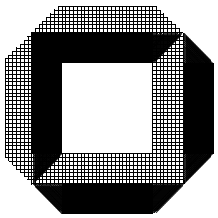

UNIVERSITÄT KARLSRUHE
FAKULTÄT FÜR INFORMATIK

Postfach 6980, D-7500 Karlsruhe 1

Towards More Flexible Schema Management in Object Bases

Guido Moerkotte Andreas Zachmann

Universität Karlsruhe
Fakultät für Informatik
7500 Karlsruhe, Germany
[*moer*|*zachmann*]@ira.uka.de

Interner Bericht Nr. 24/92 * November 1992

Abstract

There exists a current trend in database technology to make databases more extensible and flexible, or even to generate databases for specific customer needs. So far, schema management and especially schema evolution have been excluded from this trend. In this paper, we propose a new approach to schema management and topics centered around it, like schema consistency and schema evolution. This approach allows easy tailoring of schema management, high-level specification of schema consistency and development of advanced tools supporting the user during schema evolution.

We exemplify the approach by designing a simple schema manager. In order to demonstrate the achieved flexibility, this simple schema manager is then enhanced by complex schema evolution concepts and versioning mechanisms. It turns out, that the resulting necessary modifications to be carried out on the previously designed simple schema manager can be held at a minimum.

1 Introduction

Object-oriented database systems are emerging as the next generation of database systems for so called non-standard applications. Up to now, there has been little experience with these systems in real applications. Nevertheless, it seems that different applications pose different requirements on these systems. This may be one reason for a current trend emphasizing the flexibility, extensibility and easy customizability of database systems (e.g. Exodus [11, 6], Genesis [3, 4], Postgres [23], Probe [9, 14], Starburst [13, 17]).

Although the requirements of the different applications may also differ for schema management and especially schema evolution, flexibility has been included so far only into the runtime system of databases. Thus, the schema management and its schema evolution concept for object-oriented database systems have been excluded from changes. That is, all approaches rely on a fixed data model, assuming a fixed set of evolution operations, a fixed notion of schema consistency, and a fixed set of inconsistency cures like masking or conversion. From the following indications we infer that more flexible schema management is needed:

Bocionek pointed out that there exists five different semantics for a simple schema evolution operation like type deletion [5]. Since the lack of experience with real applications, it seems impossible to decide for the best semantics during system development. Thus, the customization or even the definition of new schema evolution operations by the database user should be possible.

Skarra and Zdonik [22] introduce the schema evolution concept as applied in ENCORE. They propose to use only pre and post exception handler to mask certain kinds of inconsistencies since conversion is too expensive to be performed. Thinking of applications with large amounts of data and no time for reorganization this is convincing. Nevertheless, for the O_2 system [12], Zicari proposes to cure inconsistencies by immediate conversion [25]. Thus, it might be worthwhile to have both cures built into the system, and provide the possibility to choose among these and even more, to introduce new (not yet discovered) cures. The necessary changes to be performed by the database developer should thereby be held at a minimum.

Banerjee, Kim, Kim, and Korth introduce the schema evolution concept of Orion [2]. Kim and Chou enhance this concept with a schema versioning mechanism [16]. In our opinion, it should be easy for the database developer to introduce the newly proposed mechanism. Further, if for an application it is discovered that the proposed versioning mechanism is not the best one, it should be easy to change and expand.

These findings convinced us that — at the moment and maybe also in the future — it might be impossible to define *the* schema manager with an associated schema evolution mechanism that suits all applications best. Consequently, this paper proposes a new approach to schema management in object bases. This approach will allow the design of schema managers which provide flexibility and support for both, the database developer — to ease the implementation and modification of the schema manager — and the database user — to ease the schema managers adaption to his/her specific needs.

There still exists another problem with schema evolution concepts as employed in current object-oriented database systems: there exists no formal definition of the applied notion of schema consistency. The lack of a formal model manifests itself in the observation that — opposed to the relational model where the different relations are independent from a typing point of view — the components of an object-oriented schema are highly interdependent. This need for a formal basis was also seen by other researchers like Abiteboul, Kanellakis, and Waller who defined a minimal formal model which allows to reason about formal properties of schemas and schema updates [1, 24]. Nevertheless, our intention in formalizing the notion of schema consistency is somewhat different. Since their notion of schema consistency based on a rewrite approach is quite general, it allows to state undecidable notions of consistency. Although also looking for some possibility to formally specify schema consistency, we had for pragmatic reasons

to prefer a formalism which only allows to state decidable notions of consistency.

In order to support the ease of modifications of schema consistency, another requirement is posed upon the formal basis: it should allow the declarative definition of schema consistency. Furthermore, if the user is able to define new complex schema evolution operations, the formal basis must enable the design of tools which automatically check schema consistency and — in case of a detected inconsistency — analyze the situation and generate possible repairs whose execution regains consistency. One proposal in this direction was given by Delcourt and Zicari [10]. They also give a formal framework for treating structural consistency. A tool called ICC is presented which allows the automatic detection of inconsistencies. In case of an inconsistency the update is rejected and the user receives a notification denoting the type of inconsistency together with a location (e.g. class) where the inconsistency was detected. Since they rely on a fixed set of possible update operations and a fixed notion of schema consistency this approach is not applicable to our problem.

As turned out the logical framework for deductive databases fulfilled all our requirements: it suffices to define schema consistency declaratively, there exist efficient consistency checks, e.g. [18, 20], and mechanisms to automatically generate repairs for detected inconsistencies have also been designed and implemented [19].

The outline of the rest of the paper is as follows. Section 2 summarizes our goals concerning flexibility and support for database users and database developers. It then gives a high-level overview of our approach including the proposed system architecture. In Section 3 the core of our database programming language GOM [15] is modeled and the applied notion of consistency is defined. The result is a simple schema manager for the core of GOM. Section 4 sounds out the flexibility and support induced by our approach. It hypothetically assumes an existing schema manager (the one of section 3) and then exploits the impacts of adding inconsistency cures and complex schema evolution operations. Section 5 concludes the paper. The appendix gives the notion of schema as supplied by GOM.

2 The General Idea

2.1 The Goals

As already mentioned in the introduction, we want to design more flexible schema managers. Flexibility should be supported for both, the database user and the database developer. Especially the latter should be supported in his highly difficult task to design and implement a schema manager. Within this subsection, we briefly summarize the goals we pursue with our approach.

User-Defined Complex Schema Evolution Operations: Existing approaches to schema evolution provide only a fixed set of evolution operations. Instead, the possibility should exist to compose complex schema evolution operations from a set of primitive operations which allow any schema modification. If these operations are to be used only once, they should be designable on-line in *schema evolution sessions*. If they are likely to be used more often, the possibility should exist to define schema evolution operations within some programming or macro language.

Note that there exists a severe problem. In general, it cannot be assured that neither the necessary primitives nor the complex operations transform a consistent schema into another consistent one. Even worse, allowing only schema evolution operations which guarantee in all situations the consistency of the resulting modified schema results in an unacceptable restriction of possible schema evolution operations. Take the addition of an argument to an operation as an example. This operation would be impossible since it only results in a consistent schema if all calls of this operation are modified at the time. Thus, only the execution of a whole set of primitive evolution operations may result in a consistent schema. Even worse, the necessary

modifications and their number are dependent on the current situation and, hence, no such schema evolution operation (for adding an argument to an existing and used operation) which preserves consistency in all cases can be defined. Consequently, decoupling schema evolution operations from schema consistency is a necessity, and as such one of our main goals. One consequence of this approach is that consistency checking is deferred until the end of a schema evolution session or the execution of a schema evolution operator. Further, this decoupling immediately leads us to our next goal.

Advanced User Support for Consistency Control: In order to control schema consistency efficient tools must exist which automatically check schema consistency after an evolution session. Since schema consistency and schema evolution operations may become arbitrary complex, it is unacceptable for the user if the tool would simply accept or reject the proposed modifications in a stupid "yes/no" manner. Instead, the system must at least give a detailed description of the inconsistencies. But even this is not enough support in case of very complex errors. Thus, we aim at the best support we can think of. That is, the tool should be able to automatically generate all (useful) repairs for a detected inconsistency.

Changing the Definition of Consistency: This goal concerns both, the user and the developer. Both might wish to change the current definition of schema consistency. For example, due to the conceptual mass multiple inheritance can lead to, some project leader might want to restrain inheritance to single inheritance. This modification should be possible and easy to perform.

Another situation necessitating modifications of the system's notion of consistency occurs if the system's capabilities are to be extended. Adding non-existent cures or changes to the data model like allowing overloading are typical examples. These changes are best supported if there exists a formalism in which schema consistency can be specified declaratively. This declarative statement of schema consistency should then automatically lead to a consistency checker and its associated inconsistency repair mechanism. In case of extensions, changing the definition of schema consistency often is not the only necessary adaption to be performed by the developer but as we will see the other changes can also be held at a minimum (see section 4).

