Acta Cybernetica, Vol. 11, No. 1-2, Szeged, 1993

Fundamental Concepts of Object Oriented Databases

K.-D. Schewe* B. Thalheim*

Abstract

It is claimed that object oriented databases (OODBs) overcome many of the limitations of the relational model. However, the formal foundation of OODB concepts is still an open problem. Even worse, for relational databases a commonly accepted datamodel existed very early on whereas for OODBs the unification of concepts is missing. The work reported in this paper contains the results of our first investigations on a formally founded object oriented datamodel (OODM) and is intended to contribute to the development of a uniform mathematical theory of OODBs.

A clear distinction between *objects* and *values* turns out to be essential in the OODM. *Types* and *Classes* are used to structure values and objects repectively. Then the problem of unique object identification occurs. We show that this problem can be be solved for classes with extents that are completely representable by values. Such classes are called *value-representable*.

Another advantage of the relational approach is the existence of structurally determined *generic update operations*. We show that this property can be carried over to object-oriented datamodels if classes are value-representable. Moreover, in this case *database consistency* with respect to implicitly specified referential and inclusion constraints will be automatically preserved.

This result can be generalized with respect to distinguished classes of explicitly stated static constraints. Given some arbitrary method and some integrity constraint there exists a *greatest consistent specialization* (GCS) that behaves nice in that it is compatible with the conjunction of constraints. We present an algorithm for the GCS construction of user-defined methods and describe the GCSs of generic update operations that are required herein.

1 Introduction

The shortcomings of the relational database approach encouraged much research aimed at achieving more appropriate data models. It has been claimed that the *object-oriented approach* will be the key technology for future database systems and languages [8]. Several systems [4,6,7,9,15,16,17,19,26,36,37,38] arose from these

*Cottbus Technical University, Computer Science Institute, P.O.Box 101344, D-03013 Cottbus

efforts. However, in contrast to research in the relational area there is no common formal agreement on what constitutes an object-oriented database [10,11,13].

The basic question "What is an object?" seems to be trivial, but already here the variety of answers is large. In object oriented programming the notion of an *object* was intended as a generalization of the abstract data type concept with the additional feature of *inheritance*. In this sense object orientation involves the isolation of data in semi-independent modules in order to promote high software development productivity. The development of object oriented databases regarded an object also as a basic unit of persistent data, a view that is heavily influenced by existing semantic datamodels (SDMs) [2,29,31,39,40,60]. Thus, object oriented databases are composed of independent objects but must also provide for the maintenance of inter-object consistency, a demand that is to some degree in dissonance with the basic style of object orientation.

A view that is common in OODB research is that objects are abstractions of real world entities and should have an identity [8]. This leads to a distinction between *values* and *objects* [10,11]. A value is identified by itself whereas an object has an identity independent of its value. This object identity is usually encoded by object identifiers [1,3,34]. Abstracting from the pure physical level the identifier of an object can be regarded as being immutable during the object's lifetime. Identifiers ease the sharing and update of data. However, such abstract identifiers do not relieve us from the task to provide unique identification mechanisms for objects. In object oriented programming object names are sufficient, but retrieving mass data by name is senseless.

In most approaches to OODBs an object is coupled with a value of some fixed structure. To our point of view this contradicts already the goal of objects being abstractions of reality. In real situations an object has several and also changing aspects that should be captured by the object model. Therefore, in our object model each *object* o consists of a unique identifier id, a set of (type-, value-)pairs (T_i, v_i), a set of (reference-, object-)pairs (ref_j, o_j) and a set of methods $meth_k$.

Types are used to structure values. Classes serve as structuring primitive for objects having the same structure and behaviour. It is obvious that the multiple aspects view of an object allows them to be simultaneously members of more than one class and to change class memberships. This setting also makes every discussion on "object migration" unnessecary, as migration is only a specific form of value change.

In our model a class structure uniformly combines aspects of object values and references. The extent of classes varies over time, whereas types are immutable. Relationships between classes are represented by references together with referential constraints on the object identifiers involved. Moreover, each class is accompanied by a collection of methods. A schema is given by a collection of class definitions together with explicit integrity constraints.

The Identification Problem. One important concept of object-oriented databases is *object identity*. Following [1,12] the immutable identity of an object can be encoded by the concept of abstract object-identifiers. The advantages of this approach are that sharing, mutability of values and cyclic structures can be represented easily [42]. On the other hand, object identifiers do not have a meaning for the user and should therefore be hidden.

We study whether equality of identifiers can be derived from the equality of values. In the literature the notion of "deep" equality has been introduced for objects with equal values and references to objects that are also "deeply" equal. This recursive definition becomes interesting in the case of cyclic references.

Therefore, we introduce *uniqueness constraints*, which express equality on identifiers as a consequence of the equality of some values or references. On this basis we can address the problem how to characterize those classes that are completely representable (and hence also identifiable) by values.

Generic Update Operations. The success of the relational data model is due certainly to the existence of simple query and update-languages. Preserving the advantages of the relational in OODBs is a serious goal.

The generic querying of objects has been approached in [1,12]. While querying is per se a set-oriented operation, i.e. it is not necessary to select just one single object, and hence does not raise any specific problems with object identifiers, things change completely in case of updates. If an object with a given value is to be updated (or deleted), this is only defined unambigously, if there does not exist another object with the same value. If more than one object exists with the same value or more generally with the same value and the same references to other objects, then the user has to decide, whether an update- or delete-operation is applied to *all* these objects, to only *one* of these objects selected non-deterministically or to *none* of them, i.e. to reject the operation. However, it is not possible to specify a priori such an operation that works in the same way for all objects in all situations. The same applies to insert-operations. Hence the problem, in which cases operations for the insertion, deletion and update of objects can be defined generically.

Some authors [43] have chosen the solution to abandon generic operations. Others [6,7,9] use identifying values to represent object identity, thus embody a strict concept of surrogate keys to avoid the problem. Our approach is different from both solutions in that we use the concept of hidden abstract identifiers, but at the same time formally characterize those classes for which unique generic operations for the insertion, deletion and update of single objects can be derived automatically. It turns out that these are exactly the value-representable ones.

The Consistency Problem. One of the primary benefits that database systems offer is automatic enforcement of database integrity. One type of integrity is maintained through automatic concurrency control and recovery mechanisms; another one is the automatic enforcement of user-specified integrity constraints. Most commercial database systems, especially relational database management systems enforce only a bare minimum of constraints, largely because of the performance overhead associated with updates.

The *maintenance problem* is the problem how to ensure that the database satisfies its constraints after certain actions. There are at present two approaches to this maintenance problem. The first one, more classical is the modification of methods in accordance to the specified integrity constaints. The second approach uses generation mechanisms for the specified events. Upon occurrence of certain database events like update operations the management component is activated for integrity maintenance. The first research direction did not succeed because of some limitations within the approach. The second one is at present one of the most active database research areas. One of our objectives is to show that the first approach can be extended to object-oriented databases using stronger mathematical fundamentals.

Accuracy is an obviously important and desirable feature of any database. To this end, *integrity constraints*, conditions that data must satisfy before a database is updated, are commonly employed as a means of helping to maintain consistency. In relational databases the specification and enforcement of integrity constraints has a long tradition [61], whereas in OODBs the integrity problem has only recently drawn attention [48].

In object oriented databases, integrity maintenance can be based on two different approaches. The first one uses blind update operations. In this case, any update is allowed and the system organizes the maintenance. The second approach is based on methods rewriting. This approach is more effective. Assuming a consistent database state the modified method can not lead to an inconsistent state.

In relational databases distinguished classes of static integrity constraints have been discussed such as *inclusion*, *exclusion*, *functional*, *key* and *multi-valued dependencies*. All these constraints can be generalized to the object oriented case. Then the result on the existence of integrity preserving methods can be generalized to capture also these constraints. We shall also describe the resulting methods.

The Organization of the Paper. We start with a motivating example in Section 2 then introduce in Section 3 a core OODM to formalize the concepts used intuitively in the example. In Section 4 the notions of (weak) value-representability are introduced in order to handle the identification problem. The genericity problem will be approached in Section 5. We show the relationship between value-representability and the unique existence of generic update operations. The consistency problem is dealt with in Section 6. We outline an operational approach based on the computation of greatest consistent specializations (GCSs). Since the used algorithm allows the problem to be reduced to basic update operations, we describe the GCSs hereof. We summarize our results and describe some open problems in Section 7.

2 A Motivating Example

In this section we start giving a completely informal introduction to the OODM on the basis of a simple university example. We first introduce *types* and *classes*, then show an example of a *database instance*, i.e. the content of the database at a given timepoint. The representation of an instance requires *object identifiers*. Then we extend the example by introducing user-defined *constraints*. We shall see that this enables alternative representations without using identifiers, hence leads to the notion of *value-representability*. Finally, we indicate the definition of methods as a means to model database dynamics. For the sake of simplicity we only describe a *generic update method* that can be generated by the system.

As already said in the introduction, we distinguish between values and objects with the main difference defined by values identifying themselves whereas objects require an additional external identification mechanism. Types are used to structure values. Thus, let us first give some examples of types.

Example Basically, every type can be built from a few predefined *basic types* such as $BOOL$, NAT, $STRING$, etc. and also predefined *type constructors* for records, finite sets, lists, unions, etc.

The type definition for $PERSONNAME$ uses both a set constructor $\{\cdot\}$ and a (tagged) record constructor $(\cdot)$:

Type *PERSONNAME*
  = (FirstName : *STRING* ,
  SecondName : *STRING* ,
  Titles : *STRING*)
End *PERSONNAME*

The definition of a type *PERSON* uses the type *PERSONNAME*.

```
Type PERSON
   = ( PersonIdentityNo : NAT ,
       Name : PERSONNAME )
End PERSON
```

The following defines *STUDENT* as a subtype of *PERSON*, i.e. we can naturally project each value of type *STUDENT* onto a value of type *PERSON*.

```
Type STUDENT
   = ( PersonIdentityNo : NAT ,
       StudNo : NAT ,
       Name : PERSONNAME )

End STUDENT
```

Besides these definitions of types as sets of values we may also define new type constructors as follows, where α is a parameter for this new constructor:

```
Type MPERSON(α)
   = ( PersonIdentityNo : NAT ,
       Spouse : α )
End MPERSON
```

□

Next we use these types to build the structural part of an OODM schema. We define a schema as a collection of classes and a class as a variable collection of objects.

Example Each object in a class has a structure, which combines aspects of values associated with the object and references to other objects. This structure can be based on a type definition as above or involve itself a (nameless) type definition. Moreover, class definitions involve IsA relations in order to model objects in more than one class. We use ∘ to indicate concatenation for record types.

```
Schema University
   Class PERSONC
      Structure PERSON
   End PERSONC
   Class MARRIEDPERSONC
      IsA PERSONC
      Structure ( PersonIdentityNo : NAT ,
         Spouse : MARRIEDPERSONC )
   End MARRIEDPERSONC
   Class STUDENTC
      IsA PERSONC
      Structure STUDENT ∘
         ( Supervisor : PROFESSORC ,
         Major : DEPARTMENTC ,
         Minor : DEPARTMENTC )    End STUDENTC
   Class PROFESSORC
      IsA PERSONC
```

```
    Structure ( PersonIdentityNo : NAT ,
      Age : NAT ,
      Salary : NAT ,
      Faculty : DEPARTMENTC )    End PROFESSORC
  Class DEPARTMENTC
    IsA PERSONC
    Structure ( DeptName : STRING )
  End DEPARTMENTC                                          □
```

In principle, we are now able to describe the content of the database at a given timepoint. For such database instances we need a type ID of object identifiers that is used for two purposes, first as a unique and efficient internal identification mechanism for objects and second for modelling objects in different classes and references to other objects. In this case each class will be associated with a *representation type* that can be used directly for storing objects.

Example We use $\mathcal{D}$ as a name for the instance.

