Extensible Safe Object-Oriented Design of Database Applications**

Klaus–Dieter Schewe, Bernhard Thalheim+
Ingrid Wetzel, Joachim W. Schmidt

Hamburg University, Dept. of Computer Science,
Schlüterstr. 70, D-W-2000 Hamburg 13
schewe@dbis1.informatik.uni-hamburg.de

+Rostock University, Dept. of Computer Science,
A.-Einstein-Str. 21, D-O-2500 Rostock
thalheim@informatik.uni-rostock.dbp.de

March 4, 1994

Abstract

The formal foundation of object-oriented databases is still an open problem. In this paper a formalization of kernel object-oriented design concepts is achieved by an automatizable transformation into a specification language with a clear mathematical semantics. This approach allows object-oriented design to be *extensible*.

The preservation of advantages of relational databases requires the definability of generic operations for the insertion, deletion and update of single objects. It is shown that such generic operations can only be defined for value-representable classes. The possibility to detect value-representability contributes to the *safety* for object-oriented design.

1 Introduction

The shortcomings of the relational database approach encouraged much research aimed at achieving more appropriate data models. There has been a tremendous interest in *semantic data models* which emphasize on natural mechanisms and constructs for the description of relationships between stored data. In the last decade several models such as SDM [HaMc81], TAXIS [MBW80], SAM* [Su83] and IFO [AbHu87] were proposed primarily as schema design tools. Their usefulness for this task as well as their fundamentals are widely accepted. Similar structuring mechanisms are used by *knowledge representation systems* such as KL-ONE [BrSc85], BACK [LNPS85], TELOS [MBJK90] or CLASSIC [BBMR89] which put additional emphasis on inferences on such structures.

Recently, it has been claimed that the *object-oriented approach* to be the key technology for future database systems and languages. Several systems such as EXODUS [CDV88], VISION [CaSc87], IRIS [FBC+87], Gemstone [MSOP86], ORION [KBB+88] and O_2 [BBB+88]

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

arose from these efforts; current research in databases often addresses object-orientation, e.g. [AGO91]. However, in contrast to research in the relational area there is no common formal agreement on what constitutes an object-oriented database [Bee90].

In this paper we outline a safe and extensible way to design an object-oriented database while supporting general design concepts as described in [ABD+90]. One important concept of object-oriented databases is *object identity*. Following [Abi90, BeKo90] 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 [Oho90]. On the other hand, the handling of such identifiers is itself a database problem [LoSc87], since the user can only access those objects in the database that can be completely described by values.

The generic querying of objects has been approached in [Abi90, BeKo90]. 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 [SaJu91] have chosen the solution to abandon generic operations. Others [AGO91, AGO91a, BBB+88] 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 call such classes *value-representable*. In our approach these classes are accompanied by a collection of generic operations that need not to be specified by the user. Furthermore, inclusion dependency and referential integrity are enforced by these operations. This justifies the use of the notion "safety".

We introduce a kernel high-level language featuring types and classes together with separate hierarchies on both. Classes denote sets of objects, and objects themselves are encoded via an object identifier and values of that object belonging to a collection of types or referring to other objects. Recently, the necessity of a separate mechanism for the handling of relationships was identified [AGO91a]. We allow relationships between classes to be represented in the same manner as classes with references to other classes, which allows in turn the modelling of higher-order relationships [Tha90].

Representing this kernel language in the formal specification language SAMT [SWS91] the formal foundation of SAMT, i.e., the combination of algebraic type specification and structured substitution-based module specification, carries over to object-oriented design. On the other hand SAMT offers a formally based notion of refinement [SWS91] which enables the derivation of different implementations in different target languages.

Extensibility can be achieved in our approach since additional features of high-level object-oriented design languages can also be defined by SAMT representations. Figure 1 gives an overall picture of our approach.