2.2 The Solution

A well-defined structure and modularization of the schema management with clear responsibilities of the different modules does not only support the developer in implementing the system, rather it is a necessary prerequisite to reach the overall goal of flexibility. More specifically, for a given system adoption, all involved modules of the implementation must be easy and undoubtedly localizable, and the necessary changes to each module must be limited and clearly distinguishable. Thus, we start with the proposal of a generic architecture of an object-oriented database system. Since the realization of object management and access is not our main topic, we do not discuss in detail aspects of the runtime system. Being only interested in its relation to schema management and its dependencies with consistency, we will go into more detail only for those parts of the runtime system that have an impact on the definition of schema consistency.

Figure 2.2 gives an overview of the proposed architecture. The modules *Analyzer* and *Runtime System* as well as the *Database Model* are centered around the *Consistency Control*. The *Object Base* contains the actual physical representation of all instantiated objects. This physical representation has to be modeled since there might exist inconsistencies between the physical representation of an object of some type and certain attributes associated with it. The model of the *Object Base* then consists of a set of assertions the runtime system ensures on the physical representation of the objects. It is contained in the *Object Base Model* that is part of

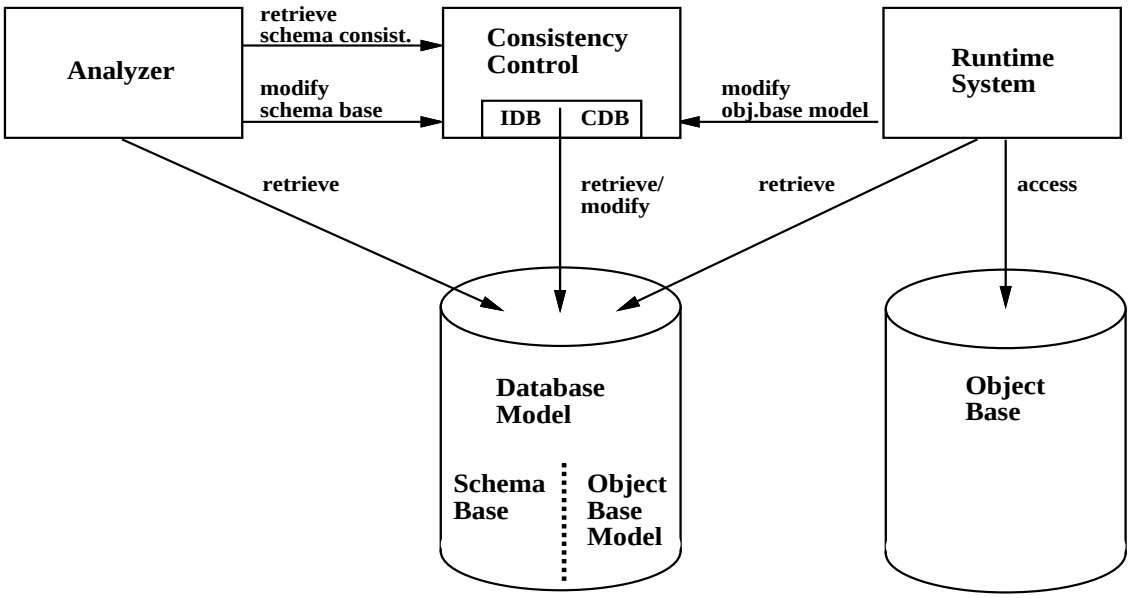

Figure 1: The Generic System Architecture

the *Database Model*. The other half of the *Database Model* is the *Schema Base* that contains the current schema definitions, i.e., abstract representations of the sources. In case the code of the operations is compiled, there exists a consistency problem between the original source code and the compiled code. Since we will not be concerned with these issues within the current paper, we assume that the source code is interpreted by the runtime system. (Note, that this is not an inherent restriction of our approach).

The *Analyzer* constitutes the *front end* to perform the user initiated schema updates: the interface of this module is identical to the update operations visible to the user, e.g., a regular introduction of a new type or the addition of a new attribute to some type[1]. Each call of an update operation will be mapped to corresponding modifications of the schema base located in the *Database Model*. These modifications have to take place via the *Consistency Control*. Thus, we assume that the *Analyzer* can change the *Schema Base* only via calling the *modify* operation of the *Consistency Control*. Possible instantiations for the *Analyzer* are interactive schema editors or compiler front ends that parse the textual update specification. The latter should for flexibility reasons be implemented using standard compiler generator tools like Lex and Yacc.

The *Runtime System* can be thought of as any runtime system of an object-oriented database system. Its main task is object management and its main responsibility is the physical object representation. Thus, it is also responsible for performing cures like conversion. As already mentioned above, we assume that the *Runtime System* interprets the schema, especially the method's source code. Further, the *Runtime System* has to correctly report changes in the object's representation via the *modify* operation.

The *Consistency Control* component is responsible for executing the reported modifications received from the *Analyzer* and the *Runtime System*. All changes have to be enclosed between a *BES* (begin of evolution session) and an *EES* (end of evolution session). At *EES* time the *Consistency Control* starts checking consistency. Here, we can identify two parts of the consistency definition:

1. schema consistency, and

[1] Of course, the user interface will offer operations to retrieve the current state of the schema, too.

2. schema/object consistency

The former captures the conditions necessary for a schema to be consistent. It thus deals with requirements such as the domain of all attributes must be defined and all invoked operations must be present. The latter deals with those conditions that specify the consistency between the schema and the object base. A typical condition here is that for each attribute in a type definition, there must exist a physical representation of it for every object being an instance of this type. The precise definition of both parts of consistency can be found in the next section.

If an inconsistency in any of these parts has been detected, the user is informed. On request, the *Consistency Control* generates possible repairs for the inconsistency. The user may then decide for one possible repair to be executed or to rollback the evolution session. The exact protocol and an example for the generation of repairs are given at the end of next section.

It is important to note that factoring out the notion of consistency, and the above separation of consistency into two disjoint parts results in a major reduction of the schema manager's complexity.

To abstractly assess this architecture with respect to flexibility we want to identify those parts of the architecture that are involved in specific changes. Instead of giving a complete taxonomy of possible (and mostly trivial) changes, we concentrate on the most interesting changes:

- *defining new evolution operators:* In this case, only the analyzer has to be expanded using the interface of the *Database Model*. The already existing parts of the analyzer do not have to be modified. In case the *Analyzer* has been build using some standard compiler generator tools this is routine.

- *expanding the data model:* If the underlying data model is expanded or changed, all components have to be expanded or changed, resp. Again, for the analyzer this is routine. The necessary changes to the runtime system are directly dependent on the extensions or changes to the data model. Nevertheless, it is most likely that in case of an extension the existing part of the runtime system does not have to be touched. Additionally, the consistency definition has to be changed, too.

- *changes in consistency definition:* This seems to be the most problematic case. We first note, that there is no need to change any module interface since the *Consistency Control* hides the consistency definition behind its interface. Nevertheless, up to now, it might be that the *Consistency Control* itself has to be reimplemented due to the high degree of interdependencies present among the components of a schema for object-oriented databases.

As can be seen from the above the most difficult task is to adopt changes in the consistency definition. If asked to specify schema consistency procedurally, most users are likely bound to fail. Even for the database developer, designing and implementing the *Consistency Control* is likely to be the most difficult part realizing the schema manager.

The remedy to make changes in the implementation of *Consistency Control* feasible is to specify schema consistency (and thus the implementation of the *Consistency Control*) in a declarative way—as a set of constraints. Each constraint definition exactly corresponds to a specific consistency assumption within the database model. More specifically, we use a deductive database for the *Consistency Control*. Besides constraint definitions (*CDB*) it contains rules (*IDB*) to define auxiliary intentional predicates. We applied [20] for efficient consistency checking and [19] for the automatic generation of repairs. The interface to the *Database Model* then consists of the operations — add (+) and delete (-) — for modifying the extensions of the base predicates.

3 A Simple Schema Manager for the Core of GOM

The last section motivated the use of a deductive database as the underlying component of the schema manager. Before a deductive database can be used, its (meta) schema has to be specified by modeling the schema information of the underlying object model. Since a deductive database consists of facts giving the extensions for base predicates, rules defining derived predicates and constraints restricting the number of "legal" extensions, these three components have to be specified.

