A Framework for Schema Evolution by Meta Object Manipulation*

Markus Tresch

Department of Computer Science, Information Systems – Databases
ETH Zürich, CH-8092 Zürich, Switzerland
e-mail: tresch@inf.ethz.ch

Abstract

In this paper we address the problem of schema evolution in object-oriented database systems. Most currently available database prototypes either completely lack schema evolution facilities, or offer a restricted set of special purpose schema evolution operators supporting simple schema changes. Our approach is different. We consider schema objects as objects like others and have the meta schema fully available at run-time. Thus any operator of our object manipulation and query algebra can be applied to meta objects in the same way as they are used for data objects. We discuss the additional problems that arise when treating schema objects as normal objects and making the algebra work on the meta schema as well. Furthermore, we overview how to solve these problems on different levels of implementing schema evolution.

1 Introduction

The logical structure of the data stored in a database is defined by the database schema. In object-oriented database management systems, this schema consists of types (arranged in an inheritance lattice) and classes (forming a subclass hierarchy). In general, such a logical structure is likely to undergo changes during the lifetime of a database. According to new requirements, it may dynamically change in several ways, even if the database is already populated with objects. In this paper, we summarize in the term *schema evolution* several types of possible changes to the logical database schema: original schema design, schema updates during database lifetime, and schema integration.

Most current object-oriented database system products and prototypes do not allow free dynamic changes to the schema with populated databases. The reason is that either the schema information is not available in an appropriate way, or it is simply a read-only data-dictionary, that cannot be changed such that the database is kept consistent. Most

*published in: Proceedings of the 3rd Int'l Workshop on Foundations of Models and Languages for Data and Objects, Aigen, Austria, September 1991. Tech. Report, Institut für Informatik, TU Clausthal.

often in these systems, after the design of the database schema is committed, no more changes to the structure are allowed.

More advanced systems support schema update facilities. There is no longer a clear cut between the design and application phases. Nevertheless, schema evolution possibilities are very limited and are normally restricted to simple changes, realized by a given set of special purpose schema update operators. Furthermore, changing the schema information is often only allowed, if the database is empty. If they allow to modify populated databases, they either include a data migration utility to adapt existing data objects to the changed schema, or some other mechanism has to ensure consistency between data and structure.

An early investigation of changing types in populated database exists for the ENCORE database management system [SZ86, SZ87]. This work addresses the effects of type changes to objects and to programs that use objects of the type. The first systematic analysis of desirable schema evolution possibilities was done for the ORION data model in [BKKK87, BCG+87]. A set of all necessary schema updates was listed and organized in a taxonomy. Similar enumerations can also be found for the O_2 [Zic90] and the GemStone DBMS [PS87], actually forming a subset of ORION's taxonomy. The schema update primitives of the above taxonomy are realized as special purpose operators for schema management. That is, for every possibility there is a function. So finally, the collection of these operators build a "schema update language".

Our approach is different in that we try to use a transparent meta level: In our object-oriented data model COCOON, we consider schema objects as objects like others, and have the meta types and meta classes available at any time. We allow the use of our object algebra COOL in the same way on the meta level as on the primary object level. So there is no difference in the syntax of the descriptive algebra operators, whether they query or update data-, schema-, or meta objects. In fact, representation of the meta schema within the same model is an old idea. However, usually such meta data serve only for documentation purposes. That is, only *queries* are allowed. In contrast, here we investigate the feasibility of *updates* to meta objects and their propagation to the instances.

The paper is organized as follows. In the next section, we sketch our object model and its algebra. We introduce the schema and meta level objects in Section 3. In Section 4 we discuss the problems that arise, if we allow the algebra to work on the meta objects.

2 The COCOON Object Model

2.1 Basic Concepts

The COCOON object model used here is an object-function model in the sense of [Bee89]. As described in [SS90, SLT91] it is a *core* object model, meaning that it is restricted to the following essential constituents:

- *Objects* are instances of abstract types, specified by a set of applicable operations. In contrast, *data* are instances of concrete types, that can be atomic (numbers, strings) or constructed (tuples, sets).
- *Functions* are the generalized abstraction of attributes (stored or computed), update methods, and relationships between objects. They can be single- or set-valued. Func-

tions are described by their signature (the name, domain- and range types).

