Identification, Genericity and Consistency in Object-Oriented Databases*

Klaus–Dieter Schewe, Joachim W. Schmidt, Ingrid Wetzel

University of Hamburg, Dept. of Computer Science,
Vogt-Kölln-Str. 30, D-W-2000 Hamburg 54

Abstract. It is claimed that object-oriented databases overcome many of the limitations of the relational data model especially by generalizing the notion of object identification. A clear distinction between objects and values turns out to be essential for the object-oriented approach whereas the relational model is based exclusively on values. Since, however, value uniqueness within scopes is a quite natural constraint for a wide class of applications, identification by value is also of interest for object-oriented datamodels.
Hence, in this paper we concentrate on those classes where the extents are completely representable by values. We formalize some basic concepts of object-oriented databases and show that the finiteness of a database and the existence of finitely representable recursive types are sufficient to decide *value-representability*.
Another advantage of the relational approach is the existence of structurally determined *canonical update operations*. We show that this property can be carried over to object-oriented datamodels iff classes are value-representable. Moreover, in this case *database consistency* with respect to implicitly specified referential and inclusion constraints will be automatically preserved.

1 Introduction

The success of the relational data model is due certainly to the existence of simple query and update-languages whereas its modelling power is often criticized for its limitations [1, 4, 7, 8]. Novel data modelling constructs are required by advanced database applications and it is claimed that object-oriented databases will satisfy a much wider range of demands [3, 4, 5, 9, 10, 11, 13, 14]. However, in contrast to the relational model where a commonly accepted theory existed very early on, there is not yet a generally accepted object-oriented data model.

Object-oriented databases are based on a clear distinction between *values* and *objects* [6]. 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, 2, 12]. The identifier of an object is immutable during the object's lifetime; the object's value is in general assumed to be mutable. Identifiers can be used to model relationships naturally by references between objects. This facilitates sharing and cyclicity.

* This work has been supported in part by research grants from the E.E.C. Basic Research Action 3070 FIDE: "Formally Integrated Data Environments".

In contrast to the relational model where identification is modelled by keys, object identifiers are system-provided and hence do not have a meaning for the user. Consequently, they have to be hidden from the user.

Values can be classified into types and objects can be grouped into classes. Roughly speaking, a type defines a fixed set of values and a class defines a finite collection of objects with the same structure. The content of a class varies over time. The dynamics of an object-oriented database is defined by methods associated with objects (classes).

Our intent here is not to describe a concrete object-oriented database language. Instead 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 *functional constraints* on classes, in particular *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 following problems:

- How to characterize those classes that are completely representable (and hence also identifiable) by values?
- How to characterize those classes for which canonical update operations can be derived?
- How to define canonical update operations that enforce integrity with respect to inclusion and to functional and referential constraints as mentioned above?

Our approach to the first and the second of these problems have been reported in [15] in a more informal way. In this paper we show that the finiteness of a database and the existence of finitely representable recursive types are sufficient to decide value-representability. Moreover, in the case of value-representability, we can derive canonical update operations that enforce consistency with respect to all implicit referential and inclusion constraints.

The remainder of the paper is organized as follows. In Section 2 we first introduce concepts of OODBs informally. For their formalization we introduce in Section 3 an algebraic framework for type specifications similar to [8]. The possibility of defining parameterized types enables us to formally define classes with implicit referential and inclusion constraints plus additional user-defined functional constraints. This will be done in Section 4.

In Section 5 we define formally the notions of value-representability and value-definability of a class and analyse the referential structure of a given schema. We derive sufficient conditions for value-representability.

Section 6 discusses the problem of canonical update operations. It consists of the results on the definability of such operations and their relation to consistency. We conclude with a short summary and outlook.

2 The Structure of Object-Oriented Databases

In our approach to OODBs 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 $meth_k$. We assume all identifiers id to belong to unique given set ID. Types represent immutable sets of values. They can be defined algebraically similar to [8]. Type constructors can be defined analogously by parameterized types. These can be used to build complex types by nesting and recursive types. We assume that the set ID of possible object identifiers is also a type. Then an instantiation of a parameterized type defines a structure that represents a combination of values and references, where references are expressed by the occurrence of a value of type ID.

