Meta Object Management and its Application to Database Evolution*

Markus Tresch and Marc H. Scholl

Department of Computer Science, Databases and Information Systems
University of Ulm, D-W 7900 Ulm, Germany
<tresch,scholl>@informatik.uni-ulm.de

Abstract. In this paper, we address the problem of supporting more flexibility on the schema of object-oriented databases. We describe a general framework based on an object-oriented data model, where three levels of objects are distinguished: *data objects*, *schema objects*, and *meta-schema objects*. We discuss the prerequisites for applying the query and update operations of an object algebra uniformly on all three levels. As a sample application of the framework, we focus on database evolution, that is, realizing incremental changes to the database schema and their propagation to data instances. We show, how each schema update of a given taxonomy is realized by direct updating of schema objects, and how this approach can be used to build a complete tool for database evolution.

1 Introduction

There is an increasing need for *database evolution facilities*, offering more flexibility on the logical structure of object-oriented databases (OODBs). On the one hand, schema evolution is the basic prerequisite for better support of database *extensibility* and *reusability*, which is a big promise of object-oriented systems; and on the other hand, database integration has a renaissance, since the integration of *federated databases* and *interoperability of multidatabase systems* has become urgent. In addition, new aspects arise from the fact that not only the logical schemas of the databases can evolve, but also existing data must be migrated and integrated. Either of these demand more dynamics of database schemas.

Evolution in databases addresses the problem that the logical structure of a database is likely to undergo changes during lifetime, even if a database is already populated with objects. There are many reasons for that [6]: *schema design* can be a stepwise development of a schema from scratch; *schema tailoring* consists in slight adaptations of existing schemas (e.g. extension with new components); *schema restructuring/reorganization* is used after significant, non-trivial changes; and *schema versioning* allows to record and manage a history of schemas.

Most current OODBS products and prototypes do not allow free and dynamic changes of the schema. Some of them provide a limited set of special purpose

* Work done while at Department of Computer Science, ETH Zürich, Switzerland

schema update methods, and restrict their application to unpopulated (empty) databases. If modifications of populated databases are allowed, the problem is how to propagate the changes to the instances. One either includes a data migration utility to adapt existing data objects to the changed schema, or an other mechanism (screening, versioning) has to ensure consistency between data and structure.

An early investigation of type changes in populated databases exists for ENCORE [26]. This work addresses the effects of type changes to objects and to programs that use objects of the type. The impact of type polymorphism on schema evolution is investigated in [18]. The first systematic analysis of desirable schema evolution possibilities was done for the ORION data model [2, 3], where a set of necessary schema updates was listed and organized in a taxonomy. Similar enumeration can also be found for the O_2 [31] and the GemStone DBMSs [21]. The schema update primitives of these taxonomies are realized as special purpose methods for schema management. Another approach is to provide a complete end-user tool to assist in transforming database schemas [15].

Schema evolution is in fact an important issue for ER databases, because even relational, hierarchical, network, or object-oriented databases use an ER-approach for conceptual DB design [12], or for representation of external schemas [16].

Our approach was to built a *uniform model* and *general framework* for investigation of the above mentioned database evolution and integration issues. In the following, we introduce the main components of the framework, which in turn reflects the structure of this paper:

Object model and algebra: As the basis of the framework, the object-oriented data model COCOON with its algebraic query and update language is used (Section 2). However, notice that the basic ideas of the framework are not strongly bound to this specific object model; it can be replaced by any object-oriented data model, especially an object-oriented entity-relationship approach [17, 20] with an algebra or a calculus [19].

Three object levels: In this model, a separation of database objects into three disjoint subsets is introduced: data objects, schema objects, and meta-schema objects (Section 3). Hence, also (meta-)schema objects are modeled as objects like others, we allow that the generic query and update operators can be applied on objects of each of these three sorts and there is no difference in the syntax and semantics of the algebraic operators, whether they query or update data, schema, or meta-schema objects. Even though representation of the meta-schema within the same model is not a new idea, such meta-data usually serve only for documentation purposes. That is, only *retrieval* is allowed. In contrast, here we investigate also the feasibility of *updates* to meta-objects.

Elementary Operations: Based on this separation, we focus into special applications, implementing elementary operations for database evolution or database integration (Section 5.1). We thereby concentrate in this paper on the application of the framework to schema evolution, and show how to re-

alize schema update operations by applying the algebraic query and update algebra to meta-objects. Consequently, the use of generic update operators as "schema evolution and integration language", instead of special purpose schema evolution methods, has the advantage that: (i) the functionality of these operators is formally defined with a clear semantics [14] and has no unpredictable side-effects; (ii) they handle integrity constraints, that is, no update leaves the database in an inconsistent state w.r.t. these constraints. The remaining problem is to propagate the modifications from the schema level down to the data object level. We will show that this can be achieved straight-forward due to the clear semantics of schema updates.

Advanced Operations: Next, the elementary operations are encapsulated into advanced, user-oriented tools for database evolution. The purpose of this is to capture more semantics and insure higher level integrity constraints (Section 5.2). Database integration using the framework, as an other sample application of the framework is presented in [25].

2 An Object Model and Algebra

The framework we present throughout this paper is based on the COCOON object model. We very briefly review the key concepts of the object model and the algebra, referring to the literature for more details [24, 23].

2.1 Basic Concepts

The COCOON object model is an object-function model in the sense of [4, 5]. Its basic constituents are objects, functions, types, and classes:

Besides **data**, which can be atomic (numbers, strings) or constructed (tuples, sets), there are **objects** that are instances of abstract object types (AOTs). Objects can be manipulated by a set of applicable operations.

Functions are the generalized abstraction of attributes (stored or computed), relationships between objects, and update methods (with side-effects). They can be single- or set-valued. Functions are described by their name and signature. The implementation is given separately, in the object implementation language (OIL), which is not described here any further.

Types describe the common interface to all of its instances. So, a type is defined by a name and a set of applicable functions. This set is the union of the functions explicitly defined to belong to the type and those inherited from the types of the acyclic isa relationship. The subtype relationship that is used for type-checking corresponds to the subset relationship of the function sets. Thus, instances of one type are also instances of its supertypes (multiple instantiation). The root of this lattice is the predefined type **object**.

Classes are strictly distinguished from types [4]. Classes are typed collections of objects. So every class c has an associated member type *mtype(c)* and an actual extension *extent(c)*, the set of objects in the class. We define the extent of a class to include the members of all its subclasses. Thus, objects can be member of

multiple classes at the same time (multiple class membership). Besides the subset property the subclass relationship states that the member type of a subclass must be the same type or a subtype of its superclasses' member types. The top class of the subclass hierarchy is the class **Objects**.