- *Types* are normally named and are defined by a set of functions that are applicable to their instances. They are arranged in a type lattice, where subtypes inherit all functions from their supertypes. The type **object** is the predefined root of the lattice. Objects can be instances of several types at the same time (multiple instantiation).

- *Classes* are strictly distinguished from types. Classes are typed collections of objects. They form a subclass hierarchy, meaning that the collection of objects in a subclass is included in the collection of its superclasses. The class **Objects** is the predefined root of the subclass hierarchy. Objects can be member of multiple classes at the same time (multiple class membership).

One important feature of our model that is usually not found in other models is *class predicates*. Constraints can be attached to a class, that must be satisfied by every member object of the class. Class predicates may either be necessary or necessary and sufficient conditions. Classes with necessary and sufficient predicates are populated automatically: whenever objects are added to their superclass(es) and the predicate evaluates to true, the new objects are included in this class, too.

As an example, consider the type *person* and the two classes *Persons* and *Youngs* below. Both classes have the same member type *person*. The class *Youngs* is defined as a subclass of *Persons*, holding exactly those persons that satisfy the class predicate $age < 30$. The selector **all** in the **where** clause indicates that the class predicate is necessary and sufficient (in contrast to "**some** " meaning only necessary), such that the member objects of *Youngs* are automatically computed from those of *Persons*.

define type *person* **isa object** = *name: string, age: integer ;*
define type *employee* **isa** *person* = *salary: integer, empl: company* **inverse** *staff ;*
define type *company* **isa object** = *name, city: string,staff:* **set of** *employee* **inverse** *empl;*

define class *Persons : person* **some** *Objects ;*
define class *Youngs : person* **all** *Persons* **where** *age<30 ;*
define class *Employees : employee* **some** *Persons ;*
define class *Companies : company* **some** *Objects ;*

2.2 Object Algebra

The query and object manipulation language COOL is an extension of (nested) relational algebra. It provides a collection of generic operators that are applied to sets of objects, namely class extensions or query results. The query operators have object-preserving semantics. That is, they return (some of) the already existing input objects, instead of generating new (copies of) objects. This makes COOL orthogonal and query operators can be nested. There is a query operator for selection (**select** *[city(empl(e))='Zuerich'](e:Employees);*), projection (**project** *[name,staff](Companies);*), extend (**extend** *[UD$:= salary*1.4] (Employees)*), and set operations (**union, intersect, difference**).

Besides query operators, COOL provides also a collection of generic set-oriented update operators. They are used to create new objects (*john:=* **insert** *[name:='J.Smith', age:=29](Persons);*) or to destroy one (**delete** *(paul);*). In addition there are two operators to include existing objects into sets (**add** *[marc](staff(ibm));*), and to exclude objects from sets (**remove** *[dec] (Companies);*). Note that in contrast to **insert/delete**, **add/remove** have no effect on the existence of objects, they simply manipulate set membership. Furthermore, there is a set iterator for update operations, applying a method on each object in a given set (**update** *[hireEmpl(john)] (Companies);*). Finally, values of functions can be assigned by the set operator (**set** *[salary:= 1.1*salary] (select[city(empl(e))='Bern'] (e:Employees));*). Due to the set-oriented style of the language, objects are typically unnamed. However, variables can be used as temporary names ("handles") for objects. In the examples above, *john, paul, marc, ibm, dec* are variables for objects.

3 A Framework for Schema Evolution

In this section we present the framework we will use for further discussion of our schema evolution approach. We give parts of the formal definition of a COCOON database and present the meta schema that describes the semantics of our object model. A similar approach has already been used in [HM90] for data model comparison.

3.1 COCOON Databases

Databases store objects. For every COCOON database, the set of persistent objects, actually contained in the database, is denoted as

$$\mathcal{O} = \{\ldots, o, \ldots\} \ .$$

The schema of a database is composed of types, functions, and classes. To describe such schemas in our model itself, we introduce the notion of schema objects: Every type, function, or class of a database schema is represented by a corresponding object. Thus, we have the following sets of schema objects:

$$\begin{array}{ll} \mathcal{T} & = \{\ldots, t, \ldots\} \subset \mathcal{O} \quad \text{, the set of type objects,} \\ \mathcal{F} & = \{\ldots, f, \ldots\} \subset \mathcal{O} \quad \text{, the set of function objects,} \\ \mathcal{C} & = \{\ldots, c, \ldots\} \subset \mathcal{O} \quad \text{, the set of class objects.} \end{array}$$