Objects can be grouped into classes with some structure built from values and references. Furthermore, we may associate methods and constraints with each class. This means of structure building involves implicit *referential constraints*. Inheritance on classes is given by IsA-relations, i.e. by set inclusion on object identifiers. Moreover we introduce subtyping and formalize this by the definition of a continuous function from a subtype to a supertype. The relation between subtyping and inheritance is given by an *inclusion constraint* on classes.

Example 1. Let us look at a university application, where the objects are persons or departments. Let $(\cdot)$, $\{\cdot\}$ and $[\cdot]$ denote type constructors for (tagged) tuples, finite sets and lists respectively. Let PERSON, DEPARTMENT, PROFESSOR and STUDENT be types such that STUDENT and PROFESSOR are subtypes of PERSON. The type specifications will be given elsewhere. Then we need the following four class definitions (omitting additional constraints):

PersonC = **structure** PERSON **end**
ProfessorC = **IsA** PersonC
  structure (prof : PROFESSOR , fac : DepartmentC) **end**
DepartmentC = **structure** (fac : DEPARTMENT , head : ProfessorC) **end**
StudentC = **IsA** PersonC
  structure (stud : STUDENT , major : DepartmentC , minor : DepartmentC ,
    supervisors : { (fac : DepartmentC , prof : [sup : ProfessorC]) }) **end**

Note that we used names for the references. In this example an object o in the class ProfessorC is given by $(i,(v,j))$, where $i :: ID$ is the oid of o, v is a value of type PROFESSOR and $j :: ID$ is the oid of an object o' in the class DepartmentC. Moreover, o also belongs to the class PersonC and is represented there by (i,w), where the subtype function f : PROFESSOR $\rightarrow$ PERSON satisfies $f(v) = w$. □

3 Type Specifications

Our approach to types is an algebraic one similar to [8]. In general a type is specified by a collection of constructors, selectors and other functions – the signature of the type – and axioms defined by universal Horn formulae. Now let N_P, N_T, N_C, N_R, N_F, N_M and V denote arbitrary pairwise disjoint, countably infinite sets representing a reservoir of parameter-, type-, class-, reference-, function-, method- and variable-names respectively.

Definition 1. A *type signature* Σ consists of a type name $t \in N_T$, a finite set of supertype-/function-pairs $T \subseteq N_T \times N_F$, a finite set of parameters $P \subseteq N_P$, a

finite set of base types $B \subseteq N_T$ and pairwise disjoint finite sets C, S, $F \subseteq N_F$ of constructors, selectors and functions such that there exist predefined arities $ar(c) \in (P \cup B^* \cup \{t\})^* \times \{t\}$, $ar(s) \in \{t\} \times (P \cup B^* \cup \{t\})$ and $ar(f) \in (P \cup B^* \cup \{t\})^* \times (P \cup B^* \cup \{t\})$ for each $c \in C$, $s \in S$ and $f \in F$.

We write $f : t \rightarrow t'$ to denote a *supertype-/function-pair* $(t', f) \in T$. We write $c : t_1 \times \ldots \times t_n \rightarrow t$ to denote a *constructor* of arity $(t_1 \ldots t_n, t)$, $s : t \rightarrow t'$ to denote a *selector* of arity (t, t') and $f : t_1 \times \ldots \times t_n \rightarrow t'$ to denote a *function* of arity $(t_1 \ldots t_n, t')$. If $t_i = b_i^0 \ldots b_i^m \in B^*$, we write $b_i^0(b_i^1 \ldots b_i^m)$. We call $\mathcal{S} = P \cup B \cup \{t\}$ the set of *sorts* of the signature Σ.