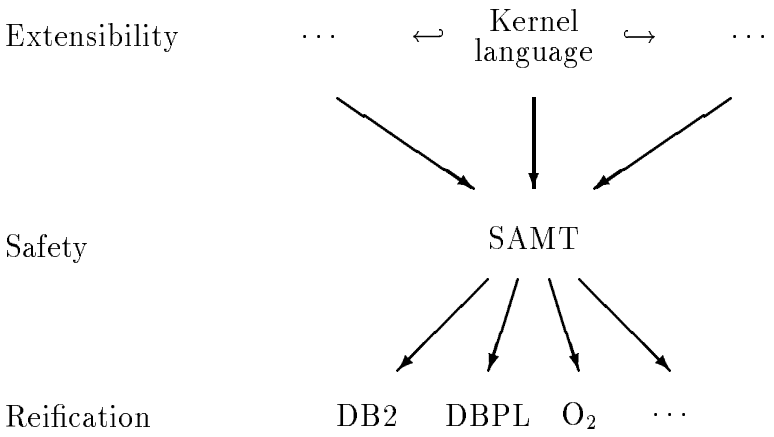

Figure 1: The general modeling approach

In section 2 we describe the kernel components of object-oriented design and illustrate the formal representation of the outlined high-level object-oriented design components together with the derived operations in SAMT. In section 3 we discuss the problem of value-representability and its relation to the automatic derivation of generic operations. In section 4 we outline the capability of extending the kernel design language on the basis of SAMT.

2 Formalization of Object-Oriented Conceptual Design

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 classified 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 [Bee90], objects have to be encoded by object identifiers regardless of the content, location or addressability [KhCo86]. In our approach each object consists of a unique, immutable *identifier*, a set of values of different types called the *value-types* of the object, *references* to other objects and *operations(methods)* associated with the object. This is formally expressed by

$$o \quad = \quad (i, \{v_j, T_j\}, \{o_k\}, \{op_l\})$$

for an object o with identifier i, values v_j of types T_j, references to objects o_k and operations op_l.

The class concept provides the classification 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 a *type declaration* and a *class definition*.

2.1 Type Declarations

The high-level type language provides STRING, NAT, INT as *basic types* with the usual operations on them. *Enumerated* and *subrange* types can also be defined as usual.

Basically, only two type constructors, the *tagged tuple* constructor denoted by $(\cdot)$ and the *set* type constructor denoted by $\{\cdot\}$, are allowed.

Example 1 *Both type constructors are used in the declaration for PERSONNAME:*

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

Example 2 *The definition for a type PERSON uses the type PERSONNAME and a type ADDRESS defined elsewhere:*

$$\begin{array}{lcll} PERSON & = & (\ PersonIdentityNo & : NAT, \\ & & \ Name & : PERSONNAME, \\ & & \ Address & : ADDRESS\ ) \end{array}$$

In SAMT a type is given by a triplet $T = (S, F, Ax)$, where S denotes the structure of the type T, i.e. the generators of T as a set, F denotes the functions on T including algebraic operations, and Ax is a collection of axioms on S and F [SWS91]. This approach is similar to algebraic specifications.

SAMT types can be used for an easy representation of tuple types such as *PERSONNAME* and *PERSON*, the tags in which give the names of selector functions:

Example 3 (SAMT representation) *Suppose the data type STRING is defined elsewhere. Then the names of persons can be defined by the following type.*

PERSONNAME ==
  `BasedOn` *STRING;*
  `Constructors PName` *: FirstName : STRING* $\times$ *LastName : STRING* $\times$
    Titles : FSETS*(STRING)* $\rightarrow$ *PERSONNAME ;*
`End` *PERSONNAME*

In this example we use the parameterized type FSETS(α) to represent the type constructor $\{\cdot\}$ of our high level kernel language and to denote the type of finite subsets for any type α. A SAMT definition of FSETS is given in [SWS91]. The representation of the type *PERSON* can be achieved analogously.

2.2 Subtyping

Types can be organized in a type hierarchy. The notion of subtype in SAMT is defined by the existence of a coercion function mapping the subtype to its supertype. In the high-level kernel language subtyping on tuples uses canonical projection as the coercion function. Subtyping of set types depends on the subtyping of the member types [AGO91].

In addition, we use concatenation on tuple types denoted by the $\circ$-operator.

Example 4 *The following example defines* STUDENT *as a subtype of* PERSON*:*

$$STUDENT \quad = \quad PERSON \circ (\ StudNo : NAT,\ Faculty : NAT\ )$$

Example 5 (SAMT representation) *The type* STUDENT *can be defined as a subtype of the type* PERSON*. In this example we use a function "student_proj_person" taking the subtype STUDENT to the supertype PERSON:*

STUDENT ==
  `SubtypeOf` *PERSON Via student_proj_person ;*
  `BasedOn` *NAT ; STRING ;*
  `Constructors` Student *: PersonIdentityNo : NAT* $\times$ *Name : PERSONNAME* $\times$
    Address : ADDRESS $\times$ *StudNo : NAT* $\times$ *FacultyNo : NAT* $\rightarrow$ *STUDENT ;*
  `Axioms` *With S :: STUDENT :*
    student_proj_person(S) = Person*(PersonIdentityNo(S),Name(S),Address(S)) ;*
`End` *STUDENT*

In [SWS91] we provided examples of more sophisticated subtypes in SAMT using non-trivial coercion functions.