$\mathcal{D}$(PERSONC) =
$\{ (i_1, (123, (\text{"John"}, \text{"Denver"}, \{\text{"Professor"}, \text{"Dr"}\}))),$
$(i_2, (124, (\text{"Mary"}, \text{"Stuart"}, \{\text{"Dr"}\}))),$
$(i_3, (456, (\text{"John"}, \text{"Stuart"}, \{\}))),$
$(i_4, (567, (\text{"Laura"}, \text{"James"}, \{\}))),$
$(i_5, (987, (\text{"Dave"}, \text{"Ford"}, \{\})))\}$

$\mathcal{D}$(MARRIEDPERSONC) =
$\{ (i_1, (123, i_2)),$
$(i_2, (124, i_1))\}$

$\mathcal{D}$(PROFESSORC) =
$\{ (i_1, (123, 48, 8000, i_6))\}$

$\mathcal{D}$(STUDENTC) =
$\{ (i_3, (456, 1023, (\text{"John"}, \text{"Stuart"}, \{\}), i_1, i_6, i_7)),$
$(i_4, (567, 2134, (\text{"Laura"}, \text{"James"}, \{\}), i_1, i_6, i_7))\}$

$\mathcal{D}$(DEPARTMENTC) =
$\{ (i_6, (\text{"Computer Science"})),$
$(i_7, (\text{"Philosophy"})),$
$(i_8, (\text{"Music"}))\}$

□

Note that the following three conditions are satisfied by the instance:

- The object identifiers are unique within a class,
- the IsA relations in the schema give rise to set inclusion relationships for the underlying sets of identifiers (inclusion integrity), and
- the identifiers occurring within an object's value at a place corresponding to a reference, always occur as an object identifier in the referenced class (referential integrity).

We shall always refer to these conditions as model inherent constraints that must be satisfied by each instance. Other integrity constraints can be defined by the user

and added to the schema in order to capture more application semantics as shown in the next example.

Example First let us express that there are no two persons with the same PersonIdentityNo, no two students with the same StudentNo and no two departments with the same name. In order to formulate this, use x_P, x_S and x_D to refer to the content of the classes PERSONC, STUDENTC and DEPARTMENTC, and let $c_P : PERSON \rightarrow (\text{PersonIdentityNo} : NAT)$ and $c_S : STUDENT \times ID^3 \rightarrow (\text{StudNo} : NAT)$ be functions that arise from the natural projection to the components PersonIdentityNo and StudNo in $PERSON$ and $STUDENT$ respectively. This gives the following ***uniqueness constraints.***

$$\begin{aligned}
&\forall i, j :: ID. \forall v, w :: \\
&PERSON. (i, v) \in x_P \wedge (j, w) \in x_P \wedge c_P(v) = c_P(w) \Rightarrow i = j. \\
&\forall i, j :: ID. \forall v, w :: \\
&STUDENT \times ID^3. (i, v) \in x_S \wedge (j, w) \in x_S \wedge c_S(v) = c_S(w) \Rightarrow i = j \\
&\forall i, j :: ID. \forall v, w :: \\
&(\text{DeptName} : STRING). (i, v) \in x_D \wedge (j, w) \in x_D \wedge v = w \Rightarrow i = j \ . \quad (1)
\end{aligned}$$

Let us further assume that the salary of a professor is determined by his/her age. For this purpose, let $\text{Age}, \text{Salary} : T_{Prof} \rightarrow NAT$ be the natural projections to the Age- and Salary-values respectively. Then we have the following *functional constraint* on the class PROFESSORC:

$$\begin{aligned}
&\forall i, j :: ID. \forall v, w :: T_{Prof}. (i, v) \in x_{Prof} \wedge (j, w) \in x_{Prof} \wedge \text{Age}(v) = \text{Age}(w) \Rightarrow \\
&\text{Salary}(v) = \text{Salary}(w) \ . \quad (2)
\end{aligned}$$

Next assume that we want to guarantee that the spouse of a person's spouse is the person itself, which gives (with the abbreviations understood) the formula

$$\begin{aligned}
&\forall i, j :: ID. \forall v, w :: \\
&T_{MP}. (i, v) \in x_{MP} \wedge (j, w) \in x_{MP} \wedge \text{Spouse}(v) = j \Rightarrow \text{Spouse}(w) = i \ . (3)
\end{aligned}$$

Note that all these constraints are also satisfied by the instance above. □

Now we have added uniqueness constraints, the object identifiers used in instances correspond one-to-one to values of some types associated with the classes. These are the so-called value identification types V_C. Hence we could remove identifiers and represent the same information in a purely value-based fashion. In our example the value representation type for the class PERSONC is simply $PERSON$, but for the class MARRIEDPERSONC we need the recursive type

$$V_{MP} = PERSON \circ (\text{Spouse} : V_{MP})$$

with values that are rational trees [45,47].

So far only structural aspects (types, classes, constraints) have been considered. Let us now add methods to classes in order to model the dynamics of the database. In the OODM methods will be modelled in a simple procedural style.

Example Let us describe an insert-method for the class PERSONC.

insert$_{PersonC}$ (in: $P :: PERSON$, out: I :: ID) =
  IF $\exists\, O \in$ PERSONC . value(O) = P
  THEN I := ident(O)
  ELSE I := NewId ;
    PERSONC := PERSONC $\cup$ { (I,P)}
  ENDIF

For an insertion into the class MARRIEDPERSONC we need a more complex input type V recursively defined as

$$V = PERSON \circ (V \cup ID)$$

For each $P :: V$ let $f(P) :: PERSON$ be the projection onto $PERSON$ corresponding to the subtype relation between V and $PERSON$. Then we have

insert$_{MarriedPersonC}$ (in: $P :: V$, out: $I :: ID$) =
  I := insert$_{PersonC}(f(P))$;
  IF $\forall\, O \in$ MARRIEDPERSONC . ident(O) $\neq I$
  THEN P' := substitute(I,P,Spouse(P)) ;
      IF $P' :: ID$
      THEN $J := P'$
      ELSE J := insert$_{MarriedPersonC}(P')$
      ENDIF ;
    MARRIEDPERSONC := MARRIEDPERSONC $\cup$ { (I,$f(P) \circ (J)$)}
  ENDIF

We used the global method NewId to denote the selection of a new identifier. The expression substitute(I,P,T) denotes the result of replacing the value I for P in the expression T. Later we shall use a more abstract syntax oriented toward guarded commands [20,41,46]. □

Later we shall see that methods as described in this example are canonical and can be automatically derived from the schema. Corresponding generic update methods look quite similar with the only difference that there is no output. Such generic update methods only exist for value representable classes in which case, however, they enforce integrity with respect to the model inherent constraints. However, generic update methods need not be consistent with respect to the user-defined constraints. To achieve this, we have to apply the GCS algorithm to user-defined methods.

In the following sections we formally define the concepts above and proof the main results on value representation, generic updates and integrity enforcement.

3 A Core Object Oriented Datamodel

In this section we present a slightly modified version of the object oriented datamodel (OODM) of [45,47,49]. We observe that an object in the real world always has an identity. Therefore, abstract (i.e. system-provided) object identifiers are introduced to capture identity. However, neither the real world object that was the basis of the abstraction nor the abstract identifier can be used for the identification of an object.

In contrast to existing object oriented datamodels [1,3,4,6,7,8,9,16,17,26,36,37, 42,43,54] an object is not coupled with a unique type. In contrast, we observe that

real world objects can have different aspects that may change over time. Therefore, a primary decision was taken to let an object be associated with more than one type and to let these types even change during the object's lifetime. The same applies to references to other objects.

In the following let N_P, N_T, N_C, N_R, N_F, N_M and V denote arbitrary pairwise disjoint, denumerable sets representing parameter-, type-, class-, reference-, function-, method- and variable-names respectively.

3.1 A Simple Type System

Relational approaches to data modelling are called value-oriented since in these models real world entities are completely represented by their values. In the object-oriented approach we distinguish between objects and values. Values can be gouped into types. In general, a type may be regarded as an immutable set of values of a uniform structure together with operations defined on such values. Subtyping is used to relate values in different types.

In [12,47,49] algebraic type specifications as in [21,23] have been used to allow open type systems. For the sake of simplicity we deviate here from this approach and follow the more classical view of [14,15,45] using a type system that consists of some *basic types* such as *BOOL*, *NATURAL*, *INTEGER*, *STRING*, etc., and *type constructors* for records, finite sets, bags, lists, etc. and a *subtyping* relation. Moreover, assume the existence of *recursive types*, i.e. types defined by (a system of) domain equations. In principle we could use one of the type systems defined in [4,5,14,15,19,24,38]. In addition we suppose the existence of an abstract identifier type ID in $\mathcal{T}$ without any non-trivial supertype. Arbitrary *types* can then be defined by nesting. A type T without occurrence of ID will be called a *value-type*. We shall proceed giving a more formal definition of types.

Definition 1 *1. A* base type *is either* $BOOL$, NAT, INT, $FLOAT$, $STRING$, ID *or* $\perp$.

2. *Let* $a_i \in N_F$ *and* $\alpha, \beta, \alpha_i \in N_P$ $(i = 1, \ldots, n)$. *A* type constructor *is either* $(a_1 : \alpha_1, \ldots, a_n : \alpha_n)$ *(record),* $\{\alpha\}$ *(finite set),* $[\alpha]$ *(list),* $\langle\alpha\rangle$ *(bag) or* $\alpha \cup \beta$ *(union).*

3. *A* type t *is either a base type, a type constructor, a generalized constructor that results from replacing some parameters in a type constructor by types or a recursive type defined by an equation* $t = \{\alpha/t\}.t'$, *where* t' *is a generalized constructor and one of its parameters* α *is replaced by* $t \in N_T$.

   In the latter two cases the remaining parameters of the type constructor together with the parameters of the replacing types yield the parameters $\alpha_1, \ldots, \alpha_n$ *of* t.

4. *A type* t *is called* proper *iff the number of its parameters is 0.* t *is called a* value type *iff there is no occurrence of* ID *in* t.

5. *A* type form *consists of a type name* $t \in N_T$ *and a type* t' *with possibly some of its parameters replaced by type names.*

6. *A* type specification $\mathcal{T}$ *is a finite collection of type forms* $t_1, \ldots, t_n$ *such that the only type names occurring herein are the names of* $t_1, \ldots, t_n$.

The semantics of such types as sets of values is defined as usual. Moreover, we assume the standard operators on base types and on records, sets, bags, ... We omit the details here.

If t' is a proper type occurring in a type t, then there exists a corresponding *occurrence relation*

$$o : t \times t' \to BOOL .$$

Finally, we introduce subtypes. For a more detailed introduction to types see either [14] or [49].

Definition 2 *1. A* subtype relation $\leq$ *on types is given by the following rules:*

(a) Every type t is its own subtype and a subtype of $\bot$.

(b) $NAT \leq INT \leq FLOAT$.

(c) $(\ldots, a_{i-1} : \alpha_{i-1}, a_i : \alpha_i, a_{i+1} : \alpha_{i+1}, \ldots) \leq (\ldots, a_{i-1} : \alpha'_{i-1}, a_{i+1} : \alpha'_{i+1}, \ldots)$ *whenever* $\alpha_j \leq \alpha'_j$.

(d) $\left\{ \begin{array}{ccc} \{\alpha\} & \leq & \{\beta\} \\ [\alpha] & \leq & [\beta] \\ \langle\alpha\rangle & \leq & \langle\beta\rangle \end{array} \right\}$ *iff* $\alpha \leq \beta$.

(e) $\{\alpha\} \leq \langle\alpha\rangle$ *and* $[\alpha] \leq \langle\alpha\rangle$.

(f) $\alpha, \beta \leq \alpha \cup \beta$.

2. A subtype function *is a function $t' \to t$ from a subtype to its supertype ($t' \leq t$) defined by (a)-(f) above.*

3.2 The Class Concept as a Structural Primitive

The class concept provides the grouping of objects having the same structure which uniformly combines aspects of object values and references. Moreover, generic operations on objects such as object creation, deletion and update of its values and references are associated with classes provided these operations can be defined unambigously. Objects can belong to different classes, which guarantees each object of our abstract object model to be captured by the collection of possible classes. As for values that are only defined via types, objects can only be defined via classes.

Each object in a class consists of an identifier, a collection of values and references to objects in other classes. Identifiers can be represented using the unique identifier type ID. Values and references can be combined into a representation type, where each occurence of ID denotes references to some other classes. Therefore, we may define the structure of a class using parameterized types.

Definition 3 *1. Let t be a value type with parameters $\alpha_1, \ldots, \alpha_n$. For distinct reference names $r_1, \ldots, r_n \in N_R$ and class names $C_1, \ldots, C_n \in N_C$ the expression derived from t by replacing each α_i in t by $r_i : C_i$ for $i = 1, \ldots, n$ is called a* structure expression.