Definition 2. A *type declaration* consists of a type signature Σ with type name t such that there exists a type declaration for each $b \in B - \{t\}$ and a set Ax of Horn formulae over Σ. Moreover, if $b_i^0(b_i^1 \ldots b_i^m)$ with $b_i^j \in B$ occurs within a constructor, selector or function, then b_i^0 must have been declared as a parameterized type with m parameters. We say that (Σ, Ax) defines the *parameterized type* $t(\alpha_1, \ldots, \alpha_n)$, iff $P = \{\alpha_1, \ldots, \alpha_n\} \neq \emptyset$ or the *proper type* t respectively.
A *type* t is defined either by a type declaration or by mutually recursive equations involving t as a variable.

The semantics of a type is given by term generated algebras that are quotients of the term algebra defined by the constructors. Functions are considered to define additional structure via operations and functions on these algebras. Subtyping is modelled by the use of a continous function taking the subtype to the supertype. Recursive types are fixpoints of functors.

Example 2. Assume the types NAT, PERSONNAME and ADDRESS to be defined elsewhere. Then a proper type PERSON as used in Example 1 can be defined as follows:

```
PERSON ==
   BasedOn NAT ; PERSONNAME ; ADDRESS ;
   Constructors
      Person : NAT × PERSONNAME × ADDRESS → PERSON ;
   Selectors
      PersonIdentityNo : PERSON → NAT ;
      Name : PERSON → PERSONNAME ;
      Address : PERSON → ADDRESS ;
   Axioms
      With P :: PERSON :
            Person(PersonIdentityNo(P),Name(P),Address(P)) = P ;
      With N :: NAT , PN :: PERSONNAME , A :: ADDRESS :
            PersonIdentityNo(Person(N,PN,A)) = N;
      With N :: NAT , PN :: PERSONNAME , A :: ADDRESS :
            Name(Person(N,PN,A)) = PN;
      With N :: NAT , PN :: PERSONNAME , A :: ADDRESS :
            Address(Person(N,PN,A)) = A;
End PERSON                                                    □
```

In [16] we presented the parameterized type $FSETS(\alpha)$ denoting finite sets with elements of type α. On this basis, it is easy to define $PFUN(\alpha, \beta)$, the type of partial functions (with finite domain) from α to β.

So far we use only single-sorted algebras, but the extension to many-sorted and order-sorted algebras is straightforward. Moreover, each value of a given type can be expressed by a closed constructor term. In the case of recursive types this term is a *rational tree*.

4 The Concept of a Class in Object-Oriented Databases

Our approach to OODBs distinguishes between values grouped into types and objects grouped into classes. 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, but methods will be postponed to Section 6.

Throughout the rest of the paper we assume a proper identifier type ID to be defined such that only the trivial type $\bot$ is a supertype of ID. A type T without any occurrence of ID will be called a *value type*.

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. (i) Let t be a value type with parameters $\alpha_1, \ldots, \alpha_n$. Let $r_1, \ldots, r_n \in N_R$ be distinct reference names and $C_1, \ldots, C_n \in N_C$ be class names. 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*.

(ii) A *class* consists of a class name $C \in N_C$, a structure expression S, a list of superclasses $D_1, \ldots, D_m$ and a list of constraints $\mathcal{I}_1, \ldots, \mathcal{I}_k$. 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.

In general a class may be considered as a variable C of type $PFUN(ID, T_C)$. However, the introduction of classes is associated inevitably with the introduction of constraints. In object-oriented databases we want to be able to model inheritance between classes, which leads to some kind of *inclusion constraint*. Modelling relationships between classes via references leads to the introduction of *referential constraints*. Moreover, generalizing the well known key constraints from the relational theory will give us *uniqueness constraints* on classes.

Definition 4. Let C, C' be classes of representation types T_C and $T_{C'}$ respectively. Let i, j be of type ID and v, w be of type T_C.

(i) A *referential constraint* on C and C' has the form

$$Pair(i, v) \in C \ \wedge \ o(v, j) = true \ \Rightarrow \ j \in dom(C') \, ,$$

where $o : T_C \times ID \rightarrow BOOL$ is a function.

