Foundations of Object Oriented Database Concepts*

Klaus–Dieter Schewe[1],
Bernhard Thalheim[2],
Ingrid Wetzel[1]

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

[2] University of Rostock, Dept. of Computer Science,
Albert-Einstein-Str. 21, D-O-2500 Rostock, FRG

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

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 outstanding.

Our research in Hamburg and Rostock is directed towards a formally founded object oriented datamodel (OODM) and to contribute to the development of a uniform mathematical theory of OODBs. This report contains the results of our first investigations on the OODM.

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*. The finiteness of a database and the existence of finitely representable *rational tree 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.

This result can be generalized with respect to distinguished classes of explicitly stated static constraints. We show that integrity enforcement is always possible. Given some arbitrary method S and some static or transition constraint $\mathcal{I}$ there exists a *greatest consistent specialization* (GCS) $S_{\mathcal{I}}$ of S with respect to $\mathcal{I}$. Such a GCS behaves nice in that it is compatible with the conjunction of constraints, inheritance and refinement. For the GCS construction of a user-defined operation, however, it is in general not sufficient to replace the involved primitive update operations by their GCSs.

From an engineering point of view an algorithm is required to generate these consistent operations. We address this construction problem by the specification of generators for them. These generators will be based on the possibility to represent syntactic components of the language as values within the language itself, which is known to form the basis of *linguistic reflection*. Moreover, the generators involve a single generic proof of correctness hence relieve the user of the burden to write basic update operations and to assure their consistency.

Contents

Chapter 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 [9]. Several systems [5, 7, 8, 10, 16, 17, 18, 19, 26, 35, 38, 40] arose from these efforts. However, in contrast to research in the relational area there is no common formal agreement on what constitutes an object-oriented database [11, 12, 14].

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, 41, 42, 61]. 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 [9]. This leads to a distinction between *values* and *objects* [11, 12]. 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, 33]. 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. 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.

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 static and dynamic constraints.

1.1 The Identification Problem

One important concept of object-oriented databases is *object identity*. Following [1, 13] 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 [44]. On the other hand, system provided identifiers can not be used for object identification, since this would lead to a database of identifiers [36]. Hence the user can only access objects in the database via values. Object identifiers do not have a meaning for the user and should be hidden from the user.

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 problem how to characterize those classes that are completely representable (and hence also identifiable) by values. We show that the finiteness of a database and the existence of finitely representable recursive types are sufficient to decide *value-representability*.

1.2 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, 13]. 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 [45] have chosen the solution to abandon generic operations. Others [7, 8, 10] 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.

It can be shown that generators for these generic update operations can be specified for any value-representable class of a schema. The generators can be used to increase the productivity of system developers while enhancing the quality of the implemented systems.

1.3 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; allmost all commercial systems provide this. 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. A database system should be designed to automatically take methods specified by the user or generated by the system upon checking integrity constraints upon occurrence of certain database operations.

The *maintenance problem* is the problem how to ensure 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 [63], whereas in OODBs the integrity problem has only recently drawn attention [27, 51].

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 contrast to the relational model the fundamental concepts of *object identity* [33] and *inheritance* imply inevitably the existence of *inclusion* and *referential constraints* that are specified with each OODB schema. Integrity with respect to these constraints must be preserved by all database updates, especially by basic *insert-*, *delete-* and *update*-operations. As usual in OODBs such operations are modeled by *methods*. Due to the main result in [50] such operations are only uniquely determined by the schema in the case of *value-representable* classes. We shall outline these methods and draw specific attention to the type of the required input-value.

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.

1.4 The Organization of the Paper

In Chapter 2 we introduce the OODM. We start first motivating the concepts and give a short review on some theoretical aspects underlying the model. Then we describe in detail the structural and the behavioural parts of the OODM.

Chapter 3 handles the identification problem. We introduce the notion of value-representability and show results on the decidability.

The genericity problem will be approached in Chapter 4. We show the relationship between value-representability and the unique existence of generic update operations. Finally we describe an algorithmic approach to generate such operations using linguistic reflection.

The consistency problem is dealt with in Chapter 5. We outline an operational approach based on greatest consistent specializations and describe them for the case of distinguished classes of static integrity constraints.

We summarize our results and describe some open problems in Chapter 6.

Acknowledgement

We would like to thank Catriel Beeri for stimulating discussions concerning object identification. We also want to thank David Stemple who contributed to the engineering aspects concerning genericity in the OODM. He convinced us on the benefits of linguistic reflection. Thanks also to Kasimierz Subieta for questioning the theme from a programming point of view.

Chapter 2

An Object Oriented Datamodel

In this chapter we present the formal object oriented datamodel (OODM) of [49, 50, 51]. 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, 5, 7, 8, 9, 10, 17, 18, 26, 35, 38, 44, 45, 56] 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.

Classes are used to abstract from individual objects providing mutable collections of a given structure and behaviour, where the latter is modelled via methods.

We start with a motivation, where we explain the general notion of an object and illustrate by using examples the notions of *type*, *class* and *method*. Throughout this section a preliminary informal syntax will be used.

Then we give a brief outline of some theoretical background underlying the OODM. This comprises an algebraic framework for type and method specifications that stems from the specification language SAMT [47, 54] together will formally defined consistency notions.

The heart of the chapter is formed by Sections 2.3 and 2.4, where we introduce formally the structural and the behavioural part of the OODM.

2.1 Motivation

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.

Whereas values are encoded by themselves [11, 12], objects have to be encoded by object identifiers regardless of the content, location or addressability [33]. 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 [13, 21, 23]. 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.

2.1.1 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. Thus, a design consists of *type* and *class definitions*.

Type Definitions.

Assume the existence of *basic types* $STRING$, NAT, INT and $BOOL$ with the usual operations on them. Additionally, there exist a basic type ID that will be used to model object identifiers and a trivial type $\perp$ with only one element and no operations.

In this paper we only use fixed type constructors, the *tagged tuple* constructor denoted by $(\cdot)$, the *finite set* constructor denoted by $\{\cdot\}$ and the *union* constructor denoted $\cup$. We use concatenation on tuple types denoted by the $\circ$-operator.

Types and type constructors in general are defined by nesting of these constructors. These types can be organized in a *subtype* hierarchy with the subtype relation defined as usual. $\perp$ is a supertype of every type.

As a last structuring feature recursive type definitions are allowed. They only occur as input-types for generated update methods, where *rational tree* values are required [4].

Example 1 The set and the tuple type constructors are both used in the declaration for PERSONNAME:

$$\begin{array}{lll} PERSONNAME & = & (\ \text{FirstName} : STRING, \\ & & \ \ \text{SecondName} : STRING, \\ & & \ \ \text{Titles} : \{STRING\}) \end{array}$$

The definition for a type PERSON uses the type PERSONNAME and a type ADDRESS defined elsewhere:

$$PERSON = (\ \text{PersonIdentityNo} : NAT,\ \text{Name} : PERSONNAME,\ \text{Address} : ADDRESS\ )$$

The following example defines $STUDENT$ as a subtype of $PERSON$:

$$STUDENT = PERSON \circ (\ \text{StudNo} : NAT, \text{Faculty} : NAT\ ) \qquad \square$$

Class Definitions.

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 type constructors.

- Let t be a type constructor with parameters $\alpha_1, \ldots, \alpha_n$ such that ID does not occur in t. For distinct reference names $r_1, \ldots, r_n$ and class names $C_1, \ldots, C_n$ 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*.
- A *class* consists of a class name C, a structure expression S, a set of class names $D_1, \ldots, D_m$ (in the following called the set of *superclasses*) and a set of methods. 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.
- A *schema* $\mathcal{S}$ is a finite collection of classes $C_1, \ldots, C_n$ closed under references and superclasses together with a collection of constraints $\mathcal{I}_1, \ldots, \mathcal{I}_n$.

Example 2 Let the types $PERSON$ and $STUDENT$ be as in Example 1. In addition assume a type $PROFESSOR$ defined as elsewhere as another subtype of $PERSON$. Then the following is a simple schema for a university application. For the moment methods are omitted.

```
Schema University
   Class PersonC
      Structure PERSON
      Methods ...
   Class MarriedPersonC
      IsA PersonC
      Structure PERSON∘ (Spouse : MarriedPersonC)
   Class StudentC
      IsA PersonC
      Structure STUDENT∘ (Supervisor : ProfessorC)
   Class ProfessorC
      IsA PersonC
      Structure PROFESSOR
```

`Constraint` Unique(PERSONC,$PERSON$)

`Constraint` EXCL(STUDENTC,PROFESSORC) □

The notation Unique(C,T) means that values of type T do not occur twice in the class C. EXCL(C_1,C_2) states that the classes C_1 and C_2 are disjoint.

At each time, C is given by a finite set of objects. More precisely, C is a set of pairs (i, v), where i is of type ID and v is of type T_C such that identifiers are unique in this set. This defines an *instance* $\mathcal{D}$ of a schema $\mathcal{S}$.

Moreover, C gives rise to *referential constraints* defined by the structure S and *IsA constraints* defined by the set of superclasses of C, i.e.

- whenever an identifier $j :: ID$ occurs in v and this occurrence corresponds to the reference $r_k : C_k$, then there must exist an object in C_k with identifier j (*referential integrity*), and
- for each superclass D_k there exist an object with identifier i (and some value $w :: T_{D_k}$ in D_k) (*inclusion integrity*). Moreover, if T_C is a subtype of T_{D_k}, there v must correspond to w.

Note, that we do not require classes to be disjoint nor that IsA relations require a subtype relation on the corresponding representation types.

2.1.2 Methods as a Basis for Behaviour Modeling

Methods in general can be described operationally with the usual control constructs. Assignments are only allowed on the class C or on a selective expression on C. Therefore, we dispense with introducing a specific method language here.

Let us now concentrate on basic update methods, i.e. insertion, deletion and update of a single object on a classes C. In contrast to the relational datamodel such update operations can not always be derived in the object-oriented case, because the abstract identifiers have to be hidden from the user. However, in [50] it has been shown that for *value-representable* classes these operations are uniquely determined by the schema and consistent with respect to the implicit referential and inclusion constraints.

Value-representability of all classes in a schema is implied, if we define a trivial *uniqueness constraint* for each class. Such a constraint requires the values of type T_C in the class extension C to be unique, which is similar to a (trivial) key definition in the relational case.

Example 3 Let us describe the insert-method for the class PERSONC of Example 2.

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 the insert on 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. □

2.2 Theoretical Background

For the moment let us abstract from the specific database context. Look at a database being defined as some *state space* X with typed state variables $x_1 :: T_1, \ldots, x_n :: T_n$. Then *state transitions* on X are expressible by (partial, non-deterministic) guarded commands with a sophisticated axiomatic semantics defined by predicate transformers. Formulae in first-order logic can be used to express both static and transition integrity *constraints* on X. We adopt this approach in order to formalize the OODM as well as the integrity enforcement problem within a strict mathematical framework that will later be applied to the OODM.

2.2.1 Types

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. This is related to algebraic specifications [21, 23]. Now let N_P, N_T, N_F, and V denote arbitrary pairwise disjoint, countably infinite sets representing a reservoir of parameter-, type-, function-, 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

$f : t_1 \times \ldots \times t_n \to t'$ 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 $b \in B \perp \{t\}$ 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. Subtyping is modelled by the use of a continous function taking the subtype to the supertype. Recursive types are fixpoints of functors. See [54] for a completely mathematical treatment of types. It can be shown that even the guarded commands give rise to a type $GC(\alpha, \beta, \gamma)$, where α (β) is the type of the input (output) and γ is the type of the underlying state space. See [47] for more details.

2.2.2 State Transitions

In general non-deterministic partial state transitions S on a state space X can be described by a subset of $\mathcal{D} \times \mathcal{D}_\perp$, where $\mathcal{D}$ denotes the set of possible states on X and $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$, where $\perp$ is a special symbol used to indicate non-termination. It can be shown 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 $wlp(S)$. They assign to some postcondition $\mathcal{R}$ the *weakest (liberal) precondition* of S to establish $\mathcal{R}$. 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.

Such operations S can be specified by guarded commands in the style of Dijkstra [20, 46, 48, 43]:

Definition 3 Let X be some state space. A *guarded command* S on X consists of a name S, a set of input-parameters $\{\iota_1, \ldots, \iota_k\}$, a set of output-parameters $\{o_1, \ldots, o_l\}$ and a body. To each input-parameter ι_i corresponds a type T_i and to each output-parameter o_j corresponds a type O_j. The *body* of S is recursively built from the following constructs:

(i) assignment $x := E$, where x is a state variable in X or a local variable within S and E is a term of the same type as x,

(ii) $skip$, $fail$, $loop$,

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

(iv) instantiation $x'_1, \ldots, x'_i \leftarrow S'(E'_1, \ldots, E'_j)$, where S' is the name of another operation ($S = S'$ is possible) on X 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.