2. A structural class *consists of a class name $C \in N_C$, a structure expression S and a set of class names $D_1, \ldots, D_m \in N_C$ (in the following called the set of* superclasses*). We call r_i the* reference *named r_i from class C to class C_i. The type derived from S by replacing each reference $r_i : C_i$ by the type ID is called the* representation type *T_C of the class C, the type $U_C = (ident : ID, value :: T_C)$ is called the* class type *of C.*

3. A (structural) schema *S is a finite collection of structural classes $C_1, \ldots, C_n$ closed under references and superclasses.*

4. *An* instance $\mathcal{D}$ *of a structural schema* $\mathcal{S}$ *assigns to each class* C *a value* $\mathcal{D}(C)$ *of type* U_C *such that the following conditions are satisfied:*

   uniqueness of identifiers: *For every class* C *we have*

   $$\forall i :: ID.\forall v, w :: T_C.(i, v) \in \mathcal{D}(C) \wedge (i, w) \in \mathcal{D}(C) \Rightarrow v = w \ . \quad (4)$$

   inclusion integrity: *For a subclass* C *of* C' *we have*

   $$\forall i :: ID.i \in dom(\mathcal{D}(C)) \Rightarrow i \in dom(\mathcal{D}(C')) \ . \quad (5)$$

   Moreover, if T_C *is a subtype of* T'_C *with subtype function* $f : T_C \rightarrow T'_C$, *then we have*

   $$\forall i :: ID.\forall v :: T_C.(i, v) \in \mathcal{D}(C) \Rightarrow (i, f(v)) \in \mathcal{D}(C') \ . \quad (6)$$

   referential integrity: *For each reference from* C *to* C' *with corresponding occurrence relation* o_r *we have*

   $$\forall i, j :: ID.\forall v :: T_C.(i, v) \in \mathcal{D}(C) \wedge o_r(v, j) \Rightarrow j \in dom(\mathcal{D}(C')) \ . \quad (7)$$

3.3 User Defined Integrity Constraints

Let us now extend the notion of schema by the introduction of explicit user-defined integrity constraints. First we define the notion of constraint schema in general, then we restrict ourselves to distinguished classes of constraints that arise as generalizations of constraints known from the relational model, e.g. functional and key constraints, inclusion and exclusion constraints [48,52].

Definition 4 *Let* $\mathcal{S} = \{C_1, \ldots, C_n\}$ *be a structural schema.*

1. *An* integrity constraint *on* $\mathcal{S}$ *is a formula* I *over the underlying type system with free variables* $fr(I) \subseteq \{x_{C_1}, \ldots, x_{C_n}\}$, *where each* x_{C_i} *is a variable of type* $\{U_{C_i}\}$. *We call* x_{C_i} *the* class variable *of* C_i.

2. *A* constrained schema *consists of a structural schema* $\mathcal{S}$ *and a finite set of integrity constraints on* $\mathcal{S}$.

3. *An* instance *of a constrained schema is an instance of the underlying structural schema. An instance* $\mathcal{D}$ *is said to be* consistent *with respect to the integrity constraint* I *iff substituting* $\mathcal{D}(C)$ *for each class variable* x_C *in* I *evaluates to true, when interpreted in the usual way.*

Note that the conditions for an instance in Definition 4 correspond to model inherent integrity constraints. We refer to these constraints as implicit *identifier*, *IsA* and *referential* constraints on the schema $\mathcal{S}$. Let us now define some distinguished classes of user-defined constraints.

Definition 5 *Let* C, C_1, C_2 *be classes in a schema* $\mathcal{S}$ *and let* $c^i : T_C \rightarrow T_i$ *(*$i = 1, 2, 3$*) and* $c_i : T_{C_i} \rightarrow T$ *(*$i = 1, 2$*) be subtype functions.*

1. *A* functional constraint *on* C *is a constraint of the form*

   $$\forall i, i' :: ID.\forall v, v' :: T_C.\ c^1(v) = c^1(v') \wedge (i, v) \in x_C \wedge (i', v') \in x_C \Rightarrow c^2(v) = c^2(v'). \quad (8)$$

2. *A* uniqueness constraint *on C is a constraint of the form*

$$\forall i, i' :: ID. \forall v, v' :: \\ T_C.\, c^1(v) = c^1(v') \wedge (i, v) \in x_C \wedge (i', v') \in x_C \Rightarrow i = i'. \qquad (9)$$

A uniqueness constraint on C is called trivial *iff $T_C = T_1$ and $c^1 = id$ hold.*

3. *An* inclusion constraint *on C_1 and C_2 is a constraint of the form*

$$\forall t :: T. \exists i_1 :: ID, v_1 :: T_{C_1}.(i_1, v_1) \in x_{C_1} \wedge c_1(v_1) = t \Rightarrow \\ \exists i_2 :: ID, v_2 :: T_{C_2}.(i_2, v_2) \in x_{C_2} \wedge c_2(v_2) = t\ . \qquad (10)$$

4. *An* exclusion constraint *on C_1, C_2 is a constraint of the form*

$$\forall i_1, i_2 :: ID. \forall v_1 :: T_{C_1}. \forall v_2 :: \\ T_{C_2}.(i_1, v_1) \in x_{C_1} \wedge (i_2, v_2) \in x_{C_2} \Rightarrow c_1(v_1) \neq c_2(v_2)\ . \qquad (11)$$

3.4 Methods as a Basis for Behaviour Modelling

So far, only static aspects have been considered. A structural schema is simply a collection of data structures called classes. Let us now turn to adding dynamics to this picture. As required in the object oriented approach operations will be associated with classes. This gives us the notion of a method.

We shall distinguish between visible and hidden methods to emphasize those methods that can be invoked by the user and others. This is not intended to define an interface of a class, since for the moment all methods of a class including the hidden ones can be accessed by other methods. The justification for such a weak hiding concept is due to two reasons.

- Visible methods serve as a means to specify (nested) transactions. In order to build sequences of database instances we only regard these transactions assuming a linear invocation order on them.
- Hidden methods can be used to handle identifiers. Since these identifiers do not have any meaning for the user, they must not occur within the input or output of a transaction.

Definition 6 *Let S be a structural schema.*
Let $T_1, \ldots, T_n, T'_1, \ldots, T'_m$ be types, $M \in N_M$ and $\iota_1, \ldots, \iota_n, o_1, \ldots, o_m \in V$.

1. *A* method signature *consists of a method name M, a set of input-parameter / input-type pairs $\iota_i :: T_i$ and a set of output-parameter / output-type pairs $o_j :: T'_j$. We write*

$$o_1 :: T'_1, \ldots, o_m :: T'_m \leftarrow M(\iota_1 :: T_1, \ldots, \iota_n :: T_n)\ .$$

2. *Let C be some structural class in S. A* method *M on C consists of a method signature with name M and a body that is recursively built from the following constructs:*

   (a) *assignment $x := E$, where x is either the class variable x_C or a local variable within S, and E is a term of the same type as x,*

(b) skip, fail, loop,

(c) sequential composition $S_1; S_2$*, choice* $S_1 \square S_2$*, projection* $x :: T \mid S$*, guard* $P \rightarrow S$*, restricted choice* $S_1 \boxtimes S_2$*, where* P *is a well-formed formula and* x *is a variable of type* T*, and*

(d) instantiation $x'_1, \ldots, x'_i \leftarrow C' : S'(E'_1, \ldots, E'_j)$*, where* S' *is a method on class* C' *with input-parameters* $\iota'_1, \ldots, \iota'_j$ *and output-parameters* $o'_1, \ldots, o'_i$*, such that the variables* o'_f*,* x'_f *have the same type and the term* E'_g *has the same type as the variable* ι'_g*.*

3. *A method* M *on a class* C *with signature* $o_1 :: T'_1, \ldots, o_m :: T'_m \leftarrow M(\iota_1 :: T_1, \ldots, \iota_n :: T_n)$ *is called* value-defined *iff all* T_i $(i = 1 \ldots n)$ *and* T'_j $(j = 1, \ldots, m)$ *are proper value types.*

As already mentioned the OODM distinguishes between transactions, i.e. methods visible to the user, and hidden methods. We require each transaction to be value-defined.

Subclasses inherit the methods of their superclasses, but overriding is allowed as long as the new method is a specialization of all its corresponding methods in its superclasses. Overriding becomes mandatory in the case of multiple inheritance with name conflicts. A method that overrides a hidden method on some superclass must also be hidden.

Definition 7 *Let* S *be a structural schema and* $C \in S$ *be a structural class as in Definition 3 with superclasses* $D_1, \ldots, D_k$*. A* method specification *on* C *consists of two sets of methods* $\mathcal{S} = \{M_1, \ldots, M_n\}$ *(called* transactions*) and* $\mathcal{H} = \{M'_1, \ldots, M'_m\}$ *(called* hidden methods*) such that the following properties hold:*

1. *Each* M_i $(i = 1, \ldots, n)$ *is value-defined.*

2. *For each transaction* M^l *on some superclass* D_l *there exists some* $i \in \{1, \ldots, n\}$ *such that* M_i *specializes* M^l*.*

3. *For each hidden method* M^l *on some superclass* D_l *there exists some* $j \in \{1, \ldots, m\}$ *such that* M'_j *specializes* M^l*.*

Let us briefly discuss what specialization means for the input- and output-types. Sometimes it is required that the input-type for an overriding method should be a subtype of the original one (covariance rule), sometimes the opposite (contravariance rule) is required. The first rule applies e.g. if we want to override an insert method. In this case the inherited method has no effect on the subclass, but simply calls the "old" method. The second rule applies if input-types required on the superclass can be omitted on the subclass. Both rules are captured by the formal notion of specialization. We omit the details [44]. Now we are prepared to generalize the definition of classes and schemata.

Definition 8 *1. A* class *consists of a class name* $C \in N_C$*, a structure expression* S*, a set of class names* $D_1, \ldots, D_m \in N_C$ *(called the set of* superclasses*) and a method specification* $(\mathcal{S} = \{M_1, \ldots, M_n\}, \mathcal{H} = \{M'_1, \ldots, M'_{n'}\})$ *on* C*.*

2. *A* (behavioural) schema S *is a finite collection of classes* $\{C_1, \ldots, C_n$ *closed under references, superclasses and method call together with a collection of integrity constraints* $I_1, \ldots, I_n$ *on* S.

3. *An* instance D *of a behavioural schema* S *is an instance of the underlying structural schema. A* database history *on* S *is a sequence* $D_0, D_1, \ldots$ *of instances such that each transition from* D_{i-1} *to* D_i *is due to some transaction on some class* $C \in S$.

Note the relation between database histories used here and the work on the semantics of object bases in [22,28].

3.5 Queries and Views

Roughly speaking the querying of a database is an operation on the database without changing its state. The emphasis of a query is on the output. While such a general view of queries can be subsumed by transactions, hence by methods in the OODM, query languages are in particular intended to be declarative in order to support an ad-hoc querying of a database without the need to write new transactions [8].

Querying a relational database can be expressed by terms in relational algebra. This view can be easily generalized to the OODM using its type system. Therefore, terms over such types occur naturally. Moreover, type specifications are based on other type specifications via constructors, selectors and functions. Hence, $\mathcal{T}$ allows arbitrary terms involving more than one class variable x_C to be built. Then a *query* turns out be be represented by term t over some type T such that the free variables of t are all class variables. This approach is in accordance with the algebraic approach in [12] and with so called *universal traversal combinators* [25].

In relational algebra a *view* may be regarded simply as a stored query (or derived relation). We shall try to generalize also this view to the OODM.

However, things change dramatically, when object identifiers come into play [13], since now we have to distinguish between queries that result in values and those that result in (collections of) objects. Therefore we distinguish in the OODM between value queries and general access expressions.

A *value query* on a schema S can then be represented by a term t of some value type T with $fr(t) \subseteq \{x_C \mid C \in S\}$. Ad-hoc querying of a database should then be restricted to value queries. This is no loss of generality, because for any type T in $\mathcal{T}$ involving identifiers there exists a corresponding type T' allowing multiple occurrences. Take e.g. a class C. If we want to get all the objects in that class no matter whether they have the same values or not, the corresponding term would be x_C. This is not a value query, but if T_C is a value type, we may take $T' = \langle T_C \rangle$ and the natural projection given by the subtype functions

$$\{(ident : ID, value : \alpha)\} \rightarrow \langle (ident : ID, value : \alpha) \rangle \rightarrow \langle \alpha \rangle \ .$$

In the case of arbitrary access expressions another problem occurs [13]. So far, we can only build terms t that involve identifiers already existing in the database. Thus, such queries are called *object preserving*. If we want the result of a query to represent "new" objects, i.e. if we want to have *object generating queries*, we have to apply a mechanism to create new object identifiers. This can be achieved by *object creating functions* on the type ID with arity $ID \times \ldots \times ID \rightarrow ID$ [32,35].