Obviously, the specification of all these components depends on the chosen object model. As an example object model we have chosen GOM ([15]) which will shortly be introduced by means of an example. In order to keep the schema model short and simple, several restrictions have been applied. Especially, some fancy features available in GOM[2] are not treated. Nevertheless both, structural and behavioral, aspects (in the sense of [25]) as well as multiple inheritance will be modeled. Hence, the core of (almost) every object-oriented data model is captured.

We will then use this example to extract the necessary entities and relationships which will be modeled by base predicates whose extensions will be contained in the *Schema Base*. A simple and quite restricted notion of schema consistency will then be defined via a set of constraints. Last, this section models the *Object Base* and gives the definition of schema/object consistency.

3.1 Example

The running example is based on the *leaded/unleaded cars*-example of [22]. It consists of a schema called *CarSchema* containing the definitions for the tuple-structured types *Person*, *Location*, *City*, and *Car*. Each type definition consists of a body specifying the attributes of the type. Available operations model the behavior. Operations are specified in two parts: declaration and implementation. The former contains the signature of the operation while the latter gives a piece of code for the operation. The (almost) complete type definitions in GOM are:

schema CarSchema **is**

type Person **is**
[name : string;
age : int;]
end type Person;

type Location **is**
[longi : float;
lati : float;]
operations
distance : || Location ⟶ float;
implementation
· · · !! uses longi and lati.
end type Location;

type City **supertype** Location **is**
[name : string;
noOfInhabitants : int;]
refine
distance : || Location ⟶ float;

[2] like overloading, parametric polymorphism, generic types, virtual types, subschemas, value receiving operations, and information hiding through the **public** clause

```
implementation
   ... !! uses longi and lati as well as city name.
end type City;

type Car is
   [ owner:     Person;
     maxspeed:float;
     milage:    float;
     location:  City; ]
operations
   changeLocation : || Person, City ⟶ float;
implementation
   changeLocation(driver,newLocation) is
   begin
      if (self.owner == driver)
      begin
         self.milage := self.milage + self.location.distance(newLocation);
         self.location := newLocation;
         return self.milage;
      end
      else return -1.0;
   end changeLocation;
end type Car;

end CarSchema;
```

3.2 Base Predicates

Taking a look at the example the needed entity-types are easily recognized: *schema*, *type*, *attribute*, *operator declaration*, *argument declarations* and *operator implementation*. These can directly be modeled by the following base predicates

1. Schema(SchemaId, UserName)
2. Type(TypeId, TypeName, SchemaId)
3. Attr(TypeId, AttrName, TypeId)
4. Decl(DeclId, TypeId, OpName, TypeId)
5. ArgDecl(DeclId, ArgNo, TypeId)
6. Code(CodeId,Code,DeclId)

where keys are underlined. The base predicates have the following attributes. Schemas and types have two attributes, an identifier and the user given name. To express the n:1 relationship *occurs* between types and schemas, *Type* has a third attribute containing schema identifiers. For attributes, we do not specify an identifier since they are uniquely determined by the first two attributes of the base predicate *Attr* containing the type in which they occur and their name. Further, by that, the n:1 relationship *occurs* between attributes and types is covered already. The third attribute of *Attr* contains the attributes' domain type. Operation declarations have again an identifier, the receiver type, the user given operation name, and its result type. Since the receiver type is identical with the type in which the declaration occurred, this relationship

has been covered already. The argument declarations for an operator declaration are modeled by a separate base predicate *ArgDecl* with the declarations identifier, the number of the argument (numbering from left to right), and the declared argument type. A piece of code is modeled by an identifier, the actual text fragment and the declaration it implements. By the latter, we cover the 1:1 relationship *implements*.[3]

Possible extensions of the base predicates have to be derived from a given schema definition by the *Analyzer* component. For the example, the *Analyzer* would derive the extensions shown in Figure 2 where the existence of types for the built-in sorts — like integer, float, string and so on — is implicitly assumed.

Schema	sid_1	CarSchema		
Type	tid_1	Person	sid_1	
	tid_2	Location	sid_1	
	tid_3	City	sid_1	
	tid_4	Car	sid_1	
Attr	tid_1	name	tid_{string}	
	tid_1	age	tid_{int}	
	tid_2	longi	tid_{float}	
	tid_2	lati	tid_{float}	
	tid_3	name	tid_{string}	
	tid_3	noOfInhabitants	tid_{int}	
	tid_4	owner	tid_1	
	tid_4	maxspeed	tid_{float}	
	tid_4	milage	tid_{float}	
	tid_4	location	tid_3	
Decl	did_1	distance	tid_2	tid_{float}
	did_2	distance	tid_3	tid_{float}
	did_3	changeLocation	tid_4	tid_{float}
ArgDecl	did_1	1	tid_2	
	did_2	1	tid_2	
	did_3	1	tid_1	
	did_3	2	tid_3	
Code	cid_1	$\cdots$	did_1	
	cid_2	$\cdots$	did_2	
	cid_3	$\cdots$	did_3	

Figure 2: Extensions for the Example

Having modeled all entities and some 1:n relationships there still exist other relationships not yet covered. These fall into two groups. The first group covers the *subtype* and *refinement* relationship:

SubTypRel(TypeId, TypeId)
DeclRefinement(DeclId,DeclId)

SubTypRel(X,Y) states that X is a subtype of Y and *DeclRefinement(X,Y)* states that X is a refinement of Y.

[3]Of course, one has to model the parameters of the code. Since this can be done in a fashion similar to modeling the arguments in the operator declarations, we will skip this straightforward part of the model.

While the *Consistency Control* should not inspect the code implementing operations,[4] it needs some information about the code: the operations called and the attributes accessed by it. These relationships are covered by the second group of base predicates:

CodeReqDecl(CodeId, DeclId)
CodeReqAttr(CodeId, TypeId, AttrName)

Again, the extensions of the relations must be derived by an *Analyzer*. For the example, we have the following extensions:

SubTypRel	tid_3	tid_2	
DeclRefinement	did_2	did_1	
CodeReqDecl	cid_2	did_1	
CodeReqAttr	cid_1	tid_2	longi
	cid_1	tid_2	lati
	cid_2	tid_2	longi
	cid_2	tid_2	lati
	cid_2	tid_3	name
	cid_3	tid_4	owner
	cid_3	tid_4	milage
	cid_3	tid_4	location

3.3 Schema Consistency

Of course, not all extensions of the above predicates are valid for a given data model. Several constraints have to be introduced in order to restrict the instances to the legal ones. Different classes of constraints are distinguished. *Uniqueness constraints* require that something must be unique, e.g., an attribute name within a given type. Keys also belong to this category. *Existence constraints* require that something must exist, e.g., for an operator declaration there must exist some code. Referential integrity constraints are a subset of the existence constraints. Beside these easy to state constraints there exist *multiple inheritance constraints* and *typing constraints*.

Keys and other Uniqueness Constraints Beside key constraints which we do not state explicitly due to their simplicity, there exists only one further uniqueness constraint for our schema model stating that every type name can be used at most once within one schema:[5]

$\forall$ X_1, X_2, Y_1, Y_2, Z
Type(X_1,Y_1,Z) $\wedge$ Type(X_2,Y_2,Z) $\Longrightarrow$ (Y_1=Y_2 $\Longrightarrow$ X_1=X_2)

Referential integrity and other existence constraints There exists a whole bunch of typical referential integrity constraints like any schema identifier in the *Type* extension must occur in the *Schema* extension etc. We do not give the formulas for these simple constraints which always have the same pattern and can easily be stated. For our simple example schema model there exists only one constraint not following this pattern. It requires that for any declaration a piece of code implementing it has to be present:

$\forall$ D, T_c, O, T_t $\exists$ C_1, C_2 Decl(D,T_c,O,T_t) $\Longrightarrow$ Code(C_1,C_2,D)

[4] We only have to model those parts of the data model that contain dependencies to other modeled entities.

[5] We use ordinary FOL syntax for expressing constraints. Variables start with a capital letter. Constraints have to be closed range-restricted formulas.

Simple Constraints for SubTypRel and DeclRefinement Before stating the constraints for *SubTypRel*, we need its transitive closure which is easily defined by the following two rules which follow "Prolog syntax":

$\text{SubTypRel}^t(X,Y)$:- SubTypRel(X,Y).
$\text{SubTypRel}^t(X,Z)$:- SubTypRel(X,Y), $\text{SubTypRel}^t(Y,Z)$.

The subtype relationship has to be acyclic. Additionally, in GOM, there must exist a unique root called ANY. This is expressed by the following two constraints, resp.:

$\forall$ X, Y, Z $\quad \neg\ \text{SubTypRel}^t(X,X)$
$\forall$ X, Y, Z $\quad \text{Type}(X,Y,Z) \Longrightarrow (X = \text{ANY} \vee \text{SubTypRel}^t(X,\text{ANY}))$

The transitive closure of the refinement relationship is also needed. We define:

$\text{DeclRefinement}^t(X,Y)$:- DeclRefinement(X,Y).
$\text{DeclRefinement}^t(X,Z)$:- DeclRefinement(X,Y), $\text{DeclRefinement}^t(Y,Z)$.