Each variable occurring in the S has a well-defined scope. The scoping rules are omitted. Furthermore, we omit the detailed definition of the predicate transformers $wlp(S)$ and $wp(S)$ [43, 47]. We only give an informal description of the less usual operations *projection*, *guard* and *restricted choice*. Projection gives the introduction of a new local variable x of the given type. A guard $\mathcal{P} \rightarrow S$ gives a precondition $\mathcal{P}$ for S. If $\mathcal{P}$ is not satisfied, the whole operation is undefined. Restricted choice $S \boxtimes T$ means to execute S unless it is undefined in which case T is taken. The basic commands *skip*, $fail$, *loop* are only introduced for theoretical completeness: *skip* does nothing, $fail$ is always undefined, and *loop* never terminates.

Here we dispense with any structuring of state spaces into modules. However, in order to define "extended operations" we need to know for each operation S the subspace $Y \subseteq X$ such that S does neither read nor change the values in $X \perp Y$. In this case we call S a Y-operation on X. We omit the formal details [43, 47].

2.2.3 Consistency Proof Obligations

General constraints and arbitrary operations on a state space X raise the problem whether consistency as defined by the constraints is always satisfied by the operations. One approach to address this problem is to use general verification techniques. The verification approach consists in the derivation (and proof) of general proof obligations expressed in the predicate transformer calculus.

The static constraints on a state space X, i.e. first-order formulae $\mathcal{I}$ with $fr(\mathcal{I}) \subseteq X$ partition the state space, i.e. the collection of the mutable classes into two distinguished subspaces. States not satisfying the constraints should never be reached.

In general inherited operations can be overwritten. Unless inheritance is simply regarded as a copying mechanism we should ensure that this can be done in a concise way, i.e., overriding should be restricted to "specialization". The intuition behind this definition is that whenever an execution of the specialized operation T establishes some post-predicate $\mathcal{R}$, then this execution should already be one of the general method S.

Transition constraints on X are expressible as first-order formulae $\mathcal{J}$ with $fr(\mathcal{J}) \subseteq X \dot{\cup} X'$, where X' is a disjoint copy of X. We may then exploit a weak equivalence between guarded commands and predicative specifications. We may associate with the transition constraint $\mathcal{J}$ the guarded command $\Phi(\mathcal{J}) = x' \mid \mathcal{J} \rightarrow x := x'$ where x (x') is used as an abbreviation for the collection $x_1, \ldots, x_n$ ($x'_1, \ldots, x'_n$) of (state) variables. Satisfying $\mathcal{J}$ is equivalent to each operation S specializing $\Phi(\mathcal{J})$. Hence the following formal definitions:

Definition 4 Let X be a state space, $Z \subseteq Y \subseteq X$ subspaces, $\mathcal{I}$ a static and $\mathcal{J}$ a transition constraint on X, S a Z-operation and T a Y-operation.

(i) S is *consistent* with respect to $\mathcal{I}$ iff $\mathcal{I} \Rightarrow wlp(S)(\mathcal{I})$ holds on X.

(ii) T *specializes* S iff $wp(S)(true) \Rightarrow wp(T)(true)$ and $wlp(S)(\mathcal{R}) \Rightarrow wlp(T)(\mathcal{R})$ hold for all Z-predicates $\mathcal{R}$ (denoted $T \sqsubseteq S$).

(iii) S is *consistent* with respect to $\mathcal{J}$ iff $\mathcal{R} \quad wlp(\Phi(\mathcal{J}))(\mathcal{R}) \Rightarrow wlp(S)(\mathcal{R})$ holds for all X-predicates.

Note that $\sqsubseteq$ defines a partial order on operations. There exists an equivalent characterization of transition consistency that avoids the quantification over all state predicates [46]. See [47] for more details.

2.3 The Structural Approach

In the following assume to be given a type system $\mathcal{T}$ as described e.g. in Section 2.2.1. Basically such a type system consists of some *basic types* such as $BOOL$, $NATURAL$, $INTEGER$, $STRING$, etc., *type constructors* (parameterized types) for record, finite sets, lists, etc. and a *subtyping* relation. Moreover, assume that (mutually) *recursive types*, i.e. types defined by (a system of) domain equations, exist in $\mathcal{T}$. As an alternative to our definition of $\mathcal{T}$ in Section 2.2.1 we may restrict $\mathcal{T}$ being one of the type systems defined in [5, 6, 15, 16, 19, 24, 39, 40]. 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*.

Now 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.

2.3.1 The Concept of a Class

The OODM in [50] 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 defined by deterministic guarded commands [43, 46, 47].

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 5 (i) 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*.

(ii) A *class* consists of a class name $C \in N_C$, a structure expression S, a set of class names $D_1, \ldots, D_m \in N_C$ (in the following called the set of *superclasses*) and a set of static 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.

(iii) A (structural) *schema* $\mathcal{S}$ is a finite collection of classes $C_1, \ldots, C_n$ closed under references and superclasses together with a collection of static constraints $\mathcal{I}_1, \ldots, \mathcal{I}_n$.

(iv) An *instance* $\mathcal{D}$ of a structural 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.

Here we dispense with giving a concrete syntax for constraints. Distinguished classes of static constraints will be introduced in Section 2.3.3.

2.3.2 The Representation of a Schema

We now associate with each schema $\mathcal{S}$ a state space X such that each class C in $\mathcal{S}$ is represented by a state variable $x_C :: PFUN(ID, T_C)$ in X. $PFUN(ID, T_C)$ is called the *class type* of C. $PFUN(\alpha, \beta)$ is the type constructor for partial function from α to β with finite domain[1]. Moreover, C gives rise to *referential constraints* defined by the structure S and *class inclusion constraints* defined by the set of superclasses of C. All other constraints on $\mathcal{S}$ and C are directly translatable in constraints on X. Let us now formally describe the form of structurally defined inclusion and referential constraints.

Definition 6 Let C, C' be classes with representation types T_C and T'_C respectively and let $o : T_C \times ID \to BOOL$ be a function.

(i) If T_C be a subtype of T'_C via $f : T_C \to T'_C$, a *class inclusion constraint* on C and C' is a constraint in the form

$$\forall i :: ID. \forall v :: T_C. \, member(Pair(i, v), x_C) = true \;\Rightarrow$$

$$member(Pair(i, f(v)), x_{C'}) = true \;, \tag{2.1}$$

where $Pair$ is the constructor of $PAIR(\alpha, \beta)$ and $member$ is a function on finite sets $FSET(\alpha)$, hence also on the subtype $PFUN(\alpha, \beta)$.

In the general case a *class inclusion constraint* on C and C' has the form

$$\forall i :: ID. \, member(i, dom(x_C)) = true \;\Rightarrow member(i, dom(x_{C'})) = true \;.$$

(ii) A *referential constraint* on C and C' is a constraint in the form

$$\forall i, j :: ID. \forall v :: T_C. \, member(Pair(i, v), x_C) = true \wedge$$

$$o(v, j) = true \;\Rightarrow\; member(j, dom(x_{C'})) = true \;. \tag{2.2}$$

It is easy to see that each class D in the set of superclasses of C gives rise to an inclusion constraint. Moreover, each reference $r : E$ occurring in the structure expression S of C gives rise to a referential constraint with the function o determined by the type underlying S. Then $o(v, j) = true$ means that the identifier j occurs within v at a place corresponding to the reference.

Let us now finalize the presentation of the datamodel by a simple example.

[1] In fact, we need a more sophisticated semantics for objects and classes as exemplified by the algebraic approach of the IS-CORE group [22] or by the evolving algebra approach [28].

Example 4 Assume the existence of a value type PERSON defined elsewhere. A class C named PERSONC may be defined as follows.

PERSONC ==
  `Structure` PAIR(PERSON , spouse : PERSONC)
  `Constraints` $\forall$ I,J :: ID . $\forall$ V :: T_C . member(Pair(I,V),x_C) = true $\wedge$
  member(Pair(J,V),x_C) = true $\Rightarrow$ I = J
`End` PERSONC $\square$

2.3.3 Static Integrity Constraints

Let us now introduce some kinds of explicit static constraints are generalizations of constraints known from the relational model, e.g. functional and key constraints, general inclusion and exclusion constraints, multi-valued dependencies and path constraints [51, 55].

Definition 7 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 functions.

(i) A *functional constraint* on C is a constraint of the form

$$\forall i, i' :: ID. \forall v, v' :: T_C . c^1(v) = c^1(v') \wedge member(Pair(i, v), x_C) = true$$
$$\wedge member(Pair(i', v'), x_C) = true \Rightarrow c^2(v) = c^2(v') \ . \tag{2.3}$$

A functional constraint is called a *value constraint* iff neither T_1 nor T_2 contains ID.

(ii) A *uniqueness constraint* on C is a constraint of the form

$$\forall i, i' :: ID. \forall v, v' :: T_C . c^1(v) = c^1(v') \wedge member(Pair(i, v), x_C) = true$$
$$\wedge member(Pair(i', v'), x_C) = true \Rightarrow i = i' \ . \tag{2.4}$$

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

(iii) A *general inclusion constraint* on C_1 and C_2 is a constraint of the form

$$\forall t :: T. \exists i_1 :: ID, v_1 :: T_{C_1} . member(Pair(i_1, v_1), x_{C_1}) = true \wedge c_1(v_1) = t$$
$$\Rightarrow \exists i_2 :: ID, v_2 :: T_{C_2} . member(Pair(i_2, v_2), x_{C_2}) = true \wedge c_2(v_2) = t. \tag{2.5}$$

(iv) An *exclusion constraint* on C_1, C_2 is a constraint of the form

$$\forall i_1, i_2 :: ID. \forall v_1 :: T_{C_1} . \forall v_2 :: T_{C_2} . member(Pair(i_1, v_1), x_{C_1}) = true$$
$$\wedge member(Pair(i_2, v_2), x_{C_2}) = true \Rightarrow c_1(v_1) \neq c_2(v_2) \ . \tag{2.6}$$

(v) An *object generating constraint* on C is a constraint of the form

$$\forall i_1, i_2 :: ID. \forall v_1, v_2 :: T_C . member(Pair(i_1, v_1), x_C) = true \wedge$$
$$member(Pair(i_2, v_2), x_C) = true \wedge c^1(v_1) = c^1(v_2) \Rightarrow$$
$$\exists i :: ID, v :: T_C . member(Pair(i, v), x_C) = true \wedge$$
$$c^1(v) = c^1(v_1) \wedge c^2(v) = c^2(v_1) \wedge c^3(v) = c^3(v_2) \ . \tag{2.7}$$

Note that the definition of uniqueness constraints is a generalization of the key concept and object generating constraints are a straightforward generalization of multi-valued dependencies in the relational model [63]. The following definition extends these constraints to path constraints.

Definition 8 (i) Let $C_1, \ldots, C_n$be classes in a schema $\mathcal{S}$ with representation types $T_{C_1}, \ldots, T_{C_n}$ and let referential constraints on $C_{i\perp 1}, C_i$ be defined via $o_i : T_{C_{i-1}} \times ID \rightarrow BOOL$. Then $C_1, \ldots, C_n$ define a *path* in $\mathcal{S}$ and the corresponding *path expression* is given by

$$\begin{aligned} & member(Pair(i_1, v_1), x_{C_1}) = true \wedge o_2(v_1, i_2) = true \wedge \\ & \quad member(Pair(i_2, v_2), x_{C_2}) = true \wedge \ldots \wedge o_n(v_{n\perp 1}, i_n) = true \wedge \\ & \quad member(Pair(i_n, v_n), x_{C_n}) = true \ . \end{aligned} \tag{2.8}$$

(ii) Let C, C' be classes in a schema $\mathcal{S}$ and let $\mathcal{P}$ be a { (general) inclusion | exclusion | functional | uniqueness | object generating } constraint on C, C' or C respectively. If $C_1, \ldots, C_n$ and $C'_1, \ldots, C'_m$ are paths in $\mathcal{S}$ with $C_n = C$ and $C'_m = C'$, then replacing the corresponding path expressions for $member(Pair(i, v), x_C) = true$ and $member(Pair(i', v'), x_{C'}) = true$ respectively in $\mathcal{P}$ defines a *path constraint* $\mathcal{P}'$ on C_1 and C'_1. We assume all free variables in $\mathcal{P}'$ other than x_C and $x_{C'}$ to be universally quantified. More precisely we call $\mathcal{P}'$ a { *(general) path inclusion* | *path exclusion* | *path functional* | *path uniqueness* | *path object generating* } *constraint.*

2.4 The Behavioural Approach

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.

In general methods describe possible sequences of database instances. In order to restrict this set of possible sequences to legal ones dynamic integrity constraints are used. In general some temporal logic is required to express such constraints [37]. In order to avoid this we restrict the OODM to allow only transition constraints to be specified.

In a second part we shall have a short look on queries and views.

2.4.1 Methods, Transactions and Transition Constraints

Let us now address the formalization of the notions *method*, *class* and *transition constraint* and then generalize the notion of *schema*.

Definition 9 Let $T_1, \ldots, T_n, T'_1, \ldots, T'_m \in N_T$ such that there exist types in $\mathcal{T}$ with these names. Let $M \in N_M$ and $\iota_1, \ldots, \iota_n, o_1, \ldots, o_m \in V$.

(i) 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

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

(ii) Let C be some class as in Definition 5. A *method* M on C consists of a method signature with name M and a body that is represented as a guarded command on $X = \{x_C\}$.

(iii) 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.