We treat schema objects as ordinary objects of the database, such that they are contained in $\mathcal{O}$. Using this notation, we can give the following definition of a COCOON database:

Definition 3.1 (*Database*) A database is a tuple $DB = < \mathcal{O}, \Sigma >$, where

1. $\mathcal{O}$ is the finite set of persistent objects in the database, and
2. $\Sigma = \{ < f_i, o_j, v_{ij} > \}$ is the state of the database, such that v_{ij} is the value, returned by applying function f_i to object o_j, where $f_i \in \mathcal{F}, o_j \in \mathcal{O}$. □

A closer look at the persistent objects $\mathcal{O}$ of a database shows, that they are built up of three sort of objects: meta, schema, and data objects ($\mathcal{O} = \mathcal{O}_{meta} \cup \mathcal{O}_{schema} \cup \mathcal{O}_{data}$):

- *meta objects:* The objects, building the database kernel. They are application independent and always part of the database. They are composed of meta types $\mathcal{T}_M$, meta functions $\mathcal{F}_M$, meta classes $\mathcal{C}_M$, the root of the type inheritance lattice (*object*), and the root of the subclass hierarchy (*Objects*).

$$\begin{aligned}\mathcal{O}_{meta} \;=\; \{\; & object, \underbrace{type, function, class}_{\mathcal{T}_M}, \; Objects, \underbrace{Types, Functions, Classes, Views}_{\mathcal{C}_M} \\ & tname, inherited, local, functs, supert, fname, dom, ran, setval, \\ & \underbrace{inverse, computed, cname, mtype, superc, sufficient, pred, extent}_{\mathcal{F}_M} \;\}\end{aligned}$$

- *schema objects:* The objects defining the schema of the database application. They are created as instances of meta types and consist of the application types $\mathcal{T}_A$, the application functions $\mathcal{F}_A$, and the application classes $\mathcal{C}_A$.

$$\mathcal{O}_{schema} \;=\; \{\underbrace{t_1, \ldots, t_k}_{\mathcal{T}_A}, \underbrace{f_1, \ldots, f_m}_{\mathcal{F}_A}, \underbrace{c_1, \ldots, c_l}_{\mathcal{C}_A}\}$$

- *data objects:* The primary level data objects, representing the user data that are stored in the database. They are created as instances of application types.

$$\mathcal{O}_{data} \;=\; \{o_1, \ldots, o_n\}$$

A valid "empty" database holds only the meta objects (called the kernel). Later, the database design process adds schema objects, and the use of the application generates finally data objects. Figure 1 gives an overview on types and its instances on the three different levels.

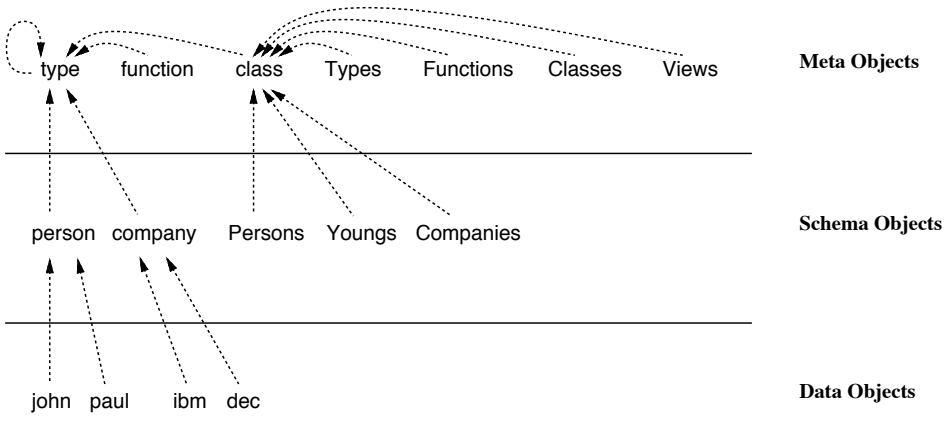

Figure 1: Instance-of Relationship between Meta, Schema, and Data Level Objects

3.2 Meta Schema

A database schema is a representation of the structure (syntax), semantics, and constraints on the use of a database in the data model. More formally, the schema of a database $DB = < \mathcal{O}, \Sigma >$ is given as the triple of types, functions, and classes defined in the database: $\mathcal{S}_{DB} = < \mathcal{T}, \mathcal{F}, \mathcal{C} >$, with $\mathcal{T}, \mathcal{F}, \mathcal{C} \subset \mathcal{O}$.