The idea that a view is a stored query then carries over easily. However, the structure of a view should be compatible with the structure of the schema, i.e. each view may be regarded as a derived class. Summarizing, we get the following formal definition.

Definition 9 *Let $S = \{C_1, \ldots, C_n\}$ be some schema.*

1. *A* value query *on S is a term t over some proper value type T with $fr(t) \subseteq \{x_{C_1}, \ldots, x_{C_n}\}$.*
2. *An* access expression *on S is a term t over some proper type T with $fr(t) \subseteq \{x_{C_1}, \ldots, x_{C_n}\}$.*
3. *A* view *on S consists of a view name $v \in N_C$ such that there is no class $C \in S$ with this name, a structure expression $S(v)$ containing references to classes in S or to views on S and a defining access expression $t(v)$ of type $\{U_v\}$, where T_v is the representation type corresponding to $S(v)$.*
4. *A* (complete) schema *is a behavioural schema together with a finite set of views. An* instance *of a complete schema is an instance of the underlying structural schema such that for every view v replacing each class variable x_C in the access expressions of v yields a value of type $\{U_v\}$ satisfying the uniqueness property for identifiers.*

4 The Object Identification Problem

From an object oriented point of view a database may be considered as a huge collection of objects of arbitrary complex structure. Hence the problem to uniquely identify and retrieve objects in such collections.

Each object in a database is an abstraction of a real world object that has a unique *identity*. The representation of such objects in the OODM uses an abstract identifier I of type ID to encode this identity. Such an identifier may be considered as being immutable. However, from a systems oriented view permutations or collapses of identifiers without changing anything else should not affect the behaviour of the database.

For the user the abstract identifier of an object has no meaning. Therefore, a different access to the identification problem is required. We show that the unique identification of an object in a class leads to the notion of *(weak) value-identifiability*, where weak value-representability can be used to capture also objects that do not exists for there own, but depend on other objects. This is related to *weak entities* in entity-relationship models [62]. The stronger notion of *value-representability* is required for the unique definition of generic update operations.

4.1 The Notion of Value-Representability

According to our definitions two objects in a class C are identical iff they have the same identifier. By the use of constraints, especially uniqueness constraints, we could restrict this notion of equality.

Let us address the characterization of those classes, the objects in which are completely representable by values, i.e. we could drop the object identifiers and replace references by values of the referred object. We shall see in Section 5 that in case of value-representable classes we are able to preserve an important advantage of relational databases, i.e. the existence of structurally determined update operations.

Definition 10 *Let C be a class in a schema S with representation type T_C.*

1. *C is called* value-identifiable *iff there exists a proper value type I_C such that for all instances $\mathcal{D}$ of S there is a function $c : T_C \to I_C$ such that the uniqueness constraint on C defined by c holds for $\mathcal{D}$.*

2. *C is called* value-representable *iff there exists a proper value type V_C such that for all instances $\mathcal{D}$ of S there is a function $c : T_C \to V_C$ such that for $\mathcal{D}$*

   (a) *the uniqueness constraint on C defined by c holds and*

   (b) *for each uniqueness constraint on C defined by some function $c' : T_C \to V'_C$ with proper value type V'_C there exists a function $c'' : V_C \to V'_C$ that is unique on $c(codom(\mathcal{D}(C)))$ with $c' = c'' \circ c$.*

It is easy to see that each value-representable class C is also value-identifiable. Moreover, the *value-representation type* V_C in Definition 10 is unique up to isomorphism.

4.2 Value-Representability in the Case of Acyclic Reference Graphs

Since value-representability is defined by the existence of a certain proper value type, it is hard to decide, whether an arbitrary class is value-representable or not. In case of simple classes the problem is easier, since we only have to deal with uniqueness and value constraints. In this case it is helpful to analyse the reference structure of the class. Hence the following graph-theoretic definitions.

Definition 11 *The* reference graph *of a class C in a schema S is the smallest labelled graph $G_{ref} = (V, E, l)$ satisfying:*

1. *There exists a vertex $v_C \in V$ with $l(v_C) = \{t, C\}$, where t is the top-level type in the structure expression S of C.*

2. *For each proper occurrence of a type $t \neq ID$ in T_C there exists a unique vertex $v_t \in V$ with $l(v_t) = \{t\}$.*

3. *For each reference $r_i : C_i$ in the structure expression S of C the reference graph G^i_{ref} is a subgraph of G_{ref}.*

4. *For each vertex v_t or v_C corresponding to $t(x_1, \ldots, x_n)$ in S there exist unique edges $e_t^{(i)}$ from v_t or v_C respectively to v_{t_i} in case x_i is the type t_i or to v_{C_i} in case x_i is the reference $r_i : C_i$. In the first case $l(e_t^{(i)}) = \{S_i\}$, where S_i is the corresponding selector name; in the latter case the label is $\{S_i, r_i\}$.*

Definition 12 *1. Let $S = \{C_1, \ldots, C_n\}$ be a schema. Let $S' = \{C'_1, \ldots, C'_n\}$ be another schema such that for all i there exists a uniqueness constraint on C_i defined by some $c_i : T_{C_i} \to T_{C'_i}$. Then an* identification graph *G_{id} of the class C_i is obtained from the reference graph of C'_i by changing each label C'_j to C_j.*

2. *The identification graph G_{id} resulting from the use of trivial uniqueness constraints is called the* standard identification graph.

Clearly, there need not exist any identification graph nor does the existence of one identification graph imply the existence of the standard one. However, if the standard identification graph exist, then it is equal to the reference graph.

Proposition 13 *Let C be a class in a schema S with acyclic reference graph G_{ref} such that there exist uniqueness constraints for C and each C_i such that C_i occurs as a label in G_{ref}. Then C is value-representable.*

Proof. We use induction on the maximum length of a path in G_{ref}. If there are no references in the structure expression S of C the type T_C is a proper value type. Since there exists a uniqueness constraint on C, the identity function id on T_C also defines a uniqueness constraint. Hence $V_C = T_C$ satisfies the requirements of Definition 10.

If there are references $r_i : C_i$ in the structure expression S of C, then the induction hypothesis holds for each such C_i, because G_{ref} is acyclic. Let V_C result from S by replacing each $r_i : C_i$ by V_{C_i}. Then V_C satisfies the requirements of Definition 10. □

Corollary 14 *Let C be a class in a schema S such that there exist an acyclic identification graph G_{id} and uniqueness constraints for C and each C_i occuring as a label in G_{id}. Then C is value-identifiable.*

4.3 Computation of Value Representation Types

We want to address the more general case where cyclic references may occur in the schema $S = \{C_1, \ldots, C_n\}$. In this case a simple induction argument as in the proof of Proposition 13 is not applicable. So we take another approach. We define algorithms to compute types V_C and I_C that turn out to be proper value types under certain conditions. In the next subsection we then show that these types are the value representation type and the value identification type required by Definition 10.

Algorithm 15 *Let $F(C_i) = T_i$ provided there exists a uniqueness constraint on C_i defined by $c_i : T_{C_i} \to T_i$, otherwise let $F(C_i)$ be undefined. If ID occurs in some $F(C_i)$ corresponding to $r_j : C_j$ $(j \neq i)$, we write ID_j.*

Then iterate as long as possible using the following rules:

1. *If $F(C_j)$ is a proper value type and ID_j occurs in some $F(C_i)$ $(j \neq i)$, then replace this corresponding ID_j in $F(C_i)$ by $F(C_j)$.*

2. *If ID_i occurs in some $F(C_i)$, then let $F(C_i)$ be recursively defined by $F(C_i) == S_i$, where S_i is the result of replacing ID_i in $F(C_i)$ by the type name $F(C_i)$.*

This iteration terminates, since there exists only a finite collection of classes. If these rules are no longer applicable, replace each remaining occurrence of ID_j in $F(C_i)$ by the type name $F(C_j)$ provided $F(C_j)$ is defined. □

Note that the the algorithm computes (mutually) recursive types. Now we give a sufficient condition for the result of Algorithm 15 to be a proper value type.

Lemma 16 *Let C be a class in a schema S such that there exists a uniqueness constraint for all classes C_i occurring as a label in some identification graph G_{id} of C. Let I_C be the type $F(C)$ computed by Algorithm 15 with respect to the uniqueness constraints used in the definition of G_{id}. Then I_C is a proper value type.*

Proof. Suppose I_C were not a proper value type. Then there exists at least one occurrence of ID in I_C. This corresponds to a class C_i without uniqueness constraint occurring as a label in G_{id}, hence contradicts the assumption of the lemma. □

4.4 The Finiteness Property

Let us now address the general case. The basic idea is that there is always only a finite number of objects in a database. Assuming the database being consistent with respect to inclusion and referential constraints yields that there can not exist infinite cyclic references. This will be expressed by the *finiteness property*. We show that this property allows the computation of value representation types.

Definition 17 *Let C be a class in a schema S and let $g_{k,l}$ denote a path in G_{ref} from v_{C_k} to v_{C_l} provided there is a reference $r_l : C_l$ in the structure expression of C_k. Then a* cycle *in G_{ref} is a sequence $g_{0,1} \cdots g_{n-1,n}$ with $C_0 = C_n$ and $C_k \neq C_l$ otherwise.*

Note that we use paths instead of edges, because the edges in G_{ref} do not always correspond to references. According to our definition of a class there exists a referential constraint on C_k, C_l defined by $o_{k,l} : T_{C_k} \times ID \to BOOL$ corresponding to $g_{k,l}$. Therefore, to each cycle there exists a corresponding sequence of functions $o_{0,1} \cdots o_{n-1,n}$. This can be used as follows to define a function $cyc : ID \times ID \to BOOL$ corresponding to a cycle in G_{ref}.

Definition 18 *Let C be a class in a schema S and let $g_{0,1} \cdots g_{n-1,n}$ be a cycle in G_{ref}. The corresponding* cycle relation *$cyc : ID \times ID \to BOOL$ is defined by $cyc(i,j) = true$ iff there exists a sequence $i = i_0, i_1, \ldots, i_n = j$ $(n \neq 0)$ such that $(i_l, v_l) \in C_l$ and $o_{l,l+1}(i_{l+1}, v_l) = true$ for all $l = 0, \ldots, n-1$.*

Given a cycle relation cyc, let cyc^m the m-th power of cyc.

Lemma 19 *Let C be a class in a schema S. Then C satisfies the finiteness property, i.e. for each instance $\mathcal{D}$ of S and for each cycle in G_{ref} the corresponding cycle relation cyc satisfies*

$$\forall i \in dom(C).\, \exists n.\, \forall j \in dom(C).\, \exists m < n.\, \left(cyc^n(i,j) = true \Rightarrow cyc^m(i,j) = true\right).$$

Proof. Suppose the finiteness property were not satisfied. Then there exist an instance $\mathcal{D}$, a cycle relation cyc and an object identifier i_0 such that

$$\forall n.\, \exists j \in dom(C).\, \forall m < n.\, \left(cyc^n(i_0,j) = true \wedge cyc^m(i_0,j) = false\right)$$

holds. Let such a j corresponding to $n > 0$ be i_n. Then the elements $i_0, i_1, i_2, \ldots$ are pairwise distinct. Hence there would be infinitely many objects in $\mathcal{D}$ contradicting the finiteness of a database. □

Lemma 20 *Let $\mathcal{D}$ be an instance of schema $S = \{C_1, \ldots, C_n\}$. Then $\mathcal{D}$ satisfies at each stage of Algorithm 15 uniqueness constraints for all $i = 1, \ldots, n$ defined by some $c'_i : T_{C_i} \to F(C_i)$.*

Proof. It is sufficient to show that whenever a rule is applied replacing $F(C_i)$ by $F(C_i)'$, then $F(C_i)'$ also defines a uniqueness constraint on C_i.

Suppose that $(i, v) \in C_i$ holds in $\mathcal{D}$. Since it is possible to apply a rule to $F(C_i)$, there exists at least one value $j :: ID$ occurring in $c_i(v)$. Replacing ID_j in $F(C_i)$ corresponds to replacing j by some value $v_j :: F(C_j)$. Because of the finiteness property such a value must exist. Moreover, due to the uniqueness constraint defined by c_j the function $f : F(C_i) \to F(C_i)'$ representing this replacement must be injective on $c_i(codo(\mathcal{D}(C_i)))$. Hence, $c_i' = f \circ c_i$ defines a uniqueness constraint on C_i. □

Now assume that we use only trivial uniqueness constraints in Algorithm 15. In order to distinguish this situation from the general case we write $G(C_i)$ instead of $F(C_i)$ to refer to this special case.