(ii) An *inclusion constraint* on C and C' has the form

$$Pair(i, v) \in C \ \Rightarrow \ Pair(i, f(v)) \in C' \, ,$$

where $f : T_C \rightarrow T_{C'}$ is a subtype function provided such an f exists. Otherwise it is simply

$$i \in dom(C) \Rightarrow i \in dom(C')$$

(iii) A *uniqueness constraint* on C has the form

$$Pair(i, v) \in C \wedge Pair(j, w) \in C \wedge f(v) = f(w) \ \Rightarrow \ i = j \, ,$$

where $f : T_C \rightarrow T$ is a subtype function.

Each reference in the structure expression S of a class C gives rise to an implicit referential constraint. Each superclass D of a class C defines an implicit inclusion constraint. Moreover, arbitrary constraints may be added as explicit constraints, but for the purpose of this paper we concentrate on uniqueness constraints.

Example 3. Let $f : PAIR(PERSON, ID) \rightarrow PAIR(NAT, ID)$ be a subtype function defined by the axiom

`With` P :: PERSON , I :: ID : f(Pair(P,I)) = Pair(PersonIdentityNo(P),I) .

Then a class named MarriedPersonC may be defined as follows:

```
MarriedPersonC == IsA PersonC
   Structure PAIR( PERSON , spouse : MarriedPersonC)
   Constraints With I,J :: ID , V,W :: PAIR(PERSON,ID) :
      Pair(I,V) ∈ MarriedPersonC ∧ Pair(J,W) ∈ MarriedPersonC ∧
        f(V) = f(W) ⇒ I = J
End MarriedPersonC                                              □
```

Definition 5. (i) A *schema* $\mathcal{S}$ is a finite collection of classes $C_1, \ldots, C_n$ closed under references and superclasses.

(ii) An *instance* $\mathcal{D}$ of a schema $\mathcal{S}$ assigns to each class C a value $\mathcal{D}(C)$ of type $PFUN(ID, T_C)$ such that all implicit and explicit constraints on $\mathcal{S}$ are satisfied.

5 Object Identification and Value-Representation

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.

The goal of this section is 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 6 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 6. Let C be a class in a schema $\mathcal{S}$ with representation type T_C.

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

(ii) C is called *value-representable* iff there exists a proper value type V_C such that for all instances $\mathcal{D}$ of $\mathcal{S}$ 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 6 is unique up to isomorphism.

Theorem 7. *Let C be a class in a schema $\mathcal{S}$. Then C is value-representable iff C is value-identifiable and C_i is value-representable for all references $r_i : C_i$ in the structure expression S.*

Proof. This follows directly from the definitions. □

5.1 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 8. The *reference graph* of a class C in a schema $\mathcal{S}$ is the smallest labelled graph $G_{rep} = (V, E, l)$ satisfying:

(i) 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.

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

(iii) 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}.

(iv) For each vertex v_t or v_C corresponding to $t(x_1, \ldots, x_n)$ in S there exist unique edges $e^{(i)}_t$ 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 $l(e^{(i)}_t) = \{S_i\}$, where S_i is the corresponding selector name; in the latter case the label is $\{S_i, r_i\}$.

Definition 9. Let $\mathcal{S} = \{C_1, \ldots, C_n\}$ be a schema. Let $\mathcal{S}' = \{C'_1, \ldots, C'_n\}$ be another schema such that for all i either $T'_{C_i} = T_{C_i}$ holds or 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.

Example 4. Let MARRIEDPERSONC be defined as in Example 3. Then the reference graph and the identification graph with respect to the uniqueness constraint of this class are shown in Figure 1. □

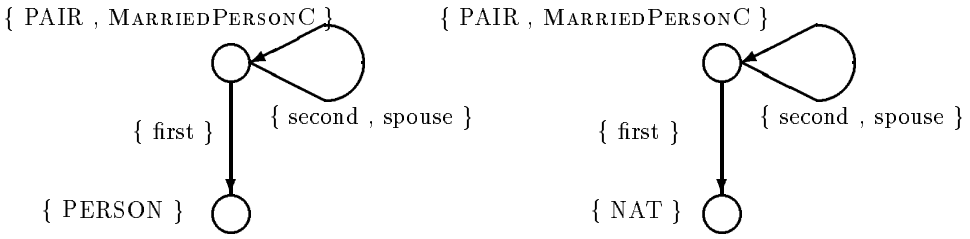

Fig. 1. The reference graph and identification graph of class MARRIEDPERSONC

Theorem 10. *Let C be a class in a schema $\mathcal{S}$ 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 6.