On the representation level (see Section 2.3.2) we use guarded commands for methods. As mentioned above 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 10 Let C be a class as in Definition 5 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:

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

(ii) 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.

(iii) 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 reuired on the superclass can be omitted on the subclass. Both rules are captured by the formal notion of specialization. We omit the details [47].

Example 5 Let us now describe methods on the class PERSONC introduced in Example 4. Some details such as the definition of the method named "exists" are omitted, but we remark that the described insert-method is in fact the canonical one [50].

```
Methods
      insert( P :: V_C = PAIR(PERSON,V_C) ) == I ← insert'(P)
(hidden)
    I :: ID ← insert'( P :: V̄_C = PAIR(PERSON,UNION(V̄_C,ID)) ) ==
     B :: BOOL | ( B ← exists(P,x_C) ;
      B = true →
       ( I :: ID | ( member(I,dom(x_C)) = false →
          P'' :: T_C |
            ( ∃ P' :: T_C . P' = P → P'' := P ⊠
              ( J :: ID | J ← insert'( substitute(P,I,second(P))) ;
                P'' := Pair(first(P),J) )) ;
         x_C := union(x_C,single(Pair(I,P''))))
       ⊠ skip                                                    □
```

The dynamic part of a schema also requires transition constraints to be specified.

Definition 11 Let C be a class as in Definition 5. A *transition constraint* on C is a first-order formula $\mathcal{R}$ with $fr(\mathcal{R}) \subseteq \{x_C, x'_C\}$, where x'_C represents the value of x_C after performing some operation.

Now we are prepared to generalize the definition of classes, schemata and instances.

Definition 12 (i) 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*), a set of static constraints $\mathcal{I}_1, \ldots, \mathcal{I}_k$, a set of transition constraints $\mathcal{J}_1, \ldots, \mathcal{J}_l$ and a method specification $(\mathcal{S} = \{M_1, \ldots, M_n\}\ ,\ \mathcal{H} = \{M'_1, \ldots, M'_{n'}\})$ on C. 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.

(ii) A *schema* $\mathcal{S}$ is a finite collection of classes $C_1, \ldots, C_n$ closed under references, superclasses and method call together with a collection of static constraints $\mathcal{I}_1, \ldots, \mathcal{I}_n$ and a collection of transition constraints $\mathcal{J}_1, \ldots, \mathcal{J}_l$.

(iii) An *instance* $\mathcal{D}$ of a structural 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.

2.4.2 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 [9].

Querying a relational database can be expressed by terms in relational algebra. This view can be easily generalized to the OODM that is built upon a sophisticated extensible type

system $\mathcal{T}$. Each type is algebraically specified and hence gives rise to an algebra – to be more precise: a G-algebra [54]. 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 [13] 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 [14], 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 $\mathcal{S}$ can then be represented by a term t of some value type T with $fr(t) \subseteq \{x_C \mid C \in \mathcal{S}\}$. 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 of type $T = PFUN(ID, T_C)$ would be x_C. This is not a value query, but if T_C is a value type, we may take $T' = BAG(T_C)$ and the natural projection given by the subtype functions

$$PFUN(\alpha, \beta) \to FSET(PAIR(\alpha, \beta)) \to BAG(PAIR(\alpha, \beta)) \to BAG(\beta) \; .$$

In Section 3.1 we shall see how to generalize this to be case where T_C is arbitrary. We then have to replace T_C by the value-representation type V_C provided this exists.

In the case of arbitrary access expressions another problem occurs [14]. 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 $ID \times \ldots \times ID \to ID$ [32, 34].

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 13 Let $\mathcal{S} = \{C_1, \ldots, C_n\}$ be some schema.

(i) A *value query* on $\mathcal{S}$ is a term t over some proper value type T with $fr(t) \subseteq \{x_{C_1}, \ldots, x_{C_n}\}$.

(ii) An *access expression* on $\mathcal{S}$ is a term t over some proper type T with $fr(t) \subseteq \{x_{C_1}, \ldots, x_{C_n}\}$.

(iii) A *view* on $\mathcal{S}$ consists of a view name $v \in N_C$ such that there is no class $C \in \mathcal{S}$ with this name, a structure expression $S(v)$ containing references to classes in $\mathcal{S}$ or to views on $\mathcal{S}$ and a defining access expression $t(v)$ of type $PFUN(ID, T_v)$, where T_v is the representation type corresponding to $S(v)$.

Let us now finalize this chapter with a simple example of a view in the OODM.

Example 6 Take again the class PERSONC of Example 4. Let *Age* be some selector on the type $PERSON$ and let the function $filter$ be defined on $FSET(\alpha)$ with arity $FSET(\alpha) \times FUN(\alpha, BOOL) \rightarrow FSET(\alpha)$ (see also [13]). Then the following defines the subset of PERSONC of all old persons.

```
View OldPersonV ==
   Structure PAIR(PERSON, spouse : PersonC)
   Definition filter(PersonC,Lambda[X](greater(Age(First(Second(X))),60)))
End OldPersonV
```
□

Chapter 3

Object Identification and Value-Representation

This chapter is devoted to the identification problem in object oriented databases. Roughly speaking databases are considered to contain persistent mass data. 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 which may be e.g. a physical address 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 notions of *value-identifiability* and *value-representability*. We discuss the identification problem in Section 3.1 under the assumption that the only explicit constraints are uniqueness constraints. Then we analyse the weaker concept of *weak value-representability* that 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 [64].

3.1 On 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.

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 4.1 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 14 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 14 is unique up to isomorphism.

Theorem 15 *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. □

3.1.1 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 16 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 17 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.

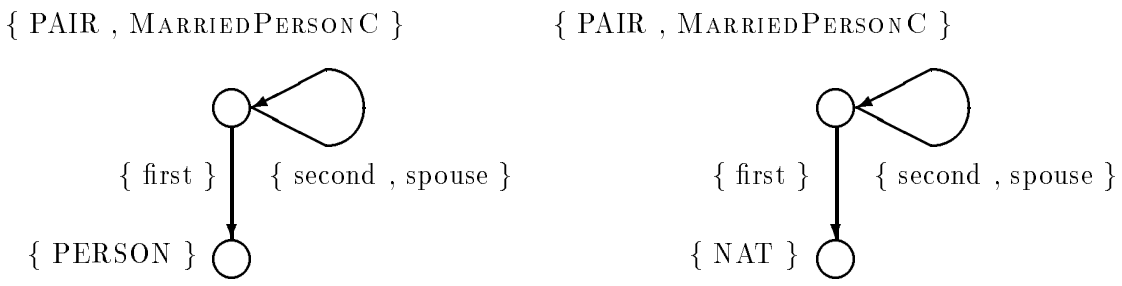

Figure 3.1: The reference graph and identification graph of class MarriedPersonC

Example 7 Let MarriedPersonC be defined as in Example 4. Then the reference graph and the identification graph with respect to the uniqueness constraint of this class are shown in Figure 3.1. □

Theorem 18 *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 14.

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 14. □

Theorem 19 *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 18. □

Theorem 20 *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 7 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 18 implies that value-representability can be decided by checking the existence of uniqueness constraints in the class definitions. □

Theorem 21 *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 20. □

3.1.2 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 18 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 14.

Algorithm 22 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 22 to be a proper value type.

Lemma 23 *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 22. 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 24 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:

(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 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 24 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 23. □

3.1.3 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 2.2.1 satisfy this property.

Definition 26 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 $g_{0,1} \cdots g_{n \perp 1,n}$ 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 $o_{0,1} \cdots o_{n \perp 1,n}$. This can be used as follows to define a function $cyc : ID \times ID \rightarrow BOOL$ corresponding to a cycle in G_{ref}.

Definition 27 Let C be a class in a schema $\mathcal{S}$ and let $g_{0,1} \cdots g_{n \perp 1,n}$ 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 $l = 0, \ldots, n \perp 1$.

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

Lemma 28 *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 29 *Let $\mathcal{D}$ be an instance of schema $\mathcal{S} = \{C_1, \ldots, C_n\}$. Then $\mathcal{D}$ satisfies at each stage of Algorithm 22 uniqueness constraints for all $i = 1, \ldots, n$ defined by some $c_i : T_{C_i} \rightarrow G(C_i)$.*

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 30 *Let $\mathcal{D}$ be an instance of schema $\mathcal{S} = \{C_1, \ldots, C_n\}$. Then $\mathcal{D}$ satisfies at each stage of Algorithm 24 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 29. □

Lemma 31 *Let $\mathcal{D}$ be an instance of schema $\mathcal{S} = \{C_1, \ldots, C_n\}$. Then at each stage of the algorithms 22 and 24 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 29 it is sufficient to show that the required property is preserved by the application of a rule from Algorithm 22 or 24. 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 29 and Lemma 30 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 $g^{\perp 1} : g(c_i(codom_{\mathcal{D}}(C_i))) \to G(C_i)$. Then $\bar{c}_i \circ h$ satisfies the required property. □

Theorem 32 *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 22. Then C is value-representable with value representation type V_C.*

Proof. V_C is a proper value type by Lemma 23. From Lemma 29 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 $\mathcal{D}$.

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 31 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 33 *Let $\mathcal{S}$ be a schema such that all classes C in $\mathcal{S}$ are value-identifiable. The all classes C in $\mathcal{S}$ are also value-representable.* □

Theorem 34 *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 24 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 32. □

Theorem 35 *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 20. □

3.2 Weak Value-Representability

Let us now ask whether there exist also weaker identification mechanisms other than value-representability. In several papers, e.g. [44] 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 [64]. 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 36 Let $\mathcal{S}$ be some schema.

(i) If r is a reference from class C to D in $\mathcal{S}$ and $o : T_C \times ID \to BOOL$ is the function of Definition 6 expressing the corresponding referential constraint, then r satisfies the *(SF)-condition* iff

(a) $o(v,i) = true \wedge o(v,j) = true \;\Rightarrow\; i = j$ and

(b) $member(j, dom(x_D)) = true \;\Rightarrow\; \exists v :: T_C . member(v, codom(x_C)) = true \wedge o(v,j) = true$

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

(ii) An *(SF)-chain* from class D to C in $\mathcal{S}$ is a sequence of classes $D = C_0, \ldots, C_n = C$ together with references r_i $(i = 1, \ldots, n)$ from $C_{i\perp 1}$ to C_i such that each r_i satisfies the (SF)-condition.

(iii) A class C in $\mathcal{S}$ 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 37 *If C is a weakly value-identifiable class in a schema $\mathcal{S}$, then there exists a proper value type I_C such that for each instance $\mathcal{D}$ of $\mathcal{S}$ 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 $(i = 1, \ldots, n)$. Since 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 $o_i(v,j) = true$ (just take some inverse image of j under the surjective reference function). Since r_i defines a function, c_i is clearly injective.

If $c' : ID \to I_D$ 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. □

Problem. *Does the converse of Lemma 37 also hold?*

Definition 38 A class C in a schema $\mathcal{S}$ is called *weakly value-representable* iff there exists a proper value type V_C such that for each instance $\mathcal{D}$ of $\mathcal{S}$ the following properties hold.

(i) There is a function $c : ID \to V_C$ that is injective on $dom_{\mathcal{D}}(C)$.

(ii) 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 37 each weakly value-representable class is also weakly value-identifiable. We shall see that also the converse of this fact is true.

Algorithm 39 Let the schema be $\mathcal{S} = \{C_1, \ldots, C_n\}$. Start with $H(C_i) = T_{C_i}$ $(i = 1, \ldots, n)$. If ID occurs in some $H(C_i)$ 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 $H(C_j)$ is a proper value type and ID_j occurs in some $H(C_i)$ $(j \neq i)$, then replace this corresponding ID_j in $H(C_i)$ by $H(C_j)$.

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

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 $H(C_i)$ by the type name $H(C_j)$. □

This algorithm is similar to the Algorithms 22 and 24. However, we completely ignore uniqueness constraints.

Lemma 40 *Let C be a class in a schema $\mathcal{S}$ and let I_C be the type $H(C)$ computed by Algorithm 39. Then I_C is a proper value type.*

Proof. The proof is analogous to that of Lemma 23. □

Lemma 41 *Let $\mathcal{D}$ be an instance of the schema $\mathcal{S} = \{C_1, \ldots, C_n\}$. 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 37 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 Algorithm 39 there exists a function $\bar{c} : H(D) \to I_C$ that is unique on $c'(dom_{\mathcal{D}}(C))$ with $c = \bar{c} \circ c'$.*

Proof. The proof is analogous to the one of Lemma 29. □

Theorem 42 *Let C be a weakly value-identifiable class in a schema $\mathcal{S}$ and let 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 Algorithm 39. Then C is weakly value-representable with weak value-representation type V_C.*

Proof. V_C is a proper value type by Lemma 40. From Lemmata 30 and 37 it follows that there exists a function $c' : ID \to V_C$ that is injective on $dom_{\mathcal{D}}(C)$.

From Lemma 41 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. □

Chapter 4

Genericity

The preservation of advantages of relational databases requires the definability of generic operations for querying and for the insertion, deletion and update of single objects. While querying [1, 13, 30, 57] 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 [45] have chosen the solution to abandon generic operations. Others [7, 8, 10] 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 generic operations for the insertion, deletion and update of single objects can be derived automatically. We show that there is a close connection between *value-representability* and the unique existence of generic operations for the insertion, deletion and update of single objects. Furthermore, inclusion and referential integrity are enforced by these operations.