Following the above definition, we define the meta schema as a special database schema: the schema of the meta database, that is, the triple $\mathcal{S}_M = < \mathcal{T}_M, \mathcal{F}_M, \mathcal{C}_M >$, where $\mathcal{T}_M = \{type, function, class\}$, $\mathcal{F}_M = \{tname, inherited, \ldots, extent\}$, and $\mathcal{C}_M = \{$ *Types, Functions, Classes, Views* $\}$. For the specification of the meta database in COOL notation, see Appendix A. Figure 2 below shows the hierarchy of meta types (with their meta functions) and meta classes.

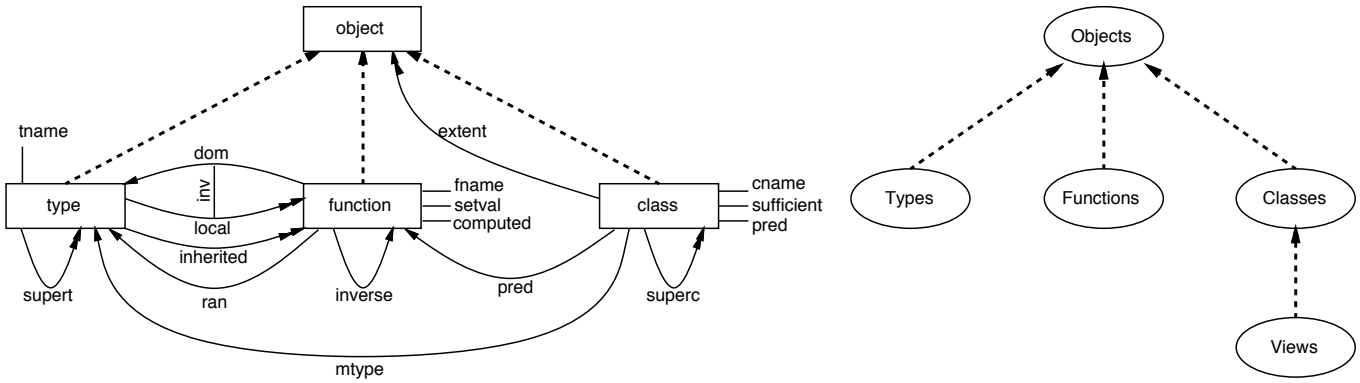

Figure 2: Hierarchy of Meta Types and Meta Classes

4 Manipulation of Meta Level Objects

The presented framework gives a formal definition of databases, where (i) objects are treated uniformly, regardingless whether they belong to the data, schema, or meta schema level, and (ii) the meta schema is well defined and fully available. These are both prerequisites for making use of the COOL language to retrieve and modify meta objects.

Querying the meta schema of Appendix A works straight forward without additional effort. As an example, query Q_1 below selects all types that are not supertype of any other type (being exactly the leaves of the type lattice), and Q_2 searchs for all functions with equal domain and range type, listing the name of the function and of the domain type.

Q_1 : **select** *[∅ =* **select** *[t ∈ supert(s)](s:Types)] (t:Types);*
Q_2 : **project** *[name,name(dom)] (***select** *[dom(f)=ran(f)] (f:Functions));*

The next step is to perform updates on meta objects. In contrast to the query language, direct meta object manipulation yields side effects, that must be regarded carefully. Consider update U_1, creating a new single-valued function with name *address*. Because its

domain type is *person*, the new function will be added to *functs(person)*. According to the type lattice, the function is inherited from type *person* to all subtypes, e.g., *employee*. What if there already exists a function with the same name in one of these classes?