Lemma 21 *Let $\mathcal{D}$ be an instance of schema $\mathcal{S} = \{C_1, \ldots, C_n\}$. Then at each stage of Algorithm 15 (applied with arbitrary uniqueness constraints and in parallel with trivial ones) there exists for all $i = 1, \ldots, n$ a function $\bar{c}_i : G(C_i) \to F(C_i)$ that is unique on $c_i(codom(\mathcal{D}(C_i)))$ with $c_i' = \bar{c}_i \circ c_i$.*

Proof. As in the proof of Lemma 20 it is sufficient to show that the required property is preserved by the application of a rule from any of the two versions of Algorithm 15. Therefore, let $\bar{c}_i$ satisfy the required property and let $g : G(C_i) \to G(C_i)'$ and $f : F(C_i) \to F(C_i)'$ be functions corresponding to the application of a rule to $G(C_i)$ and $F(C_i)$ respectively. Such functions were constructed in the proofs of Lemma 20 and Lemma 20 respectively.

Then $f \circ \bar{c}_i$ satisfies the required property with respect to the application of f. In the case of applying g we know that g is injective on $c_i(codom(\mathcal{D}(C_i)))$. Let $h : G(C_i)' \to G(C_i)$ be any continuation of $g^{-1} : g(c_i(codom(\mathcal{D}(C_i)))) \to G(C_i)$. Then $\bar{c}_i \circ h$ satisfies the required property. □

Theorem 22 *Let C be a class in a schema $\mathcal{S}$ such that there exists a uniqueness constraint for all classes C_i occurring as a label in the reference graph G_{ref} of C. Let V_C be the type $G(C)$ computed by Algorithm 15 with respect to trivial uniqueness constraints and let I_C be the type $F(C)$ computed by Algorithm 15 with respect to arbitrary uniqueness constraints. Then C is value-representable with value representation type V_C and each such I_C is a value identification type.*

Proof. V_C is a proper value type by Lemma 16. From Lemma 20 it follows that if $\mathcal{D}$ is an instance of $\mathcal{S}$, then there exists a function $c : T_C \to V_C$ such that the uniqueness constraint defined by c holds for $\mathcal{D}$. The same applies to I_C.

If V_C' is another proper value type and $\mathcal{D}$ satisfies a uniqueness constraint defined by $c' : T_C \to V_C'$, then V_C' is some value-identification type I_C. Hence by Lemma 21 there exists a function $c'' : V_C \to V_C'$ that is unique on $c(codom(\mathcal{D}(C)))$ with $c' = c'' \circ c$. This proves the Theorem. □

Corollary 23 *Let $\mathcal{S}$ be a schema such that all classes C in $\mathcal{S}$ are value-identifiable. Then all classes C in $\mathcal{S}$ are also value-representable.*

□

4.5 Weak Value-Representability

Let us now ask whether there exist also weaker identification mechanisms other than value-representability. In several papers, e.g. [42] a navigational approach on the

basis of the reference structure has been favoured. This leads to dependent classes similar to "weak entities" in the entity-relationship model [62]. We shall show that such an approach requires at least a value-identifiable "entrance" of some path and the hard restriction on references to be representable by surjective functions.

Definition 24 *Let S be some schema.*

1. If r is a reference from class C to D in S and $o : T_C \times ID \to BOOL$ is the function of Definition 4 expressing the corresponding referential constraint, then r satisfies the *(SF)-condition* iff

   (a) $o(v,i) \wedge o(v,j) \Rightarrow i = j$ and
   (b) $j \in dom(x_D) \Rightarrow \exists v :: T_C . v \in codom(x_C) \wedge o(v,j)$

   hold for all $i, j :: ID, v :: T_C$.

2. An *(SF)-chain* from class D to C in S is a sequence of classes $D = C_0, \ldots, C_n = C$ such that for all i $(i = 1, \ldots, n)$ either C_i is a subclass of C_{i-1} or there exists a reference r_i from C_{i-1} to C_i satisfying the (SF)-condition.

3. A class C in S is called *weakly value-identifiable* iff there exists a value-identifiable class D and an (SF)-chain from D to C.

The notation (SF)-condition has been chosen to emphasize that such a reference represents a surjective function. It is easy to see taking $n = 0$ that each value-identifiable class is also weakly value-identifiable.

Lemma 25 *If C is a weakly value-identifiable class in a schema S, then there exists a proper value type I_C such that for each instance $\mathcal{D}$ of S there exists a function $c : ID \to I_C$ such that c is injective on $dom(\mathcal{D}(C))$.*

Call I_C a *weak value-identification type* of the class C.

Proof. Let $D = C_0, \ldots, C_n = C$ be an (SF)-chain from the value-identifiable class D to C with corresponding references r_i $(i = 1, \ldots, n)$. If r_i satisfies the (SF)-condition, there exists a function $c_i : ID \to ID$ such that $j \in dom(\mathcal{D}(C_i)) \Rightarrow (c_i(j), v) \in x_{C_{i-1}}$ for some v with $o_i(v,j)$ (just take some inverse image of j under the surjective reference function). Since r_i defines a function, c_i is clearly injective. If C_i is a subclass of C_{i-1}, then take $c_i = id$.

If $c' : ID \to I_D$ is the function defined by the uniqueness constraint on D and $c'' : ID \to ID$ is the concatenation $c_1 \circ \ldots \circ c_n$, then $c = c' \circ c''$ satisfies the required property. □

Definition 26 *A class C in a schema S is called* weakly value-representable *iff there exists a proper value type V_C such that for each instance $\mathcal{D}$ of S the following properties hold.*

1. *There is a function $c : ID \to V_C$ that is injective on $dom(\mathcal{D}(C))$.*
2. *For each proper value type V'_C and each function $c' : ID \to V'_C$ that is injective on $dom(\mathcal{D}(C))$ there exists a function $c'' : V_C \to V'_C$ that is unique on $c(dom(\mathcal{D}(C)))$ with $c' = c'' \circ c$.*

We call V_C the weak value-representation type *of the class C.*

Note that the weak value-representation type is unique provided it exists. Again it is easy to see that value-representability implies weak value-representability. Moreover, due to Lemma 25 each weakly value-representable class is also weakly value-identifiable. We shall see that also the converse of this fact is true.

We want to compute weak value representation types. This can be done using a slight modification of Algorithm 15 that completely ignores uniqueness constraints. We refer to this algorithm as the *blind* version of Algorithm 15 and to emphasize this, we write $H(C_i)$ instead of $F(C_i)$. Analogous to Lemmata 16 and 20 the following results holds.

Lemma 27 *Let C be a class in a schema S and let I_C be the type $H(C)$ computed by the blind version of Algorithm 15. Then I_C is a proper value type.*

Lemma 28 *Let D be an instance of the schema $S = \{C_1, \ldots, C_n\}$. Let C, D be classes such that C is weakly value-identifiable, D is value-identifiable and there exists some (SF)-chain from D to C. Let $c : ID \to I_C$ be the function of Lemma 25 corresponding to this chain. Let $c' : ID \to H(D)$ be a function corresponding to the uniqueness constraint on D and the instance $\mathcal{D}$. Then at each stage of the blind version of Algorithm 15 there exists a function $\bar{c} : H(D) \to I_C$ that is unique on $c'(dom_{\mathcal{D}}(C))$ with $c = \bar{c} \circ c'$.*

Based on these two lemmata we can now state the main result on weak value representability.

Theorem 29 *Let C be a weakly value-identifiable class in a schema S andlet V_C be the product of all types $H(D)$, where D is the leading value-identifiable class in some maximal (SF)-chain corresponding to C and $H(D)$ is the result of the blind version of Algorithm 15. Then C is weakly value-representable with weak value-representation type V_C.*

Proof. V_C is a proper value type by Lemma 27. From Lemmata 20 and 25 it follows that there exists a function $c' : ID \to V_C$ that is injective on $dom_{\mathcal{D}}(C)$.

From Lemma 28 it follows that there exists a function $\bar{c} : V_C \to I_C$ that is unique on $c'(dom(\mathcal{D}(C)))$ with $c = \bar{c} \circ c'$. This proves the Theorem. □

5 The Genericity Problem

The preservation of advantages of relational databases requires generic operations for querying and for the insertion, deletion and update of single objects. While querying [1,12,30,55] is per se a set-oriented operation, i.e. it is not necessary to select just one single object, and hence does not raise any specific problems with object identifiers, things change completely in case of updates. If an object with a given value is to be updated (or deleted), this is only defined unambigously, if there does not exist another object with the same value. If more than one object exists with the same value or more generally with the same value and the same references to other objects, then the user has to decide, whether an update- or delete-operation is applied to *all* these objects, to only *one* of these objects selected non-deterministically or to *none* of them, i.e. to reject the operation. However, it is not possible to specify a priori such an operation that works in the same way for all objects in all situations. The same applies to insert-operations. Hence the problem, in which cases operations for the insertion, deletion and update of objects can be defined generically.

Some authors [43] have chosen the solution to abandon generic operations. Others [6,7,9] use identifying values to represent object identity, thus embody a strict concept of surrogate keys to avoid the problem. Our approach is different from both solutions in that we use the concept of hidden abstract identifiers, but at the same time formally characterize those classes for which unique generic methods for the insertion, deletion and update of single objects exist. At the same time inclusion and referential integrity have to be enforced. We show that these classes are the value-representable ones.

5.1 Generic Update Methods

The requirement that object-identifiers have to be hidden from the user imposes the restriction on canonical update operations to be value-defined in the sense that the identifier of a new object has to be chosen by the system whereas all input- and output-data have to be values of proper value types.

We now formally define what we mean by generic update methods. For this purpose regard an instance $\mathcal{D}$ of a schema $\mathcal{S}$ as a set of objects. For each recursively defined type T let $\bar{T}$ denote by replacing each occurrence of a recursive type T' in T by $UNION(T', ID)$.

Definition 30 *Let C be a class in a schema $\mathcal{S}$.* Generic update methods *on C are $insert_C$, $delete_C$ and $update_C$ satisfying the following properties:*

1. *Their input types are proper value types; their output type is the trivial type $\perp$.*
2. *In the case of insert applied to an instance $\mathcal{D}$ there exists some $o :: U_C$ such that*
   (a) *the result is an instance $\mathcal{D}'$ with $o \in \mathcal{D}'$ and $\mathcal{D} \subseteq \mathcal{D}'$ hold and*
   (b) *if $\bar{\mathcal{D}}$ is any instance with $\mathcal{D} \subseteq \bar{\mathcal{D}}$ and $o \in \bar{\mathcal{D}}$, then $\mathcal{D}' \subseteq \bar{\mathcal{D}}$.*
3. *In the case of delete applied to an instance $\mathcal{D}$ there exists some $o :: U_C$ such that*
   (a) *the result is an instance $\mathcal{D}'$ with $o \notin \mathcal{D}'$ and $\mathcal{D}' \subseteq \mathcal{D}$ hold and*
   (b) *if $\bar{\mathcal{D}}$ is any instance with $\bar{\mathcal{D}} \subseteq \mathcal{D}$ and $o \notin \bar{\mathcal{D}}$, then $\bar{\mathcal{D}} \subseteq \mathcal{D}'$.*
4. *In the case of update applied to an instance $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$, where $\mathcal{D}_2 = \{o\}$ if $o \neq o'$ and $\mathcal{D}_2 = \emptyset$ otherwise there exist $o, o' :: U_C$ with $o = (i, v)$ and $o' = (i, v')$ such that*
   (a) *the result is an instance $\mathcal{D}' = \mathcal{D}_1 \cup \mathcal{D}_2'$ with $\mathcal{D}_2 \cap \mathcal{D}_2' = \emptyset$,*
   (b) *$o \in \mathcal{D}$, $o' \in \mathcal{D}'$,*
   (c) *if $\bar{\mathcal{D}}$ is any instance with $\mathcal{D}_1 \subseteq \bar{\mathcal{D}}$ and $o' \in \bar{\mathcal{D}}$, then $\mathcal{D}' \subseteq \bar{\mathcal{D}}$.*

Canonical update methods *on C are $insert'_C$, $delete'_C$ and $update'_C$ defined analogously with the only difference of their output type being ID and their input-type being $\bar{T}$ for some value-type T.*

Note that this definition of genericity includes the consistency with respect to the implicit constraints on S. We show that value-representability is necessary and sufficient for the existence and uniqueness of such operations.