COCOON features that are usually not found in other object-oriented models are the possibilities to define class predicates and views. Class predicates are either necessary or necessary and sufficient conditions that have to be fulfilled by the class members. **Views** can be defined by queries of arbitrary complexity and can be regarded as a special kind of classes, because their extent and their member type is sufficiently defined by the query. Thus, classes with necessary and sufficient predicates are regarded as a special kind of views, since also their extent is populated automatically.[1]

EXAMPLE 1: As a running example, we use the following database *Business*, defined by COOL type and class declaration statements.[2]

define database *Business ;*
define type *person* **isa object** = *name:* **unique** *string* **not null** ,
age: integer ;
define type *employee* **isa** *person* = *salary: integer* ,
empl: company **inverse staff ;**
define type *company* **isa object** = *ident:* **unique** *string, city: string* ,
staff: **setof** *employee* **inverse** *empl* ,
bran: **setof** *company* **not cyclic** *;*
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** *;*
define view *PublEmpls* **as project** [*name,empl*] (*Employees*) ;
enddb .

The two classes *Persons* and *Youngs* have both the same type, *person*. Moreover, 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 (in contrast to **some**) indicates that the class predicate is necessary and sufficient, such that the member objects of class *Youngs* are automatically computed from those of *Persons*. The projection view *PublEmpls* hides the *age* and *salary* properties, such that users of this view only see *name* and *empl* functions.

In addition to class predicates, there is a possibility to define constraints for functions: (i) **not null** functions are not allowed to have undefined values, (ii) **unique** functions must have distinct values, (iii) **not cyclic** functions ensure that no object has itself as result of that function (even if applied repeatedly), and (iv) **inverse** means that two functions are inverse to each other. ◇

[1] Updating views is discussed in [23].

[2] We use the convention that type names are in lower-case letters and in singular, whereas class or view names are in plural and start with an upper-case letter.

2.2 Generic Query and Update Operations

The query and object manipulation language COOL was designed as an extension of (nested) relational algebra. It provides a collection of generic query operators with object-preserving semantics. That is, they return (some of) the already existing input objects, instead of generating new (copies of) objects.

As query operations we provide selection of objects (**select** $[P](C)$), projection (**project** $[f_1, \ldots, f_n](C)$), extension (**extend** $[f_i :=< expr_i >, \ldots](C)$), and the set operations (**union, intersection, difference**). Variables can be used as temporary names ("handles") for objects, since objects are typically unnamed, due to the set-oriented style of the language. So this is the way how to refer to objects and results of previous algebra expressions.

Besides query operators, COOL also provides a collection of generic update operators (cf. [14] for a formal definition). The main advantage of general purpose update operations is that their semantics is known by the system. That is, they maintain model-inherent integrity constraints like uniqueness or acyclicity of functions, and class predicates. We will see later how we exploit by this property, when schema updates are defined by such generic update operations. In the remainder of this paper we will make use of the following operations:

Insert takes as argument a class C and a list of assignments of values to functions. If the function assignments in the assignment list do not conflict with the respective constraints of all functions, a new object as instance of C's member type is created, and initialized with the function values given in the assignment list. In case the new object satisfies C's class predicate, it is also added to the extent of class C. As a result, a reference to the newly created object is returned, and can therefore be assigned to a variable, e.g.:

john := **insert** [*name:='John Smith', age:=31*] (*Persons*);

Delete destroys objects consistently. It takes an object as its only argument. This object is removed from all classes, sets, and variables, in case that no function constraint is violated. For example:

delete (*paul*);

Add and **remove** have a "weaker" effect: they have no impact on the existence of objects. Rather, an existing object can be added to or removed from classes or sets. They take an object and a set of objects as parameter:

add [*john*] (*staff(ibm)*);
remove [*dec*] (*Companies*);

Both operators may change the type of objects dynamically. Consider *john*, which is an instance of type *person*. Adding it to the class of *Employees*, makes *john* an instance of type *employee*, in addition to including him in IBM's staff.

Set assigns new function values for given arguments. It takes two arguments: a list of assignments and an object. As in the case of the **insert** operation, the newly assigned function values must respect the function constraints.

set [*salary:= 1.1*salary*] (john);

It is essential to notice that generic update operations are refused and therefore not processed, if they would conflict with any function constraints (unique, not null, not cyclic). Consider for example the **insert** -statement above, where a new person object with name "John Smith" is created. Since the *name* function is defined to be unique, the update would be rejected, if there already were a person with this name.

Violation of class predicates is handled different [23]. The class predicate is checked after execution of the update operation, and if necessary, the changed objects are reclassified. Note that changing values of objects, for example with the **set**-operator, may need such a reclassification of objects. Consider for example the object *john*. Changing its age from 31 to 29 would immediately classify it down in the hierarchy, by adding it to the class *Youngs*. The treatment of inverse functions is similar to reclassification: If a function value is changed, this update is propagated automatically to the inverse function, such that the constraint is maintained.

3 Three Levels of Objects

For every COCOON database, the set of actually stored persistent objects is denoted by $\mathcal{O} = \{\ldots, o, \ldots\}$. Each of these objects has a specific state σ, such that $v_{ij} := \sigma(f_i)(o_j)$ is the actual value returned by applying function f_i on object $o_j \in \mathcal{O}$.[3] The states of all objects in $\mathcal{O}$ together form the actual state of the 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. σ is the state of the database. □

Consider for example the variable *john* of type *person* and the function *name* that is applicable on instances of that type (*name: person* $\rightarrow$ *string*). The actual value of function *name* might be the set: $\{\langle john, \text{'John Smith'}\rangle, \langle mary, \text{'Mary Hughs'}\rangle\}$, such that for instance, *σ(name)(john) = 'John Smith'*.

3.1 Meta-Schema, Schema, and Data Objects

A closer look at the set of objects $\mathcal{O}$ reveals, that they are build up of three pairwise disjoint sorts, each of which is placed on a different level: the meta-schema level, the schema level, and the data level:

[3] In our model, all functions are partial, such that when applied on an object, their values can be undefined. For this case, the state $\sigma(f_i)(o_j)$ is defined to be the null value $\perp$. Later, we will see, that this lazy evaluation strategy is very useful for schema update propagation.