U_1 : *addr:=***insert** *[fname:='address',dom:=person,ran:=string,setval:=false](Functions);*
U_2 : **delete** *(person);*

Update U_2 deletes an existing type from the database. If there are some functions with the destroyed type as their domain (e.g., *name, age*), should they be deleted too? When the type *person* no longer exists, what will be the member type of the classes *Persons* and *Youngs*? What will be the new supertype of *employee*? What happens to the instances of the deleted type? Should they be destroyed too, or are they still kept as instances of *object*, for example?

These two examples show, that manipulating one single schema object may call for several other operations, or otherwise results in an incorrect database state. In the sequel, we focus on some particular details, namely the following three problem areas: (i) *completeness* of update possibilities, (ii) *correctness* of schema changes, and (iii) *propagation* of schema level updates to the instance level.

4.1 Completeness

Completeness concerns the problem whether all desirable schema changes can be performed using the proposed operations. For this purpose, we use the ORION taxonomy of schema changes, which is usually considered to be complete [1].

Proposition 1. Direct manipulation of the COCOON meta schema by COOL operators provides *complete schema change possibilities* in the sense of the ORION taxonomy. □

This can easily be seen, by adapting the ORION taxonomy to the COCOON model (see the table below), and showing that each entry can be performed by a sequence of COOL operators applied on meta objects. As an example, notice that schema change (1.1) is performed by sample update U_1 and schema change (3.2) by update U_2.

This taxonomy is not only complete, but also minimal. If there were only changes on unpopulated databases, the number of needed schema change primitives would be even smaller. E.g., deleting a function, and then creating a new one with different name equals (on the schema level) to renaming a function. In contrast, on the instance level data is lost. A more detailed discussion of the COCOON taxonomy is contained in [Tre90].

4.2 Correctness

A database schema is a well defined collection of types and classes. Changing one of the meta objects alone may result in an incorrect schema. To ensure consistency, constraints have to be defined for the schema objects, that must be preserved during schema object manipulation.

[1] A "proof" is given in [BKKK87] based on a simple formal model, called *property inheritance graph* (PIG), which captures the essential characteristics of the ORION schema evolution model.

Schema Updates	Consistency Constraints				Propagation	
	Type-Lattice	Unique Naming	Classification	Range Closure	Instance Convers	Object Reclass
Updating Types						
(1.1) creation/deletion of a type object						
(1.1.1) create a new type object	•	•			√	
(1.1.2) delete an existing type object	•			•	√	
(1.2) changes to a type object						
(1.2.1) change the name		•				
(1.2.2) add a new inherited function	•			•	√	
(1.2.3) remove an inherited function	•			•	√	
Updating Functions						
(2.1) creation/deletion of a function object						
(2.1.1) create a new function object	•	•		•	√	
(2.1.2) delete a function object	•				√	
(2.2) changes to a function object						
(2.2.1) change the name		•				
(2.2.2) change the domain type	•			•	√	
(2.2.3) change the range type				•	√	
(2.2.4) change the set/single-valued tag					√	
(2.2.5) change the inverse function						
(2.2.6) change the computed/stored tag					√	
Updating Classes						
(3.1) creation/deletion of a class object						
(3.1.1) create a new class object		•	•			√
(3.1.2) delete an existing class object			•			√
(3.2) changes to a class object						
(3.2.1) change the name		•				
(3.2.2) change the member type	•			•	√	
(3.2.3) add a new classification edge			•			√
(3.2.4) remove a classification edge			•			√
(3.2.5) change the suff/necess tag			•		√	√
(3.2.6) change the class predicate			•		√	√

• - may be violated √ - may be affected

For this purpose, the ORION, O_2, and GemStone data models provide a set of *schema invariants*. These are conditions that have to be satisfied before and after changing the schema. In O_2 there is an interactive consistency checker, that proves for every schema update, whether structural or behavioral inconsistencies could arise. Similar, in our model there are four schema constraints, determining the basic characteristics of a COCOON schema:

- *type-lattice constraint:* types are arranged in a function inheritance lattice, forming a directed acyclic graph (DAG) with root type **object**,
- *unique naming constraint:* types and classes must have unique names in the database, and functions must have different names within their domain type,
- *classification constraint:* classes build a subclass hierarchy with root class **Objects**,

objects are automatically classified to subclasses, if they fulfill the class predicate, and the class has an all-selector,

- *closure constraint:* domain, range, and member type must be existing types of the lattice.