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 6. □

Theorem 11. *Let C be a class in a schema $\mathcal{S}$ 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.*

Proof. The proof is analogous to that of Theorem 10 □

Theorem 12. *Let C be a class with acyclic reference graph in a schema $\mathcal{S}$. Then the value-representability of C is decidable.*

Proof. So far the only explicit constraints in our model are uniqueness constraints. According to Definition 4 equality of identifiers occurs only as a positive literal in such constraints. Therefore, it is impossible to derive a uniqueness constraint for a class C that has not one a priori. Theorem 10 implies that value-representability can be decided by checking the existence of uniqueness constraints in the class definitions. □

Theorem 13. *Let C be a class in a schema $\mathcal{S}$ such that there exist an acyclic identification graph. Then the value-identifiability of C is decidable.*

Proof. The proof is analogous to that of Theorem 12. □

5.2 Computation of Value Representation Types

We want to address the more general case where cyclic references may occur in the schema $\mathcal{S} = \{C_1, \ldots, C_n\}$. In this case a simple induction argument as in the proof of Theorem 10 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 6.

Algorithm 14. Let $G(C_i) = T_{C_i}$ provided there exists a uniqueness constraint on C_i, otherwise let $G(C_i)$ be undefined. If ID occurs in some $G(C_i)$ corresponding to $r_j : C_j$ $(j \neq i)$, we write ID_j.

Then iterate as long as possible using the following rules:

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

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

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 $G(C_i)$ by the type name $G(C_j)$ provided $G(C_j)$ is defined. □

Note that the the algorithm computes (mutually) recursive types. Now we give a sufficient condition for the result of Algorithm 14 to be a proper value type.

Lemma 15. *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 14. Then V_C is a proper value type.*

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

Algorithm 16. Let $F(C_i) = T_i$ provided there exists a uniqueness constraint on C_i defined by $c_i : T_{C_i} \rightarrow T_i$, 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:

(i) 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 $F(C_j)$.

(ii) 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. □

Lemma 17. *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 some identification graph G_{id} of C. Let I_C be the type $F(C)$ computed by Algorithm 16 with respect to the uniqueness constraints used in the definition of G_{id}. Then I_C is a proper value type.*

Proof. The proof is analogous to that of Lemma 15. □

5.3 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 implies the decidability of value-representability provided the type system allows recursive types to be defined in such a way that all their values are finitely representable, i.e. representable as rational trees. Note that the type specifications introduced in Section 3 satisfy this property.

Definition 18. Let C be a class in a schema $\mathcal{S}$ 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 $o_{k,l} : T_{C_k} \times ID \rightarrow BOOL$ 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 $cyc : ID \times ID \rightarrow BOOL$ corresponding to a cycle in G_{ref}.

Definition 19. Let C be a class in a schema $\mathcal{S}$ and let $g_{0,1} \cdots g_{n-1,n}$ be a cycle in G_{ref}. The corresponding *cycle relation* $cyc : ID \times ID \rightarrow BOOL$ 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 20. *Let C be a class in a schema $\mathcal{S}$. Then C satisfies the finiteness property, i.e. for each instance $\mathcal{D}$ of $\mathcal{S}$ and for each cycle in G_{ref} the corresponding cycle relation cyc satisfies*

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

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

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

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 21. *Let $\mathcal{D}$ be an instance of schema $\mathcal{S} = \{C_1, \ldots, C_n\}$. Then $\mathcal{D}$ satisfies at each stage of Algorithm 14 uniqueness constraints for all $i = 1, \ldots, n$ defined by some $c_i : T_{C_i} \to G(C_i)$.*