In Section 4.1 we describe these operations. In Section 4.2 we then specify an algorithm to compute generic update methods [52, 53]. The specification is built on the same theoretical ground as the OODM, hence sets up a specific case of linguistic reflection [60].

4.1 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 [47].

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.

4.1.1 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 43 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 [47] for these update operations.

Lemma 44 *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*. □

4.1.2 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 45 *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 14 c is injective on $codom_{\mathcal{D}}(C)$, hence any continuation h of $d \circ g \circ c^{\perp 1}$ 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 14 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 46 *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 32 the value-representation type V_C is the result of Algorithm 22, 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 45 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 45 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 47 *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

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

Then the quasi-canonical insert operation can be defined as follows:

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

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 43 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 $J_{i,j} \leftarrow insert'_{C_i}(V_{i,j}^{(I)})$. 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 48 *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 44 and Lemma 46 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 47. □

4.2 A Generator Approach to Achieve Higher-Level Genericity

Our aim is to generate canonical update methods $insert_C$, $delete_C$ and $update_C$ for each class C of a database schema. These operations demand the identification of objects without accessing the object identifier, since *oid*s are an internal concept and do not have a meaning for the user

of a database. Hence the need for *value-representability*. Besides this identification problem we also have to cope with the enforcement of implicit integrity constraints. In Section 3.1 it has been shown that value-representability is a necessary and sufficient condition for the existence of consistent canonical update operations.

These update operations are "generic" in the sense, that they are applicable to each class of a schema. Our aim now is to provide an algorithmic solution to the generation of canonical update operations. A natural first idea is to exploit polymorphism as in [16] for this task. However, canonical consistent updates on a class C require an input-type V_C without any occurrence of ID. Such an input-type has to be computed from the schema. Hence the generation of such operations requires meta-information. It has been shown in [58, 59] that the need for meta-information exceeds the capability of polymorphism. Two solutions are then possible:

- introduce polymorphic meta-types or
- use linguistic reflection as proposed in [60].

The first approach is fine as long as we do not care about decidability problems in type checking, however, reflection is more practical.

The basic idea of linguistic reflection is to use representation types such as $SCHEMA_{rep}$, $CLASS_{rep}$ and $TYPE_{rep}$ for the representation of abstract syntax expressions representing schemata, class definitions and type declarations respectively. For each of these, there exists a function *raise* associating with this syntactic expression a true schema, class or type respectively. Moreover, we need functions i and v with signature $SCHEMA_{rep} \times CLASS_{rep} \rightarrow TYPE_{rep}$. $v(\mathcal{S},\mathcal{C})$ represents a value-type needed for the insertion of a new object into $raise(\mathcal{C})$. Clearly, this type is also required for updates. $i(\mathcal{S},\mathcal{C})$ represents a value-type needed for the identification of some object, hence is needed for delete and update-operations.

If $OPER_{rep}(\mathcal{I},\mathcal{S})$ represents the operations on the schema defined by $\mathcal{S}$ (defined via methods) with input type represented by $\mathcal{I}$, then the problem is to define three reflective functions

$insert : \mathcal{S} :: SCHEMA_{rep} \times \mathcal{C} :: CLASS_{rep} \rightarrow OPER_{rep}(v(\mathcal{S},\mathcal{C}),\mathcal{S})$,
$delete : \mathcal{S} :: SCHEMA_{rep} \times \mathcal{C} :: CLASS_{rep} \rightarrow OPER_{rep}(i(\mathcal{S},\mathcal{C}),\mathcal{S})$ and
$update : \mathcal{S} :: SCHEMA_{rep} \times \mathcal{C} :: CLASS_{rep} \rightarrow OPER_{rep}(i(\mathcal{S},\mathcal{C}) \times v(\mathcal{S},\mathcal{C}),\mathcal{S})$.

$OPER_{rep}$ is a type constructor applicable only to representation types. Clearly, we should have $OPER_{rep}(I_{rep}, S_{rep}) = (OPER(I,S))_{rep}$, where $OPER$ is again a type constructor and $OPER(I,S)$ is the type of the operations on S with input-type I.

4.2.1 Basic Assumptions

Since we are concerned with providing generic update operations we have to discuss the uniqueness and existence of these operations in general.

Whenever object oriented data models include the principle of hiding identifiers from the user insert, delete and update operations demand the unique accessibility of objects from "outside", i.e. by the user, in a way different from using the internal identifiers. This is in accordance with the identifier being only an implementation concept [12, 14]. Otherwise database management of the identifiers themselves would be required [36]. The approach of naming objects in a program scope usually applied in object oriented programming languages

is to be ruled out in the database context. Therefore we have to require that objects within a class need to be distinguished by values and referenced objects.

Our previous work on these topics [50, 49] gives a characterization of those classes having objects that are completely representable by values, i.e. we can replace references by values of referred objects and identify an object by its values. These classes are called value-representable i.e. it is possible to identify each object by some value of a proper value type (without occurrence of oids). (In the case of cyclic references the finiteness of the database imposes the instances of such a type to be finitely representable).

It is shown that the value-representability of classes is decidable under certain conditions and that a unique value type (serving as input type for canonical update operations) can be derived for any value-representable class. Moreover, the unique existence of canonical insert, delete and update operations of a class C turns out to be guaranteed iff C is value-representable.

4.2.2 A Framework for Generator Application

Applying the approach discussed above to practically support meta polymorphism we need to introduce in our context:

- *Representation types* for syntactic components of our language capturing meta information, e.g. $Schema_{rep}$, $Class_{rep}$, $Type_{rep}$ about schemata, class and type definitions or representing code to be generated, e.g. $Meth_{rep}$. For each of these a function *raise* exists associating with this abstract syntax a true schema, class, type or method respectively.

- *Generator functions* that are defined on these representation types and produce again abstract syntax which extends – after a *raise* – our specification. In particular we need a generator $V_{\mathbf{gen}}$ with signature

  $$V_{\mathbf{gen}} \; : \quad Schema_{rep} \times Class_{rep} \rightarrow Type_{rep}$$

  deriving a value type definition serving as input type in insert and update operations. Moreover, we need at least one generator $I_{\mathbf{gen}}$ with the same signature that derives an identification type used in delete and update operations. For the sake of simplicity we neglect $I_{\mathbf{gen}}$ and use also $V_{\mathbf{gen}}$ for identification. Canonical update operations are generated by functions $Insert_{\mathbf{gen}}$, $Delete_{\mathbf{gen}}$, $Update_{\mathbf{gen}}$ with signatures:

  $$\begin{array}{lll} Insert_{\mathbf{gen}} & : & Schema_{rep} \times Class_{rep} \rightarrow Meth_{rep} \\ Delete_{\mathbf{gen}} & : & Schema_{rep} \times Class_{rep} \rightarrow Meth_{rep} \\ Update_{\mathbf{gen}} & : & Schema_{rep} \times Class_{rep} \rightarrow Meth_{rep} \end{array}$$

4.2.3 Representations Types

In this section we present the definition of representation types forming the framework in which generator functions are defined. Here we make use of general algebraic type specifications as in [47, 13]. However, we distinguish between constructors, selectors and other functions. Axioms are given by conditional equations. A type definition may refer to other type definitions which is indicated by the keyword *basedOn*.

We concentrate on those representation types that are necessary for defining $Insert_{\mathbf{gen}}$. Constructions for $Delete_{\mathbf{gen}}$ and $Update_{\mathbf{gen}}$ can be given analogously. Although needed we

omit all details concerning the representation type for types and the definition of the generator for value-representation types that are needed for input. The following type definitions provide abstract syntax values for *structures, classes, methods* and *commands.*

The type definition for $Structure_{rep}$ uses representations of types as defined in [50] on top of which structures are built. The list of reference-name/class-name pairs indicates the references of a class whereby class objects can have a complex structure with references occurring as parameters at any level in nested type constructors. The axiom indicates that a reference list of a structure has to be as long as its parameter list whereby *params* is a function on types returning lists of their parameters.

$Structure_{rep}$ ==
  basedOn
    $ClassName_{rep}$, $Type_{rep}$, $List(\alpha)$, $Pair(\alpha,\beta)$, $RefName_{rep}$
  constructors
    $structure_{con}$: $Type_{rep} \times List(Pair(RefName_{rep} , ClassName_{rep})) \rightarrow Structure_{rep}$
  selectors
    $structureType_{sel}$: $Structure_{rep} \rightarrow Type_{rep}$
    $references_{sel}$: $Structure_{rep} \rightarrow List(Pair(RefName_{rep} , ClassName_{rep}))$
  axioms
    $With\ s : Structure_{rep}$.
    $length(params(structureType_{sel}(s)) = length(references_{sel}(s))$
  end

The definition of $Class_{rep}$ can be given in the following way omitting for now user defined methods.

$Class_{rep}$ ==
  basedOn
    $ClassName_{rep}$, $Structure_{rep}$, $Fset(\alpha)$
  constructors
    $class_{con}$: $ClassName_{rep} \times Fset(ClassName_{rep}) \times Structure_{rep} \rightarrow Class_{rep}$
  selectors
    $className_{sel}$: $Class_{rep} \rightarrow ClassName_{rep}$
    isA_{sel} : $Class_{rep} \rightarrow Fset(ClassName_{rep})$
    $structure_{sel}$: $Class_{rep} \rightarrow Structure_{rep}$
  axioms...
  end

An abstract representation of a method consists of its name, the declaration of its input and output parameters together with a representation of its body.

$Meth_{rep}$ =
  basedOn
    $MethName_{rep}$, $VarName_{rep}$, $Command_{rep}$, $Type_{rep}$, $Fset(\alpha)$, $Pair(\alpha, \beta)$
  constructors
    $meth_{con}$: $MethName_{rep} \times Fset(Pair(VarName_{rep} , Type_{rep})) \times$
      $Fset(Pair(VarName_{rep} , Type_{rep})) \times Command_{rep} \rightarrow Meth_{rep}$
  selectors
    $methName_{sel}$: $Meth_{rep} \rightarrow MethName_{rep}$

$input_{sel}$: $Meth_{rep} \rightarrow Fset(Pair(VarName_{rep}, Type_{rep}))$
$output_{sel}$: $Meth_{rep} \rightarrow Fset(Pair(VarName_{rep}, Type_{rep}))$
$command_{sel}$: $Meth_{rep} \rightarrow Command_{rep}$
end

Finally, the representation of commands is inductively defined by:

$Command_{rep}$ ==
basedOn
$MethName_{rep}$, $VarName_{rep}$, $List(\alpha)$, $Pair(\alpha, \beta)$, $Expr_{rep}$, $Pred_{rep}$
constructors
$skip_{con}$: $\rightarrow Command_{rep}$
$call_{con}$: $MethName_{rep} \times List(VarName_{rep}) \times List(VarName_{rep}) \rightarrow Command_{rep}$
$assign_{con}$: $VarName_{rep} \times Expr_{rep} \rightarrow Command_{rep}$
seq_{con} : $List(command_{rep}) \rightarrow Command_{rep}$
if_{con} : $Pred_{rep} \times Command_{rep} \times Command_{rep} \rightarrow Command_{rep}$
$while_{con}$: $Pred_{rep} \times Command_{rep} \rightarrow Command_{rep}$
any_{con} : $VarName_{rep} \times Type_{rep} \times Pred_{rep} \times Command_{rep} \rightarrow Command_{rep}$
selectors...
axioms...
end

Predicates will have the constructors $true_{con}$, $false_{con}$, $forall_{con}$, $exist_{con}$, not_{con}, $implies_{con}$, eq_{con}, $isIn_{con}$.... We omit the details.

4.2.4 Generators for Generic Update Operations

The generator to be presented produces abstract syntax for *insert* operations, i.e. it is defined in terms of $meth_{con}$. The case of *update* and *delete* can be handled analogously. Its task is the generation of the appropriate actual parameters for $meth_{con}$ from class and schema representations. The parameters are the operation's name, a list of input parameters, a list of output parameters and a representation of a command forming the body of the insert operation. Omitting other details we concentrate on parts of the body managing inclusion constraints, dealing with references in the acyclic and cyclic case and changing the class extent. For each subtask we will briefly discuss the code to be produced followed by a discription of the algorithm to produce the actual parameters forming this code and then present the generator definition together with its (raised) output.

Managing Inclusion Constraints

In the case of *IsA*-relationships we have to generate calls of insert operations on each of the direct superclasses while providing appropriate input values for these operations.

Since class representations capture *IsA*-information through a set of class names we can produce the insert calls by applying a general *set-reduce* function as defined in [58]. It will return a list of calls which, serving as parameter for a sequence constructor, will lead to a sequence of commands. Roughly speaking *set-reduce* applies a function to each element of a set and accomodates the results by the application of another function.