Whereas other systems satisfy these invariants by an appropriate implementation of their schema change operators, we have three different possibilities to guarantee correct schemas: (i) to find an appropriate formal definition of the model, such that many invariants are ensured automatically; (ii) to define constraints on the meta schema, that must be fulfilled by every schema object; (iii) to use schema update transactions.

Of course, we tend to include as much as possible into the definition of the model. For example, the type-lattice constraint is ensured automatically by the way how we formally defined our model: we consider types as (usually named) sets of functions, thus the position of a type in the lattice is implicitly defined as $\tau \preceq \tau' \Longleftrightarrow \mathit{functs}(\tau) \supseteq \mathit{functs}(\tau')$. In other words, whenever the definition of a type changes, its position in the lattice is recalculated automatically. Or vice-versa, the subtype hierarchy can be manipulated by adding/removing functions applicable to a type.

Other invariants are guaranteed by model-inherent constraints, namely as class predicates. For example, the unique naming rule is implemented by class predicates, as shown in the definition of the meta classes in Appendix A. The table in Section 4.1 gives a complete overview of schema updates and their potential effects on schema constraints.

The third possibility of keeping a schema consistent, is to execute complete schema change transactions, that is sequences of COOL operators. This way can be chosen, if a single COOL operation alone cannot implement a schema transformation correctly, but a sequence of them can. For example, destroying a type like in update U_2, removes the functions *name* and *age* automatically from the class *Functions*, because they do not any more satisfy the class predicate (cf. Appendix A). To avoid deleting those functions together with the type, the following transaction could be run:

BOT
set *[dom:= <new_domain_type>]* (**select** *[dom=person](Functions));*
set *[ran:= <new_range_type>]* (**select** *[ran=person](Functions));*
set *[mtype:= <new_member_type>]* (**select** *[mtype=person](Classes));*
delete *(person);*
EOT

4.3 Propagation

In a populated database, objects exist as instances of types and are collected in classes. Schema level updates, changing the definition of types or classes, may have an impact on the existing instances. We distinguish in our model two different ways of schema change propagation:

- *instance conversion*, where instances must be transformed to fit into the modified type definition (see also [SZ86, SZ87, TK89, LH90]);

- *object reclassification*, where classification of objects must be reconsidered, according to changing class definition (cf. [NR89a, NR89b]).

The above table gives an overview on which schema changes propagate to data objects. Instance conversion is necessary for those changes manipulating a type or the type-lattice. As an example, consider again update U_1. Instances of type *person*, that have been created before U_1 was executed, do not have a value defined for the function *addr* in the database. This is the reason why most existing systems demand such instances to be converted to the new type immediately after schema change. However, this is not necessary. Since a type may have many instances, such *eager conversion* of all objects after each schema modification may be to expensive.

In contrast, a *lazy conversion* strategy delays any instance conversion as long as possible. With a slight modification of the formal definitions of the COCOON model in [SLTS91] we can even build this lazy conversion into the model. The state function $\sigma : \mathcal{F} \rightarrow (\mathcal{O} \rightarrow \mathcal{O})$ returns for each function object $f \in \mathcal{F}$, the actual function $\mathcal{O} \rightarrow \mathcal{O}$ from the database state:

$$\sigma(f_i)(o_j) := \begin{cases} v_{ij} & \text{, if } \exists < f_i, o_j, v_{ij} > \in \Sigma \text{ and } v_{ij} \in ran(f_i); \\ \bot & \text{, if } \nexists < f_i, o_j, v_{ij} > \in \Sigma, \\ cast(v_{ij}, t_r) & \text{, if } \exists < f_i, o_j, v_{ij} > \in \Sigma \text{ and } v_{ij} \notin ran(f_i), \text{ with } t_r = ran(f_i). \end{cases}$$

Applying $\sigma(f)$ to an object o yields the current value v_{ij} of $f(o)$. It returns a null value ($\bot$), if there is no value actually defined in the database state, and in addition, $\sigma(f_i)(o_j)$ transforms the retrieved value into another type, if it is not identical to the actual range type of f_j. This third alternative was added to realize lazy conversion.

Consider, for example, schema update (2.2.3) of the taxonomy, where the range of the *age* function could be changed from *integer* to *date*. When the age of a person is accessed next, the σ-operator detects a type inconsistency and tries to transform the existing age-value to the new type.