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

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

Lemma 22. *Let $\mathcal{D}$ be an instance of schema $\mathcal{S} = \{C_1, \ldots, C_n\}$. Then $\mathcal{D}$ satisfies at each stage of Algorithm 16 uniqueness constraints for all $i = 1, \ldots, n$ defined by some $c_i' : T_{C_i} \to F(C_i)$.*

Proof. The proof is analogous to the proof of Lemma 21. □

Lemma 23. *Let $\mathcal{D}$ be an instance of schema $\mathcal{S} = \{C_1, \ldots, C_n\}$. Then at each stage of the algorithms 14 and 16 for all $i = 1, \ldots, n$ there exists 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 21 it is sufficient to show that the required property is preserved by the application of a rule from Algorithm 14 or 16. 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 21 and Lemma 22 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 24. *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 14. Then C is value-representable with value representation type V_C.*

Proof. V_C is a proper value type by Lemma 15. From Lemma 21 it follows that if $\mathcal{D}$ is an instance of $\mathcal{S}$, then there exists a function $c : T_C \rightarrow V_C$ such that the uniqueness constraint defined by c holds for $\mathcal{D}$.

If V'_C is another proper value type and $\mathcal{D}$ satisfies a uniqueness constraint defined by $c' : T_C \rightarrow V'_C$, then V'_C is some value-identification type I_C. Hence by Lemma 23 there exists a function $c'' : V_C \rightarrow V'_C$ that is unique on $c(codom_{\mathcal{D}}(C))$ with $c' = c'' \circ c$. This proves the lemma. □

Theorem 25. *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 some identification graph G_{id} of C. Let I_C be the type $F(C)$ computed by Algorithm 16 with respect to the uniqueness constraints used in the definition of G_{id}. Then C is value-identifiable with value identification type I_C.*

Proof. The proof is analogous to that of Theorem 24. □

Theorem 26. *Let C be a class in a schema $\mathcal{S}$. Then the value-representability and the value-identifiability of C are decidable.*

Proof. The proof is analogous to that of Theorem 12. □

6 Existence and Consistency of Generic Update Operations

Methods are used to specify the dynamics of an object-oriented database. Here, we do not want to give a concrete language for methods. In general methods can be specified in the style of Dijkstra focussing on deterministic operations [16].

In this paper we are only interested in *canonical update operations*, i.e. we want to associate with each class C in a schema $\mathcal{S}$ *methods* for *insertion*, *deletion* and *update* on single objects. These operations should be consistent with respect to the constraints in $\mathcal{S}$. Thus, they are sufficient to express the creation, deletion and change of objects including the migration between classes. However, we would like to regard these operations as being "generic" in the sense of polymorphic functions, since insert, delete and update should be defined for each class. The problem is that the input-type and the body of these operations require information from the schema. This leads to polymorphism with respect to meta-types. For the purpose of this paper we do not discuss this problem.

6.1 Canonical Update Operations

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 canonical update operations. 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 27. Let C be a class in a schema $\mathcal{S}$. *Canonical update operations* on C are $insert_C$, $delete_C$ and $update_C$ satisfying the following properties:

(i) Their input types are proper value types; their output type is the trivial type $\perp$.
(ii) In the case of *insert* applied to an instance $\mathcal{D}$ there exists a distinguished object $o :: PAIR(ID, T_C)$ 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 $\mathcal{D}' \subseteq \bar{\mathcal{D}}$.
(iii) In the case of *delete* applied to an instance $\mathcal{D}$ there exists a distinguished object $o :: PAIR(ID, T_C)$ 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 $\bar{\mathcal{D}} \subseteq \mathcal{D}'$.
(iv) In the case of *update* applied to an instance $\mathcal{D} = \mathcal{D}_1 \dot{\cup} \mathcal{D}_2$, where $\mathcal{D}_2 = \{o\}$ if $o \neq o'$ and $\mathcal{D}_2 = \emptyset$ otherwise there exist distinguished objects $o, o' :: PAIR(ID, T_C)$ with $o = Pair(i, v)$ and $o' = Pair(i, v')$ such that
  (a) the result is an instance $\mathcal{D}' = \mathcal{D}_1 \dot{\cup} \mathcal{D}'_2$ 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 $\mathcal{D}' \subseteq \bar{\mathcal{D}}$.