We require the refinement relationship to be acyclic:

$\forall$ X $\neg\text{DeclRefinement}^t(X,X)$

Multiple Inheritance Constraints For our simple schema manager where we do not model any resolution strategies for conflicts arising from multiple inheritance, we require any two inherited attributes with the same name to have the same codomain. For any two inherited operations we require that there exists a refinement, if they have the same name and different origins. Thus, the following constraints are needed:

$\forall$ T, A, D_1, D_2 $\quad \text{Attr}^i(T,A,D_1) \wedge \text{Attr}^i(T,A,D_2) \Longrightarrow D_1 = D_2$
$\forall$ T, T_1, T_2, O, T_{t1}, T_{t2}, D_1, D_2 $\exists$ D
SubTypRel(T,T_1) $\wedge$ SubTypRel(T,T_2) $\wedge$
$\text{Decl}^i(D_1,T_1,O,T_{t1}) \wedge \text{Decl}^i(D_2,T_2,O,T_{t2})$
$\Longrightarrow$
DeclRefinement(D,D_1) $\wedge$ DeclRefinement(D,D_2)

where the intentional predicate $Attr^i$ defined by

$\text{Attr}^i(T,A,D)$:- Attr(T,A,D).
$\text{Attr}^i(T_1,A,D)$:- $\text{SubTypRel}^t(T_1,T_2)$, Attr($T_2$,A,D).

and $Decl^i$ is similar but we have to respect that some of the supertypes' operations have been refined already:

$\text{Decl}^i(X, Y_{1,1}, Z, Y_{1,2})$:- Decl(X, $Y_{1,1}$, Z, $Y_{1,2}$).
$\text{Decl}^i(X, Y_{1,1}, Z, Y_{1,2})$:- $\text{SubTypRel}^t(Y_{1,1}, Y_{2,1})$, Decl(X, $Y_{2,1}$, Z, $Y_{1,2}$),
$\neg$ Refined(X, $Y_{1,1}$).

where *Refined(X, Y)* is another intentional predicate which holds, if there is a refinement of declaration X associated to type Y or one of its supertypes:

Refined(X_1, $Y_{2,1}$) :- Decl(X_1, $Y_{1,1}$, Z_1, $Y_{1,2}$), $\text{DeclRefinement}^t(X_2, X_1)$,
Decl(X_2, $Y_{2,1}$, Z_2, $Y_{2,2}$).
Refined(X_1, Y) :- Decl(X_1, $Y_{1,1}$, Z_1, $Y_{1,2}$), $\text{DeclRefinement}^t(X_2, X_1)$,
Decl(X_2, $Y_{2,1}$, Z_2, $Y_{2,2}$), $\text{SubTypRel}^t(Y, Y_{2,1})$.

Refinement Constraints In GOM we adhere to contravariance to ensure strong typing (see [15] for more details). Contravariance states that for any two declarations D_1 and D_2 where D_2 is a refinement of D_1 that

1. for each parameter of D_2, its type must be a supertype of (or the same as) the type of the corresponding parameter of D_1, and
2. the result type of D_2 must be a subtype of (or the same as) the result type of D_1.

Further, the names must be the same, the type in which D_2 occurs must be a subtype of the type where D_1 occurs, and the number of arguments specified for D_1 must be the same as for D_2. Thus we specify:

$$
\begin{array}{l}
\forall\ D_1, D_2, T_{c_1}, T_{c_2}, O_1, O_2, T_{t_1}, T_{t_2} \\
\quad \text{DeclRefinement}(D_2, D_1) \wedge \text{Decl}(D_1, T_{c_1}, O_1, T_{t_1}) \wedge \text{Decl}(D_2, T_{c_2}, O_2, T_{t_2}) \\
\quad \Longrightarrow (O_1 = O_2 \\
\quad \wedge \quad (T_{t_1} = T_{t_2} \vee \mathit{SubTypRel}^t(T_{t_2}, T_{t_1})) \\
\quad \wedge \quad (\forall\ N, TA_1, TA_2 \\
\qquad \text{ArgDecl}(D_1, N, TA_1) \wedge \text{ArgDecl}(D_2, N, TA_2) \\
\qquad \Longrightarrow \\
\qquad (TA_1 = TA_2 \vee \text{SubTypRel}^t(TA_1, TA_2))) \\
\quad \wedge \quad (\forall\ N, TA_1\ \exists TA_2 \quad \text{ArgDecl}(D_1, N, TA_1) \Longrightarrow \text{ArgDecl}(D_2, N, TA_2)) \\
\quad \wedge \quad (\forall\ N, TA_2\ \exists TA_1 \quad \text{ArgDecl}(D_2, N, TA_2) \Longrightarrow \text{ArgDecl}(D_1, N, TA_1)))
\end{array}
$$

3.4 Schema/Object Consistency

So far, we have only been concerned with schema consistency. In this subsection, we will touch the topic of schema/object consistency. Assume that for each type there exists exactly one physical representation for all objects of this type, the following base predicate is introduced to model physical representations:

PhRep(PhRepId,TypeId)

with *PhRepId* being the key and *TypeId* denoting the identifier of the unique type whose objects have this representation. We require that a fact is present in the extension of *PhRepId* if and only if there exists at least one object of the type equal to the second argument of this fact. We assume the implicit existence of physical representations of built-in sorts like integer, float, string and so on.

In order not to confuse the logical and the physical level, we introduce the notion of *slot*s for attributes at the physical level: a slot is meant to be a piece of memory where the value of a logical attribute as defined in the type definition is stored. The physical representation of an object then corresponds to a number of slots. Slots are modeled by a base predicate named *Slot*:

Slot(PhRepId,AttrName,PhRepId)

A slot can be identified uniquely giving *PhRepId* and *AttrName*. The third argument *PhRepId* denotes the physical representation of the value of the slot.

Opposed to the other base predicates, it is the *Runtime System*'s responsibility to keep the data for the *PhRep* and *Slot* up to date. Beside key and referential integrity constraints, three other constraints are needed in order to guarantee the consistency between the physical and the logical part. The first two constraints are uniqueness constraints whereas the third is an existential constraint. We require that there exists only one physical representation for each type:

$$\forall\ C_1, T, C_2 \quad \text{PhRep}(C_1,T) \wedge \text{PhRep}(C_2,T) \Longrightarrow C_1 = C_2$$

The second uniqueness constraint states that the slots for each attribute for a given type must be unique:

$$\forall\ C_{1,1}, A, C_{1,2}\ C_{2,1}, C_{2,2}$$
$$\text{Slot}(C_{1,1}, A, C_{1,2}) \wedge \text{Slot}(C_{2,1}, A, C_{2,2})$$
$$\Longrightarrow C_{1,1} = C_{2,1} \wedge C_{1,2} = C_{2,2}$$

The last constraint (subsequently referred to as (*)) states that for every type there must exist a corresponding slot for every associated attribute including the inherited ones:

$$\forall\ T, A, T_A, C\ \exists\ C_A$$
$$\text{Attr}^i(T,A,T_A) \wedge \text{PhRep}(C,T)$$
$$\Longrightarrow \text{Slot}(C,A,C_A) \wedge \text{PhRep}(C_A,T_A)$$

We continue our example and give consistent extensions (not containing the definitions for base types) of the newly defined base predicates:

PhRep	$clid_1$	tid_1	
	$clid_2$	tid_2	
	$clid_3$	tid_3	
	$clid_4$	tid_4	
Slot	$clid_1$	name	$clid_{string}$
	$clid_1$	age	$clid_{int}$
	$clid_2$	longi	$clid_{float}$
	$clid_2$	lati	$clid_{float}$
	$clid_3$	name	$clid_{string}$
	$clid_3$	noOfInhabitants	$clid_{int}$
	$clid_4$	owner	$clid_1$
	$clid_4$	maxspeed	$clid_{float}$
	$clid_4$	milage	$clid_{float}$
	$clid_4$	location	$clid_3$

3.5 Incorporating conversion

If the last constraint is violated by some schema modifications there exist two brute force methods for repairing the constraint. The first is to provide more schema modifications, which results in deleting attributes for which there exists no appropriate slot. The second is to delete all instances. Both of these two methods are not very satisfactory. Thus, object conversion and masking have been introduced (see e.g. [22, 25]) as more subtle methods to regain the consistency between the schema and the object base. Since the incorporation of masking into a given schema management is used as an example to demonstrate the flexibility of our approach, and as such is deferred to the next section, we will indicate solely how conversion is incorporated into our simple schema manager.

