Towards More Flexible Schema Management in Object Bases

Guido Moerkotte Andreas Zachmann

Universität Karlsruhe
Fakultät für Informatik
D-7500 Karlsruhe
West Germany
Netmail: moer/zachmann@ira.uka.de

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.

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 [9, 6], Genesis [3, 4], Postgres [21], Probe [7, 12], Starburst [11, 13]).

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 [20] 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 O2 system [10], Zicari proposes to cure inconsistencies by immediate conversion [23]. 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 [14]. 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 proper-
ties of schemas and schema updates [1, 22]. 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 [8]. 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. [17], and mechanisms to automatically generate repairs for detected inconsistencies have also been designed and implemented [16].

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 [13] 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.

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. 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 [19]. Thus, only the execution of a whole set of primitive evolution operations may result in a consistent schema. Consequently, decoupling schema evolution operations from schema consistency is a necessity, and as such one of our main goals. 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 leaders 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.

2.2 The Solution

We start with the proposal of a generic architecture of an object-oriented database system. It is designed such that all involved modules are easily and undoubtedly localizable, and the necessary changes to each module are limited and clearly distinguishable.

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 mo-
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.

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 built 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.

Designing, implementing and modifying 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. 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 [17] for efficient consistency checking and [16] 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 ([13]). In order to keep the schema model short and simple, several restrictions have been applied. Nevertheless both, structural and behavioral, aspects (in the sense of [23]) as well as multiple inheritance will be modeled. Hence, the core of (almost) every object-oriented data model is captured.

3.1 Example

The running example is based on the loaded/unloaded cars-example of [20]. The following schema is (hopelessly) self-explanatory:

```
schema CarSchema is
  type Person is [name: string; age: int];
  type Location is [longi: float; lati: float];
    distance : [Location ——> float; 
  implementation
    ••!!! uses longi and lati!! as well as city name.
end type Location;
  type City supertype Location is [name: string; noOfInhabitants: int;]
    refine distance : [Location ——> float; 
  implementation
    ••!!! implementation uses longi and lati!! as well as city name.
end type City;
  type Car is
    [owner: Person; maxspeed: float;
     milage: float; location: City;]
    changeLocation : [Person, City ——> float;
  implementation
    changeLocation (driver, newLocation) is
      begin
        self.milage := self.milage +
            self.location.distance(newLocation);
        self.location := newLocation;
      return self.milage;
      end changeLocation;
end type Car;
end CarSchema;
```

where keys are underlined. 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 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:n relationship implements.

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.

```
<table>
<thead>
<tr>
<th>Schema</th>
<th>sid1</th>
<th>CarSchema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>tId1</td>
<td>Person</td>
</tr>
<tr>
<td></td>
<td>tId2</td>
<td>Location</td>
</tr>
<tr>
<td></td>
<td>tId3</td>
<td>City</td>
</tr>
<tr>
<td></td>
<td>tId4</td>
<td>Car</td>
</tr>
<tr>
<td>Attr</td>
<td>tId1</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Decl</td>
<td>did1</td>
<td>distance</td>
</tr>
<tr>
<td></td>
<td>did2</td>
<td>distance</td>
</tr>
<tr>
<td></td>
<td>did3</td>
<td>changeLocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Code</td>
<td>cid1</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>cid2</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>cid3</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
```

Figure 2: Extensions for the Example

Having modeled all entities and some 1:n relationships, we proceed with

```
SubTRel(Typeld, Typeld)
DeclRefine(DeclId, DeclId)
```

SubTRel(X, Y) states that X is a subtype of Y and DeclRefine(X, Y) states that X is a refinement of Y.

While the Consistency Control should not inspect the code implementing operations, it needs some information about the code: the operations called and the attributes accessed by it. These relationships are covered by:
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. Besides these easy to state constraints there exist multiple inheritance constraints and typing constraints. We state only one example for each group:

Keys and other Uniqueness Constraints Every type name can be used at most once within one schema:  
\[ \forall X_1, X_2, Y_1, Y_2, Z \quad \text{Type}(X_1, Y_1, Z) \land \text{Type}(X_2, Y_2, Z) \implies (Y_1 = Y_2 \implies X_1 = X_2) \]

Referential integrity and other existence constraints For any declaration a piece of code implementing it has to be present:
\[ \forall D, T_e, O, T_i \exists C_1, C_2 \quad \text{Decl}(D, T_e, O, T_i) \implies \text{Code}(C_1, C_2, D) \]

Simple Constraints for SubTRel and DeclRefine For the following we need the transitive closure for SubTRel and DeclRefine:
- SubTRel\’(X, Y) \land \exists \text{SubTRel}(X, Y)
- SubTRel\’(X, Y) \land \exists \text{SubTRel}(X, Y, \text{SubTRel}\’(Y, Z))
- DeclRefine\’(X, Y) \land \exists \text{DeclRefine}(X, Y)

Both relationships have to be acyclic. Additionally, in GOM, there must exist a unique root called any:
\[ \forall X \quad \neg \text{DeclRefine}(X, X) \land \exists Y, Z \quad \text{SubTRel}(X, Y) \land \forall X, Y, Z \quad \text{Type}(X, Y, Z) \implies (X = \text{ANY} \lor \exists \text{SubTRel}(X, \text{any})) \]