Quasi-canonical update operations 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 canonical update operations includes the consistency with respect to the implicit and explicit constraints on $\mathcal{S}$. We show that value-representability is sufficient for the existence and uniqueness of such operations. We use a guarded command notation as in [16] for these update operations.

Lemma 28. *Let C be a class in a schema $\mathcal{S}$ such that there exist quasi-canonical update operations on C. Then also canonical update operations 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 quasi-canonical operation and ignore its output. The same argument applies to *delete* and *update*. □

6.2 Existence of Canonical Updates in the Case of Value-Representability

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

Lemma 29. *Let C, D be value-representable classes in a schema $\mathcal{S}$ such that C is a subclass of D with subtype function $g : T_C \to T_D$. Then there exists a function $h : V_C \to V_D$ such that for each instance $\mathcal{D}$ of $\mathcal{S}$ with corresponding functions $c : T_C \to V_C$ and $d : T_D \to V_D$ we have $h(c(v)) = d(g(v))$ for all $v \in codom_{\mathcal{D}}(C)$.*

Proof. By Definition 6 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 6 it follows $d_2(g(v_2)) = d_1(g(v_1))$, i.e. $h_2(w) = h_1(w)$. □

In the following let $\mathcal{S}_0$ be a schema derived from a schema $\mathcal{S}$ by omitting all IsA relations.

Lemma 30. *Let C be a value-representable class in $\mathcal{S}$ such that all its superclasses $D_1 \ldots D_n$ are also value-representable. Then quasi-canonical update operations exist on C in $\mathcal{S}$ iff they exist on C and all D_i in $\mathcal{S}_0$.*

Proof. By Theorem 24 the value-representation type V_C is the result of Algorithm 14, hence V_C does not depend on the inclusion constraints of $\mathcal{S}$. Then we have

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

where $h_i : V_C \to V_{D_i}$ is the function of Lemma 29 and $insert^0_C$ denotes a quasi-canonical insert on C in $\mathcal{S}_0$. 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 29 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. □

Now assume the existence of a global operation $NewId$ that produces a fresh identifier $I :: ID$.

Lemma 31. *Let C be a value-representable class in $\mathcal{S}_0$. Then there exist unique quasi-canonical update operations on C.*

Proof. Let $r_i : C_i$ $(i = 1 \ldots n)$ denote the references in the structure expression of C. If V be a value of type $\bar{V_C}$, then there exist values $V_{i,j} :: \bar{V_{C_i}}$ $(i = 1 \ldots n, j = 1 \ldots k_i)$ occurring in V. Let $\bar{V} = \{V_{i,j}/J_{i,j} \mid i = 1 \ldots n, j = 1 \ldots k_i\}.V$ 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 quasi-canonical insert operation can be defined as follows:

$$
\begin{array}{l}
I :: ID \leftarrow insert'_C(V :: \bar{V_C}) \; == \\
IF \; \exists \, I' :: ID \, , \, V' :: T_C . \, (Pair(I', V') \in C \; \wedge \; c(V') = V) \\
THEN \; I := I' \\
ELSE \; I \leftarrow NewId \, ; \; J_{1,1} \leftarrow insert'_{C_1}(V^{(I)}_{1,1}) \, ; \; \ldots \, ; \; J_{n,k_n} \leftarrow insert'_{C_n}(V^{(I)}_{n,k_n}) \, ; \\
\qquad\qquad C := C \cup \{Pair(I, \bar{V})\} \\
FI
\end{array}
$$

It remains to show that this operation is indeed quasi-canonical. Apply the operation to some instance $\mathcal{D}$. If there already exists some object $o = Pair(I', V')$ in C with $c(V') = V$, the result is $\mathcal{D}' = \mathcal{D}$ and the requirements of Definition 27 are trivially satisfied. Otherwise let the distinguished object be $o = Pair(I, \bar{V})$. 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 $i = 1 \ldots n$, $j = 1 \ldots k_i$, 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 $i = 1 \ldots n$, $j = 1 \ldots k_i$, where $\mathcal{D}_{i,j}$ is the result of $J_{i,j} \leftarrow insert'_{C_i}(V^{(I)}_{i,j})$. Hence $\mathcal{D}' = \underline{\cup}_{i,j} \mathcal{D}_{i,j} \cup \{o\} \subseteq \bar{\mathcal{D}}$. The uniqueness follows from the uniqueness of V_C.