The implementation of the conversion routines must be present in the *Runtime System*. These conversion routines must be able to, e.g., add or delete slots. Since these changes can be reflected already in our model, the only remaining problem is to detect the need of executing them. This can be either left to the user or supported by the system. The latter approach relies on the repair mechanism of the *Consistency Control*. Let us illustrate this approach with our example. Since now, all cars drove on leaded fuel. This changes and the first cars using unleaded fuel appear. In order to capture this change the attribute *fuelType* of type *string* (with occurring values "leaded" and "unleaded") could be added to the type *Car* by $+Attr(tid_4,fuelType,tid_{string})$. Clearly, constraint (*) is violated since

$$\text{Attr}^i(\text{tid}_4,\text{fuelType},\text{tid}_{string}) \wedge \text{PhRep}(\text{clid}_4,\text{tid}_4) \Longrightarrow \text{Slot}(\text{clid}_4,\text{fuelType},\text{clid}_{string})$$

does not hold. This implication can be made true by either invalidating the premise or by validating the conclusion. Thus, the resulting repairs are:

1. -$\text{Attr}^i(\text{tid}_4,\text{fuelType},\text{tid}_{string})$,
2. -$\text{PhRep}(\text{clid}_4,\text{tid}_4)$, and
3. +$\text{Slot}(\text{clid}_4,\text{fuelType},\text{clid}_{string})$.

The first possibility is to undo just the proposed change to the schema. Since the tuple *PhRep(clid$_4$,tid$_4$)* is present in the *Object Base Model* if and only if there exist instances of *Car*, its deletion results in deleting all cars. Now comes the crucial point for conversion: the third change can be achieved by executing the conversion routines, or — the other way round — with the repair we have detected the possibility to remedy the inconsistency by the execution of the conversion routine which adds a slot to every object of a type. The conversion routine itself must be supplied with information on the values to write into the new slots. This can be done by providing a default value, by asking the user for every instance, or by providing an operation that—called on the old instances—provides a value for the new slot. In our example, the last possibility would be chosen: an operation is provided that selects the fuel types depending on the car model and its production date.

There still exists a minor problem concerning the readability of repairs. If presented to the user as changes to the extensions of the base predicates the repairs might not necessarily be easy to interpret by the user. This can be remedied, too. Since the *Consistency Control* is not aware of the actual changes in the *Object Base* necessary to derive the proposed changes in the *Database Model*, we assume that for each change to a base predicates' extension either the *Analyzer* or the *Runtime System* can explain the changes to be performed. These explanations can be ordered by the *Consistency Control* to add more information to the generated repairs.

All together, we are now prepared to state the general protocol of a schema evolution session for our schema manager:

1. The user starts a schema evolution session.
2. Then, the user proposes (a) change(s) to the schema and suggests to end the session.
3. The *Analyzer* extracts the necessary changes to the extensions of the base predicates.
4. The *Consistency Control* performs a consistency check.
5. If no consistency violation was detected, the schema evolution session can end successfully.
6. If an inconsistency was detected, the *Consistency Control* derives — upon user request — repairs for the detected inconsistency. These repairs are stated in the form of changes to the base predicate extensions.
7. In order to make the user aware of the consequences of these changes, the *Consistency Control* asks the *Analyzer* and the *Runtime System* for the necessary actions which have to performed in order to gain the necessary changes for the base predicate extensions.
8. The *Consistency Control* then prepares this information, presents it to the user, and asks him/her to make a choice — undoing the evolution session is always among the repairs.
9. The *Consistency Control* initiates the execution of the chosen repair by the *Analyzer* and/or *Runtime System* and ends the schema evolution session successfully.

The repairs are computed by building a derivation tree for each consistency violation and subsequent combination of its leaves into a repair ([19]).

4 Sounding out Flexibility

As the abstract assessment of the flexibility of our approach has already been presented in section 2, the topic of this section is to sound out the flexibility by means of concrete examples. Since our goal is to provide flexibility to both, developers and users of a database system, there exist two corresponding subsections.

4.1 Developer's Flexibility

Assume that the above very simple and restricted schema model has been developed by some company. They released this system as a prototype version GOM-V0.1 to some selected companies. Of course, this prototype cannot at all satisfy the customers' needs. One deficiency is the limited facility to repair schema-object consistency by conversion of all instances. This is too rigid for practical use, additional adoption mechanisms—like masking of objects—should be available. Another shortcoming was the lack of schema versioning. Despite the fact that the development team expected an enormous expense needed to improve their prototype, they dared to tackle implementing both versioning of schemas and masking of objects. In the following section, we want to accompany their design of the intended release GOM-V1.0 and subsequently summarize the actual implementational efforts to be undertaken.

Design In [7, 8] a schema versioning mechanism has been proposed which we like to incorporate into our schema manager. Therefore, we simply have to extend our schema model by two new base predicates capturing the evolution of schemas and types:

1. evolves_to_S(SchemaId, SchemaId)
2. evolves_to_T(TypeId, TypeId)

We skip the constraints needed to express the referential integrity of types and schemas occurring in the extensions, since they are in the same fashion as the integrity constraints of section 3. Beside this, we want to constrain the version graph, spanned by *evolves_to_S*, respectively *evolves_to_T*:

1. The version graph of types and schemas must be acyclic (forming a DAG).
2. We require some kind of digestibility of evolution of types and schemas: Types may evolve from each other only if the corresponding schemas do also evolve from each other.

To formalize this, we need the definition of the transitive closures $evolves\_to\_S^t$ and $evolves\_to\_T^t$ of the schema and the type evolution relation, which can be defined analogously to the definition of $SubTypRel^t$ (see section 3). Now, the constraints to capture the DAG restriction are

$\forall$ X $\neg$ evolves_to_S^t(X,X) and
$\forall$ X $\neg$ evolves_to_T^t(X,X)

The second requirement can also easily be stated:

$\forall$ X_1, X_2, Y_1, Y_2, Z_1, Z_2
Type(X_1,Y_1,Z_1) $\wedge$ Type(X_2,Y_2,Z_2) $\wedge$ evolves_to_T^t(X_1,X_2) $\Longrightarrow$
evolves_to_S^t(Z_1,Z_2)

Since the old schema version is available still, we cannot get into schema-object inconsistencies as long as we do not change the old schema, but simply add new schema versions. Schema evolution problems didn't go up in smoke, of course. They just appear in another light:

Still, we have to care for the objects. If there exists no possibility to access objects which are instantiated in another schema version, we have to start from scratch each time we have added a new schema version. Thus, we have to extend visibility of objects to other schema versions. This can easily be done, if the corresponding type has not been changed. Problems arise, if the new type version is incompatible to the old one. Such a compatibility check cannot be performed automatically since it requires to reason about the semantics of the operations associated to this type. There also exists a test which could be done automatically—testing the refinement condition—but this condition captures only a syntactical prerequisite for compatibility.

Instead, we want the schema designer to manifest explicitly the *semantic compatibility* of types: he/she has to define explicitly which objects instantiated of another type version shall be substitutable for objects instantiated from the current[6] type (and schema) version. In other words, the schema designer explicitly has to extend the substitutability of types. Obviously, the refinement condition has to be satisfied. Since it is too restrictive to allow access to objects of types which fulfill the refinement condition with the corresponding type, we introduce masking. For this, a syntactical means is the **fashion** construct [21] which perfectly fits the purposes of schema evolution.

With **fashion** one can imitate the behavior of a given type in terms of the signature of another type allowing the instances of the second type to be substitutable for instances of the first type. This is exactly what we need to bridge the gap between two versions of the same type. Take an evolution of type *Person* from the *CarSchema* introduced in section 3 as an example: We want to replace the *age*-attribute of *Person@CarSchema*[7] by an attribute *birthday: date* within the new version *Person@NewCarSchema* of this type. The resulting definition of *Person@NewCarSchema* will be as follows:

```
type Person is
   [ name :     string;
     birthday : date;]
end type Person;
```

If we want the instances of *Person@CarSchema* to be substitutable for instances of *Person@NewCarSchema*, the following **fashion**-declaration will be appropriate:

```
fashion Person@CarSchema as Person@NewCarSchema
   where
      birthday : → date is /* derive birthday from age */
      birthday : ← date is /* derive age from birthday */
      name     : string   is self.name;
end fashion;
```

By means of this declaration, the *Runtime System* is told that each time a *Person@CarSchema* appears within an attribute or a variable of type *Person@NewCarSchema*, read and write accesses to the (not existing) *birthday* attribute[8] are redirected to the specified code to derive the birthday from *age* (and vice versa). Similar to this, accesses to *name* will be "redirected" to the corresponding *name* attribute of *Person@CarSchema*.