Multiple Inheritance Constraints For our simple schema manager, 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:

\[ \forall T, A, D_1, D_2 \quad \text{Attr}'(T, A, D_1) \land \text{Attr}'(T, A, D_2) \implies D_1 = D_2 \]
\[ \forall T, T_1, T_2, O, T_{12}, D_1, D_2 \exists D \]
- SubTRel\’(T, T_1) \land \exists \text{SubTRel}\’(T, T_2) \land \exists \text{Decl}(D, T_1, O, T_{12}) \land \exists \text{Decl}(D, T_2, O, T_{12}) \implies \text{DeclRefine}(D, T_1) \land \text{DeclRefine}(D, T_2) \]

where the following rules capture inheritance and refinement:
- Attr'(T, A, D) \implies Attr(T, A, D).
- Attr'(T, A, D) \land \exists \text{SubTRel}\’(T_1, T_2) \land \text{Attr}(T_2, A, D).
- Decl'(X, Y_{12}, Z, Y_{12}) \land \exists \text{Decl}(X, Y_{12}, Z, Y_{12}).

3.4 Schema/Object Consistency

Assume that for each type there exists exactly one physical representation for all objects of this type, the base predicate \text{PhRep}(\text{PhRepId}, \text{TypeId}) is introduced to model physical representations. \text{TypeId} denotes the identifier of the unique type whose objects have this representation. We assume the implicit existence of physical representations of built-in sorts.

In order not to confuse the logical and the physical level, we introduce the notion of slots 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:
\[ \text{Slot}(\text{PhRepId}, \text{AttrName}, \text{PhRepId}) \]

The third argument \text{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 \text{PhRep} and \text{Slot} up to date. Besides key and referential integrity constraints, three other constraints are needed in order to guarantee the consistency between the physical and the logical part. We require that there exists only one physical representation for each type, that the slots for each attribute for a given type must be unique, and that for every type there must exist a corresponding slot for every associated attribute including the inherited ones. These constraints can easily be expressed.

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

\[ \text{PhRep} \begin{array}{|c|c|c|} \hline \text{clId} & \text{tid1} & \text{tid2} \\ \hline \text{clId} & \text{tid1} & \text{tid3} \\ \hline \text{clId} & \text{tid1} & \text{tid4} \\ \hline \text{Slot} & \text{clId1} & \text{name} & \text{clIdString} \\ \hline \end{array} \]

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. [20, 23]) 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(fid4, fuelType, tid_string). Clearly, constraint (*) is violated since

\[ \text{Attr}(\text{tid}_4, \text{fuelType}, \text{tid}_{\text{string}}) \land \text{PhRep}((\text{clid}_4, \text{tid}_4) \Rightarrow \text{Slot}((\text{clid}_4, \text{fuelType}, \text{clid}_{\text{string}}) \text{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. -Attr(\text{tid}_4, \text{fuelType}, \text{tid}_{\text{string}}),
2. -PhRep(\text{clid}_4, \text{tid}_4), and
3. +Slot((\text{clid}_4, \text{fuelType}, \text{clid}_{\text{string}}).

The first possibility is to undo just the proposed change to the schema. Since the tuple \text{PhRep}((\text{clid}_4, \text{tid}_4) is present in the Object Base Model if and only if there exist instances of \text{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 ([16]).

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 satisfy the customers’ needs at all. One deficiency is the limited facility to repair schema-object consistency by conversion of all
instances. Thus, extending the prototype by an additional adoption mechanism—like masking of object—would be nice. In [19] we did show how Consistency Control can easily be extended to model versioning of schemas and masking of objects along the version graph by means of the fashion clause (see [18]). Due to space limitations, we skip the detailed description: we solely accompany the development team's effort needed to implement the intended release GOM-V1.0.

After finishing the design to incorporate versioning and masking, the newly introduced base predicates, rules and constraints have to be inserted into the system. This simple keyboard exercise can be performed within an hour. To accept the syntax for the fashion clause and to perform the necessary modifications to the newly introduced base predicates, the Analyzer has to be expanded. Since Lex and Yacc have been employed, this task takes a single day. Second, the Runtime System has to be enabled to work with objects which are not instances of a subtype of the expected type but are instances of another version of this type for which a fashion clause guarantees 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 all necessary modifications.

### 4.2 User's Flexibility

As is our goal, the user gains more flexibility by 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 in section 3

sort Fuel is enum (leaded, unleaded);

type PolluterCar supertype Car is
    fuel:  -> Fuel is return unleaded;
end type PolluterCar;

type CatalystCar supertype Car is
```

fuel:  -> 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:

3. Adding an operation fuel:  -> Fuel to this (repeated) 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 Fashion-Type) 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 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 simple schema manager. The necessary design effort consisted solely in modeling the data model the schema manager has to handle. In order to partly assess the achieved flexibility of our approach (a more thorough investigation can be found in [19]), the simple schema manager was enhanced. It was argued that the necessary implementational effort is held at a minimum.

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

References