2.3 Class Definitions

Objects are grouped into classes. Classes denote mutable sets of objects with values of the same type and the same set of operations.
In addition to having values an object may refer to another object. For instance, a person may refer to his/her spouse if this person is married. References can be modelled at the class level. Therefore, we extend the class definition in accordance to the type definition. This means that instead of giving a simple value-type expression for a class, we use a structure expression built analogously. Not only the type names but also class names are allowed in such expressions. Class names occurring in structure expressions are interpreted as references. Tags preceding class names are interpreted as reference names. The subscript $\perp$ can be used to indicate references that are only partial.

Example 6 *The class* PERSONS *with an additional reference to the spouse of a person – if it exists – can be defined as follows:*

$$\text{PERSONS} = \text{CLASS } PERSON \circ (\text{Spouse} : \text{PERSONS}_{\perp})$$

Classes can be represented by state variables of persistent SAMT modules [SWS91]. The types of these variables are defined as partial functions from a system-provided unique identifier type *ID_TYPE* either to some other value type, to the identifier type in case of references, or to type expressions involving both these possibilities.

Value types have been specified by the type declaration in the kernel language. References to other classes give rise to additional constraints to be represented by static invariants. The domain of the reference variable is included in the domain of the class variable in case of partial references or otherwise even equal. The range of the reference is included in the domain of the corresponding class.

Example 7 (SAMT representation) *The following part of a module* PERSONMODULE *represents the class* PERSONS*:*

VARIABLES *Persons* :: PFUN(*ID_TYPE,PERSON*)
Spouse :: PFUN(*ID_TYPE,ID_TYPE*)
INVARIANT
STATIC *With I :: ID_TYPE :*
member(I,domain(Spouse)) = true $\Rightarrow$ *member(I,domain(Persons)) = true ;*

2.4 IsA-Relationships

Classes may overlap allowing objects belonging to several classes to have values of different types. Classes may include other classes which is expressed by an IsA-relationship.

Example 8 *The class* STUDENTS *shows the use of an IsA-relationship:*

STUDENTS = CLASS *STUDENT* ISA PERSONS

For the SAMT representation of subclasses we exploit submodules. Suppose we are given two classes and an IsA-Relation between them. This means that the objects in the subclass form a subset of the objects of the superclass. Since we assume a uniform identifier type for all objects, this means to require an injection on the corresponding sets of identifiers.

The value type of a subclass need not be a subtype of the superclass' value type. However, if it is then an additional invariant appears. We must guarantee that the value associated with an object in the subclass as a member of the superclass via the subtyping function is identical to its value as a member of the superclass via the identifier inclusion. This relates IsA-relationships on classes with the subtype concept of SAMT.

Example 9 (SAMT representation) *For example, the class* STUDENTS *will be represented by the following part of a persistent module* STUDENTMODULE*:*

INHERITS PERSONMODULE
VARIABLES *Students* :: PFUN(*ID_TYPE,STUDENT*)
INVARIANT
STATIC With I :: ID_TYPE :
member(I,domain(*Students*)) = true $\Rightarrow$ member(I,domain(*Persons*)) = true $\wedge$
stud_proj_person(*Students*(I)) = *Persons*(I) ;

The representation of the classes PROFESSORS and SUPERVISORS can be achieved analogously.

2.5 Constraints

If each object belonging to a class is identifiable by the value it is related to in that class we denote this property by using the keyword UNIQUE in that class definition.

References can be used in the *UNIQUE*-construct using the reference tag. This forces the object-identifier to depend not only on the object's value, but also on its references to other objects. Classes with references can be used for the representation of relationships as shown in example 10.

Example 10 *A supervisor class is defined as a relationship between professors and sets of students with the time interval of the supervisorship as attribute. Therefore we obtain the following definition:*

SUPERVISOR = CLASS (*Prof :* PROFESSORS,
Studs : { (*Stud :* STUDENTS) $\circ$ *(From : DATE, Till : DATE)* })
*UNIQUE(*Studs*)*

Each UNIQUE constraint defines a substructure $U(exp)$ derived from the structure expression exp used in the class definition. $U(exp)$ can be defined as follows:

- If none of the tags t_i in some expression exp occurs in the UNIQUE constraint, then we have $U(exp) = ()$, where $()$ denotes the empty tuple type.
- If t occurs in the UNIQUE constraint, then $U((t : exp)) = exp$.
- In all other cases we have $U((t_1 : exp_1, \ldots, t_n : exp_n)) = U((t_1 : exp_1)) \circ \ldots \circ U((t_n : exp_n))$, $U((t : exp)) = (t : U(exp))$ and $U(\{exp\}) = \{U(exp)\}$.