For generating a call we need the method's name together with a list of input and output variables. Of further interest is the input value. In parallel to the generation of insert

operations we have to generate functions on the value types involved. For sake of simplicity we assume that *IsA*-relationship between classes C_1 and C_2 (C_1 *IsA* C_2) includes subtype relationships on their structure types (T_C_1 *subtypeOf* T_C_2). This is not required in general. *IsA*-relation on classes means inclusion relation on their corresponding sets of oids. If we had no subtype relationship on the value-types we would have to produce a different input type for a subclass including all values necessary in superclasses. This is easy compared to the case of an existing subtype relation. In Lemma 45 it has been shown that in the case of subtype relationship a subtype function between the value types of sub- and superclasses exists. Thus, a generator for these functions exists. At this point we omit the definition. We merely make use of the name of the generated function which we built by the concatenation instead of applying a name selector to the generator's output. The function taking V_C to V_D_i is called h_C,D_i.

Again for simplicity we make the further assumption for the case of more than one distinct superclasses. We assume the existence of a common superclass of these superclasses. Otherwise we would run into problems of renaming or having synonyms of identifiers. A technical solution for this problem is indicated in [62].

$Insert_{\mathbf{gen}}$ has input values C of type $Class_{rep}$ and S of type $Schema_{rep}$. The insert operation being produced uses variable name *'v'* for its input value.

seq_{con}(*setReduce*(
  $isA_{sel}(C)$,
  lambda D.
    $call_{con}$(*concat('insert_',D)*,
      singleList($apply_{con}$(*concat('h_',* $concat(className_{rep}\ (C),D)$,*'v'*))
      singleList(*concat('i_',D)*)),
  append,
  nil))

If we apply *raise* to the produced output code for superclasses $D_1, \ldots D_n$ the result will be:

$i\_D_1 \leftarrow insert\_D_1(h\_C,D_1(v));$
$\ldots;$
$i\_D_n \leftarrow insert\_D_n(h\_C,D_n(v));$

Enforcing Referential Constraints in the Acyclic Case

In this section we handle classes with an acyclic reference structure. In this case the insert operation includes calls with appropriate input values for each class being referred to. The operation has somehow to gather the returned identifiers for producing a value of type T_C which forms a prerequisite for adding the new object to the class extent.

In order to produce the sequence of insert calls for referenced objects we proceed in the same way as shown above. We apply *list-reduce* to the list of references which has to be selected from the structure representation of the class. *list-reduce* is defined analogously to *set-reduce*.

Again we have to provide input values for which we assume generated functions doing the job [50]. For each reference r_i to a class C_i we have a function val_r_i taking an input

value of type V_C to the set of all those values of type V_C_i that occur at the corresponding place in v. These values correspond to existing or new objects in C_i to which the future object in C will refer to via r_i. As we explained above the value type of a class includes for each reference the value type of the class referenced to. Since references can occur in deeply nested type constructors (e.g. we can have lists of sets of referred objects) they are in general multivalued. This requires the call of a method *insertObjects_*C_i before the final insert. The method *insertObjects_*C_i receives a set of input values for C_i and returns sets of identifier/input-value pairs. Its definition is given below.

The calling insert operation gathers these sets of identifier-value pairs in a list l.[1] It should be indicated that in the case of references to $C_1, \ldots C_n$ the list l has type *List(Set(Pair(Id, Union(*$V\_C_1, \ldots V\_C_n$*))))* where *Union* means a variadic union type constructor. Since we neglect the representation type for types we introduce a function *makeTypeRep* taking a type definition and turning it into its representation.

$any_{con}($
  'l',makeTypeRep(List(Set(Pair(Id,Union(V_C_1*,...*V_C_n*))))),*
  eq_{con}*('l','nil'),*
  seq_{con}*(listReduce(*
    $references_{sel}(structure_{sel}(C))$*,*
    lambda D.cons
      $call_{con}$*(concat('insertObjects_',second(D)),*
        $apply_{con}$*(concat('val_', first(D)),v), 'j'),*
      singleList($assign_{con}$*('l', 'append(Pair(j,l))')))),*
    append,
    nil)))

The generator will produce the following code for references to classes C_1 to C_n.

l : List(Set(Pair(Id,Union(V_C_1*,..,*V_C_n*)))) | l = nil* $\perp\!\!\!\rightarrow$
(j $\leftarrow$ *insertObjects_*C_1*(val_*r_1*(v)); l := append(Pair(j,l));*
...;
j $\leftarrow$ *insertObjects_*C_n*(val_*r_n*(v)); l := append(Pair(j,l)));*

The called methods *insertObjects_*C_i have the following form:

k $\leftarrow$ *insertObjects_*C_i *(vs : Set(*V_C_i*)) =*
  s:Set(Pair(Id,V_C_i*) | s = empty* $\perp\!\!\!\rightarrow$
  (j:Nat | j = 1 $\perp\!\!\!\rightarrow$
  (do vs $\neq$ *empty*
    v : V_C_i *| v* $\in$ *vs* $\perp\!\!\!\rightarrow$
    $(id_{r_i,j}$ $\leftarrow$ *insert_*C_i*(v);*
    s := insert(Pair($id_{r_i,j}$*,v),s);*
    vs := vs - {v};
    j := j+1);
  od);
  k := s)

[1] At this place a list structure is needed which corresponds to the order in which the references are given in the reference list. Referenced classes might have the same structure so that a value alone would not suffice to indicate to which class the object with a value belongs to.

Clearly, there also exists a generator for *insertObjects_*C_i, but for the sake of brevity we omit its description here.

Changing the Class Extent

This part deals with adding the new object to the class' extent. According to our data modelling approach we have to insert a pair consisting of an identifier and a value of type T_C. This requires a function *f_C* on V_C producing a value of type T_C. It is applied to the input value of the insert operation together with the provided list *l* of sets of identifier-/values-pairs and substitutes "reference-wise" values of type V_C_i by their related identifiers as provided in list *l*. Generator code and raised generated code are the following:

$assign_{con}(name_{sel}(C),\ unionExpr_{rep}\ (name_{sel}(C),$
$\quad singleSet(concat('Pair',\ concat('i',apply(concat('f_',name_{sel}(C)),'Pair(v,l)'))))))$

$C := C \cup \{Pair(i,f\_C(Pair(v,l)))\}$

Dealing with Cyclic Reference Structures

Now we extend our generator to the case of a cyclic reference structure. In this case the value type V_C will be recursively defined. In general, even mutual recursive definitions are required [50]. Recursive types have to be defined in such a way that all their values are finitely representable, i.e. representable as rational trees. The finiteness of the database which imposes the finiteness of cycles fits perfectly with rational trees as input values.

Let us consider the insert call arising by a cycle on class C. To simplify the explanation, we concentrate on single-valued and direct cyclic references first. In this case $val\_r_c(v)$ is unary, say $\{v'\}$, where v' is of type V_C. We need a function *occurs* defined on two values of type V_C. It checks the input value whether v occurs in v' which is true iff the object corresponding to v' refers to the object corresponding to v, i.e. the cycle on class C is closed just at this input object. Furthermore, this means that the calls of insertions on C will stop with the second occurrence of the rational tree v in v. It requires the already created identifier to be available on the later applied insert operation. It also serves as indication that the cycle has to be closed at this point and no further insert call on C need to follow. This passing on of the identifier is simply managed by substituting the identifier for v in v' and calling the next inserts on C for reducing the cycle with the substituted value. This requires to change V_C by substituting $Union(V\_C,Id)$ for V_C in the defining expression of V_C. Let the resulting type be denoted $\overline{V\_C}$. Here $Union(\alpha,\beta)$ denotes a simple union type. Consider, that function val_r_i provides only values of type V_C, hence without identifiers. Therefore, if an identifier was substituted in the case of a cycle no input value will be provided for a further input call and, moreover, the identifier is already part of the value to be inserted.

General cycles are handled analogously. The function *occurs* in this case has to check if v occurs in some value v' in the set $val\_r_c(v)$.

However, since substitution of *i* for v in v' does not change v' if $occurs(v,v') = false$, we only have to replace the parameter val_C_i (in the acyclic case) by $substitute(val\_C_i(v),(v,i))$. Hence all insert operations may take values of the extended type $\overline{V\_C}$ as input which includes V_C.

Chapter 5

Integrity Enforcement

Consistency is a crucial property of database application systems. In general a database may be considered as a triplet $(\mathcal{S}, \mathcal{O}, \mathcal{C})$, where $\mathcal{S}$ defines a structure, $\mathcal{O}$ denotes a collection of state changing operations and $\mathcal{C}$ is a set of constraints. Constraints can be classified into static, transition and general dynamic constraints describing legal states, state transitions or state sequences respectively. Then the *consistency problem* is to guarantee that each specified operation $o \in \mathcal{O}$ will never violate any constraint $\mathcal{I} \in \mathcal{C}$. *Integrity enforcement* aims at the derivation of a new set $\mathcal{O}'$ of operations such that $(\mathcal{S}, \mathcal{O}', \mathcal{C})$ satisfies this property.

Let us now address the integrity enforcement problem with respect to static and transition constraints. In general, verification techniques based on predicate transformers are applicable [47] but an obvious disadvantage of the verification approach is that it does not help the user in writing consistent operations. An alternative is to generate a *Greatest Consistent Specialization* (GCS) of a given method with respect to the given constraints. This comprises the following problems:

(i) Does a GCS exist in general? Is it compatible with the conjunction of constraints, optimization, inheritance and refinement?

(ii) How does a GCS look like in the OODM with respect to distinguished classes of constraints?

(iii) How to enforce integrity of general user-defined operations with respect to arbitrary constraints? Is it sufficient to replace involved primitive operations by their GCSs?

In Section 5.1 we address these problems for static constraints. We show the existence of GCSs and also discuss compatibility results with respect to the conjunction of constraints, specialization and refinement. In Section 5.2 we describe the structure of GCSs in the OODM with respect to specific classes of constraints. Moreover, we show that the general enforcement problem can not be reduced to primitive operations. In Section 5.3 we derive the existence of GCSs for transition constraints and also discuss compatibility properties.

5.1 Enforcing Static Integrity

An alternative to consistency verification is the computation of methods that enforce all constraints of a schema. We now address this problem first for static constraints and generalize

Theorem 48. Our approach starts with a formalization of the integrity enforcement problem focussing on GCSs. We show that GCSs always exist and are unique (up to semantic equivalence). On this formal basis we are able to describe certain compatibility results and outline the structure of GCSs with respect to basic update operations and distinguished classes of static constraints.

5.1.1 The Problem

Suppose now to be given an update operation S and a static constraint $\mathcal{I}$. Assume that S is an X-operation, whereas $\mathcal{I}$ is defined on Y with $X \subseteq Y$. The idea is to construct a "new" Y-operation $S_{\mathcal{I}}$ that is consistent with respect to $\mathcal{I}$ and can be used to replace S. Roughly speaking this means that the effect of $S_{\mathcal{I}}$ on the state variables in X should not be different from the effect of S. Formally this is expressed by the specialization relation introduced in Definition 4. Clearly, if any there will be more than one such specialization. Hence the idea to distinguish one of them as the "greatest", i.e. all others should specialize it.

Before giving now the definition of a GCS let us first remark that for any predicate transformer f the *conjugate predicate transformer* f^* is defined by $f^*(\mathcal{R}) = \neg f(\neg \mathcal{R})$. Hence the following definition of a *greatest consistent specialization*:

Definition 49 Let $X \subseteq Y$ be state spaces, S an X-operation and $\mathcal{I}$ a static integrity constraint on Y. A Y-operation $S_{\mathcal{I}}$ is a *Greatest Consistent Specialization* (GCS) of S with respect to $\mathcal{I}$ iff

(i) $wlp(S)(\mathcal{R}) \Rightarrow wlp(S_{\mathcal{I}})(\mathcal{R})$ holds on Y for all formulae $\mathcal{R}$ with $fr(\mathcal{R}) \subseteq X$,

(ii) $wp(S(true) \Rightarrow wp(S)_{\mathcal{I}})(true)$ holds on Y,

(iii) $\mathcal{I} \Rightarrow wlp(S_{\mathcal{I}})(\mathcal{I})$ holds on Y and

(iv) for each Y-operation T satisfying properties (i) – (iv) (instead of $S_{\mathcal{I}}$) we have

(a) $wlp(S_{\mathcal{I}})(\mathcal{R}) \Rightarrow wlp(T)(\mathcal{R})$ for all formulae $\mathcal{R}$ with $fr(\mathcal{R}) \subseteq X$ and

(b) $wp(S_{\mathcal{I}})(true) \Rightarrow wp(T)(true)$.

Note that properties (i) and (ii) require $S_{\mathcal{I}}$ to be a specialization of S. Property (iii) requires $S_{\mathcal{I}}$ to be consistent with respect to the constraint $\mathcal{I}$. Finally, property (iv) states that each consistent specialization T of S with $T \sqsubseteq S$ also specializes $S_{\mathcal{I}}$.

Based on the formal definition of a GCS we can now raise the following questions:

- Does such a GCS always exist? If it does, is it uniquely determined by S and $\mathcal{I}$ (up to semantic equivalence)?
- Is the GCS $S_{\mathcal{I}}$ of a deterministic operation S itself deterministic? If not, how to achieve a deterministic consistent operation?
- Does a GCS $(S_{\mathcal{I}_1})_{\mathcal{I}_2}$ (provided it exists) with respect to more than one integrity constraint depend on the order of enforcement? Is it sufficient to take $(S_{\mathcal{I}_1})_{\mathcal{I}_2}$ in order to enforce integrity with respect to $\mathcal{I}_1 \wedge \mathcal{I}_2$?