[6] i.e. the corresponding type version of the schema version which is used by an application

[7] We use the *at*-notation "'<TypeName>@<UsersSchemaName>" as a syntactical means to identify the type version. This is possible due to the uniqueness of <TypeName> within a given schema and due to the uniqueness of <UsersSchemaName> as stated in section 3.

[8] the signature of the read-access is *birthday : → date*,
the signature of the write-access is *birthday : ← date*

Modeling **fashion** is similar to modeling subtyping and refinement which we already did in section 3. Analogous to *SubTypRel*, we have one base predicate *FashionType* which materializes how substitutability will be affected:

FashionType(TypeId, TypeId)

FashionType(X, Y) states that instances of type X shall be substitutable for the instances of type Y, i.e., wherever an object of type Y is expected, an object of type X suffices. Opposed to [21], we want to restrict the use of **fashion** for our schema manager solely to schema evolution purposes. This can easily be expressed by the following constraint:

$\forall$ X, Y FashionType(X, Y) $\Longrightarrow$ evolves_to_T(X, Y) $\vee$ evolves_to_T(Y, X)

But this is not the only condition related to **fashion** which has to be asserted by *Consistency Control*. As mentioned above, the relation of two types connected via *FashionType* is similar to the relation of two types connected via *SubTypRel*. But there are two differences between **fashion** and subtyping. First, there is no inheritance between Y and X. Thus, every operation available for Y has to be (re-)defined in X. Since we do not want to redeclare[9] these operations, no distinction between declaration and definition is needed. The second difference is a direct consequence of the first one: even the attributes of Y have to be (re-)defined in X. We do not want to insist in the existence of a corresponding attribute, instead of this, the implementation of a pair of read and write operations is accepted. All together, two additional base predicates are needed to model the **fashion** construct:

1. FashionDecl(DeclId, TypeId, Code)
2. FashionAttr(TypeId, AttrName, TypeId, Code, Code)

FashionDecl(X, Y, Z) reflects that operation X will be imitated within type Y by means of code Z. The same does *FashionAttr*(X_1, Y, X_2, Z_1, Z_2) for attributes: The pair X_1, Y identifies the attribute whose behavior will be made available for instances of X_2 by providing the implementations of the read-operation Z_1 and the write-operation Z_2.

Again, the key constraints and referential integrity constraints are obvious. Additionally, it must be guaranteed that for *FashionType(X, Y)*, the complete behavior of Y will be provided by means of *FashionDecl*s and *FashionAttr*s:

1. $\forall$ X, Y, Z, U, V $\exists$ W
   FashionType(X, Y) $\wedge$ Decl^i(Z, Y, U, V) $\Longrightarrow$ FashionDecl(Z, X, W)
2. $\forall$ X, Y, Z, U $\exists$ V_1, V_2
   FashionType(X, Y) $\wedge$ Attr^i(Y, Z, U) $\Longrightarrow$ FashionAttr(Y, Z, X, V_1, V_2)

Now, we stated all constraints which have to be asserted if we want the instances of one type version to be substitutable for another type version by means of **fashion**.

Implementation After finishing the above design, implementation of these extensions — adding versioning and masking — proceeds as follows. First, the above base predicates, rules, and constraints have to be inserted into the system. This simple keyboard exercise can be performed within an hour. Second, the *Analyzer* has to be expanded such that it can accept the syntax for the fashion clause and perform the necessary modifications to the extensions of the base predicates introduced above. Since Lex and Yacc have been employed, this task takes a single day. Third, the *Runtime System* has to be enabled to work with objects which are not

[9] i.e. we do not want to change parameter types, for example.

instances of a subtype of the expected type but are instances of another version of this type which is guaranteed to possess the necessary behavior. Since dynamic binding had already been present in the system due to the possible refinement, the extension to the *Runtime System* could be held at a minimum. Nevertheless, lasting one week, this is the hardest of the three necessary modifications.

4.2 User's Flexibility

As is our goal, the user gains more flexibility by means of our approach, too. Since we decoupled schema modification from asserting the schema to be consistent, the user may be allowed to perform any possible complex schema change. Regardless to what he is really changing, *Consistency Control* never looses track of possible errors.

Again, take the schema of section 3 as an example. This schema modeled the world before there was the need to distinguish between cars with and without a catalyst: there didn't exist a car with catalyst. Thus, the information not to have a catalyst was redundant and was therefore not part of the database schema. But some years later, things got more complicated: cars with catalyst have to tank unleaded fuel while cars without catalyst still need leaded fuel. Since there remain still some cases where this distinction is not needed, our schema designer decided that tailoring the existing hierarchy is the best thing to cope with the changed situation: the new schema version *NewCarSchema* should contain not only a type *Car* but also two new subtypes of *Car*: *PolluterCar* and *CatalystCar*, both equipped with an operation *fuel* returning the sort of fuel the corresponding car needs:

```
type Car is
   ...    !! see definition of Car in section 3
end type Car;

sort Fuel is enum (leaded, unleaded);

type PolluterCar supertype Car is
   operations
      declare fuel: → Fuel;
   implementation
      define fuel is return leaded;
end type PolluterCar;

type CatalystCar supertype Car is
   operations
      declare fuel: → Fuel;
   implementation
      define fuel is return unleaded;
end type PolluterCar;
```

The evolution of the old schema *CarSchema* to the new schema *NewCarSchema* is not just adding two new subtypes to the existing schema. Even if this change would result in the same type definitions, it does not reflect the meaning of the changed situation in the world we want to model.

If we take a closer look at the semantics of this evolution, we have to do the following:

1. Defining a new type *PolluterCar* within schema *NewCarSchema*.
2. Defining *PolluterCar* as an evolution of type *Car* from schema *CarSchema*.

3. Adding an operation *fuel:* $\rightarrow$ *Fuel* to this (renamed) type.
4. Defining a new type *Car* by using the same textual definition as *Car* in schema *CarSchema*.
5. Defining a new type *CatalystCar*.
6. Defining both *PolluterCar* and *CatalystCar* as subtypes of *Car*.
7. Defining an adoption mechanism (via *FashionType*) to be able to reuse the instances of the "old" *Car* type definition as instances of *PolluterCar* in the new schema.

We do not want to give the complete list of primitive evolution operators which have to be executed to get the expected result—they are obvious. What we want to stress at this point is, that the user is really able to execute exactly those changes which reflect the specific evolution in the modeled world. To perform these changes, the user can call the corresponding operations of the *Analyzer*'s interface in a step-by-step manner within a schema modification session. If he did everything right—i.e. if none of the constraints will be violated—*Consistency Control* will accept his modification and the user can rely that the runtime system will interpret his schema in exactly the intended and appropriate way. But beside the manual execution of these steps, the user also has the possibility to abstract from this concrete case and to program a new parameterized complex schema evolution operator which will be added to the implementation of the *Analyzer*. Note, that all other modules of the system are not touched by this extension. If we assume the *Analyzer* as a dedicated (and probably graphic) schema editor, such a program can be realized by an editing macro.

Of course, the system developer can easily provide the system with libraries containing lots of such complex evolution operators, like "deleting nodes within the type hierarchy" or "restructuring the type hierarchy". We did introduce another example of such a complex evolution operation in section 2 already: if we want to change the argument list of an operation, even those locations within the code of (other) operations have to be changed, which contain calls of this operation. This case could be supported by a complex evolution operator which finds out all relevant locations and offers them to the user to do the necessary change. The set of such complex evolution operations will never be complete, there will always remain cases which are specific to the modeled situation and cannot be foreseen by the developer. With our approach, this does not matter at all—the user can easily make up for the developer's lack of foresight by defining the complex operations for his/her own.

5 Conclusion

A new approach to schema management was introduced. The generic architecture of our approach centers the schema management tasks around the consistency control component. Besides the runtime system this component is the most difficult to implement. Deciding to rely on deductive database technology cuts the implementational efforts for this component down to zero — provided that a deductive database is available — and exhibits the further advantage that schema consistency can be stated declaratively, easing its definition.

The proposed generic architecture has been instantiated by designing a specific simple schema manager. The necessary design effort consisted solely in modeling the data model the schema manager has to handle. In order to assess the achieved flexibility of our approach, the simple schema manager was enhanced by advanced features for inconsistency cures and versioning. As for the simple schema manager, it was shown that the main task consisted in designing the cures and versioning concepts to be incorporated. The necessary "implementation" of these features consisted in feeding some additional definitions into the consistency control component,

performing some slight extensions to the analyzer and the necessary modifications to the runtime system.

Acknowledgement This work was partially supported by the German research council (DFG) under contract SFB 346 (A1).