UNIQUE constraints require the partial function representing the class and its references in SAMT to be injective.

Example 11 (SAMT representation) *UNIQUE(Studs) on the class* SUPERVISORS *can be represented as follows:*

INVARIANT
STATIC *With I,J :: ID_TYPE :*
member(I,domain(Supervisors)) = true $\wedge$ *member(J,domain(Supervisors)) = true* $\wedge$
Studs(I) = Studs(J) $\Rightarrow$ *I = J* ;

Additional semantics can be captured if we can define a key on the set of object values belonging to each class. This property is denoted by VALUE-KEY followed by a sequence of value tags. If a class is unique and has a value-key, then objects of this class can be identified by their values of the value-keys.

Example 12 *Extend the class* PERSONS *by additional constraints.*

PERSONS = CLASS *PERSON* $\circ$ (Spouse : $\text{PERSONS}_{\perp}$)
UNIQUE(PERSON) VALUE-KEY(PersonIdentityNo)

VALUE-KEY gives the usual formulation of key on the range of the partial function representing the class.

Example 13 (SAMT representation) *VALUE-KEY(PersonIdentityNo) on* PERSONS *can be expressed in* PERSONMODULE *as follows:*

INVARIANT
STATIC *With* P_1, P_2 *:: PERSON :*
member(P_1,*range(PersonClass)) = true* $\wedge$ *member*(P_2,*range(PersonClass)) = true* $\wedge$
PersonIdentityNo(P_1) = *PersonIdentityNo*(P_2) $\Rightarrow$ $P_1 = P_2$;

3 Safety

In the tradition of value-based database programming *generic operations* for insertion, deletion, update and querying have been provided. The benefits of object identification in the object-oriented approach should not force us to go beyond this capability. Hence the need to provide operations for creating, deleting, updating and querying objects. Queries are per se set-oriented, hence their generalization to object-oriented databases does not cause problems with object identifiers [Abi90, BeKo90]. Therefore, we concentrate on the other operations. We show that generic operations can only be defined unambigously on classes the objects in which can be uniquely identified by values. Subsequently, we call this property value-representability.

3.1 Value Representability

In our approach objects are encoded in the database by their immutable identifiers. The concept of object identifiers has the advantage that these can be provided by the system and hidden to the user. However, problems arise if we allow different objects to share the same values and references.

In this case some objects may not be completely updateable and accessible, e.g., if the different objects o_1 and o_2 with different identifiers i_1 and i_2 share the same values and references, it is not a priori decidable what is meant by a delete- or update-operation. Shall we delete (update) both objects, none of them or arbitrarily select one of them for deletion (update)? The same applies to insert-operations. If we want to insert an object o with given values and references to existing objects and there exists already an object with the same values and references, shall we introduce another copy and create a new identifier, or reject the insert? Moreover, how shall we detect the objects referred to, if the user provides only values?

Example 14 *We can define a class of books the objects in which are not distinguishable by their values.*

BOOKS = CLASS *Book : (Author: STRING , Title : STRING , Price : NAT)*
∘ (Owner : PERSONS)

Objects of the class BOOKS can not be uniquely identified. In this case, however, the problem can be resolved if we add the constraint *UNIQUE(Book,Owner).*

Several solutions are possible:

- Abandon generic operations as in [SaJu91],
- use surrogate-keys to represent object identity as in [AGO91, AGO91a, BBB+88] or
- characterize those classes which can be represented by values.

We follow the third approach. Generally, we distinguish between *object-oriented* and *value-oriented* databases depending on whether they embody the concept of object identifiers or not. Among object-oriented databases we can distinguish [AlTh90] three different kinds:

Value-based databases All objects are *value-identifiable*, i.e. can be identified by values of their value types.

