be found in [Wo 89]. It is the topic of current research to prove that the generated elementary actions together with the graph replacement rules for basic actions describe transitions between correct database states, as they were defined in section 1.

4. CONCLUSIONS

Let us summarize the ideas of this paper:
We have shown that the description of the static structure of a database can be used to derive automatically corresponding modifying actions, termed elementary actions, which describe correct state transitions. The semantics of these elementary actions was given by programmed graph replacements. For this purpose, we have shown how a database state can be represented by an attributed graph, so that the semantics of basic actions, i.e. the constituents of elementary actions, can be given by the application of graph replacement rules.

These automatically derived elementary actions can then be used by a database designer to compose complex actions, by which greater parts of the database state are modified and which model specific actions of an application area.

The language for complex actions, the generator for elementary actions, and a corresponding interpreter are realized within CADDY ([Wo 89], [Sc 90]). CADDY is a database design environment, which supports a user in designing and testing a database on a conceptual level ([EHHLE 89]).

REFERENCES

[AbH 87] Abiteboul, S. / Hull, R.: *IFO - A Formal Semantic Database Model.* ACM Transactions on Database Systems 1987, Vol. 12, No. 4 (525 - 565)

[Ch 76] Chen, P.P.: *The Entity-Relationship Model - Towards a Unified View of Data.* ACM Transactions on Database Systems 1976, Vol. 1, No. 1 (9 - 36)

[EHHLE 89] Engels, G. / Hohenstein, U. / Hülsmann, K. / Löhr-Richter, P. / Ehrich,H.-D.: *CADDY: Computer-Aided Design of Non-Standard Databases.* In: N. Madhavji, H. Weber, W. Schäfer (eds.): Int. Conf. on System Development Environments & Factories. Berlin, May 1989. Pitman Publ., London 1990

[EK 80] Ehrig, H. / Kreowski, H.-J.: Applications of Graph Grammar Theory to Consistency, Synchronization, and Scheduling in Data Base Systems. In: Information Systems, Vol. 5, 1980, (225-238)

[ELS 87] Engels, G. / Lewerentz, C. / Schäfer, W.: Graph Grammar Engineering: A Software Specification Method. In: [ENRR 87], (186-201)

[En 86] Engels, G.: Graphen als zentrale Datenstrukturen in einer Software-Entwicklungsumgebung, Fortschrittber. VDI, Nr. 62, Düsseldorf, VDI-Verlag, 1986

[ENR 83] Ehrig, H. / Nagl, M. / Rozenberg, G. (eds.): Graph Grammars and Their Application to Computer Science, 2nd Intern. Workshop, LNCS 153, Berlin, Springer 1983

[ENRR 87] Ehrig, H. / Nagl, M. / Rozenberg, G. / Rosenfeld, A. (eds.): Graph Gram- mars and Their Application to Computer Science, 3rd Intern. Workshop, Warrenton (Virginia) 1986, LNCS 291, Berlin, Springer 1987

[ERA 80] P.P. Chen (ed.): *Proc. of the 1st Intern. Conference on Entity-Relationship Approach.* Los Angeles (California), 1980

[ERA 88] C. Batini (ed.): *Proc. of the 7th Int. Conference on Entity-Relationship Approach.* Rome (Italy) 1988

[ERA 89] F. Lochovski (ed.): *Proc. of the 8th Int. Conference on Entity-Relationship Approach.* Toronto (Canada) 1989

[EWH 85] Elmasri, R.A. / Weeldreyer, J. / Hevner, A.: *The Category Concept: An Extension to the Entity-Relationship Model.* Data & Knowledge Engineering 1985, Vol. 1 (75 - 116)

[FV 83] Furtado, A.L. / Veloso, P.: Specification of Data Bases Through Rewriting Rules. In: [ENR 83], (102 - 114)

[Gö 88] Göttler, H.: Graphgrammatiken in der Softwaretechnik. Informatik-Fachberichte 178, Berlin, Springer 1988

[HaM 81] Hammer, M. / McLeod, D.: *Database Description with SDM: A Semantic Database Model.* ACM Transactions on Database Systems 1981, Vol. 6, No. 3 (351 - 386)

[HK 87] Hull, R. / King, R.: *Semantic Database Modeling: Survey, Applications, and Research Issues.* ACM Computing Surv. 1987, Vol. 19, No. 3 (201 - 260)

[HNSE 87] Hohenstein, U. / Neugebauer, L. / Saake, G. / Ehrich, H.-D.: *Three-Level Specification Using an Extended Entity-Relationship Model.* In R.R. Wagner, R. Traunmüller, H.C. Mayr (eds.): Informationsbedarfsermittlung und -analyse für den Entwurf von Informationssystemen. Informatik-Fachberichte Band 143, Springer 1987 (58 - 88)

[HoGo 88] Hohenstein, U. / Gogolla, M.: *A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions.* In: [ERA 88] (129 -148)

[LyK 86] Lyngbaek, P. / Kent, W.: *A Data Modeling Methodology for the Design and Implementation of Information Systems.* In K.R. Dittrich, U. Dayal (ed.): Proc. of the Int. Workshop on Object-Oriented Database Systems, Pacific Grove (California) 1986 (6 - 17)

[MMR 86] Makowski, J.A. / Markowitz, V.M. / Rotics, N.: *Entity-Relationship Consistency for Relational Schemes.* In G. Ausiello, P. Atzeni (eds.): Proc. International Conference on Database Theory ICDT 1986, Springer LNCS 243 (306 - 322)

[MyW 80] Mylopoulos, J. / Wong, H.K.T.: *Some Features of the TAXIS Data Model.* In: Proc. 6th International Conference on Very Large Data Bases 1980, Montreal (Canada) (399 - 410)

[Na 79] Nagl, M. : Graph-Grammatiken: Theorie, Implementierung, Anwendungen. Braunschweig, Vieweg 1979

[Na 87] Nagl, M.: *A Software Development Environment Based on Graph Technology.* In: [ENRR 87], (458 - 478)

[PRYS 89] Parent, C. / Rolin, H. / Yètongon, K. / Spaccapietra, S.: *An ER Calculus for the Entity- Relationship Complex Model.* In: [ERA 89]

[Sc 90] Schmidt, R. : Entwurf und Implementierung eines Interpreters für eine Sprache zur Beschreibung schema-abhängiger Aktionen in einem erweiterten Entity-Relationship Modell, Diploma Thesis, TU Braunschweig, 1990

[SSW 80] Scheuermann, P. / Schiffner, G. / Weber, H. : *Abstraction Capabilities and Invariant Properties Modelling within the Entity-Relationship Approach.* In: [ERA 80], (121 - 140)

[TYF 86] Teorey, T.J. / Yang, D. / Fry, J.P.: *A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model.* ACM Computing Surveys 1986, Vol. 18, No. 2 (197 - 222)

[Wo 89] Wolff, M.: Eine Sprache zur Beschreibung schema-abhängiger Aktionen in einem erweiterten Entity-Relationship-Modell, Diploma Thesis, TU Braunschweig, 1989