References

[1] S. Abiteboul, P. Kanellakis, and E. Waller. Method schemas. In *PODS*, pages 16–27, 1990.

[2] J. Banerjee, W. Kim, H.-J. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriente d database systems. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 311–322, San Francisco, 1987.

[3] D. S. Batory. Concepts for a database system compiler. In *Proc. of the 17nth ACM SIGMOD*, pages 184–192, 1988.

[4] D. S. Batory, T. Y. Leung, and T. E. Wise. Implementation concepts for an extensible data model and data language. *ACM Trans. on Database Systems*, 13(3):231–262, Sep 1988.

[5] S. Bocionek. Dynamic Flavors. Technical report, Institut der Technischen Universität München, 1987.

[6] M. Carey and D. J. DeWitt. An overview of the EXODUS project. *IEEE Database Engineering*, 10(2):47–53, Jun 1987.

[7] W. Cellary and G. Jomier. Consistencies of Versions in Object-Oriented Databases. In *VLDB*, pages 432–441, Brisbane, Australia, Aug. 1990.

[8] W. Cellary, G. Jomier, and T. Koszlajda. Formal model of an object-oriented database with versioned objects and schema. In *Proc. Intl. Conf. on Database and Expert Systems Applications*, pages 239–244, 1991.

[9] U. Dayal and J. Smith. PROBE: A knowledge oriented database management system. In *Proc. Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, 1985.

[10] C. Delcourt and R. Zicari. The design of an integrity consistency checker (icc) for an object-oriented database system. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, pages 97–117, 1991.

[11] M. J. Carey et al. The architecture of the EXODUS extensible DBMS. In *Workshop on Object-Oriented Database Systems*, pages 52–65, 1986.

[12] O. Deux et al. The o_2 system. *Communications of the ACM*, 34(10):34–48, 1991.

[13] P. Schwarz et al. Extensibility in the starburst database system. In *Proc. Int. Workshop on Object-Oriented Database Systems*, 1986.

[14] D. Goldhirsch and J. Orenstein. Extensibility in the PROBE database system. *IEEE Database Engineering*, 10(2):24–31, Jun 1987.

[15] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM: A strongly typed persistent object model with polymorphism. In *Proc. der GI Fachtagung "Datenbanken für Büro, Technik und Wissenschaft" (BTW)*, 1991.

[16] W. Kim and H.-T. Chou. Versions of schema for object-oriented databases. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 148–159, Los Angeles, 1988.

[17] J. McPherson and H. Pirahesh. An overview of extensibility in Starburst. *IEEE Database Engineering*, 10(2):32–39, Jun 1987.

[18] G. Moerkotte and S. Karl. Efficient consistency checking in deductive databases. In *2nd. Int. Conf. On Database Theory*, pages 118–128, August 1988, Bruges, Belgium, 1988.

[19] G. Moerkotte and P.C. Lockemann. Reactive consistency control in deductive databases. *ACM Trans. on Database Systems*, 16(4):670–702, 1991.

[20] G. Moerkotte and K. Rösch. On the compilation of consistency constraints. In *Proc. 2nd. Int. Workshop on the Deductive Approach to Information Systems and Databases*, 1991.

[21] G. Moerkotte and A. Zachmann. Multiple substitutability without affecting the taxonomy. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*, pages 120–135, Wien, Mar 1992.

[22] A. Skarra and S. Zdonik. Type evolution in an object-oriented database. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 393–415. MIT Press, 1987.

[23] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proc. of the 15nth ACM SIGMOD*, pages 340–355, 1986.

[24] E. Waller. Schema updates and consistency. In *DOOD*, pages 167–188, München, Germany, 1991.

[25] R. Zicari. A Framework for O_2 Schema Updates. Technical Report 38-89, GIP Altair, Oktober 1989.

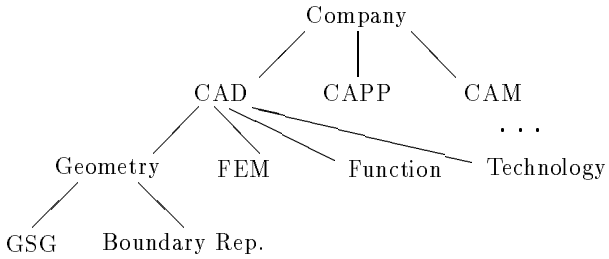

Figure 3: A Sample Schema Hierarchy

A Schema Management

A.1 The Notion of Schema in GOM

Consider a company manufacturing mechanical artifacts. Typically, this company has at least a CAD, CAPP (Computer Aided Production Planning), CAM, and a marketing department. Not all types required to represent the objects needed in one department are also necessary for all the other departments. For example, the CAD department utilizes types for representing geometric data. Dealing with 3-D models, boundary representations as well as constructive solid geometry representations are applied. These types may not be needed, e.g., in the marketing department. Covering the whole spectrum of data to be processed within the company, thousands of types are needed. Thus, it might be worthwhile not to have a large unstructured set containing all these types but, instead, to structure the set of all types, e.g., according to the varying requirements of different departments. Further, it is useful to structure the data for each department according to other aspects. Therefore, a hierarchical structuring mechanism of the set of all available types is needed.

As an example, consider the hierarchical structure of the information as found in a typical company manufacturing mechanical artifacts. In Figure 3 the division of data processed in the whole company into three aspects corresponding to the departments *CAD*, *CAPP*, *CAM*, and *Marketing* is shown. On a second level, the data processed in the *CAD* department is further divided into data concerning geometry (*Geometry*), finite element models (*FEM*), functional aspects (*Function*), and technological aspects (*Technology*). On the last level of nesting of the structuring of the data, the data types necessary to capture geometry are classified into two partitions, one class comprising constructive solid geometry models (*CSG*) and the other comprising boundary representation models (*BoundaryRep*).

Note that, so far, not even a single type has been visible. Instead, only a division of all possible types into several sets has been sketched. By the mechanisms presented before in this book, it is impossible to express this partitioning of the company's information and make it known to the database. Consequently, a new mechanism—the notion of *schema*—has to be introduced. As a first approximation, a schema can be thought of as a set of types. Then, Figure 3 represents a *schema hierarchy*. The descendant schemas of a schema are called *subschemas*, e.g., *CAD*, *CAPP*, and *CAM* are subschemas of *Company*. The schema *BoundaryRep* will then, e.g., contain the types *Cuboid*, *Surface*, *Edge*, and *Vertex*. Note, that these types are of no relevance to the *Marketing* department. A typical example type of the *CAPP* subschema is the object type *Schedule*—representing a processing schedule for manufacturing—which is fundamental for the *CAPP* department and of no interest for the *CAD* department.

Partitioning the set of all types into subsets—that is, providing a structuring mechanism—is

not the only functionality a schema provides. In summary, there exist three different equally important aspects of schema:

1. structuring the set of all types and governing their persistence,
2. allowing high-level information hiding, and
3. providing distinct name spaces.

These points are discussed in more detail in subsequent sections.

A.2 The Schema Definition Frame

A schema with its subschemata can be specified by the *schema definition frame*. Part of the schema hierarchy of Figure 3 can be specified by the following three (preliminary) schema definition frames:

```
schema Company is
        subschema CAD;
        subschema CAPP;
        subschema CAM;
        subschema Marketing;
end schema Company;

schema CAD is
        subschema Geometry;
        subschema FEM;
        subschema Function;
        subschema Technology;
end schema CAD;

schema Geometry is
        subschema CSG;
        subschema BoundaryRep;
end schema Geometry;

schema BoundaryRep is
        type Cuboid is ...;
        type Surface is ...;
        type Edge is ...;
        type Vertex is ...;
        var exampleCuboid: Cuboid;
end schema BoundaryRep;
```

Here, only the schema *BoundaryRep* contains type definitions —i.e., the type specifications of *Cuboid*, *Surface*, *Edge*, and *Vertex*. However, in general every schema may contain actual type definitions. Note, that we also defined a variable *exampleCuboid* within the schema *BoundaryRep*. Thus, a schema is not only used to structure the set of all types but also to group variables. Types, variables, and (sub-) schemata are generalized to *schema components*. Then, a schema is a collection of schema components.

A schema also implicitly governs persistence:

> A schema is always persistent, and with it, all its schema components!

This means, that the keyword **persistent** is of no meaning within a schema definition frame. It might as well be dropped as we already did in the example schema definitions. Thus, all the schemata defined above and all the types and variables included are persistent.