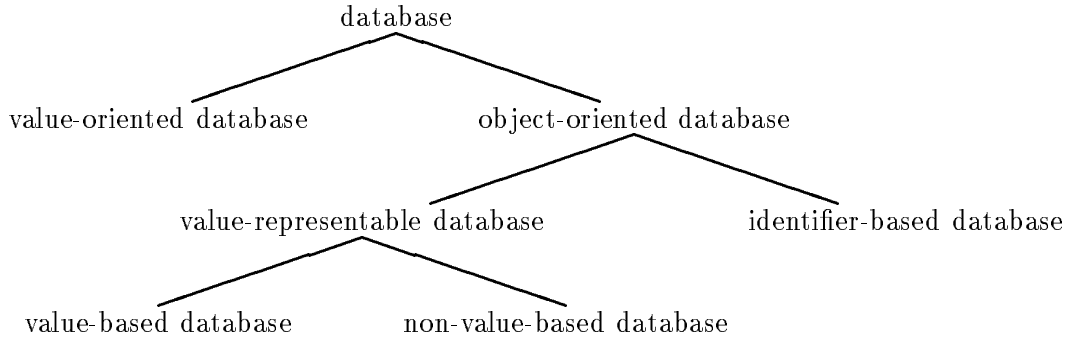

Figure 2: Classification of databases

Value-representable databases All objects are *reference-identifiable*, where reference-identifiability can be recursively defined as follows:

- Each value-identifiable object is also reference-identifiable.
- If an object is identified by values of its value-type and references to a set of objects such that each object in this set is reference-identifiable, then the object is itself reference-identifiable.

Identifier-based databases There are objects which are not reference-identifiable.

Figure 2 illustrates these different classes of databases.

Now we show how value-representability can be detected for *simple schemas* defined in the kernel language. A *simple schema* is a collection of class definitions $C_1, ..., C_m$ with constraints only defined by UNIQUE, VALUE-KEY and IsA.

For a given simple schema we construct *a value-identification type* $F(C)$ and a *value-representation type* $G(C)$ for all the classes C. Then a value-representable class C is characterized by $F(C) \neq ()$.

$F(C)$ can be used to identify the objects in this class by values. $G(C)$ can be used to provide the necessary values for insert- and update-operations.

Now let exp_C denote the expression used to define the class C and let $U(exp_C)$ denote the subexpression defined by a UNIQUE-constraint on C provided it exists, otherwise let $U(exp_C) = ()$. Then the value-representability of a given schema can be detected using the following algorithm.

Algorithm A

INPUT: *A set of class definitions* $\{C_1, ..., C_m\}$.

`Step 0.` *Let* $F(C_i) = ()$ *and* $G(C_i) = exp_{C_i}$ *for* $1 \leq i \leq m$.

`Step 1.` *For all classes* C_i *such that* $U(exp_{C_i})$ *is a type expression define* $U(exp_{C_i})$.

`Step 2.` *If* C_i *is a class with* $F(C_i) = ()$ *and* $F(D_j) \neq ()$ *for all classes* D_j *occurring in* $U(exp_{C_i})$ *let* $F(C_i) = U(exp_{C_i})$ *and replace all occurrences of classes* D_j *in* $F(C_i)$ *and* $G(C_i)$ *by* $F(D_j)$ *and* $G(D_j)$ *respectively.*

Step 3. *Repeat step 2 until it is no longer applicable.*
Step 4. *For all classes* C_i *such that* $F(C_i) \neq ()$
replace each occurence of a class D_j *in* $G(C_i)$ *by* $F(D_j)$
OUTPUT: *Classes* C_i *with their value-identification type* $F(C_i)$ *and their value-representation type* $G(C_i)$.

3.2 Definability of Generic Operations

The requirement that object identifiers are to be hidden from users imposes restrictions on the definability of generic operations. In case objects can not be distinguished by their values and references to other objects it is not possible to provide all these operations automatically. On the basis of algorithm A, however, we can decide whether generic functions are definable. Obviously, the functions **insert_object**, **delete_object** and **update_object** are definable for a class C iff C is value-representable. In this case we can use the *value representation type* of the class C and the *value identification type* of C to derive the generic operations.

In general we need operations of the following kind:

- **insert_object**:
  This operation takes a value of the *value representation type* of the corresponding class as input-parameter. A new identifier for this object is provided by the system.
- **delete_object**:
  This operation takes a value of the *value identification type* of the corresponding class as input-parameter such that the object in question can be identified.
- **update_object**:
  This operation also takes values of the *value identification type* and of the *value representation type* of the corresponding class as input-parameter such that the object in question can be identified and its value can be changed.

The dynamics of SAMT is described by Dijkstra's guarded commands [DiSc89]. Since Dijkstra's calculus is computationally complete, the representation of generic operations by transactions in persistent SAMT modules is easy.

Example 15 *Let the class* PERSONS *be defined without the* Spouse *reference. The insertion of a new person into this class can be expressed as follows:*

TRANSACTIONS
insert_Person(P :: PERSON) =
IF (member(P,range(*Persons*)) = false $\longrightarrow$
@ I :: *ID_TYPE*
(*member*(*I*, *domain*(*Persons*)) = false $\longrightarrow$
Persons := union(*Persons*,singleton(Pair(I,P)))))

The @-substitution used in this example denotes an arbitrary selection of a new identifier. A more formal introduction to this construct is given in [SSW+91a].