- *meta-schema level objects:* the objects describing the meta-schema of the database. There is a predefined fixed set of meta-objects, building the database kernel, being always part of a database. Meta level objects are the meta-types $\mathcal{T}_M$, the meta-functions $\mathcal{F}_M$, and the meta-classes $\mathcal{C}_M$.

$$\begin{aligned}\mathcal{O}_{meta} = \{\ & \textit{type, set-type, fcn-type, object-type,} \\ & \underbrace{\textit{function, class, class-def, view-def,}}_{\mathcal{T}_M} \\ & \textit{Types, Set-Types, Fcn-Types, Object-Types,} \\ & \underbrace{\textit{Functions, Classes, Class-Defs, View-Defs,}}_{\mathcal{C}_M} \\ & \textit{tname}, \textit{functs}, \textit{localf}, \textit{supert}, \textit{etype}, \textit{dom}, \textit{ran}, \\ & \underbrace{\textit{fname}, \textit{sign}, \textit{unique}, \textit{notnull}, \textit{notcyclic}, \textit{inverse}, \\ & \textit{cname}, \textit{extent}, \textit{auto}, \textit{mtype}, \textit{pred}, \textit{superc}, \textit{query}}_{\mathcal{F}_M}\ \}\end{aligned}$$

- *schema level objects:* the objects describing the schema of the application database. They are application dependent and are created as instances of meta-types. Schema objects are distinguished in application types $\mathcal{T}_A$, application functions $\mathcal{F}_A$, and 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 level objects:* the primary level objects, representing the user data stored in the database. They are created as instances of application-schema types.

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

Objects are created top-down: Whereas an "empty" database holds only meta-level objects, later, the schema level objects are created during the database design phase. Finally, the use of the database generates data level objects. Figure 1 illustrates instance-of relationship between objects and types of the three different levels.

3.2 Database Schemas

A database schema is a representation of the structure (syntax), semantics, and constraints on the use of a database in the data model. In our model, this is given by classes, types, and functions.

Thus, the schema of a COCOON database $DB = < \mathcal{O}, \sigma >$ is represented as a triple $< T, F, C >$, with T a set of objects representing types, F a set of objects representing functions, and C a set of objects representing classes. The set of all types, functions, and classes in the database is denoted as $\mathcal{T}, \mathcal{F}, \mathcal{C}$. These sets are identical with the active domain of the meta-types (*type, function, class*) and the extent of the meta-classes (*Types, Functions, Classes*).

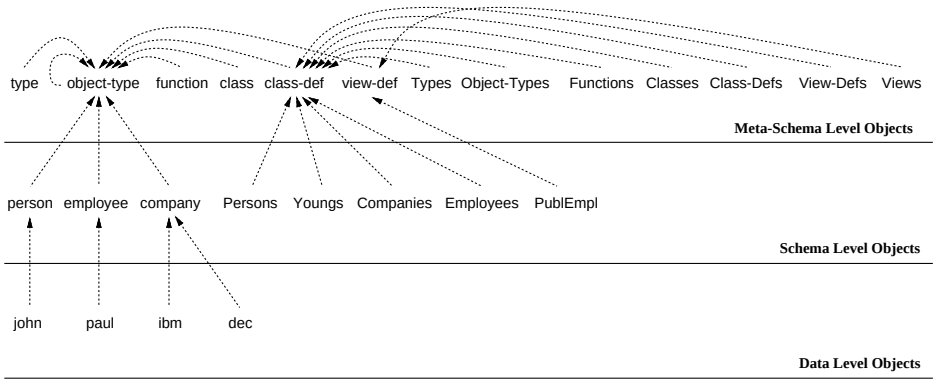

Fig. 1. *Instance-of relationship (drawn as dashed arrows) between meta-schema, schema, and data level objects. Data level objects are instances of schema level types. Schema level types are themselves represented by type objects, that are instances of meta-level types. The meta-level types are represented by meta-level objects. Notice, that the object* object-type *is instance of the type that it represents (which does not mean that it is "instance of itself").*

The meta-schema is a special schema, the schema of the meta-database. It is represented by the meta-level objects of Section 3.1 and defines the data model itself. Figure 2 gives a graphical overview of the meta-schema, and a definition in COOL notation can be found in Appendix A.

Definition 3.2 (Meta Database Schema) The meta-schema of a database $DB = < \mathcal{O}, \sigma >$ is given as the triple $\mathcal{S}_{Meta} = < \mathcal{T}_{\mathcal{M}}, \mathcal{F}_{\mathcal{M}}, \mathcal{C}_{\mathcal{M}} >$. □

The application-schema holds application level objects. It defines the conceptual schema of the database application.

Definition 3.3 (Application Database Schema) The application schema of a database $DB = < \mathcal{O}, \sigma >$ is given as the triple $\mathcal{S}_{Appl} = < \mathcal{T}_{\mathcal{A}}, \mathcal{F}_{\mathcal{A}}, \mathcal{C}_{\mathcal{A}} >$. □

3.3 Queries and Updates to Objects of Different Levels

So far, we introduced a distinction of objects into three sorts, according to the role they play within a database application. Nevertheless, all of them are ordinary objects, such that queries and updates of the COOL language apply. Of course, the effect of an operation depends on the level of its input objects.

Data Level Operations retrieve or change data level objects. These are the ordinary operations as we know them from Section 2.2.