Other schema changes propagate by object reclassification. For example, changing the class predicate of class *Youngs* from *age(p) < 30* to *age(p) ≠ 25* demands that some persons have to be removed from the class (those with age below 30 and not equal 25), and some persons have to be added to the class (those older than 30). In our framework, such reclassification is automatically performed. The definition of the meta type *class* in Appendix A shows, that the extent of a class with an **all**-selector is computed from the members of its superclass(es) and the predicate.

5 Conclusion and Perspectives

The contribution of this paper is threefold. We presented a formal framework, where meta and schema objects are treated as ordinary objects, and the schema is fully available to the user (see Appendix A). We defined the semantics of the operators of our object-oriented algebra such that they can be applied on any object, independent of whether it belongs to the schema or data level. We showed, that this mechanism is powerful enough to realize schema evolution, because all schema updates given in a taxonomy can be implemented as a (sequence of) COOL operations applied on schema objects.

We discussed a couple of examples, showing that correctness of schema object manipulation and propagation to the instance level can be ensured on three different levels of implementation: (i) by an appropriate formalization of the data model and algebra; (ii) by model-inherent constraints (class predicates) on the meta schema; (iii) by schema update transactions, that is, sequences of statements to be executed together.

Future work will address implementation issues of the presented framework. For this purpose, a prototype is under development, that realizes evolution of COCOON schemas on top of Oracle and the ONTOS object-oriented database system [TS91].

We expect to achieve further interesting results by investigating *mixed level operations*. Notice, that in this paper we simply presented queries and updates, operating on one single level. However, the sequence

apply *[* **select** *[dom=person] (Functions)] (Persons);*

searchs for all functions with domain type *person* and applies them on all objects in class *Persons*. Such a mixed level facility requires an **apply** operator that gives additional expressive power to the query language. On the other hand, static type checking becomes impossible in some cases.

Acknowledgement The author is indebted to Marc Scholl for numerous discussions and improving an earlier draft of this paper.

References

[BCG+87] J. Banerjee, H. Chou, J.F. Garza, W. Kim, D. Woelk, and N. Ballou. Data model issues for object-oriented applications. *ACM Trans. on Office Information Systems*, 5(1):3–26, January 1987.

[Bee89] C. Beeri. Formal models for object-oriented databases. In Kim et al. [KNN89], pages 370–395.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD Record 1987*, pages 311–322, San Francisco, CA, February 1987. ACM Press.

[HM90] S. Hong and F. Maryanski. Using a meta model to represent object-oriented data models. In *Proc. 6th Int'l IEEE Conf. on Data Engineering*, pages 11 – 19, Los Angeles, USA, February 1990. IEEE Comp. Soc. Press.

[KNN89] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, December 1989. North-Holland.

[LH90] B.S. Lerner and A.N. Habermann. Beyond schema evolution to database reorganization. In M. Meyrowitz, editor, *Proc. joint Int'l Conf. OOPSLA / ECOOP*, pages 67–76, Ottawa, Canada, October 1990. ACM Press.

[NR89a] G.T. Nguyen and D. Rieu. Schema change propagation in object-oriented databases. In *Information Processing 89. Proc. 11th IFIP World Computer Congress*, pages 815–820, San Francisco, CA, August 1989. IFIP, North-Holland, Amsterdam.

[NR89b] G.T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data & Knowledge Engineering, North-Holland*, 4(1):43–67, July 1989.

[PS87] D.J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *Proc. Int'l Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 111–117. ACM Press, October 1987.

[SLT91] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proc. 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, December 1991.

[SLTS91] M.H. Scholl, C. Laasch, M. Tresch, and H.-J. Schek. The COCOON core object model. Technical report, ETH Zürich, Dept. of Computer Science, 1991. in preparation.

[SS90] M.H. Scholl and H.-J. Schek. A relational object model. In *Proc. 3rd Int'l Conf. on Database Theory (ICDT'90)*, Paris, 1990.

[SZ86] A.H. Skarra and S.B. Zdonik. The management of changing types in an object-oriented database. In *Proc. Int'l Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 483–495. ACM Press, September 1986.

[SZ87] A.H. Skarra and S.B. Zdonik. Type evolution in an object-oriented database. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 393–413. MIT Press, 1987.