In case of integrity constraints these operations have to be augmented to enforce the constraints. For instance, IsA-relationships impose the following constraints on generic operations:

- Each insert into a subclass must include an insert into its superclass, e.g., an insertion of a student must include also a person insert.
- Each delete on the superclass must include a delete operation on the subclass.
- Update operations on subclasses (superclasses) may enforce updates on superclasses (subclasses).

In case of an insert this means to provide as additional input the required data of the superclass. This can be easily expressed exploiting the SAMT subtype facility, thus to compute these data via the subtype coercion function.

Example 16 (SAMT representation) *The insertion of a new student into the class* STUDENTS *requires the insertion of a new person in the class* PERSONS*:*

TRANSACTIONS
insert_Student(S :: STUDENT) =
IF (member(stud_proj_person(S),range(*Persons*)) = false $\longrightarrow$
@ I :: *ID_TYPE*
(*member*(*I*, *domain*(*Persons*)) = false $\longrightarrow$
Students := union(*Students*,singleton(Pair(I,S))) ||
Persons := union(*Persons*,singleton(Pair(I,stud_proj_person(S)))))

In addition there must exist operations to make an existing object in the superclass also a member of the subclass, e.g. *make_Person_a_Student*, which generalizes the required insert operations. The same applies in the opposite direction, e.g., if we want a student in the database, who has finished his/her studies, to remain in the database simply as a person.

References also impose constraints on generic operations:

- Each insert into a class must include an insert into the referenced class, provided the referenced object does not exist already. It may also include an update on the referenced object. E.g., an insertion of a person may include another person insert, the person's spouse. It may also include adding a reference from an already existing (unmarried) person to the inserted person.
- Each delete on a class must include delete operations on referenced classes. It must also include delete- or update-operations on objects referencing to the deleted object.
- Update operations on classes require updates, deletes or inserts on all other classes referenced by or referencing to the class in question.

The necessary values to enforce referential integrity are included in the value representation type of a class.

The representation in SAMT can be done in the same manner as for IsA-relationships. provided there is no cyclic reference. However, if there is one, generic operations become more complex since the necessary input should be given as a *rational tree*, i.e. a finite or infinite tree which has only finite number of subtrees. Such data structures can be represented e.g. by ψ-terms [Ait90] or as values of recursive SAMT types.

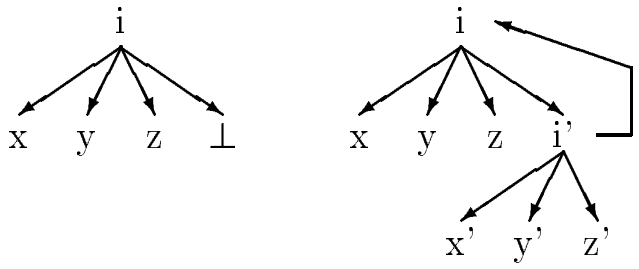

Figure 3: The graphs of the class PERSONS

3.3 Extensions

So far, for the detection of value-representability only simple schemas were considered. However, it is possible to enhance the computation of a class' value-representation by using semantic constraints.

Example 17 *Let the UNIQUE-constraint on the class* PERSONS *in example 12 be changed to* UNIQUE(PERSON,Spouse). *In this case* $F(\text{PERSONS}) = ()$. *Using the additional constraint* Spouse(x) = y $\Rightarrow$ Spouse(y) = x *the class* PERSONS *gets value-representable, since each person is either not married and represented by its value of type* PERSON *or is married and represented by its value of type* PERSON *together with the value of his/her spouse. The possible structures of objects in this class are illustrated in figure 3.*

Generally, if in a schema several semantic constraints are defined, then for value representability of a class cycles of references need to be considered. Given a cycle from a class C to itself by the reference function f, we can distinguish several cases:

- $\forall x.\ \exists n \neq n'.\quad f^n(x) = f^{n'}(x') \vee f^n(x) = \bot$
- $\exists n.\ \forall x.\quad f^n(x) = \bot \vee f^n(x) = x$

In the second case, the value-representability can be derived as in example 17. In the first case, the value-representability is based on the finiteness of the database. In this case the value representation type is a *rational tree*, i.e. a finite or infinite tree which has only finite number of subtrees. Such data structures can be represented e.g. by ψ-terms [Ait90] or as values of recursive SAMT types. Consequently the derived generic operations become more complex.