- What is the relation between integrity enforcement and inheritance, i.e. is the GCS $T_{\mathcal{I}}$ of a specialization T of S a specialization of the GCS $S_{\mathcal{I}}$ of S?
- How does a GCS look like (if it exists)?

Partial results for these questions will be discussed in the next subsections.

5.1.2 Greatest Consistent Specializations

Let us first address the existence problem. Based on the axiomatic semantics via predicate transformers we can show that a GCS always exists and is uniquely determined up to semantic equivalence. First, however, we need some general properties concerning the specialization order $\sqsubseteq$.

Lemma 50 *Let $\mathcal{T}$ be a set of X-operations. Then there exists the least upper bound $S_0 = \bigsqcup_{S \in \mathcal{T}} S$ with respect to $\sqsubseteq$. S_0 is uniquely determined (up to semantic equivalence) and satisfies*

$$(i) \qquad wlp(S_0)(\mathcal{R}) \Leftrightarrow \bigwedge_{S \in \mathcal{T}} wlp(S)(\mathcal{R}) \text{ and} \tag{5.1}$$

$$(ii) \qquad wp(S_0)(\mathcal{R}) \Leftrightarrow \bigwedge_{S \in \mathcal{T}} wp(S)(\mathcal{R}) \tag{5.2}$$

for all formulae $\mathcal{R}$ with $fr(\mathcal{R}) \subseteq X$.

Proof. Let S_0 be defined (up to semantic equivalence) by (5.1) and (5.2). Then the universal conjunctivity and the pairing condition are trivially satisfied. It follows that S_0 is an X-operation.

Let $T \in \mathcal{T}$ and $\mathcal{R}$ be an arbitrary formula with $fr(\mathcal{R}) \subseteq X$. Then we have:

- $\bigwedge_{S \in \mathcal{T}} wlp(S)(\mathcal{R}) \Rightarrow wlp(T)(\mathcal{R})$ and
- $\bigwedge_{S \in \mathcal{T}} wp(S)(\mathcal{R}) \Rightarrow wp(T)(\mathcal{R})$.

Thus, S_0 is an upper bound of $\mathcal{T}$. If T_0 is any upper bound of $\mathcal{T}$, we get

- $wlp(T_0)(\mathcal{R}) \Rightarrow wlp(T)(\mathcal{R})$ and
- $wp(T_0)(\mathcal{R}) \Rightarrow wp(T)(\mathcal{R})$

for all $T \in \mathcal{T}$, hence also

- $wlp(T_0)(\mathcal{R}) \Rightarrow \bigwedge_{S \in \mathcal{T}} wlp(S)(\mathcal{R})$ and
- $wp(T_0)(\mathcal{R}) \Rightarrow \bigwedge_{S \in \mathcal{T}} wp(S)(\mathcal{R})$.

It follows that $S_0 \sqsubseteq T_0$, i.e. S_0 is the least upper bound. □

Note that for $\mathcal{T} = \emptyset$ the least upper bound is $fail$. Now we are prepared to present our result concerning the unique existence of a GCS.

Theorem 51 *Let S be an X-operation, $X \subseteq Y$ and $\mathcal{I}$ a static integrity constraint on Y. Then there exists a greatest consistent specialization $S_{\mathcal{I}}$ of S with respect to $\mathcal{I}$. Moreover, $S_{\mathcal{I}}$ is uniquely determined (up to semantic equivalence) by S and $\mathcal{I}$.*

Proof. Let $\mathcal{T}$ be the set of Y-operations T satisfying (in place of $S_{\mathcal{I}}$ the properties (i) – (iii) of Definition 49 and let $S_{\mathcal{I}} = \bigsqcup_{T \in \mathcal{T}} T$. By definition of a least upper bound $S_{\mathcal{I}}$ satisfies properties (i), (ii) and (iv). Property (iii) follows from Lemma 50(i). □

Although Theorem 51 states that there is always a (unique) solution of the integrity enforcement problem, it does not help us in constructing a GCS, since its proof is non-constructive. Another general problem is that there are usually more than just one static integrity constraint. If we successively build GCSs, can we guarantee that the final result will be independent of the order of constraints? Is the final result the same, if we simply take the conjunction of all constraints? We address these two problems next.

Theorem 52 *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be static constraints on Y_1 and Y_2 respectively. If for any operation S the GCS with respect to $\mathcal{I}_i$ is denoted by $S_{\mathcal{I}_i}$ $(i = 1, 2)$, then for any X-command with $X \subseteq Y_1 \cap Y_2$ the GCSs $(S_{\mathcal{I}_1})_{\mathcal{I}_2}$ and $(S_{\mathcal{I}_2})_{\mathcal{I}_1}$ are semantically equivalent.*

Proof. For symmetric reasons it is sufficient to show

$$(S_{\mathcal{I}_1})_{\mathcal{I}_2} \sqsubseteq (S_{\mathcal{I}_2})_{\mathcal{I}_1} \ .$$

We have $\mathcal{I}_2 \Rightarrow wlp((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\mathcal{I}_2)$. Because of the transitivity of the implication $(S_{\mathcal{I}_1})_{\mathcal{I}_2}$ then satisfies properties (i) – (iii) with respect to S and $\mathcal{I}_2$, hence by property (iv) we get $(S_{\mathcal{I}_1})_{\mathcal{I}_2} \sqsubseteq S_{\mathcal{I}_2}$. We have

$$\mathcal{I}_1 \Rightarrow wlp(S_{\mathcal{I}_1})(\mathcal{I}_1) \Rightarrow wlp((S_{\mathcal{I}_1})_{\mathcal{I}_2})(\mathcal{I}_1) \ .$$

The first implication is the consistency of $S_{\mathcal{I}_1}$ with respect to $\mathcal{I}_1$, the second implication follows from property (i) for $(S_{\mathcal{I}_1})_{\mathcal{I}_2}$ with $\mathcal{R} = \mathcal{I}_1$.

Thus, $(S_{\mathcal{I}_1})_{\mathcal{I}_2}$ satisfies properties (i) –(iii) with respect to $S_{\mathcal{I}_2}$ and $\mathcal{I}_1$, i.e.

$(S_{\mathcal{I}_1})_{\mathcal{I}_2} \sqsubseteq (S_{\mathcal{I}_2})_{\mathcal{I}_1}$ by property (iv).

□

Theorem 53 *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be static constraints on Y_1 and Y_2 respectively. If for any operation S the GCS with respect to $\mathcal{I}_i$ is denoted by $S_{\mathcal{I}_i}$ $(i = 1, 2)$, then for any X-command with $X \subseteq Y_1 \cap Y_2$ the GCSs $(S_{\mathcal{I}_1})_{\mathcal{I}_2}$ and $S_{(\mathcal{I}_1 \wedge \mathcal{I}_2)}$ coincide on initial states satisfying $\mathcal{I}_1 \wedge \mathcal{I}_2$, i.e., $\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow (S_{\mathcal{I}_1})_{\mathcal{I}_2}$ and $\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{(\mathcal{I}_1 \wedge \mathcal{I}_2)}$ are semantically equivalent.*

Proof. From the transitivity of the implication and Definition 49 it follows that

$(S_{\mathcal{I}_1})_{\mathcal{I}_2} \sqsubseteq S_{\mathcal{I}_1 \wedge \mathcal{I}_2}$ holds.

Then also

$$\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow (S_{\mathcal{I}_1})_{\mathcal{I}_2} \sqsubseteq \mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{\mathcal{I}_1 \wedge \mathcal{I}_2} \text{ holds.}$$

We have to show the converse. Let

$$\bar{S}_{\mathcal{I}_1} = \mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow S_{\mathcal{I}_1 \wedge \mathcal{I}_2} \boxtimes S_{\mathcal{I}_1} \ .$$

Then we get $wlp(S)(\mathcal{R}) \Rightarrow wlp(\bar{S}_{\mathcal{I}_1})(\mathcal{R})$ and $wp(S)(\mathcal{R}) \Rightarrow wp(\bar{S}_{\mathcal{I}_1})(\mathcal{R})$ for all formulae $\mathcal{R}$ with $fr(\mathcal{R}) \subseteq X$.

Moreover, $\mathcal{I}_1 \Rightarrow wlp(\bar{S}_{\mathcal{I}_1})(\mathcal{I}_1)$ holds, hence by Definition 49 it follows that $\bar{S}_{\mathcal{I}_1} \sqsubseteq S_{\mathcal{I}_1}$.

If we define

$$\bar{S}_{\mathcal{I}_1,\mathcal{I}_2} = \mathcal{I}_2 \rightarrow \bar{S}_{\mathcal{I}_1} \boxtimes (S_{\mathcal{I}_1})_{\mathcal{I}_2} \ ,$$

then by analogous arguments we derive $\bar{S}_{\mathcal{I}_1,\mathcal{I}_2} \sqsubseteq (S_{\mathcal{I}_1})_{\mathcal{I}_2}$. In particular we conclude

$$\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow \bar{S}_{\mathcal{I}_1,\mathcal{I}_2} \sqsubseteq \mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow (S_{\mathcal{I}_1})_{\mathcal{I}_2} \ ,$$

but $\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow \bar{S}_{\mathcal{I}_1,\mathcal{I}_2}$ is semantically equivalent to $\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow \bar{S}_{\mathcal{I}_1}$ and this to $\mathcal{I}_1 \wedge \mathcal{I}_2 \rightarrow (S_{\mathcal{I}_1})_{\mathcal{I}_2}$. Hence the theorem. □

In Section 2.3.1 we required methods on subclasses that override inherited methods to be specializations. Since we regard methods as operations on some state space defined by the schema, the problem occurs, whether the GCS of a specialized operation T of S remains to be a specialization of the GCS of S. Fortunately this is also true.

Theorem 54 *Let $S_{\mathcal{I}}$ be the GCS of the X-operation S with respect to the static integrity constraint $\mathcal{I}$ defined on Y with $X \subseteq Y$. Let T be a Z-operation that specializes S. If $\mathcal{I}$ is regarded as a constraint on $Y \cup Z$, then the GCS $T_{\mathcal{I}}$ of T with respect to $\mathcal{I}$ is a specialization of $S_{\mathcal{I}}$.*

Proof. From the transitivity of the implication and the consistency of $T_{\mathcal{I}}$ with respect to $\mathcal{I}$ it follows that $T_{\mathcal{I}}$ satisfies properties (i) – (iii) of Definition 49, hence $T_{\mathcal{I}} \sqsubseteq S_{\mathcal{I}}$. □

In Section 2.3.1 methods were introduced as deterministic guarded commands. Hence the question whether this property is preserved under GCS construction. Unfortunately this is not true as the following example shows.

Example 8 Let x, y and z be state variables, all of type $FSETS(T)$, where $FSETS(\alpha)$ is the finite set constructor (see e.g. [47]) and T is any value type. Let $X = \{y\}$ and $Y = \{x, y, z\}$. Then we define an X-operation S by

$$S(t :: T) == \ y' :: FSET(T) \mid$$
$$y = Union(y', Single(t)) \wedge member(t, y') = false \rightarrow y := y'$$

and a static constraint $\mathcal{I}$ on Y by

$$x = Union(y, z) \ \wedge \ \forall t :: T.member(t, y) = true \Rightarrow member(t, z) = false \ .$$

Then the GCS $S_{\mathcal{I}}$ has the form

$$
\begin{aligned}
S_{\mathcal{I}}(t :: T) \;==\; & \mathcal{I} \;\rightarrow S(t)\,; \\
& (\, z := Union(z, Single(t)) \,\Box \\
& \quad x' :: FSET(T) \mid x = Union(x', Single(t)) \wedge member(t, x') = false \;\rightarrow \\
& \qquad x \;:=\; x' \,) \,\boxtimes\; S(t)\ .
\end{aligned}
$$

We omit the formal proof of the properties of Definition 49. □

A general approach to remove non-determinism is *operational refinement* as defined in [47]. However, operational refinement allows to "complete" a specification of an operation S whenever S is undefined are never terminating. In this paper we do not regard completion via refinement. Therefore, we regard the notion of specialization instead. Due to [47, Proposition 4.1] it is easy to see that whenever S is consistent with respect to a static constraint $\mathcal{I}$, then each specialization T of S does so, too.

Hence a deterministic specialization of $S_{\mathcal{I}}$ is still a consistent specialization of S with respect to $\mathcal{I}$. The next theorem states that we can choose a maximal one.

Theorem 55 *Let $S_{\mathcal{I}}$ be the GCS of a deterministic operation S with respect to the static constraint $\mathcal{I}$ and let T be some deterministic specialization of $S_{\mathcal{I}}$. Then T specializes S. Moreover, if T' is any deterministic specialization of S that is consistent with respect to $\mathcal{I}$, then T' also specializes some deterministic specialization T of $S_{\mathcal{I}}$.*

Proof. The first statement is trivial, since $\sqsubseteq$ is a partial order.