Lemma 31 *Let C be a class in a schema S such that there exist canonical update methods on C. Then also generic update methods exist on C.*

Proof. In the case of *insert* define $insert_C(V :: V_C) == I \leftarrow insert'_C(V)$, i.e. call the corresponding canonical operation and ignore its output. The same argument applies to *delete* and *update*. □

Theorem 32 *Let C be a class in a schema S such that there exist generic update methods on C. Then C is value-representable. Moreover, all super- and subclasses of C are also value-representable.*

Proof. First consider the *delete* method with input type I_C which is by definition a proper value type. We show that it is already a value identification type.

If not, then for all instances $\mathcal{D}$ and all functions $c : T_C \rightarrow I_C$ there exist $i, j :: ID$ and $v, w :: T_C$ with

$$i \neq j \wedge (i, v) \in \mathcal{D}(C) \wedge (j, w) \in \mathcal{D}(C) \wedge c(v) = c(w) \ . \tag{12}$$

Now take $o = (i, v)$ and $o' = (j, w)$. Then there exist two distinct instances $\mathcal{D}'$ and $\mathcal{D}''$ satisfying the conditions of Definition 30(iii) with respect to o and o' respectively, hence contradict the assumption of a unique generic *delete*-method on C.

The same argument applies to the input-type V_C. Moreover, since insertion requires all values of referenced object to be provided, we derive from Algorithm 15 and Theorem 22 that V_C is a value representation type. Therefore, C is value-representable.

The value-representability on superclasses is implied, since *insert* (and *update*) on C involve the corresponding method on each superclass. The value-representability of subclasses follows from the propagation of *update* through them. We omit the technical details. □

5.2 Generic Updates in the Case of Value-Representability

Our next goal is to reduce the existence problem of canonical update operations to schemata without IsA relations.

Lemma 33 *Let C, D be value-representable classes in a schema S such that C is a subclass of D with subtype function $g : T_C \rightarrow T_D$. Then there exists a function $h : V_C \rightarrow V_D$ such that for each instance $\mathcal{D}$ of S with corresponding functions $c : T_C \rightarrow V_C$ and $d : T_D \rightarrow V_D$ we have $h(c(v)) = d(g(v))$ for all $v \in codom(\mathcal{D}(C))$.*

Proof. By Definition 10 c is injective on $codom(\mathcal{D}(C))$, hence any continuation h of $d \circ g \circ c^{-1}$ satisfies the required property.

It remains to show that h does not depend on $\mathcal{D}$. Suppose $\mathcal{D}_1$, $\mathcal{D}_2$ are two instances such that $w = c_1(v_1) = c_2(v_2) \in V_C$, where c_1, d_1, h_1 correspond to $\mathcal{D}_1$ and c_2, d_2, h_2 correspond to $\mathcal{D}_2$. Then there exists a permutation π on ID such that $v_2 = \pi(v_1)$. We may extend π to a permutation on any type. Since ID has no non-trivial supertype, g permutes with π, hence $g(v_2) = \pi(g(v_1))$. From Definition 10 it follows $d_2(g(v_2)) = d_1(g(v_1))$, i.e. $h_2(w) = h_1(w)$. □

In the following let S_0 be a schema derived from a schema S by omitting all IsA relations.

Lemma 34 *Let C be a value-representable class in S such that all its superclasses and subclasses $D_1 \dots D_n$ are also value-representable. Then canonical update operations exist on C in S iff they exist on C and all D_i in S_0.*

Proof. By Theorem 22 the value-representation type V_C is the result of Algorithm 15, hence V_C does not depend on the inclusion constraints of S. Then we have

$$I :: ID \leftarrow insert'_C(V :: V_C) ==$$
$$I \leftarrow insert'_{D_1}(h_1(V)); \dots; I \leftarrow insert'_{D_n}(h_n(V)); I \leftarrow insert^0_C(V)$$

where $h_i : V_C \to V_{D_i}$ is the function of Lemma 33 and $insert^0_C$ denotes a canonical insert on C in S_0. Hence in this case the result for the *insert* follows by structural induction on the IsA-hierarchy.

If the subtype function g required in Lemma 33 does not exist for some superclass D then simply add V_D to the input type. We omit the details for this case.

The arguments for *delete* and *update* are analogous. The value-representability of subclasses is required for the update case. □

From now on we use a global operation *NewId* that produces a fresh identifier $I :: ID$. This can be represented as a method using projection.

Lemma 35 *Let C be a value-representable class in S_0. Then there exist unique quasi-canonical update operations on C.*

Proof. Let $r_i : C_i$ $(i = 1 \dots n)$ denote the references in the structure expression of C. If V be a value of type V_C, then there exist values $V_{i,j} :: V_{C_i}$ $(i = 1 \dots n, j = 1 \dots k_i)$ occurring in V. Let $\bar{V} = \{V_{i,j}/J_{i,j} \mid i = 1 \dots n, j = 1 \dots k_i\}.V$ denote the value of type T_C that results from replacing each $V_{i,j}$ by some $J_{i,j} :: ID$. Moreover, for $I :: ID$ let

$$V^{(I)}_{i,j} = \begin{cases} \{V/I\}.V_{i,j} & \text{if } V \text{ occurs in } V_{i,j} \\ V_{i,j} & \text{else} \end{cases}$$

Then the canonical insert operation can be defined as follows:

$$I :: ID \leftarrow insert'_C(V :: \bar{V_C}) ==$$
$$\exists I' :: ID, V' :: T_C . (Pair(I', V') \in C \wedge c(V') = V) \to I := I'$$
$$\boxtimes \exists V' :: T_C . V = V' \to I \leftarrow NewId; x_C := x_C \cup \{(I, V)\}$$
$$\boxtimes I \leftarrow NewId; J_{1,1} \leftarrow insert'_{C_1}(V^{(I)}_{1,1}); \dots; J_{n,k_n} \leftarrow insert'_{C_n}(V^{(I)}_{n,k_n});$$
$$x_C := x_C \cup \{(I, \bar{V})\}$$

It remains to show that this operation is indeed canonical. Apply the method to some instance $\mathcal{D}$. If there already exists some $o = (I', V')$ in C with $c(V') = V$, the result is $\mathcal{D}' = \mathcal{D}$ and the requirements of Definition 30 are trivially satisfied. Otherwise let $o = (I, \bar{V})$. If $\bar{\mathcal{D}}$ is an instance with $\mathcal{D} \subseteq \bar{\mathcal{D}}$ and $o \in \bar{\mathcal{D}}$, we have $J_{i,j} \in dom(C_i)$ for all $i = 1 \dots n$, $j = 1 \dots k_i$, since $\bar{\mathcal{D}}$ satisfies the referential constraints. Hence $\bar{\mathcal{D}}$ contains the distinguished objects corresponding to the involved quasi-canonical operations $insert'_{C_i}$. By induction on the length of call-sequences $\mathcal{D}_{i,j} \subseteq \bar{\mathcal{D}}$ for all $i = 1 \dots n$, $j = 1 \dots k_i$, where $\mathcal{D}_{i,j}$ is the result of $J_{i,j} \leftarrow insert'_{C_i}(V^{(I)}_{i,j})$. Hence $\mathcal{D}' = \bigcup_{i,j} \mathcal{D}_{i,j} \cup \{o\} \subseteq \bar{\mathcal{D}}$. The uniqueness follows from the uniqueness of V_C.

The definitions and proofs for *delete* and *update* are analogous. □

Theorem 36 *Let C be a value-representable class in a schema S such that all its super- and subclasses are also value-representable. Then there exist unique generic update operations on C.*

Proof. By Lemma 31 and Lemma 34 it is sufficient to show the existence of canonical update operations on C and all its super- and subclasses in the schema S_0. This follows from Lemma 35. □

In [50] it has been shown, how linguistic reflection [56] can be exploited to generate the generic update operations for value-representable classes in an OODM schema.

6 The Consistency Problem

In general a database may be considered as a triplet (S, O, C), where S defines a structure, O denotes a collection of state changing operations and C is a set of constraints. Then the *consistency problem* is to guarantee that each specified operation $o \in O$ will never violate any constraint $I \in C$. *Integrity enforcement* aims at the derivation of a new set O' with $\mid O' \mid = \mid O \mid$ of operations such that (S, O', C) satisfies this property.

Suppose we are given a database schema S and a static integrity constraint I on that schema. Regard I as a logical formula defined on S. Consistency requires that only those instances D of S are allowed that satisfy I. Call the set of such instances $sat(S, I)$. Each transaction is a *database transformation*. Such a database transformation T takes an arbitrary instance D and possibly some input values $v_1, \ldots, v_n$ and produces a new instance D' and possibly some output values $v'_1, \ldots, v'_m$. T is *consistent* with respect to I iff for each $D \in sat(S, I)$ we also have $D' \in sat(S, I)$.

Classically consistency is maintained at run-time by transaction monitors. Whenever an inconsistent instance is produced the transaction that caused the inconsistency will be rolled back. This "everything or nothing" approach has been critized, since it causes enormous run-time overhead for consistency checking and rollback. Moreover, it leaves the burden of writing consistent transactions to the user. In principle the first problem vanishes, if verification techniques are used at design time [44,57,58], whereas the second one still remains.

As an alternative a lot of attention has been paid to integrity enforcement. In most cases the envisioned solution is an active database [18,27,59,64,65], where production rules are used to repair inconsistencies instead of rolling back. Although this is sometimes coupled with design time (or even run-time) analysis of the rules [18,27,33,63], the approach is not always successfull. Moreover, a satisfying theory for rule triggering systems with respect to the integrity enforcement problem is still missing. Therefore, we favour an operational approach [51,48,52,53], which aims at replacing inconsistent database transactions by consistent specializations.

6.1 Greatest Consistent Specializations

In general non-deterministic partial state transitions S as used in our method language can be described by a subset of $D \times D_\perp$, where D denotes the set of possible states and $D_\perp = D \cup \{\perp\}$, where $\perp$ is a special symbol used to indicate non-termination. It can be shown [20,41,46,44] that this is equivalent to defining two predicate transformers $wp(S)$ and $wlp(S)$ associated with S satisfying the *pairing condition* $wp(S)(\mathcal{R}) \Leftrightarrow wlp(S)(\mathcal{R}) \wedge wp(S)(true)$ and the *universal conjunctivity* of $wlp(S)$,i.e.

$$wlp(S)(\forall i \in I.\, \mathcal{R}_i) \;\Leftrightarrow\; \forall i \in I.\, wlp(S)(\mathcal{R}_i)\ .$$

The predicate transformers assign to some postcondition $\mathcal{R}$ the *weakest (liberal) precondition* of S to establish $\mathcal{R}$. Clearly, pre- and postconditions are X-constraints. Informally these conditions can be characterized as follows:

- $wlp(S)(\mathcal{R})$ characterizes those initial states such that all terminating executions of S will reach a final state characterized by $\mathcal{R}$ provided S is defined in that initial state, and
- $wp(S)(\mathcal{R})$ characterizes those initial states such that all executions of S terminate and will reach a final state characterized by $\mathcal{R}$ provided S is defined.

The use of these predicate transformers for the definition of language semantics is usually called "axiomatic semantics". Based on this consistency and specialization can be formally defined and used for the formal description of the consistency problem. For this purpose we define "extended operations" and therefore need to know for each operation S the set of classes $\mathcal{S}'$ such that S does neither read nor change the class variables x_C with $C \notin \mathcal{S}'$. In this case we call S a $\mathcal{S}'$-operation. We omit the formal definition [41,51].

Definition 37 *Let $\mathcal{S}$ be a schema, I a constraint and S, T methods defined on $S_1 \subseteq \mathcal{S}$ and $S_2 \subseteq \mathcal{S}$ respectively with $S_1 \subseteq S_2$.*

1. *S is* consistent *with respect to I iff $I \Rightarrow wlp(S)(I)$ holds.*
2. *T* specializes *S iff $wp(S)(true) \Rightarrow wp(T)(true)$ and $wlp(S)(\mathcal{R}) \Rightarrow wlp(T)(\mathcal{R})$ hold for all constraints $\mathcal{R}$ with free variables x_C such that $C \in S_1$ (denoted $T \sqsubseteq S$).*

Hence the following definition of a *greatest consistent specialization*:

Definition 38 *Let $\mathcal{S}$ be a schema, I a constraint and S a method defined on $S_1 \subseteq \mathcal{S}$. A method S_I is a* Greatest Consistent Specialization *(GCS) of S with respect to I iff*

1. *$S_I \sqsubseteq S$,*
2. *S_I is consistent with respect to I and*
3. *for each method T satisfying properties (i) and (ii) (instead of S_I) we have $T \sqsubseteq S_I$.*

If only properties (i) and (ii) are satisfied, we simply talk of a consistent specialization.