4 Extensibility

The kernel language can be seen as a short notation of SAMT constructs. Let us now indicate some extensions to this language by other constructs that are also expressible in SAMT. This style of extending the kernel language has the advantage that all new constructs are well-founded. We show this for some static and for one dynamic integrity constraint. In the same manner, new types and complex transactions can be introduced.

Static constraints in databases can be distinguished according to their abstraction level and their importance for values and objects [Tha88]. Now we show how integrity constraints can be used in an extended kernel language.

Example 18 *Given the two classes* PROFESSORS *and* STUDENTS. *If objects representing professors and objects representing students have to be mutually different then we express this by the* exclusion dependency*:*

$$\text{PROFESSORS} \parallel \text{STUDENTS} .$$

This integrity constraint is directly represented in SAMT by the following static invariant:

INVARIANT
  STATIC *With I ::* ID_TYPE *:*
    $member(I,domain(Professors)) = true \Rightarrow member(I,domain(Students)) = false$;
    With I :: ID_TYPE *:*
    $member(I,domain(Students)) = true \Rightarrow member(I,domain(Professors)) = false$;

Example 19 *Given the two classes* PROFESSORS *and* STUDENTS *which are subclasses of the class* PERSONS. *If person represented in the database is a professor or a student then we can express this by the cover constraint:*

$$\text{PERSONS} = \text{PROFESSORS} \bigcup \text{STUDENTS},$$

which can be represented in SAMT by the following static constraint:

INVARIANT
  STATIC *With I ::* ID_TYPE *:*
    $member(I,domain(Persons)) = true \Rightarrow$
      $member(I,domain(Students)) = true \vee member(I,domain(Professors)) = true$;

Let us define another constraint restricting the migration of objects from one class to another.

Example 20 *Given the two classes* PROFESSORS *and* STUDENTS *which are subclasses of the class* PERSONS. *Students can become professors but professors can not become students. Therefore, we restrict the scheme by the integrity constraint*
  ImpossibleMake(PROFESSORS, STUDENTS) .
This integrity constraint is directly represented in SAMT by the following transition constraint.

INVARIANT
  TRANSITION *With I ::* ID_TYPE *:*
    $member(I,domain(Professors)) = true \wedge member(I,domain(Students)) = false$
    $\Rightarrow member(I,domain(Students')) = false$;

5 Conclusion

In this paper we propose kernel constructs of a high-level strongly-typed language for object-oriented database design. Schemata written in this kernel language can be transformed automatically into representations in the formal specification language SAMT, a strongly-typed modular language supporting general subtyping and module inheritance within a uniform formal framework. This transformation process includes the derivation of generic operations on classes enforcing inclusion dependency and referential integrity. As a consequence the object-oriented design inherits the formal semantics of SAMT. Other benefits of the SAMT representation are seen in the possibility to prove transaction safety and the reification dimension provided by SAMT's refinement capability that does not depend on a specific target database language or DBMS.

Moreover, we discuss the problems of value representability of object-oriented designs. The question whether a class can be completely characterized by values or not is crucial for avoiding ambiguities in object definitions. On the basis of our kernel language we describe an algorithm for the detection of value representability. Thus, safety in our approach is achieved in several ways:

- by its foundation on a uniform formal framework,
- by the detection of value representability, and
- by consistency proof obligations for transaction safety.

Another advantage of this approach is its extensibility. SAMT can be used to meet all the requirements for object-oriented databases [ABD+90] except those that can only be solved at the level of implementation. This can be exploited to extend the outlined kernel language without losing any of its advantages. We demonstrate this feature by the introduction of additional static and dynamic constraints. Moreover, we discusse the increase in operation complexity due to such extensions.

The outlined approach combines functional, imperative and object-oriented concepts in a uniform formal framework (see also [Ait90a]). The specification of operations in SAMT types provides the functional part, operations in modules support the imperative style, and the encapsulation feature of modules enables object-orientation.