A.3 Information Hiding for Schemata

Remember the type definition frame. There, the **public** clause controlled the possible operations to be executable on the instances of the type. Only those operations which appeared in it were executable by clients of the type—or, more precisely, by clients of isntances of the type. The same occurs on a different level when considering schemata and contained types. Not all types have to be visible to all other schemata. In the above example, the type *Cuboid* in the schema *BoundaryRep* should be visible to other schemata whereas the other types *Surface*, *Edge*, and *Vertex* should not be visible, since they are only used to implement the type *Cuboid*. This leads to information hiding on a higher level. Therefore, the **public** clause is also introduced for schemata:

```
schema BoundaryRep is
        public Cuboid;
        interface
            type Cuboid is ...;
        implementation
            type Surface is ...;
            type Edge is ...;
            type Vertex is ...;
end schema BoundaryRep;
```

This schema realizes hiding the types *Surface*, *Edge*, and *Vertex*. Or, expressing it from a different viewpoint, only the type *Cuboid* may be used by other schemata. Further note, that the type definitions of the types made public are defined in the **interface** section of the schema definition frame whereas those only used for internal purposes are defined within the **implementation** section. Summarizing, a *schema definition frame* consists of three sections:

1. the **public** section where all the schema components made public to the super schema and—as we will see—possibly other schemata are listed,
2. the **interface** section where all the schema components made public are to be specified, and
3. the **implementation** section where all the schema components used for internal implementation purposes are specified.

Remember that schema components can be types, variables, free operations, and (sub-) schemata.

A.4 Name Spaces of Schemata

The third aspect of a schema is that it provides a *name space*. So far, without the notion of schema, all type names had to be distinct. Also, all names of global variables had to be distinct. There existed a single *global name space*. This is not necessarily easy to guarantee. In fact, in both of the schemata *CSG* and *BoundaryRep* a type representing cuboids may be appropriate. By allowing each schema to have its own *local name space*, it is possible to define a type *Cuboid* in the schema named *CSG* and another type *Cuboid* in the schema *BoundaryRep* without inducing a name conflict. Thus, the following two schema definitions are both valid, even in conjunction:

```
schema CSG is
    public Cuboid;
    interface
        type Cuboid is ...;
    implementation
        ...
end schema CSG;
```

```
schema BoundaryRep is
    public Cuboid;
    interface
        type Cuboid is ...;
    implementation
        ...
end schema BoundaryRep;
```

This becomes a very important feature when considering different designers developing the types for constructive solid geometry and boundary representation independently and in parallel. Thus, providing different name spaces for different schemata allows to design schemata independent of each other—thereby avoiding the necessity of a global consensus of naming.

Unfortunately, this advantage is not for free. As soon as the type *Cuboid*, which is a public type in both *CSG* and *BoundaryRep*, is referenced (i.e., used) in *Geometry*, this reference cannot be resolved uniquely. Both *Cuboid* types qualify. This problem is solved by explicitly renaming both types in the *Geometry* schema. Under the assumption that both *Cuboid* definitions are made public in *Geometry*, this is done by extending the *subschema* entries in the schema definition frame of *Geometry*:

```
schema Geometry is
    public CSGCuboid, BRepCuboid;
    interface
        subschema CSG with
            type Cuboid as CSGCuboid;
        end subschema CSG;
        subschema BoundaryRep with
            type Cuboid as BRepCuboid;
        end subschema BoundaryRep;
end schema Geometry;
```

The **subschema** entry has been expanded as indicated by the keyword **with**. Following the keyword **with**, a list of schema components follows. Each entry in this list is preceded by the kind of the schema component to be renamed, followed by the "old" name, followed by **as**, followed by the "new" name. Thus, in the example, the type *Cuboid* of schema *CSG* is renamed to *CSGCuboid* and can be referred to by this new name within the schema *Geometry* and its super schema *CAD*. The latter holds because both new names for the *Cuboid* types have been made public. This **public** clause implies also that the renaming, that is the statement of the **subschema** entry, has to be in the **interface** section. If the types had not been made public, the subschema entry would have to be in the **implementation** section. For renaming imported types the qualifier **type** is utilized. If variables or operations are to be renamed, the qualifiers **var** and **operation** are used, respectively.

One might ask, whether the advantage of different name spaces for schemata becomes obsolete since if name conflicts occur they have to be explicitly resolved by renaming and if they do not occur, there is no effect of different name spaces. Since a name conflict occurs only if two public schema components have the same name and both are used within a single schema, the number of name conflicts is limited by the different name spaces of schemata. Further, independent development of schemata is not affected by the name conflicts. These have to be resolved within the single schema using the components whose names conflict.

A.5 Importing Schemata

Public types or other schema components implemented in a specific schema should be available to the implementor of other schemata. Consider, for example, the implementor of the schema *CAD*.

He or she has free access to the public types specified in the schemata *Geometry*, *FEM*, *Function*, and *Technology* but to no other schema components. This is the—somewhat restrictive—default. That is, exactly those schema components of direct subschemata which are specified as being **public** are available to the implementor of the super schema.

There exist several reasons for this restrictive default. In general, a schema corresponds to a certain abstraction level of the application. The consequence of abstraction hierarchies often is that details at lower levels are of no relevance to levels higher than the direct upper level. Thus, the default provides for automatically hiding the details.

Of course, this is an idealized situation and there often exist good reasons to use schema components defined in a schema several levels down or even to use schema components of a schema which is not a direct or indirect subschema. If this is the case, the implementor can *import* other schemata. Since importing a schema implies a dependency, this mechanism should be handled with care. Therefore, it is required that any import of a schema has to be stated explicitly. Hence the restriction on the default.

Yet another reason for the restrictive default is the name space. Since the imported schemata have most likely been developed independently, name conflicts may occur within the total set of locally defined schema components unioned with all imported schema components. Assume a type *Cuboid* has also been defined in schema *CAPP*. Then, if the components of *CAPP* would automatically be visible to all other schemata, i.e., if there were an unconstrained import of all other schemata, another name conflict on *Cuboid* occurs within *Geometry* although the type definition of *Cuboid* within *CAPP* is of no relevance to the schema *Geometry*. Resolving name conflicts for irrelevant schema components is a tedious task. The whole advantage of different name spaces would be nihilated.

Consider a tool which allows the automatic conversion of CSG into boundary representation. To integrate this tool into the schema hierarchy, a third subschema *CSG2BoundRep* of schema *Geometry* containing its implementation is introduced. Due to its functionality, it has to deal with both types of cuboids, the type *Cuboid* in *CSG* and the type *Cuboid* in *BoundaryRep*. Nevertheless, *CSG2BoundRep* has no access to either of the *Cuboid* type definitions. What is needed is the explicit import of the schemata *CSG* and *BoundaryRep*. These are realized by utilizing the **import** clause within the schema definition frame:

```
schema CSG2BoundRep is
        public convert;
        interface
            import /Company/CAD/Geometry/CSG with
                type Cuboid as CSGCuboid;
            end schema CSG;
            import /Company/CAD/Geometry/BoundaryRep with
                type Cuboid as BRepCuboid;
            end schema BoundaryRep;
            ...
end schema CSG2BoundRep;
```

The **import** clause is followed by a *schema path* specifying the schema to be imported. A schema path is a sequence of schemata separated by backslashes, i.e., "/". The first backslash indicates that the specified path starts at the root. Thus, "/Company" refers to the root schema *Company*. "/Company/CAD" refers to its subschema *CAD*, and so on until the needed schema is reached. A path starting at the root is called an *absolute schema path*. A relative schema path starts with either

- a schema name, in which case this schema name refers to the subschema of the enclosing schema and this is the start of the path, or

- a double dot, i.e., "..", in which case the super schema of the enclosing schema is the start of the path.

Thus, in our example, the following paths are equivalent:

- /Company/CAD/Geometry/CSG and
- ../CSG

The double dot can also be iterated, e.g., "../.." refers to *Company*, if utilized directly within *Geometry*. It refers to *CAD*, if it occurs within *BoundaryRep* or *CSG*.

By starting with a schema name, only direct or indirect subschemata can be reached. In the example of Figure 3, the schema *CSG* can be imported into *CAD* by specifying the **import** clause utilizing the schema path "Geometry/CSG".

The renaming within the **import** clause is analogous to the renaming as employed in the **subschema** clause. Additionally to the above definition of the schema *CSG2BoundRep*, it has to be defined as a subschema of *Geometry* by adding the appropriate **subschema** entry.

Let us briefly summarize the effects of importing a schema B by a schema A using the **import** clause: all the schema components —types, variables, free operations—defined in B are—by specifying **import** B in A—readily available and can be accessed in the same manner as any other schema component directly defined in A. Also, all the schema components public in any subschema of A are available transparently in A. The only problem occurrs in the case of a name conflict. Name conflicts in a schema A can only occur if

1. the same name was used at least twice for the same kind of schema component, e.g., a type, within the set union of all schema component names of A, its subschemata and its imported schemata, and
2. this schema component was used within A, e.g. in an attribute definition.