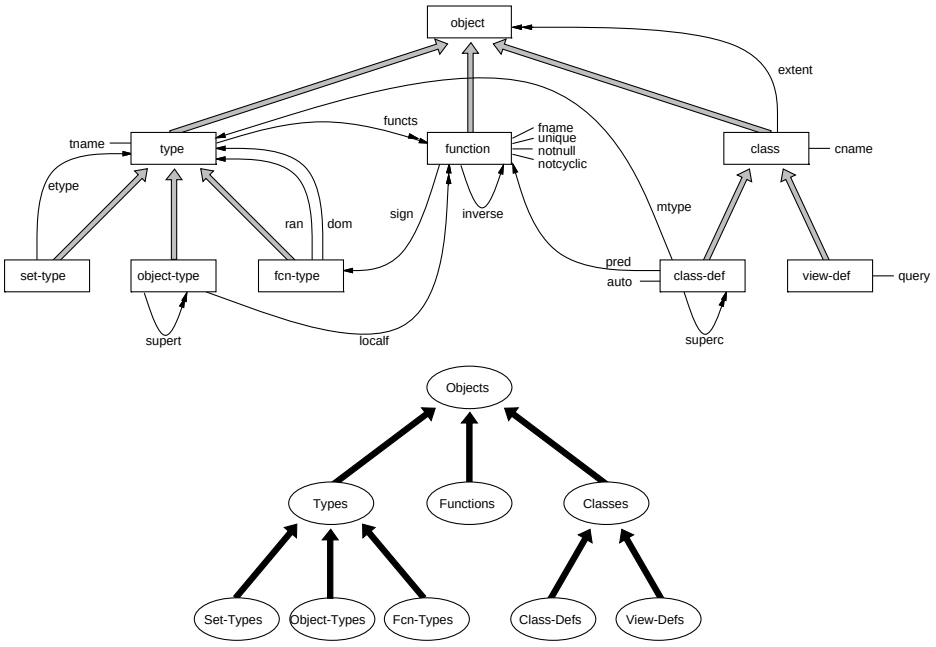

Fig. 2. *The meta-database schema is a hierarchy of meta-types $\mathcal{T}_M$ (rectangles) with their meta-functions $\mathcal{F}_M$ (lines with one arrowhead mean single-valued functions, with two arrowheads set-valued functions), and a hierarchy of meta-classes $\mathcal{C}_M$ (ellipses). Subtype relationship is shown with gray arrows, whereas subclass hierarchy is indicated using black arrows.*

Q_1 : *name(john)* $\qquad$ U_1 : **set** [*name:='hans'*](*john*)
Q_2 : **select** [*name='john'*](*Persons*) $\qquad$ U_2 : **delete** (*john*)

For example, getting the name of object *john* (Q_1), selecting all persons with name *'john'* (Q_2), changing John's name to *'hans'* (U_1), or deleting the object *john* (U_2) .

Schema Level Operations operate on objects representing the schema. Retrievals work straightforward without additional effort. As an example, query Q_3 gets the name of the application type *person*, and Q_4 returns all objects representing types with name *'person'*.

Q_3 : *tname(person)* $\qquad$ U_3 : **set** [*tname:='people'*](*person*)
Q_4 : **select** [*tname='person'*](*Types*) $\qquad$ U_4 : **delete** (*person*)

In contrast to query operators, direct manipulation of schema objects (U_3, U_4) needs further consideration, since these updates finally realize **schema evolu-**

tion. The usual way to make basic updates to the schema objects is by using a data definition language (DDL), as it was introduced in Section 2.1 (e.g., **define type** *employee* **isa** ...). Advanced changes to schema objects, like e.g. modifying types, are normally implemented in a schema manipulation language (SML).

Such a special language is used, because these operations can yield "side-effects", that must be regarded carefully. Consider for example update U_3, changing the name of type *person* to *'people'*, or U_4 even deleting the type *person*. We will cover the following questions: There are some functions applicable to instances of type *person* (*name, age, ...*), should they be deleted together with the type? When the type *person* is removed, what will be the member type of the classes *Persons* and *Youngs*? What will be the new supertype of *employee*? What about propagation of schema changes to existing data objects like *john*. What happens with the instances of a deleted type?

Meta-Schema Level Operations operate on objects representing the meta-schema. Again, queries can be used quite straightforward. So, Q_5 returns the name of the meta-type *type*, and Q_6 selects a set of all types with name *'type'*.

Q_5 : *tname(type)* $\qquad$ U_5 : **set** [*tname:='interface'*](*type*)
Q_6 : **select** [*tname='type'*](*Types*) $\qquad$ U_6 : **delete** (*type*)

Updates U_5, U_6 are changes to the meta-schema. As an example, U_5 renames the type *type* to *'interface'*, and U_6 even destroys the type *type*. These updates are generally not allowed, since they change the basic ingredients of the data model. Anyway, some elementary modification to the meta-schema could be very useful, i.e. for **model tailoring** by adding specialized types and classes, derived functions, or meta-views [13], and for preparing for the coupling of multiple DBMS [25].

Mixed Level Operations involve different levels within one operation. To do this, we need an additional operator, that provides the possibility to change the level: **apply** [*f*] (<*set-expr*>). It applies a function *f* to a set of objects in <*set-expr*>. E.g. query Q_7 first finds all classes *c* from the meta-class *Class-Defs*, where the function *name* can be applied; then the expression **select** [*name = 'john'*](*d*) is run for each of these classes, now called *d*.

Q_7 : **apply** [**select** [*name = 'john'*](*d*)]
(*d* : **select** [*name* ∈ *functs(mtype(c))*] (*c: Class-Defs*));

Notice, that the selection on *Class-Defs* returns a set of schema level objects, whereas the later selection gives data level objects. Such mixed level facilities give additional expressive power to the language, but static type checking becomes impossible in some cases [10].

4 Prerequisites for Updating Schema Objects

Direct changes to schema objects (cf. U_3, U_4, U_5, U_6 in Section 3.3) have additional "side-effects". Thus, before we show how the schema updates are implemented, we investigate the feasibility of these operations, namely completeness and correctness, as well as the propagation of meta-schema and schema level updates to other levels.

4.1 A Taxonomy of Schema Updates (Completeness)

Some schema changes are quite simple, whereas others need complete reorganization of the database. The latter can often be decomposed in a sequence of more elementary changes.

Below, a taxonomy of primitive schema updates is presented, which is minimal and complete in the sense that all possible schema transformations can be built up by (a combination of) these updates. Since a schema is a triple $< T, F, C >$, we categorize schema changes into updates to type, function, and class objects respectively:

(1) *UPDATING TYPES*
(1.1) create/delete a type object
(1.1.1) create a new type object
(1.1.2) delete an existing type object
(1.2) changes a type object
(1.2.1) change the name
(1.2.2) change the supertypes
(1.2.3) change the local functions

(2) *UPDATING FUNCTIONS*
(2.1) create/delete a function object
(2.1.1) create a new function object
(2.1.2) delete an existing function object
(2.2) change a function object
(2.2.1) change the name
(2.2.2) change the function signature
(2.2.3) change the function constraints

(3) *UPDATING CLASSES*
(3.1) create/delete a class object
(3.1.1) create a new class object
(3.1.2) delete an existing class object
(3.2) change a class object
(3.2.1) change the name
(3.2.2) change the auto tag
(3.2.3) change the member type
(3.2.4) change the class predicate
(3.2.5) change the superclasses
(3.2.6) change the query expression

A similar taxonomy was first introduced for the ORION data model [3], where all supported schema changes are classified into changes to an edge, changes to a node, and changes to the contents of a node.