References

[Abi90] S. Abiteboul: *Towards a deductive object-oriented database language*, Data & Knowledge Engineering, vol. 5, 1990, pp. 263 – 287

[AbHu87] S. Abiteboul, R. Hull: *IFO: A Formal Semantic Database Model*, ACM ToDS, vol. 12 (4), December 1987, pp. 525 – 565

[Ait90] 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

[Ait90a] H. Aït-Kaci: *A Glimpse of Paradise*, in J. W. Schmidt, A. A. Stognij (Eds.): Proc. Next Generation Information Systems Technology , Springer LNCS, vol. 504, 1991, pp. 17 – 25

[AGO91] A. Albano, G. Ghelli, R. Orsini: *Objects and Classes for a Database Programming Language*, FIDE technical report 91/16, 1991

[AGO91a] 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

[AlTh90] S. Al Fedaghi, B. Thalheim: *Fundamentals of databases - The key concept*, submitted for publication, 1990

[ABD+90] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, S. Zdonik: *The Object-Oriented Database System Manifesto*, SIGMOD 1990

[BBB+88] 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

[Bee89] C. Beeri: *Formal Models for Object-Oriented Databases*, Proc. 1st DOOD 1989, pp. 370 – 395

[Bee90] C. Beeri: *A formal approach to object-oriented databases*, Data and Knowledge Engineering, vol. 5 (4), 1990, pp. 353 – 382

[BeKo90] C. Beeri, Y. Kornatzky: *Algebraic Optimization of Object-Oriented Query Languages*, in S. Abiteboul, P. C. Kanellakis (Eds.): Proceedings of ICDT 90, Springer LNCS 470, pp. 72 – 88

[BBMR89] A. Borgida, R. J. Brachmann, D. L. McGuiness, L. A. Resnick: *CLASSIC: A Structural Data Model for Objects*, Proc. ACM SIGMOD, June 1989, pp. 59 – 67

[BrSc85] R. J. Brachmann, J. G. Schmolze: *An Overview of the KL-ONE Knowledge Representation System*, Cognitive Science, vol. 9 (2), April 1985, pp. 171 – 216

[BrWe87] K. B. Bruce,P. Wegner: *An Algebraic Model of Subtype and Inheritance*, in F. Bancilhon, P. Buneman (Eds.): *Workshop on Database Programming Languages*, Roscoff, France, September 1987

[CDV88] M. Carey, D. DeWitt, S. Vandenberg: *A Data Model and Query Language for EXODUS*, Proc. ACM SIGMOD 88

[CaSc87] M. Caruso, E. Sciore: *The VISION Object-Oriented Database Management System*, Proc. of the Workshop on Database Programming Languages, Roscoff, France, September 1987

[DiSc89] E. W. Dijkstra, C. S. Scholten: *Predicate Calculus and Program Semantics*. Springer-Verlag, 1989.

[FBC+87] D. Fishman, D. Beech, H. Cate, E. Chow et al.: *IRIS: An Object-Oriented Database Management System*, ACM ToIS, vol. 5(1), January 1987

[HaMc81] M. Hammer, D. McLeod: *Database Description with SDM: A Semantic Database Model*, J. ACM, vol. 31 (3), 1984, pp. 351 – 386

[HeSa91] 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

[KhCo86] S. Khoshafian, G. Copeland: *Object Identity*, Proc. 1st Int. Conf. on OOPSLA, Portland, Oregon, 1986

[KBB+88] 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

[LoSc87] P. Lockemann, J. W. Schmidt: *Datenbankhandbuch*, Springer, 1987

[LNPS85] K. v.Luck, B. Nebel, C. Peltason, A. Schmiedel: *BACK to Consistency and Incompleteness*, in H. Stoyan (Ed.): *GWAI-85*, Springer IFB 1986, pp. 245 – 257

[MSOP86] D. Maier, J. Stein, A. Ottis, A. Purdy: *Development of an Object-Oriented DBMS*, OOPSLA, September 1986

[MBW80] 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

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis: *Telos: Representing Knowledge About Information Systems*, ACM ToIS, vol. 8 (4), October 1990 pp. 325 – 362

[Oho90] A. Ohori: *Representing Object Identity in a Pure Functional Language*, Proc. ICDT 90, Springer LNCS, pp. 41 – 55

[SaJu91] G. Saake, R. Jungclaus: *Specification of Database Applications in the TROLL Language*, in Proc. Int. Workshop on the Specification of Database Systems, Glasgow, 1991

[SSW+91a] K.-D. Schewe, J. W. Schmidt, I. Wetzel, N. Bidoit, D. Castelli, C. Meghini: *Abstract Machines Revisited*, FIDE technical report 1991/11

[SWS91] K.-D. Schewe, I. Wetzel, J. W. Schmidt: *Towards a Structured Specification Language for Database Applications*, in Proc. Int. Workshop on the Specification of Database Systems, Glasgow, 1991

[Su83] S. Y. W. Su: *SAM*: A Semantic Association Model for Corporate and Scientific-Statistical Databases*, Inf. Sci., vol. 29, 1983, pp. 151 – 199

[Tha88] B. Thalheim: *Dependencies in Relational Databases*, Teubner Leipzig, 1988

[Tha90] 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