If T' is a deterministic specialization of S consistent with respect to $\mathcal{I}$, then according to Definition 49 T' must be a specialization of $S_{\mathcal{I}}$, hence the second statement. □

5.2 Enforcing Static Integrity in the OODM

Let us now apply the results of Section 5.1.2 to the OODM presented in Section 2.3. In this section we restrict ourselves to the basic update operations of Theorem 48 and describe the structure of their GCSs with respect to distinguished classes of static constraints. Moreover, we discuss whether the general problem of GCS construction could be reduced to these basic operations. Unfortunately this is not true.

5.2.1 Transforming Static Constraints into Primitive Operations

Let us now try to generalize the result of theorem 48 with respect to explicit static constraints. Let $\mathcal{S}$ be some schema and let $\mathcal{I}$ be an explicit static constraint on $\mathcal{S}$. We want to derive again insert-, delete- and update-operations for each class C in $\mathcal{S}$ such that these are consistent with respect to $\mathcal{I}$. Based on the Conjunctivity Theorem 53 we only approach the problem separately for the classes of constraints introduced in the previous subsection.

Moreover, we apply a specific kind of operational refinement in order to reduce the non-determinism of the resulting GCS. Whenever an arbitrary value of some proper value type is required, then we extend the input-type. However, this can not be applied to reduce the non-determinism arising from choice operations. In general, there exists a *choice normal form* for the GCS such that its components are the maximal deterministic operational refinements of the GCS that exist by Theorem 55.

In the following we restrict ourselves to the case of a single explicit constraint in addition to the one (trivial) uniqueness constraint that is required to assure value-representability and that has been used in [50] to construct *canonical update operations*. Then we look at an example with more than one constraint. We illustrate that although Theorem 53 holds, the structure of $S_{(\mathcal{I}_1 \wedge \mathcal{I}_2)}$ may be not at all obvious.

General (Path) Inclusion Constraints.

Let $\mathcal{I}$ be a general 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 45.

Lemma 56 *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 $\mathcal{D}$*

$$f_1(d_1^{\mathcal{D}}(v_1)) = f_2(d_2^{\mathcal{D}}(v_2)) \;\Leftrightarrow\; c_1(v_1) = c_2(v_2) \tag{5.3}$$

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 14 we may define $f_i = c_i \circ (d_i^{\mathcal{D}})^{\perp 1}$ 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

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

since c_i permutes with $\pi^{\perp 1}$. Then the stated equality follows. □

Now let V_{C_1,C_2} be a subtype of $V_{C_1} \times V_{C_2}$ defined via the equality $f_1(v_1) = f_2(v_2)$, where $v_i :: V_{C_i}$ are the components of a value and f_i are the functions of Lemma 56. We omit the details. Then we can define the new insert-operation on C_1 by $(insert_{C_1})_{\mathcal{I}}(V :: V_{C_1,C_2}) ==$

$$insert_{C_1}(first(V)) \;;\; insert_{C_2}(second(V)) \;. \tag{5.4}$$

Now we are able to generalize Theorem 48 with respect to general inclusion constraint.

Theorem 57 *Let $\mathcal{I}$ be a general inclusion constraint on C_1, C_2 defined via $c_i : T_{C_i} \to T$ and let $S_{\mathcal{I}}$ be the insert-operation of (5.4). Suppose that C_1 is not referenced by C_2. Then $S_{\mathcal{I}}$ is the GCS of the canonical insert-operation of Theorem 48 with respect to $\mathcal{I}$.*

Proof. We use the abbreviations $S_i = insert_{C_i}(V_i :: V_{C_i})$ $(i = 1, 2)$. Then

$$wlp(S_{\mathcal{I}})(\mathcal{R}) \equiv \{V_1/first(V)\}.wlp(S_1)(\{V_2/second(V)\}.wlp(S_2)(\mathcal{R})) \ .$$

Since S_i is total and always terminating, we have $wlp(S_i) = wp(S_i)$. Since C_1 is not referenced by C_2, we know that S_2 is a $\{x_{C_2}\}$-operation. Therefore, $wlp(S_2)(\mathcal{R})$ is a logical combination of $\mathcal{R}$ without any substitution, hence $wlp(S_1)(\mathcal{R}) \Rightarrow wlp(S_{\mathcal{I}})(\mathcal{R})$. This proves (i) and (ii).

In particular $wlp(S_{\mathcal{I}})(\mathcal{I}) \equiv$

$$\{\, x_{C_1}/Union(x_{C_1}, Single(Pair(I_1, V_1))) \,, \\ x_{C_2}/Union(x_{C_2}, Single(Pair(I_2, V_2)))\}.\mathcal{I}$$

with $I_1, I_2 :: ID$ and $V_i :: T_{C_i}$ with $c_1(V_1) = f_1(first(V))$ and $c_2(V_2) = f_2(first(V))$, where f_i are the functions of Lemma 56. Then property (iii) follows immediately. We omit the proof of (iv). □

Note there there is no need to require $C_1 \neq C_2$. Delete- and update-operations can be defined analogously to (5.4). Then a result analogous to Theorem 57 holds. We omit the details here. The generalization to path constraints is also straightforward.

(Path) Functional and Uniqueness Constraints.

Now let $\mathcal{I}$ 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 *quasi-canonical insert* on C [50]. Then we define

$$\begin{array}{l} (insert_C)_{\mathcal{I}}(V :: V_C) == \\ \quad I :: ID \mid I \leftarrow insert'_C(V) \ ; \\ \quad V' :: T_C \mid member(Pair(I, V'), x_C) = true \ \to \\ \quad (\ \forall J :: ID, W :: T_C \,.\, (member(Pair(J, W), x_C) = true \\ \quad \wedge c^1(W) = c^1(V') \ \Rightarrow \ c^2(W) = c^2(V'))\ \to \\ skip \end{array} \tag{5.5}$$

Note that in this case there is no change of input-type.

Theorem 58 *Let $\mathcal{I}$ be a functional constraint on the class C defined via $c^1 : T_C \to T_1$ and $c^2 : T_C \to T_2$ and let $S_{\mathcal{I}}$ be the insert-operation of (5.5). Then $S_{\mathcal{I}}$ is the GCS of the canonical insert-operation on C defined by Theorem 48 with respect to $\mathcal{I}$.*

Proof. The proof is analogous to the one of Theorem 57. □

For delete- and update-operations an analogous result holds. We omit the details. The generalization to path constraints is also straightforward.

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.

(Path) 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 on the insert-operation. Let $\mathcal{I}$ 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 56. Then we define a new insert-operation on C_1 by

$$
\begin{array}{l}
(insert_{C_1})_{\mathcal{I}}(V :: V_{C_1}) == \\
\quad insert_{C_1}(V)\ ; \\
\quad \mu S.\,((I :: ID \mid V' :: T_{C_2} \mid member(Pair(I, V'), x_{C_2}) = true \\
\qquad \wedge c^2(V') = f_1(V) \ \rightarrow\ delete_{C_2}(V')\ ;\ S\ ) \\
\boxtimes\ skip\ )\ .
\end{array}
\tag{5.6}
$$

Theorem 59 *Let $\mathcal{I}$ be an exclusion constraint on the classes C_1 and C_2 defined via $c_i : T_{C_i} \to T$ $(i = 1, 2)$ and let $S_{\mathcal{I}}$ be the insert-operation of (5.6). Then $S_{\mathcal{I}}$ is the GCS of the canonical insert-operation on C_1 defined by Theorem 48 with respect to $\mathcal{I}$.*

Proof. The proof is analogous to the one of Theorem 57. □

For delete- and update-operations an analogous result holds. We omit the details. The generalization to path constraints is also straightforward.

(Path) Object Generating Constraints.

Let $\mathcal{I}$ be an object generating constraint on a class C defined via the functions $c^i : T_C \to T_i$ $(i = 1, 2, 3)$. Then integrity enforcement requires to add additional inserts (deletes, updates) to each insert- (delete-, update-) operation. Let us illustrate the new insert-operation. The handling of delete and update-operations is analogous. As in the case of inclusion constraints we need a preliminary lemma.

Lemma 60 *Let $c^i : T_C \to T_i$ be functions $(i = 1, 2, 3)$ such that $c^1 \times c^2 \times c^3$ defines a uniqueness constraint on the class C. Then there exist functions $f_i : V_C \to T_i$ such that $c^i = f_i \circ c^{\mathcal{D}}$ holds for all instances $\mathcal{D}$, where $c^{\mathcal{D}} : T_C \to V_C$ corresponds to the value-representability of C.*

Proof. Since $c^1 \times c^2 \times c^3$ defines a uniqueness constraint on C, it follows from Definition 14 that there exists some $f : V_C \to T_1 \times T_2 \times T_3$ with $c^1 \times c^2 \times c^3 = f \circ c^{\mathcal{D}}$. Then $f_i = \pi_i \circ f$, where π_i is the projection to T_i $(i = 1, 2, 3)$, satisfy the required property. □

Then we define a new insert-operation on C as follows:

$$
\begin{array}{l}
(insert_C)_{\mathcal{I}}(V :: V_C)\ == \\
\quad \exists I' :: ID, V' :: T_C\,.\,member(Pair(I', V'), x_C) = true \\
\qquad\qquad \wedge\, c^1(V') = f_1(V)\ \rightarrow \\
\quad (\,(\,V'' :: V_C \mid f_1(V'') = f_1(V) \wedge f_2(V'') = f_2(V)
\end{array}
$$

$$\wedge\, f_3(V'') = c^3(V') \;\rightarrow\; insert_C(V'')\,)\,;$$
$$(\,V''' :: V_C \mid f_1(V''') = f_1(V) \wedge f_2(V''') = c^2(V') \wedge$$
$$f_3(V''') = f_3(V) \;\rightarrow\; insert_C(V''')\,)\,)\,;$$
$$\boxtimes\; skip\,)\,;$$

$$insert_C(V) \tag{5.7}$$

Theorem 61 *Let $\mathcal{I}$ be an object generating constraint on a class C defined via the functions $c^i : T_C \rightarrow T_i$ $(i = 1,2,3)$ such that $c^1 \times c^2 \times c^3$ defines a uniqueness constraint on the class C and let $S_{\mathcal{I}}$ be the insert-operation of (5.7). Then $S_{\mathcal{I}}$ is the GCS of the canonical insert-operation on C with respect to $\mathcal{I}$.*

Proof. The proof is analogous to the one of Theorem 57. □

For delete- and update-operations an analogous result holds. We omit the details. The generalization to path constraints is also straightforward.

Theorem 62 *Let $\mathcal{S}$ be a schema such that each class C in $\mathcal{S}$ is value representable. Suppose all explicit constraints in $\mathcal{S}$ are general inclusion constraints, exclusion constraints, functional constraints, uniqueness constraints, object generating constraints and path constraints. Then there exist generic update methods $insert_C$, $delete_C$ and $update_C$ for each class C in $\mathcal{S}$ that are consistent with respect to all implicit and explicit static constraints on $\mathcal{S}$.*

Proof. The result has been shown above in the Theorems 57–61 for a single explicit constraint. Then the general result follows from Theorem 53. □

5.2.2 Transforming Static Constraints into Transactions

Let us now consider the case of arbitrary methods which can be used to model transactions. We concentrate on the question whether it is possible to achieve general consistent methods as combinations of consistent primitive operations as e.g. given by Theorem 62. Regard the following simple example:

Example 9
ACCOUNTCLASS ==
`Structure` TRIPLE(PERSON,NAT,NAT)
`End` ACCOUNTCLASS

`Constraints`
$Pair(I,V) \in$ ACCOUNTCLASS $\Rightarrow second(V) + third(V) \geq 0$
$Pair(I,V) \in$ ACCOUNTCLASS $\wedge\; Pair(I',V') \in$ ACCOUNTCLASS $\wedge$
$first(V) = first(V') \Rightarrow I = I'$

The second component gives the account of a person and the third component gives his/her credit limit. Let

$$transfer(P_1 :: PERSON, P_2 :: PERSON, T :: NAT) ==$$
$$I_1, I_2 :: ID, N_1, N_2, M_1, M_2 :: NAT \mid$$

$$
\begin{aligned}
&Pair(I_1, Triple(P_1, N_1, M_1)) \in \textsc{AccountClass} \wedge \\
&Pair(I_2, Triple(P_2, N_2, M_2)) \in \textsc{AccountClass} \ \rightarrow \\
&update_{AccountClass}(P_1, Triple(P_1, N_1 \perp T, M_1)) \,; \\
&update_{AccountClass}(P_2, Triple(P_2, N_2 + T, M_2))
\end{aligned}
$$

In this case the precondition to be added is simply the weakest precondition, i.e. $N_1 + M_1 \perp T \geq 0$. Now regard the operation

$$S = transfer(P_1, P_2, T_1) \,; transfer(P_2, P_1, T_2) \ .$$

The precondition to be added to S in order to enforce the first constraint simply is $N_1 + M_1 \perp T_1 + T_2 \geq 0 \wedge N_2 + M_2 \perp T_2 + T_1 \geq 0$, which is trivially satisfied if $T_1 = T_2$. However, for large values of $T_1 = T_2$ this condition is weaker then adding three precondition separately. □

Theorem 63 *Let S be a method and let $S_1 \ldots S_n$ be canonical update operations on a schema $\mathcal{S}$ occurring within S. Let S' result from S by replacing each S_i by its GCS $(S_i)_{\mathcal{I}}$ with respect to some static constraint $\mathcal{I}$. Then $S_{\mathcal{I}}$ and S' are in general not semantically equivalent.*

Proof. A counterexample has been given in Example 9. □

As a consequence of this theorem it is even not sufficient to know explicitly the GCS of basic update operations with respect to a single constraint. Although Theorem 53 allows the GCS with respect to more than one constraint to be built successively, we have seen in Theorems 57–61 that the GCS with respect to one constraint is no longer a basic update operation. Let us illustrate this open problem by a simple example.