4.2 Insuring Schema Correctness

Since not every arbitrary triple $< T, F, C >$ of types, functions, and classes is a correct schema, we must make sure that schema updates transform a database structure into another correct state. Thus, the ORION, O_2, and GemStone data

models provide a set of *schema invariants*. These are conditions that have to be satisfied by any valid schema. Similar, in our model the schema constraints R1, ..., R14 below determine the basic characteristics of a COCOON schema, such that a schema is correct, if it satisfies these conditions.

To ensure that these conditions are respected at any time, they are translated into integrity constraints of the meta-schema (unique, not null, not cyclic). In the sequel, we explain the constraints in the COCOON meta-schema: [4]

Unique Naming Constraints (R1, R2, R3) guarantee, that there are no two types, functions, or classes with the same name. Notice that these are the internal names, after solving any naming conflicts from multiple inheritance, or overloading respectively.

$(R1)$ for all types $t, t' \in T, t \neq t'$: $tname(t) \neq tname(t')$
$(R2)$ for all functions $f, f' \in F, f \neq f'$: $fname(f) \neq fname(f')$
$(R3)$ for all classes $c, c' \in C, c \neq c'$: $cname(c) \neq cname(c')$

The unique naming constraints are expressed by declaring the meta-functions *tname, fname, cname* to be **unique** in the meta-type definition.

Closure Constraints (R4, ..., R12) ensure schema closure in the sense that the following objects must be part of the schema:

$(R4)$ for all object-types $t \in T$: $localf(t) \subseteq F,\ supert(t) \subseteq T$
$(R5)$ for all set-types $t \in T$: $etype(t) \subseteq T$
$(R6)$ for all function-types $t \in T$: $dom(t) \subseteq T,\ ran(t) \subseteq T$
$(R7)$ for all functions $f \in F$: $sign(f) \in T$
$(R8)$ $inverse(f) \in F \ \vee \ inverse(f) = \perp$
$(R9)$ $inverse(f) = f' \iff inverse(f') = f$
$(R10)$ for all class-defs $c \in C$: $mtype(c) \in T,\ superc(c) \subseteq C$
$(R11)$ $pred(c) \in F$
$(R12)$ for all view-defs $v \in C$: $query(v) \in expression$

To implement closure constraints, the meta-schema must follow two restrictions. First, in the meta-schema, all functions, that are not allowed to be undefined, have a **not null** constraint. Second, the active domain of the meta-types are by definition identical to the extent of the corresponding meta-class (cf. Section 3.2).

Acyclicity Constraints (R13, R14) ensure that supertype and superclass relationship do not end up in a cycle. Thus, no type can be supertype of itself, and no class superclass of itself.

$(R13)$ for all object-types $t \in T$: $t \notin supert^*(t)$
$(R14)$ for all class-defs $c \in C$: $c \notin superc^*(c)$

The acyclicity of the supertype/superclass relationship is implemented by adding a **not cyclic** constraint to the meta-function *supert*/*superc*, and is therefore checked automatically.

[4] The semantics of the meta-functions *tname, localf, supert, ...* is explained in the meta-schema in Appendix A. In addition, $supert^*$, $superc^*$ are defined as the transitive closure of *supert*, and *superc* respectively.

Implementing schema invariants for unique naming, closure, and acyclicity as constraints in the meta-schema is a very natural approach: since the application schema includes constraints for data level objects, constraints for schema level objects must be defined in the meta-schema. We use the generic update operations of COOL to change the schema level. Since they are defined to respect constraints, schema invariants are maintained automatically.

Thus, whenever a COOL update is performed to a schema level object in order to implement schema modifications, which would result in an incorrect database schema, this update will be rejected by the system.

4.3 Propagation to Data Level Objects (Type-Validness)

Figure 1 showed the *instance_of* relationship between types and objects. We mentioned that types define the interface (a set of functions) to their instances. The following definition gives the notion of *type-valid databases* [27], which means that every instance must match the definition of its type at any time:

Definition 4.1 (Type-Valid) A database $DB = < \mathcal{O}, \sigma >$ is type-valid, iff $\forall$ types $t \in \mathcal{T} : o$ *instance_of* $t \Rightarrow \exists! \, \sigma(f)(o) \in range(f), \; \forall f \in functs(t)$. $\square$

In other words, a database must fulfill two requirements to be type-valid: (i) whenever the type-checker allows a function f to be applied on an object o (that is, iff f is in the interface of a type t and o is an instance of t), then the state $\sigma(f)(o)$ must be well defined; and (ii) the value of the state must match the range type of f.

Since types are represented by schema objects, we have therefore strong consistency requirements between objects of the schema level and of the data level, such that updates to objects representing types must propagate to updates on objects of the data level below.[5] However, we allow direct updating of schema level objects, but want to avoid that afterwards a migration utility must be called to adapt data level objects.

4.4 Implementation Considerations

A type evolution strategy, where all objects are kept type-valid on the physical level at any time, is usually called propagation by *immediate* (*eager*) *conversion* (e.g., realized in the GemStone system [21]). In the context of schema evolution, this is a very costly strategy: after any type change all instances may have to be converted. Therefore, two main alternatives to eager instance conversion have been proposed: (i) *delayed (lazy) conversion* is a strategy, where instances are only converted on demand, i.e., at the moment when they are touched the first time after the type change by a read or write operation. In combination with *screening*, one can even avoid to convert instances after a read, because screening is a kind of "logical" instance conversion, where objects are interpreted in the

[5] Notice, that the same situation shows between the meta-schema level and the schema level as well.

new definition (e.g., ORION [3]); and (ii) *type versioning* is a strategy where a new version of a type is created whenever a type is modified. Instances created after the change, belong to the new version [26, 1], and the same object may be viewed through different versions of the schema.[6]

Our approach was to realize a more "sophisticated" state function $\overline{\sigma}$, based on a combination of the above strategies, that keeps databases type-valid without any instance conversion. For this purpose, the database state of Section 3 was enhanced, such that each value of a function is carrying in addition information about its type (*type(f)(o)*). Whenever, the type checker allows the application of function f to object o, $\overline{\sigma}$ looks up in the database for the state $\sigma(f)(o)$.

$$\overline{\sigma}(f)(o) := \begin{cases} \sigma(f)(o) & \text{, if } \mathit{type}(f)(o) \preceq \mathit{range}(f), \\ \mathit{cast}(\,\sigma(f)(o)\,) & \text{, if } \mathit{type}(f)(o) \preceq \mathit{object} \text{ and } \mathit{range}(f) \preceq \mathit{object}, \\ \mathit{transf}(\,\sigma(f)(o)\,) & \text{, if } \mathit{type}(f)(o) \not\preceq \mathit{object} \text{ or } \mathit{range}(f) \not\preceq \mathit{object}, \\ \bot & \text{, otherwise.} \end{cases}$$

If there is no actually defined value for $f(o)$ in the database – for example, because f was added by a schema update after o has been created – $\overline{\sigma}$ returns a null value ($\bot$). If there is a value and it matches the range type of f, which is the ordinary case, $\sigma(f)(o)$ is retrieved.