[TK89] L. Tan and T. Katayame. Meta operations for type management in object-oriented databases. In Kim et al. [KNN89].

[Tre90] M. Tresch. Semantics of schema evolution in the COCOON object model. Manuscript, ETH Zürich, Dept. of Computer Science, 1990.

[TS91] M. Tresch and M.H. Scholl. Implementing an object model on top of commercial database systems. In *Proc. 3rd GI Workshop on Foundations of Database Systems*, Volkse, Germany, May 1991. Technical Report 158, Dept. of Computer Science, ETH Zürich.

[Zic90] R. Zicari. A framework for schema updates in an object-oriented database system. Report 90-025, Politechnico die Milano, Dipartimento di Elettronica, 1990.

A The Meta Schema

The COCOON meta schema is composed of the meta types (*type, function, class*) and the meta classes (*Types, Functions, Classes, Views*). In COCOON notation, they are given as follows:

type *type* **isa** *object* **=**
tname **:** *string* **,**
inherited **:** **set of** *function* **,**
local **:** **set of** *function* **inverse** *dom* **,**
functs **:** **set of** *function* **,**
supert **:** **set of** *type* **;**

where:

$tname(t) = n_t$		the name of the type
$inherited(t) = \{f_1, \ldots, f_n\}$		the set of inherited functions
$local(t) := \{f_i \in \mathcal{F} \mid dom(f_i) = t\}$		the functions with type t as its domain
$functs(t) := inherited(t) \cup local(t)$		the functions, applicable to t's instances
$supert(t) := \{t_i \in \mathcal{T} \mid functs(t_i) \subset functs(t)\}$		the set of supertypes

type *function* **isa** *object* **=**

fname **:**	*string* **,**
dom **:**	*type* **inverse** *functs* **,**
ran **:**	*type* **,**
setval **:**	*boolean* **,**
inverse **:**	*function* **inverse** *inverse* **,**
computed **:**	*boolean* **;**

where:

$fname(f) = n_f$	the name of the function
$dom(f) = t_D$	the type where the function is locally defined, that is $f \in local(t_D)$, with $t_D \in \mathcal{T}$
$ran(f) = t_R$	the range type of the function ($t_R \in \mathcal{T}$)
$setval(f) = true/false$	the function is set-valued / single-valued
$inverse(f) = f_v/\perp$	another function which is inverse to f
$computed(f) = true/false$	the value of f is computed / stored

type *class* **isa** *object* **=**

cname **:**	*string* **,**
mtype **:**	*type* **,**
superc **:**	**set of** *class* **,**
sufficient **:**	*boolean* **,**
pred **:**	*function* **,**
extent **:**	**set of** *object* **;**

where:

$cname(c) = n_c$	the name of the class
$mtype(c) = t_E$	the member type of the class ($t_E \in \mathcal{T}$)
$superc(c) = \{c_1, \ldots, c_l\}$	the set of immediate superclasses, $c_i \in \mathcal{C} (i = 1 \ldots l)$
$sufficient(c) = true/false$	the class has an **all**-/**some**-selector
$pred(c) = p_c/\perp$	a boolean function $p_c \in \mathcal{F}$ ($\perp$ if no predicate specified)
$extent(c) := supers$	if the class has an **all**-selector
$extent(c) \subseteq supers$	if the class has a **some**-selector
	where $supers := \{\ o_i \in \bigcap_{c_k \in superc(c)} extent(c_k) \mid p_c(o_i)\ \}$

class *Types* **:** *type* **some** *o: Objects* **where**
select [*tname(o)* = *tname(t)* **and** $o \neq t$] (*t: Types*) = {}
and $inherited(o) \subset Functions$ **;**

class *Functions* **:** *function* **some** *o: Objects* **where**
select [*fname(o)* = *fname(f)* **and**
dom(o) $\neq$ *dom(f)* **and** $o \neq f$] (*f: Functions*) = {}
and $dom(o) \in Types$**and** $ran(o) \in Types$ **;**

class *Classes* **:** *class* **some** *o: Objects* **where**
select [*cname(o)* = *cname(c)* **and** $o \neq c$] (*c: Classes*) = {}
and $mtype(o) \in Types$
and $superc(o) \subset Classes$ **;**

class *Views* **:** *class* **some** *c: Classes* **where** *select(c)* = *true* **;**