Example 10 Let C_1, C_2, C_3 be classes and $c_i : T_{C_i} \rightarrow T$ $(i = 1,2,3)$ be functions and suppose there are defined

- a general inclusion constraint $\mathcal{I}_1$ on C_1 and C_2 via c_1 and c_2,
- a general inclusion constraint $\mathcal{I}_2$ on C_1 and C_3 via c_1 and c_3 and
- an exclusion constraint $\mathcal{I}_3$ on C_2 and C_3 via c_2 and c_3.

Clearly, the GCS of the insert-operation on C_1 with respect to $\mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_3$ is *loop*, which is hard to build successively. □

5.3 Enforcing Transition Integrity

In Section 5 we discussed integrity enforcement in a general framework with respect to static constraints. Let us now try to generalize the results for transition constraints. Thus, let S be an X-operation and $\mathcal{J}$ a transition constraint on Y with $X \subseteq Y$. The idea is again to construct a "new" Y-operation $S_{\mathcal{J}}$ that is consistent with respect to $\mathcal{J}$ and specializes S. Again this leads to the idea to construct a *Greatest Consistent Specialization* of S with respect to $\mathcal{J}$. The difference to the case of static constraints is the use of a different proof obligation for consistency according to Definition 4.

5.3.1 GCSs with Respect to Transition Constraints

We start giving a formal definition of a *Greatest Consistent Specialization* (GCS) of an operation S with respect to a transition constraint $\mathcal{J}$.

Definition 64 Let $X \subseteq Y$ be state spaces, S an X-operation and $\mathcal{J}$ a transition integrity constraint on Y. A Y-operation $S_{\mathcal{J}}$ is a *Greatest Consistent Specialization* (GCS) of S with respect to $\mathcal{J}$ iff

(i) $wlp(S)(\mathcal{R}) \Rightarrow wlp(S_{\mathcal{J}})(\mathcal{R})$ holds on Y for all formulae $\mathcal{R}$ with $fr(\mathcal{R}) \subseteq X$,

(ii) $wp(S)(true) \Rightarrow wp(S_{\mathcal{J}})(true)$ holds on Y,

(iii) $wlp(\Phi(\mathcal{J}))(\mathcal{R}) \Rightarrow wlp(S_{\mathcal{J}})(\mathcal{R})$ holds on Y for all formulae $\mathcal{R}$ with $fr(\mathcal{R}) \subseteq Y$ and

(iv) for each Y-operation T satisfying properties (i) – (iv) (instead of $S_{\mathcal{J}}$) we have

  (a) $wlp(S_{\mathcal{J}})(\mathcal{R}) \Rightarrow wlp(T)(\mathcal{R})$ for all formulae $\mathcal{R}$ with $fr(\mathcal{R}) \subseteq X$ and
  (b) $wp(S_{\mathcal{J}})(true) \Rightarrow wp(T)(true)$.

Note that properties (i), (ii) and (iii) say that $S_{\mathcal{J}} \sqsubseteq S$, where $\sqsubseteq$ is the specialization order of Definition 4. Property (iv) requires $S_{\mathcal{J}}$ to be consistent with respect to the constraint $\mathcal{J}$. Finally, property (v) states that each consistent specialization T of S with $T \sqsubseteq S$ also specializes $S_{\mathcal{J}}$.

Based on this formal definition of a GCS we can now raise the same questions as in the static case. Before we state the result on the (unique) existence of GCSs, let us first examine the relation between static and transition constraints. Suppose $\mathcal{I}$ is a static constraint on Y, then we may regard $\mathcal{I}$ also as a transition constraint. Let $\mathcal{I}'$ result from $\mathcal{I}$ by replacing each state variable x_i occurring freely in $\mathcal{I}$ by x_i'. Then $\mathcal{I} \Rightarrow \mathcal{I}'$ defines the *corresponding transition constraint* denoted as $\mathcal{J}_{\mathcal{I}}$. Then we know that consistency with respect to $\mathcal{I}$ is equivalent to consistency with respect to $\mathcal{J}_{\mathcal{I}}$. This implies the following result:

Theorem 65 *Let S be an X-operation and $\mathcal{I}$ a static constraint on Y with $X \subseteq Y$. If $S_{\mathcal{I}}$ and $S_{\mathcal{J}_{\mathcal{I}}}$ are the GCSs of S with respect to $\mathcal{I}$ and the transition constraint $\mathcal{J}_{\mathcal{I}}$ respectively, then these two GCSs are semantically equivalent.*

Proof. This follows directly from Definitions 49 and 64. Using the equivalence of consistency proof obligations with respect to $\mathcal{I}$ and $\mathcal{J}_{\mathcal{I}}$ implies the two definitions to coincide. □

Next, we are able to proof the (unique) existence of a GCS also for transition constraints. As in the static case the proof will be non-constructive, hence does not help to construct a GCS.

Theorem 66 *Let S be an X-operation, $X \subseteq Y$ and $\mathcal{J}$ a transition integrity constraint on Y. Then there exists a greatest consistent specialization $S_{\mathcal{J}}$ of S with respect to $\mathcal{J}$. Moreover, $S_{\mathcal{J}}$ is uniquely determined (up to semantic equivalence) by S and $\mathcal{J}$.*

Proof. The proof is analogous to the one of Theorem 51. □

5.3.2 Compatibility Results

Let us now address the compatibility problems with respect to the conjunction of constraints, inheritance and refinement. Note that due to Theorem 65 some of the results in Section 5.1.2 occur as special cases of the results here.

Theorem 67 *Let $\mathcal{J}_1$ and $\mathcal{J}_2$ be transition constraints on Y_1 and Y_2 respectively. If for any operation S the GCS with respect to $\mathcal{J}_i$ is denoted by $S_{\mathcal{J}_i}$ $(i = 1, 2)$, then for any X-command S with $X \subseteq Y_1 \cap Y_2$ the GCSs $(S_{\mathcal{J}_1})_{\mathcal{J}_2}$ and $(S_{\mathcal{J}_2})_{\mathcal{J}_1}$ are semantically equivalent.*

Proof. The proof is analogous to Theorem 52. □

Theorem 68 *Let $S_{\mathcal{J}}$ be the GCS of the X-operation S with respect to the transition constraint $\mathcal{J}$ defined on Y with $X \subseteq Y$. Let T be a Z-operation that specializes S. If $\mathcal{J}$ is regarded as a constraint on $Y \cup Z$, then the GCS $T_{\mathcal{J}}$ of T with respect to $\mathcal{J}$ is a specialization of $S_{\mathcal{J}}$.*

Proof. The proof is analogous to the one of Theorem 54. □

Theorem 69 *The GCS $S_{\mathcal{J}}$ of a deterministic X-operation S with respect to a transition constraint $\mathcal{J}$ is in general non-deterministic.*

Proof. This follows from Theorem 65, since determinism is not even preserved by GCSs with respect to static constraints as shown by the counter-example in Example 8. □

Hence the problem remains to remove the non-determinism. Due to [47, Proposition 4.2] it is easy to see that whenever S is consistent with respect to a transition constraint $\mathcal{J}$, then each specialization T of S does so, too.

Theorem 70 *Let $S_{\mathcal{J}}$ be the GCS of a deterministic operation S with respect to the transition constraint $\mathcal{J}$ and let T be some deterministic operational refinement of $S_{\mathcal{J}}$. Then T specializes S. Moreover, if T' is any deterministic specialization of S that is consistent with respect to $\mathcal{J}$, then T' also specializes some deterministic operational refinement T of $S_{\mathcal{J}}$.*

Proof. The proof is analogous to Theorem 55. □

An unsolved open problem concerns the generalization to arbitrary dynamic constraints. Lipeck has shown in [37] that dynamic constraints expressed in some generalized propositional temporal logic give rise to transition graphs. It should be possible to derive a suitable proof obligation in the predicate transformer calculus also for such constraints. Then the idea of GCS construction should carry over even to this class of constraints that comprises static and transition constraints.

Chapter 6

Conclusion

In this report we describe results from first investigations in Hamburg and Rostock 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.
- Types are used to structure values. In our approach general algebraic type specifications are allowed including parameterization, subtyping and mutual recursion.
- 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 static and transition constraints to classes and also to a schema. Certain kinds of such constraints can be obtained by generalizing corresponding constraints in the relational model.

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. Assuming that only uniqueness constraints are defined we can show that value-representability is decidable.

An advantage of database systems is to provide generic update operations. We show that the unique existence of such generic operations on some class requires also value-representability. However, in this case referential and IsA integrity can be enforced automatically. From an engineering point of view an algorithm is required to generate these consistent operations. We address this construction problem by the specification of generators for them. These generators will be based on the possibility to represent syntactic components of the language as values within the language itself, which is known to form the basis of *linguistic reflection*. Moreover, the generators involve a single generic proof of correctness hence relieve the user of the burden to write basic update operations and to assure their consistency.

The existence result can be generalized with respect to distinguished classes of explicitly stated static constraints. We show that integrity enforcement is always possible. Given some arbitrary method S and some static or transition constraint $\mathcal{I}$ there exists a *greatest consistent specialization* (GCS) $S_{\mathcal{I}}$ of S with respect to $\mathcal{I}$. Such a GCS behaves nice in that it is compatible with the conjunction of constraints, inheritance and refinement. For the GCS construction of a user-defined operation, however, it is in general not sufficient to replace the involved primitive update operations by their GCSs.

This report is far from giving a complete mathematical foundation of OODB concepts. A lot of problems are still left open and are the matter of current investigations or future research.

- OODBs have been claimed to support engineering applications without proving this. We believe that our approach to types will allow really complex objects to be defined and that the general notion of object will ease the support of versions. In order to support multiple objects in a class the parameterization seems to be a good idea. We shall work on this idea.

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

- In this report we left aside much of the formal semantics which is based on the specification language SAMT. However, there also exist some open problems with this, e.g. a more general approach to transactions without assuming linear order of execution and traces on classes.

- The problem of updatable views is still open. We work on it.

- 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.

- So far, we have shown an existence result for greatest consistent specializations that are used to achieve integrity enforcement. We do not know how to find such a GCS in the general case.

- The axiomatic semantics used 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 or stepwise refinement approaches.

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

Bibliography

[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] H. Aït-Kaci: *An Overview of LIFE*, in J. W. Schmidt, A. A. Stognij (Eds.): Proc. Next Generation Information Systems Technology , Springer LNCS, vol. 504, 1991, pp. 42 – 58

[5] 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

[6] 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

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

[8] 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

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

[10] 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

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

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

[13] 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

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

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

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

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

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

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

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

[35] 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

[36] P. Lockemann, J. W. Schmidt: *Datenbankhandbuch*, Springer, 1987

[37] U. W. Lipeck: *Dynamische Integrität von Datenbanken* (in German), Springer IFB 209, 1987

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

[39] F. Matthes, J. W. Schmidt: *The Type System of DBPL*, in Type Systems and Database Programming Languages, University of St. Andrews, Dept. of Mathematical and Computational Sciences, Research Report CS/90/3, 38 – 44

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

[41] 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

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

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

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

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

[46] K.-D. Schewe, J. W. Schmidt, I. Wetzel, N. Bidoit, D. Castelli, C. Meghini: *Abstract Machines Revisited*, FIDE Technical Report 1991/11, February 1991

[47] 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)

[48] K.-D. Schewe, J. W. Schmidt, I. Wetzel: *Specification and Refinement in an Integrated Database Application Environment*, Proc. VDM 91, Noordwijkerhout, October 1991

[49] 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

[50] 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

[51] K.-D. Schewe, B. Thalheim, J. W. Schmidt, I. Wetzel: *Integrity Enforcement in Object-Oriented 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

[52] K.-D. Schewe, J. W. Schmidt, D. Stemple, B. Thalheim, I. Wetzel: *Generating Methods to Assure Global Integrity*, submitted 1992

[53] 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. 13, 1992

[54] K.-D. Schewe: *Class Semantics in Object Oriented Databases*, submitted 1992

[55] K.-D. Schewe, B. Thalheim, I. Wetzel: *Integrity Preserving Updates in Object Oriented Databases*, in M. Orlowska (Ed.) : Proc. Australian Database Conference, Brisbane, February 1993 (to appear)

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

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

[58] D. Stemple, T. Sheard: *A Recursive Base for Database Programming Primitives*, in Proceedings of the First International East/West Database Workshop, Kiev, October 1990

[59] D. Stemple, T. Sheard: *Exceeding the Limits of Polymorphism*, in Proc. EDBT '90

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

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

[62] K. Subieta: *A Persistent Object Store for the LOQIS Programming System*, to appear in: International Journal of Microcomputer Applications

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

[64] 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