Moreover, $\overline{\sigma}$ transforms values that do not match the range type of f – for example, because the signature of f was modified by a schema update in between –, using the *cast* or *transf* operator. The *cast* operator maps objects from one object type into an other object type, as it is known from other strictly typed languages [14]. The *transf* operator does a similar transformation for value types. It is mainly a table of mapping rules between integers, strings, booleans, etc.

With this definition of $\overline{\sigma}$, we can show:

Theorem All schema changes of the taxonomy, performed on a COCOON database $< \mathcal{O}, \overline{\sigma} >$ using COOL algebra operators, leave the database type-valid, w.r.t. to the modified state function $\overline{\sigma}$.

Implementing the database state like this, direct updates to schema objects can be performed, and the database is kept type-valid without explicit propagation to instances. The proof is just a case analysis following the explanations in Section 5.1.

5 Database Evolution as a Sample Application

So far, the framework is built such that we can allow a user to make direct schema object updates. We make now use of the above prerequisites and turn to a specific purpose of the framework: database evolution.

[6] It is finally a matter of performance trade-offs, which of these alternatives is chosen ([11] gives a performance comparison of immediate vs. delayed conversion).

5.1 The Taxonomy with Direct Updates to Schema Objects

The first step towards realizing schema evolution operations is to find a corresponding COOL algebra expression on schema objects for each of the update primitives in the presented taxonomy. We do that in the following and mention the constraints that are checked to satisfy schema correctness as well as how propagation to existing instances (data level objects) is resolved.

Creating new Schema Objects. Schema objects are created with the generic **insert** statement, by inserting a new object into one of the meta-classes *Object-Types, Functions, Class-Defs* or *View-Defs*. Thereby, the initial values of the new objects are to be specified. All given parameters must satisfy unique naming, not null, and not cyclic constraints.

(1.1.1) $t :=$ **insert** [*tname:=* n_t, *localf:=* { $f_1, \ldots, f_n$ }, *supert:=* { $t_1, \ldots, t_l$ }] (*Object-Types*);

Since the set of functions *localf(t)* that is associated with the newly created type is already existing, this schema update is simply the assignment of a type name the a set of functions, and does not propagate to data level objects.

(2.1.1) $f :=$ **insert** [*fname:=* n_f, *sign:= dom_ran, unique:= uq, notnull:= nn, notcyclic:= nc, inverse:=* f_i] (*Functions*);

After creation, the new function is not in a type interface, no object has a value for the new function, and therefore no explicit propagation is needed.

(3.1.1) $c :=$ **insert** [*cname:=* n_c, *auto:= as, mtype:=* t_m, *pred:=* p_c, *superc:=* { $c_1, \ldots, c_n$ }] (*Class-Defs*);
$v :=$ **insert** [*cname:=* n_c, *query:=* qr] (*View-Defs*);

Classes can be created either by a class definition (insertion into the meta-class *Class-Defs*) or as a view (insertion into the meta-class *View-Defs*). Since the extents of views and *all*-classes are defined by necessary and sufficient conditions, they are populated automatically if necessary.

Deleting schema objects. Applying the **delete** operator to a type, function, or class object removes it from the database schema. Deletion of schema objects may be refused, if it violates the closure constraint.

(1.1.2) **delete** (*t*);

Deletion of a type does not propagate to the instance level. This operation simply removes the assignment of a named type object to a set of applicable functions (cf. generating a type). The instances of that type remain in the database, and all functions of the type interface are still applicable.

(2.1.2) **delete** (*f*);

After deletion of a function object, that state of this function becomes useless. Nevertheless, there is no need for immediate removal of the values of f for data objects, because the type checker would no longer permit the application of f to an object.[7]

In case that the function f is part of a class predicate or query defining a view, deleting f could result in a run-time error. We discuss this problem below.

(3.1.2) **delete** (c);

Deletion of a class has no influence on the class members.

Changing the name of schema objects. Names of types, functions, or classes can be changed, using the generic **set** operator to assign a new name string.

(1.2.1) **set** [$tname{:=}\ n_t$] (t);
(2.2.1) **set** [$fname{:=}\ n_f$] (f);
(3.2.1) **set** [$cname{:=}\ n_c$] (c);

These changes must follow the unique naming constraint. Changing object names does not need to be propagated any further.

Changing the type interface. Using the generic **add** and **remove** operators, a type object t_i can be added or removed to/from the type interface in order to change the set of supertypes. Similarly, a new function object f_i can be added or removed to/from the set of locally defined functions.

(1.2.2) **add** [t_i] (*supert(t)*);
remove [t_i] (*supert(t)*);
(1.2.3) **add** [f_i] (*localf(t)*);
remove [f_i] (*localf(t)*);

In case of adding supertypes or local functions, new functions become applicable to all instances of t: e.g., if o is an existing instance of t, then $\sigma(f_i)(o)$ becomes type-valid. But, following our lazy propagation strategy (cf. Section 4.3), the state of the data objects can remain unchanged, because σ returned a null value anyway.

After removing the function f_i from the type interface, the application of f_i to an object o in context of this type would not pass the type checker. Thus, it does not matter whether the actual value is kept in the database state.

Changing the function signature. The signature of a function object can be changed directly, by assigning a new object *ft* (of type *fcn-type*), using the generic **set** operator.[8]

(2.2.2) **set** [$sign{:=}\ ft$] (f);

[7] Deleting the values of f from the database is an implementation issue. It could be done later, e.g. in a clean-up phase, cf. Section 4.3.

[8] For the considerations here, we assume, that for each desired combination of domain and range, there already exists a function type object that represents it. For example, *obj_set_employee* for signature with domain *object* and range *set of employee*.

Altering the range of a function may cause that existing instance values must be transformed to other types. This is done by the cast-option in the state function. If the domain is restricted, the type-checker avoids the use of these function for an object that is not instance of the new domain; if the range gets more general, substitutability guarantees that function values are still valid. Anyhow, all changes of the signature are handled by the σ-function according to Section 4.3.