Let us first state the main results from [48].

Theorem 39 *Let $\mathcal{S}$ be a schema, I, J constraints and S a method defined on $S_1 \subseteq \mathcal{S}$.*

1. *There exists a greatest consistent specialization S_I of S with respect to I. Moreover, S_I is uniquely determined (up to semantic equivalence) by S and I.*
2. *The GCSs $(S_I)_J$ and $S_{(I \wedge J)}$ coincide on initial states satisfying $I \wedge J$.*

The proof of these results heavily uses predicate transformers and is therefore omitted here.

In [51] it has been shown that a GCS—that is in general non-deterministic—can be written as a finite choice of *maximal quasi-deterministic specializations* (MQCSs), where quasi-determinism means determinism up to the selection of some values. In most cases this value selection can be shifted to the input, but the selection of object identifiers should be left to the system.

Next, we formally define quasi-determinism and then present the main result from [51], an algorithm for the computation of MQCSs.

Definition 40 *A method S is called* quasi-deterministic *iff there exist types $T_1, \ldots, T_n$ such that S is semantically equivalent to*

$$y_1 :: T_1 \mid \ldots y_n :: T_n \mid S' \ ,$$

where S' is a deterministic method.

Algorithm 41 **In:** *An X-operation S and constraints $I_1, \ldots, I_n$ defined on extensions $Y_1, \ldots, Y_n$ of X.*

Let ℓ be the list of the constraints. As long as $\ell \neq nil$ proceed as follows:

1. *Set $S'_I = S$.*
2. *Choose and remove one constraint I_i from ℓ.*
3. *Check whether S'_I is I_i-reduced. If not, stop with no result, otherwise continue.*
4. *Make S'_I $\boxtimes$ -free by replacing each occurring $S_1 \boxtimes S_2$ by $S_1 \,\square\, wlp(S_1)(false) \rightarrow S_2$.*
5. *Replace each basic assignment in S'_I by some (subsumption-free) MQCS with respect to I_i.*
6. *Compute $P(S_I)$ as*

$$P(\bar{S}_I) \equiv \{z_1/x_1, \ldots, z_n/x_n\}.wlp(\{x_1/z_1, \ldots \\ \ldots, x_n/z_n\}.\bar{S}_I)(\neg wlp(S)(z_1 \neq x_1 \vee \ldots \vee z_n \neq x_n)) \ ,$$

   where the x_i are the class variables occurring in I or in S and the z_i are used as a disjoint copy of these.

7. *Set $S = P(S_I) \rightarrow S'_I$.*

Set $S'_I = S$.

Out: *An operation $I \rightarrow S'_I$, where S'_I is a (subsumption-free) MQCS of the original S with respect to the conjunction I of the constraints.*

□

An extension of the GCS algorithm to compute all (subsumption-free) MQCSs is easy.

It has been shown in [51] that Algorithm 41 is correct. However, it depends on checking a very technical condition, I-reducedness. We omit this condition here.

6.2 Enforcing Integrity in the OODM

Since Algorithm 41 allows integrity enforcement to be reduced to the case of assignments, we may restrict ourselves to the case of a single explicit constraint in addition to the trivial uniqueness constraints that are required to assure value-representability and that are used to construct *generic update operations*. In the following we describe MQCSs with respect to the constraints introduced in Definition 5.

6.2.1 Inclusion Constraints.

Let I be an inclusion constraint on C_1, C_2 defined via $c_i : T_{C_i} \to T$ $(i = 1, 2)$. Then each insertion into C_1 requires an additional insertion into C_2 whereas a deletion on C_2 requires a deletion on C_1. Update on one of the C_i requires an additional update on the other class.

Let us first concentrate on the insert-operation on C_1 (for an insert on C_2 there is nothing to do). Insertion into C_1 requires an input-value of type V_{C_1}; an additional insert on C_2 then requires an input-value of type V_{C_2}. However, these input-values are not independent, because the corresponding values of type T_{C_1} and T_{C_2} must satisfy the general inclusion constraint. Therefore we first show that the constraint can be "lifted" to a constraint on the value-representation types. Note that this is similar to the handling of IsA-constraints in Lemma 33.

Lemma 42 *Let C_1, C_2 be classes, $c_i : T_{C_i} \to T$ functions and let V_{C_i} be the value-representation type of C_i $(i = 1, 2)$. Then there exist functions $f_i : V_{C_i} \to T$ such that for all database instances $\mathcal{D}$*

$$f_1(d_1^{\mathcal{D}}(v_1)) = f_2(d_2^{\mathcal{D}}(v_2)) \Leftrightarrow c_1(v_1) = c_2(v_2) \tag{13}$$

for all $v_i \in codom(\mathcal{D}(x_{C_i}))$ $(i = 1, 2)$ holds. Here $d_i^{\mathcal{D}} : T_{C_i} \to V_{C_i}$ denotes the function used in the uniqueness constraint on C_i with respect to $\mathcal{D}$.

Proof. Due to Definition 10 we may define $f_i = c_i \circ (d_i^{\mathcal{D}})^{-1}$ on $c_i(codom(\mathcal{D}(x_{C_i})))$ $(i = 1, 2)$.

Then we have to show that this definition is independent of the instance $\mathcal{D}$. Suppose $\mathcal{D}_1$, $\mathcal{D}_2$ are two different instances. Then there exists a permutation π on ID such that $d_i^{\mathcal{D}_2} = d_i^{\mathcal{D}_1} \circ \pi$, where π is extended to T_{C_i}. Then

$$c_i \circ (d_i^{\mathcal{D}_2})^{-1} = c_i \circ \pi^{-1} \circ (d_i^{\mathcal{D}_1})^{-1} = \pi^{-1} \circ c_i \circ (d_i^{\mathcal{D}_1})^{-1} ,$$

since c_i permutes with π^{-1}. Then the stated equality follows. □

Now let $V_{C_1,C_2} = V_{C_1} \times V_{C_2}$ and define the new insert-operation on C_1 by $(insert_{C_1})_I((v_1, v_2) :: V_{C_1,C_2}) ==$

$$f_1(v_1) = f_2(v_2) \to insert_{C_1}(v_1) \; ; \; insert_{C_2}(v_2) \; , \tag{14}$$

where the f_i are the functions of Lemma 42. Note there there is no need to require $C_1 \neq C_2$. Delete- and update-operations can be defined analogously.

6.2.2 Functional and Uniqueness Constraints.

Now let I be a functional constraint on C defined via $c^1 : T_C \to T_1$ and $c^2 : T_C \to T_2$. In this case nothing is required for the delete operation whereas for inserts (and updates) we have to add a postcondition. Moreover, let $c^{\mathcal{D}} : T_C \to V_C$ denote the function associated with the value-representability of C and the database instance $\mathcal{D}$ and let all other notations be as before. Let us again concentrate on the insert-operation. Let $insert'_C$ denote the *canonical insert* on C. Then we define

$$\begin{aligned}
&(insert_C)_I(V :: V_C) == \\
&\quad I :: ID \mid I \leftarrow insert'_C(V)\,; \\
&\quad V' :: T_C \mid (I,V') \in x_C \to \\
&\quad (\forall J :: ID, W :: T_C.\,((J,W) \in x_C \\
&\quad \wedge c^1(W) = c^1(V') \Rightarrow c^2(W) = c^2(V'))) \to skip\ . \qquad (15)
\end{aligned}$$

Note that in this case there is no change of input-type. For delete- and update-operations we have analogous definitions.

A uniqueness constraint defined via $c^1 : T_C \to T_1$ is equivalent to a functional constraint defined via c^1 and $c^2 = id : T_C \to T_C$ plus the trivial uniqueness constraint. Since trivial uniqueness constraints are already enforced by the canonical update operations, there is no need to handle separately arbitrary uniqueness constraints.

6.2.3 Exclusion Constraints.

The handling of exclusion constraints is analogous to the handling of inclusion constraints. This means that an insert (update) on one class may cause a delete on the other, whereas delete-operations remain unchanged.

We concentrate again on the insert-operation. Let I be an exclusion constraint on C_1 and C_2 defined via $c_i : T_{C_i} \to T$ $(i = 1,2)$. Let $f_i : V_{C_i} \to T$ denote the functions from Lemma 42. Then we define a new insert-operation on C_1 by

$$\begin{aligned}
&(insert_{C_1})_I(V :: V_{C_1}) == \\
&\quad insert_{C_1}(V)\,; \\
&\quad \mu S.\,((I :: ID \mid V' :: T_{C_2} \mid (I,V') \in x_{C_2} \\
&\quad \wedge c^2(V') = f_1(V) \to delete_{C_2}(V')\,;\, S) \boxtimes skip)\ . \qquad (16)
\end{aligned}$$

For delete- and update-operations an analogous result holds.

Theorem 43 *The methods S_I in (14), (15) and (16) are MQCSs of generic insert-methods with respect to inclusion, functional and exclusion constraints respectively.*

The proof involves detailed use of predicate transformers and is therefore omitted here [48,49]. Analogous results hold for *delete* and *update*.

7 Conclusion

In this paper we describe first results concerning the formal foundations of object oriented database concepts. For this purpose we introduced a formal object oriented datamodel (OODM) with the following characteristics.

- Objects are considered to be abstractions of real world entities, hence they have an immutable identity. This identity is encoded by abstract identifiers that are assumed to form some type ID. This identifier concept eases the modelling of shared data and cyclic references, however, it does not relieve us from the problem to provide unique identification mechanisms for objects in a database.

- In our approach there is not only one value of a given type that is associated with an object. In contrast we allow several values of possibly different types to belong to an object, and even this collection of types may change.

- Classes are used to structure objects. At each time a class corresponds to a collection of objects with values of the same type and references to objects in a fixed set of classes. Inheritance is based on IsA relations that express an inclusion at each time of the sets of objects. Moreover, referential integrity is supported.

- We associate with each class a collection of methods. Methods are specified by guarded commands, hence the method language is computationally complete. In order to allow the handling of identifiers that are always hidden from the user as well as user-accessible transactions a hiding operator on methods is introduced. Generic update operations, i.e. insert, delete and update on a class are assumed to be automatically derived whenever this is possible.

- We associate integrity constraints to schemata. Certain kinds of such constraints can be obtained by generalizing corresponding constraints in the relational model. We assume that methods are automatically changed in order to enforce integrity.

On this basis of this formal OODM we study the problems of identification, genericity and integrity. We show that the unique identification of objects in a class requires the class to be value-representable.

An advantage of database systems is to provide generic update operations. We show that the unique existence of such generic methods requires also value-representability. However, in this case referential and inclusion integrity can be enforced automatically. This result can be generalized with respect to distinguished classes of user-defined integrity constraints. Given some arbitrary method S and some constraint I there exists a *greatest consistent specialization* (GCS) S_I of S with respect to I. Such a GCS behaves nice in that it is compatible with the conjunction of constraints. For the GCS construction of a user-defined transaction we apply the GCS algorithm developped in [48,51,52,53].

This work on mathematical foundations of OODB concepts is not yet completed. A lot of problems are still left open and are the matter of current investigations and future research.

- In our approach classes are sets. What are other bulk types? Does it make sense to abstract from classes in this way?

- The problem of updatable views is still open.

- Our approach to genericity only handles the worst case expressed by the value representation type. We assume that polymorphism will help to generalize our results to the general case. Moreover, we must integrate communication aspects at least with respect to the user.

- The usual axiomatic semantics for guarded commands abstracts from an execution model. All results are true for semantic equivalence classes. However, we also need optimization, especially with respect to the derived GCSs.
- We only presented a formal OODM without looking into methodological aspects such as the characterization of good designs.

We express the hope that others will also contribute to solve open problems in OODB foundation or in the implementation of more sophisticated object oriented database languages on a sound mathematical basis.

Acknowledgement

We would like to thank Catriel Beeri, Joachim W. Schmidt, and Ingrid Wetzel for many stimulating discussions especially concerning object identification. We also want to thank David Stemple and Kasimierz Subieta for questioning the theme from an engineering point of view.