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

Theorem 32. *Let C be a value-representable class in a schema $\mathcal{S}$ such that all its superclasses are also value-representable. Then there exist unique canonical update operations on C.*

Proof. By Lemma 28 and Lemma 30 it is sufficient to show the existence of quasi-canonical update operations on C and all its superclasses in the schema $\mathcal{S}_0$. This follows from Lemma 31. □

7 Conclusion

In this paper we introduce a structural object-oriented datamodel with a clear distinction between types and classes. Types are algebraically specified such that each value of a type is finitely representable. This can be exploited to define constraints on a schema in a uniform way. Moreover, parameterized type declarations and a unique object identifier type can be used to represent classes by a partial function type.

An advantage of the uniform type and constraint declarations is the possibility to study alternatives for the representation of objects. Our main interest are objects that could also be represented without using abstract object identifiers. We call them value-representable. Due to the finiteness of a database value-representability is decidable if uniqueness constraints are the only explicit constraints. Moreover, we show that value-representability is sufficient for the existence of unique structurally determined canonical update operations that enforce implicit referential and inclusion constraints.

This constitutes the basis of a behavioural object-oriented datamodel. Therefore, the next steps will be to extend the outlined approach to a complete formal object-oriented datamodel including a complete query language, views and transactions.

References

1. S. Abiteboul: *Towards a deductive object-oriented database language*, Data & Knowledge Engineering, vol. 5, 1990, pp. 263 – 287
2. S. Abiteboul, P. Kanellakis: *Object Identity as a Query Language Primitive*, in Proc. SIGMOD, Portland Oregon, 1989, pp. 159 – 173
3. 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
4. M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, S. Zdonik: *The Object-Oriented Database System Manifesto*, Proc. 1st DOOD, Kyoto 1989
5. 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
6. C. Beeri: *Formal Models for Object-Oriented Databases*, Proc. 1st DOOD 1989, pp. 370 – 395
7. C. Beeri: *A formal approach to object-oriented databases*, Data and Knowledge Engineering, vol. 5 (4), 1990, pp. 353 – 382
8. C. Beeri, Y. Kornatzky: *Algebraic Optimization of Object-Oriented Query Languages*, in S. Abiteboul, P. C. Kanellakis (Eds.): Proceedings of ICDT 90, Springer LNCS 470, pp. 72 – 88
9. M. Carey, D. DeWitt, S. Vandenberg: *A Data Model and Query Language for EXODUS*, Proc. ACM SIGMOD 88
10. M. Caruso, E. Sciore: *The VISION Object-Oriented Database Management System*, Proc. of the Workshop on Database Programming Languages, Roscoff, France, September 1987
11. D. Fishman, D. Beech, H. Cate, E. Chow et al.: *IRIS: An Object-Oriented Database Management System*, ACM ToIS, vol. 5(1), January 1987
12. S. Khoshafian, G. Copeland: *Object Identity*, Proc. 1st Int. Conf. on OOPSLA, Portland, Oregon, 1986
13. 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
14. D. Maier, J. Stein, A. Ottis, A. Purdy: *Development of an Object-Oriented DBMS*, OOPSLA, September 1986
15. K.-D. Schewe, B. Thalheim, I. Wetzel, J. W. Schmidt: *Extensible Safe Object-Oriented Design of Database Applications*, University of Rostock, Technical report, September 1991
16. K.-D. Schewe, I. Wetzel, J. W. Schmidt: *Towards a Structured Specification Language for Database Applications*, in Proc. Int. Workshop on the Specification of Database Systems, Glasgow, Springer WICS 1991

This article was processed using the LaTeX macro package with LLNCS style