Changing the function constraints. For changing the function constraints, one must distinguish between two cases: those making the constraint more restrictive (setting unique, not null, not cyclic to true) and those making it less restrictive (setting constraints to false). Only the former ones give problems, because there could be data objects already in the database that do not follow the new more restrictive constraints.

(2.2.3) **set** [*unique:= uq*] (*f*); **set** [*notnull:= nn*] (*f*);
set [*notcyclic:= nc*] (*f*); **set** [*inverse:= f*$_i$] (*f*);

Changing class properties. Properties of classes can be changed by assigning new selector *as*, membertype t_m, or class predicate p_c.

(3.2.2) **set** [*auto:= as*] (*c*);
(3.2.3) **set** [*mtype:=* t_m] (*c*);
(3.2.4) **set** [*pred:=* p_c] (*c*);
(3.2.5) **add** [c_i] (*superc(c)*);
remove [c_i] (*superc(c)*);
(3.2.6) **set** [*query:= qr*] (*c*);

Since the extent of a class is not stored but computed (no matter whether defined as a class or as a view), changing these class properties has no effect on instances, but simply needs recomputation of the extent. Assigning a new query or an new predicate needs of type-checking (see below).

5.2 Embedding Schema Updates into Methods

Although all schema update primitives have been mapped into an algebra expression, isolated use of these updates may not be adequate or desirable for many reasons:

- Since a small update on a schema object may have serious effects on databases, it may be useful to ensure additional pre and post conditions for such an update.
- If a direct update violates model inherent integrity constraints (uniqueness, not null, or not cyclic), the operation is just rejected. It may be more adequate, if meaningful error messages were returned.
- The user's authorization to make the schema update must be checked. In general, this is done using a normal authorization model for objects [22], and applying it to the schema objects. Anyway, some special restrictions to schema updates may be checked globally, e.g. forbidding that meta-types are deleted.

- The meta-schema does not contain any information about physical design issues. Physical descriptors for the classes and functions may be given explicitly together with a schema update. As an example, consider that some views or derived (computed) functions may be materialized, indexes should be created, objects may be clustered, or statistics on the transaction load should be kept.
- Run-time information for dynamic type-checking or function bindings is not part of the meta-schema as well. If for example the predicate of a class is changed, it must be type-checked for that no run-time error can occur; and since functions (e.g., queries of a view definition) can be derived from others, we have to represent dependencies between functions. This problem is known as "behavioral consistency" [30].
- Finally, user defined semantics may be added. For example, one may want to enforce, that deleting a class includes deletion of all instances.

EXAMPLE 2: Consider deletion of type objects ($U_{1.1.2}$). The following method requires that before type t is deleted, t is not a meta-type and that t is not used somewhere else.

procedure *delete_type (t : object-type)* **is**
    — *delete an existing type object (update 1.1.2 of the taxonomy)*
require
    $t \notin$ *MetaTypes* **and**
    select *[t ∈ supert] (Object-Types)* $= \emptyset$ **and**
    select *[t = etype] (Set-Types)* $= \emptyset$ **and**
    select *[t = dom* **or** *t = ran] (Fcn-Types)* $= \emptyset$ **and**
    select *[t = mtype] (Class-Defs)* $= \emptyset$
begin
    delete *(t);*
end — *delete_type* ◇

We therefore propose that updates to schema objects should only happen encapsulated into save schema update methods. In an object-oriented system, these methods can be implemented as overriding of the standard methods of the meta-types.

6 Conclusion and Future Work

We presented a general framework for investigation of database evolution based on an object-oriented data model and an object algebra for queries and updates.

The contribution of this framework is threefold. First, we introduced a system where schema and meta-schema objects are treated as ordinary objects, and the meta-schema is fully available to the user. Beyond, we defined the semantics of the generic update operators such that they can be applied on any object, independent of whether it belongs to a schema or the data level. Finally, we showed that this approach is powerful enough to realize schema evolution, such

that a taxonomy of desirable schema change primitives can be implemented as direct updates to schema objects. They will respect integrity constraints on the schema and they will also leave the database in a type-valid state.

The object model COCOON and its algebra COOL are currently being implemented as a prototype system, respecting the separation of database object into the three different levels [29]. We have defined all primitives for schema updates, and have been setting up a collection of higher level evolution operations, based on these primitives. They will facilitate complete schema design, tailoring, and restructuring. Finally, as a long time goal, we are working towards a tool that supports users in extending and reusing databases. The framework is open in the sense that schema evolution is only one sample application. As a second one, we started to study the feasibility of database integration [25]. We showed, how the distinction of the three object levels can be used for making multidatabases interoperable.

Furthermore, we are extending our meta-schema for physical database design and "behavioral consistency". Information must be included about the implementation (computed/stored) of functions, materialization of class/view extents, and clustering of objects. To detect run-time type errors, function bindings and dependencies between function and query expressions are to be stored.

Acknowledgments. The authors are indebted to Hans-Jörg Schek, Christian Laasch, and Klaus Gaßner for their helpful discussion on the paper. A preliminary version of the framework was presented in [28].

References

1. J. Andany, M. Leonard, and C. Palisser. Management of schema evolution in databases. In *Proc. 17th Int'l Conf. on Very Large Data Bases (VLDB)*, Barcelona, Spain, September 1991.
2. 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), January 1987.
3. 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*, San Francisco, February 1987.
4. C. Beeri. Formal models for object-oriented databases. In DOOD89 [7].
5. C. Beeri. New data models and languages – the challenge. In *Proc. ACM Symp. on Principles of Database Systems*, San Diego, California, June 1992.
6. E. Casais. *Managing Evolution in Object-Oriented Environments: An Algorithmic Approach*. Phd thesis, Centre univ. d'informatique, Université Genève, 1991.
7. *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, December 1989.
8. *Proc. 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, December 1991.
9. *Proc. 7th Int'l Conf. Entity-Relationship Approach*, Rome, Italy, November 1988.
10. J. Göers and A. Heuer. Definition and application of metaclasses in an object-oriented database model. Inst. für Informatik, TU Clausthal, Germany, June 1991.