References

[1] S. Abiteboul: *Towards a deductive object-oriented database language*, Data & Knowledge Engineering, vol. 5, 1990, pp. 263 – 287

[2] S. Abiteboul, R. Hull: *IFO: A Formal Semantic Database Model*, ACM ToDS, vol. 12 (4), December 1987, pp. 525 – 565

[3] S. Abiteboul, P. Kanellakis: *Object Identity as a Query Language Primitive*, in Proc. SIGMOD, Portland Oregon, 1989, pp. 159 – 173

[4] A. Albano, G. Ghelli, R. Orsini: *Types for Databases: The Galileo Experience*, in Type Systems and Database Programming Languages, University of St. Andrews, Dept. of Mathematical and Computational Sciences, Research Report CS/90/3, 27 – 37

[5] A. Albano, A. Dearle, G. Ghelli, C. Marlin, R. Morrison, R. Orsini, D. Stemple: *A Framework for Comparing Type Systems for Database Programming Languages*, in Type Systems and Database Programming Languages, University of St. Andrews, Dept. of Mathematical and Computational Sciences, Research Report CS/90/3, 1990

[6] A. Albano, G. Ghelli, R. Orsini: *Objects and Classes for a Database Programming Language*, FIDE technical report 91/16, 1991

[7] A. Albano, G. Ghelli, R. Orsini: *A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language*, in A. Sernadas (Ed.): *Proc. VLDB 91*, Barcelona 1991

[8] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik: *The Object-Oriented Database System Manifesto*, Proc. 1st DOOD, Kyoto 1989

[9] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard, F. Velez: *The Design and Implementation of O_2, an Object-Oriented Database System*, Proc. of the ooDBS II workshop, Bad Münster, FRG, September 1988

[10] C. Beeri: *Formal Models for Object-Oriented Databases*, Proc. 1st DOOD 1989, pp. 370 – 395

[11] C. Beeri: *A formal approach to object-oriented databases*, Data and Knowledge Engineering, vol. 5 (4), 1990, pp. 353 – 382

[12] C. Beeri, Y. Kornatzky: *Algebraic Optimization of Object-Oriented Query Languages*, in S. Abiteboul, P. C. Kanellakis (Eds.): Proc. ICDT '90, Springer LNCS 470, pp. 72 – 88

[13] C. Beeri: *New Data Models and Languages - the Challange* in Proc. PODS '92

[14] L. Cardelli, P. Wegner: *On Understanding Types, Data Abstraction and Polymorphism*, ACM Computing Suerveys 17,4, pp 471 – 522

[15] L. Cardelli: *Typeful Programming*, Digital Systems Research Center Reports 45, DEC SRC Palo Alto, May 1989

[16] M. Carey, D. DeWitt, S. Vandenberg: *A Data Model and Query Language for EXODUS*, Proc. ACM SIGMOD 88

[17] M. Caruso, E. Sciore: *The VISION Object-Oriented Database Management System*, Proc. of the Workshop on Database Programming Languages, Roscoff, France, September 1987

[18] S. Ceri, J. Widom: *Deriving Production Rules for Constraint Maintenance*, Proc. 16th Conf. on VLDB, Brisbane (Australia), August 1990, pp. 566 – 577

[19] A. Dearle, R. Connor, F. Brown, R. Morrison: *Napier88 - A Database Programming Language?*, in Type Systems and Database Programming Languages, University of St. Andrews, Dept. of Mathematical and Computational Sciences, Research Report CS/90/3, 10 – 26

[20] E. W. Dijkstra, C. S. Scholten: *Predicate Calculus and Program Semantics*, Springer-Verlag, 1989

[21] H.-D. Ehrich, M. Gogolla, U. Lipeck: *Algebraische Spezifikation abstrakter Datentypen*, Teubner-Verlag, 1989

[22] H.-D. Ehrich, A. Sernadas: *Fundamental Object Concepts and Constructors*, in G. Saake, A. Sernadas (Eds.): Information Systems – Correctness and Reusability, TU Braunschweig, Informatik Berichte 91-03, 1991

[23] H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specification*, vol.1, Springer 1985

[24] L. Fegaras, T. Sheard, D. Stemple: *The ADABTPL Type System*, in Type Systems and Database Programming Languages, University of St. Andrews, Dept. of Mathematical and Computational Sciences, Research Report CS/90/3, 45 – 56

[25] L. Fegaras, T. Sheard, D. Stemple: *Uniform Traversal Combinators: Definition, Use and Properties*, University of Massachusetts, 1992

[26] D. Fishman, D. Beech, H. Cate, E. Chow et al.: *IRIS: An Object-Oriented Database Management System*, ACM ToIS, vol. 5(1), January 1987

[27] P. Fraternali, S. Paraboschi, L. Tanca: *Automatic Rule Generation for Constraint Enforcement in Active Databases*, in U. Lipeck (Ed.): Proc. 4th Int. Workshop on Foundations of Models and Languages for Data and Objects "MODELLING DATABASE DYNAMICS", Volkse (Germany), October 19-22, 1992

[28] G. Gottlob, G. Kappel, M. Schrefl: *Semantics of Object-Oriented Data Models – The Evolving Algebra Approach*, in J. W. Schmidt, A. A. Stognij (Eds.): Proc. Next Generation Information Systems Technology, Springer LNCS, vol. 504, 1991

[29] M. Hammer, D. McLeod: *Database Description with SDM: A Semantic Database Model*, J. ACM, vol. 31 (3), 1984, pp. 351 – 386

[30] A. Heuer, P. Sander: *Classifying Object-Oriented Results in a Class/Type Lattice*, in B. Thalheim et al. (Ed.): *Proceedings MFDBS 91*, Springer LNCS 495, pp. 14 – 28

[31] R. Hull, R. King: *Semantic Database Modeling: Survey, Applications and Research Issues*, ACM Computing Surveys, vol. 19(3), September 1987

[32] R. Hull, M. Yoshikawa: *ILOG: Declarative Creation and Manipulation of Object Identifiers*, in Proc. 16th VLDB, Brisbane (Australia), 1990, pp. 455 – 467

[33] A. P. Karadimce, S. D. Urban, *Diagnosing Anomalous Rule Behaviour in Databases with Integrity Maintenance Production Rules*, in Proc. 3rd Int. Workshop on Foundations of Models and Languages for Data and Objects, Aigen (Austria), September 1991, pp. 77 – 102

[34] S. Khoshafian, G. Copeland: *Object Identity*, Proc. 1st Int. Conf. on OOPSLA, Portland, Oregon, 1986

[35] M. Kifer, J. Wu: *A Logic for Object-Oriented Logic Programming (Maier's O-Logic Revisited)*, in PODS'89, pp. 379 – 393

[36] W. Kim, N. Ballou, J. Banerjee, H. T. Chou, J. Garza, D. Woelk: *Integrating an Object-Oriented Programming System with a Database System*, in Proc. OOPSLA 1988

[37] D. Maier, J. Stein, A. Ottis, A. Purdy: *Development of an Object-Oriented DBMS*, OOPSLA, September 1986

[38] F. Matthes, J. W. Schmidt: Bulk Types – Add-On or Built-In?, in Proc. DBPL III, Nafplion 1991

[39] J. Mylopoulos, P. A. Bernstein, H. K. T. Wong: *A Language Facility for Designing Interactive Database-Intensive Applications*, ACM ToDS, vol. 5 (2), April 1980, pp. 185 – 207

[40] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis: *Telos: Representing Knowledge About Information Systems*, ACM ToIS, vol. 8 (4), October 1990 pp. 325 – 362

[41] G. Nelson: *A Generalization of Dijkstra's Calculus*, ACM TOPLAS, vol. 11 (4), October 1989, pp. 517 – 561

[42] A. Ohori: *Representing Object Identity in a Pure Functional Language*, Proc. ICDT 90, Springer LNCS, pp. 41 – 55

[43] G. Saake, R. Jungclaus: *Specification of Database Applications in the TROLL Language*, in Proc. Int. Workshop on the Specification of Database Systems, Glasgow, 1991

[44] K.-D. Schewe, I. Wetzel, J. W. Schmidt: *Towards a Structured Specification Language for Database Applications*, in D. Harper, M. Norrie (Eds.): Proc. Int. Workshop on the Specification of Database Systems, Springer WICS, 1991, pp. 255 – 274 (an extended version appeared as FIDE technical report 1991/30, October 1991)

[45] K.-D. Schewe, B. Thalheim, I. Wetzel, J. W. Schmidt: *Extensible Safe Object-Oriented Design of Database Applications*, University of Rostock, Preprint CS-09-91, September 1991

[46] K.-D. Schewe: *Spezifikation datenintensiver Anwendungssysteme* (in German), lecture manuscript, University of Hamburg, Winter 1991/92

[47] K.-D. Schewe, J. W. Schmidt, I. Wetzel: *Identification, Genericity and Consistency in Object-Oriented Databases*, in J. Biskup, R. Hull (Eds.): Proc. ICDT '92, Springer LNCS 646, pp. 341-356

[48] K.-D. Schewe, B. Thalheim, J. W. Schmidt, I. Wetzel: *Integrity Enforcement in Object-Oriented Databases*, in U. Lipeck, B. Thalheim (Eds.): Proc. 4th Int. Workshop on Foundations of Models and Languages for Data and Objects "MODELLING DATABASE DYNAMICS", Volkse (Germany), October 19-22, 1992

[49] K.-D. Schewe, B. Thalheim, I. Wetzel: *Foundations of Object Oriented Database Concepts*, University of Hamburg, Report FBI-HH-B-157/92, October 1992

[50] K.-D. Schewe, J. W. Schmidt, D. Stemple, B. Thalheim, I. Wetzel: *A Reflective Approach to Method Generation in Object Oriented Databases*, University of Rostock, Rostocker Informatik Berichte, no. 14, 1992

[51] K.-D. Schewe, B. Thalheim: *Computing Consistent Transactions*, University of Rostock, Preprint CS-08-92, December 1992

[52] K.-D. Schewe, B. Thalheim, I. Wetzel: *Integrity Preserving Updates in Object Oriented Databases*, in M. Orlowska, M. Papazoglou (Eds.) : Proc. 4th Australian Database Conference, Brisbane, February 1993, World Scientific, pp. 171-185

[53] K.-D. Schewe, B. Thalheim: *Exceeding the Limits of Rule Triggering Systems to Achieve Consistent Transactions*, submitted for publication

[54] M. H. Scholl, H.-J. Schek: *A Relational Object Model*, in Proc. ICDT 90, Springer LNCS, pp. 89 – 105

[55] G. M. Shaw, S. B. Zdonik: *An Object-Oriented Query-Algebra*, IEEE Data Engineering, vol. 12 (3), 1989, pp. 29 – 36

[56] D. Stemple, T. Sheard, L. Fegaras: *Reflection: A Bridge from Programming to Database Languages*, in Proc. HICSS '92

[57] D. Stemple, S. Mazumdar, T. Sheard: *On the Modes and Meaning of Feedback to Transaction Designer*, in Proc. SIGMOD 1987, pp. 375 – 386

[58] D. Stemple, T. Sheard: *Automatic Verification of Database Transaction Safety*, ACM ToDS vol. 14 (3), September 1989

[59] M. Stonebraker, A. Juingran, J. Goh, S. Potaminos: *On Rules, Procedures, Caching and Views in Database Systems*, in Proc. SIDMOD 1990, pp. 281 – 290

[60] S. Y. W. Su: *SAM*: A Semantic Association Model for Corporate and Scientific-Statistical Databases*, Inf. Sci., vol. 29, 1983, pp. 151 – 199

[61] B. Thalheim: *Dependencies in Relational Databases*, Teubner Leipzig, 1991

[62] B. Thalheim: *The Higher-Order Entity-Relationship Model*, in J. W. Schmidt, A. A. Stognij (Eds.): Proc. Next Generation Information Systems Technology, Springer LNCS, vol. 504, 1991

[63] S. D. Urban, L. Delcambre: *Constraint Analysis: a Design Process for Specifying Operations on Objects*, IEEE Trans. on Knowledge and Data Engineering, vol. 2 (4), December 1990

[64] J. Widom, S. J. Finkelstein: *Set-oriented Production Rules in Relational Database Systems*, in Proc. SIGMOD 1990, pp. 259 – 270

[65] Y. Zhou, M. Hsu: *A Theory for Rule Triggering Systems*, in Proc. EDBT '90, Springer LNCS 416, pp. 407 – 421

(Received April 7, 1993)