11. G. Harrus, F. Velez, and R. Zicari. Implementing schema updates in an object-oriented database system: A cost analysis. Technical report, GIP Altair, Le Chesnay Cedex, France, 1990.
12. G. Kappel and M. Schrefl. A behavior integrated entity-relationship appraoch for the design of object-oriented databases. In ER88 [9].
13. W. Klas. *A Metaclass System for Open Object-Oriented Data Models.* PhD thesis, Technische Universität Wien, January 1990.
14. C. Laasch and M. H. Scholl. Generic update operations keeping object-oriented databases consistent. In *Proc. 2nd GI-Workshop on Information Systems and Artificial Intelligence (IS/KI)*, FAW Ulm, Germany, February 1992.
15. B.S. Lerner and A.N. Habermann. Beyond schema evolution to database reorganization. In *Proc. Int'l Conf. OOPSLA/ECOOP*, Ottawa, Canada, October 1990.
16. T.W. Ling. External schemas of entity-relationship based data base management systems. In ER88 [9].
17. S.B. Navathe and M.K. Pillalamarri. OOER: toward making the E-R appraoch object-oriented. In ER88 [9].
18. S.L. Osborn. The role of polymorphism in schema evolution in an object-oriented database. *IEEE Trans. on Know. and Data Engineering*, 1(3), September 1989.
19. C. Parent, H. Rolin, K. Yétongnon, and S. Spaccapietra. An ER calculus for the entity-relationship complex model. In *Proc. 8th Int'l Conf. Entity-Relationship Approach*, Toronto, Canada, October 1989.
20. C. Parent and S. Spaccapietra. About entities, complex objects and object-oriented data models. In *Information System Concepts - An In-depth Analysis, Proc. of IFIP WG 8.1 Working Conference*, Namur, October 1989.
21. 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)*, October 1987.
22. F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model for authorization for next-generation database systems. *ACM TODS*, 16(1), March 1991.
23. M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In DOOD91 [8].
24. M.H. Scholl and H.-J. Schek. A relational object model. In *Proc. 3rd Int'l Conf. on Database Theory (ICDT'90)*, Paris, 1990.
25. M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for multi-objectbases. In *Proc. Int'l Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.
26. A.H. Skarra and S.B. Zdonik. Type evolution in an object-oriented database. In *Research Directions in Object-Oriented Programming*, 1987.
27. L. Tan and T. Katayame. Meta operations for type management in object-oriented databases. In DOOD89 [7].
28. M. Tresch. A framework for schema evolution by meta object manipulation. In *Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1991.
29. M. Tresch and M.H. Scholl. Implementing an object model on top of commercial database systems. In *Workshop on Foundations of Database Systems*, Volkse, Germany, May 1991.
30. E. Waller. Schema updates and consistency. In DOOD91 [8].
31. R. Zicari. A framework for schema updates in an object-oriented database system. In *Proc. 7th Int'l Conf. on Data Engineering (ICDE)*, Kobe, Japan, April 1991.

A The Meta Schema

In the sequel, we define the COCOON meta-schema. Usually the purpose of a meta-schema is twofold: (i) to describe the object model using its own notation, and (ii) to represent data dictionary information. Nevertheless, since the scope of this paper is schema evolution, our meta-schema is only given and explained as far as it is necessary to define the semantics of schema updates.

As proposed in Definition 3.2, the meta-schema is composed of meta-types and meta-classes. The first meta-type represents data and object types. That is, each COCOON type is represented by an object, being instance of the following meta-type:

type *type* **isa** *object* =
tname : **unique** *string* , // type name
functs : **set of** *function* ; // functions, applicable to the type's instances

Most types are defined by users in order to specify the interface of an abstract object type (i.e. to define the signatures of the applicable functions). As usual in object-oriented systems, such types can be ordered in type-hierarchies. The meta-type *object-type* is a specialized subtype:

type *object-type* **isa** *type* =
localf : **set of** *function* **not null** , // local functions, independent of inheritance
supert : **set of** *object-type* // explicitly defined
not null not cyclic , supertypes of the type

Whereas *localf(t)* are the functions defined to be applicable to t's instances independent of inheritance, for an abstract object type, the set of all applicable functions *functs(t)* is derived by the union of the local functions *localf(t)* and the functions inherited from the supertypes:

$$functs(t) := localf(t) \bigcup_{t_i \in supert(t)} functs(t_i)$$

Notice, that type checking is based on all functions *functs(t)*, not only on the local ones. We distinguish for each type t between explicit and implicit supertypes. The former ones are those explicitly assigned with the meta-function *supert(t)*, whereas the implicit ones are derived from the set of applicable functions as follows:

$$t \preceq t' \iff functs(t) \supseteq functs(t')$$

That is, a type t' is supertype of t, if the applicable functions of t are a superset the functions of t'.

In addition to abstract object types, two more subtypes represent constructed data types: the set and function types. Since these types are normally created and managed internally by the system, most of them are unnamed.

type *set-type* **isa** *type* =
etype : *type* **not null** ; // the type of the elements in the set

type *fcn-type* **isa** *type* =
dom : *type* **not null** , // the domain type of the function
ran : *type* **not null** ; // the range type of the function

The second meta-type represents COCOON functions. They are named, have a signature, and their values can be restricted by a set of constraints.

type *function* **isa** *object* =
fname : **unique** *string* **not null** , // function name
sign : *fcn-type* **not null** , // function signature
unique : *boolean* **not null** , // uniqueness constraint
notnull : *boolean* **not null** , // not undef constraint
notcyclic : *boolean* **not null** , // cycle free constraint
inverse : *function* **inverse** *inverse* ; // inverse function

Information about the implementation of the function (e.g. whether the function result is stored or computed), is intentionally excluded from the meta-schema, since this is irrelevant for schema evolution.

The third meta-type represents COCOON classes.

type *class* **isa** *object* =
cname : **unique** *string* **not null** , // class name
extent : **set of** *object* ; // actual class members

Since we treat views as classes with implicitly defined type and extension, two subtypes of meta-type *class* are distinguished: those defined as a class, and those defined as a view.

type *class-def* **isa** *class* =
auto : *boolean* **not null** , // (see below)
mtype : *type* , // explicit member type
pred : *function* , // the class predicate
superc : **set of** *class* // explicit super classes
not null not cyclic ,
pmemb : **set of** *object* ; // potential class members

type *view-def* **isa** *class* =
query **not null** : *expression* , // query defining the view

The value of *auto(c)* is true, iff the class *c* is defined with the selector **all**. In these cases the system can decide whether an object belongs to the extent of a class. If classes are defined by the selector **some** there are just necessary conditions defined. The information about class membership is specified by the user in terms of adding and removing objects to/from a class explicitly. This information is stored by the set *pmemb* that represents the potential members

of a class. These are objects that are added to a class, but need not to fulfill the class predicate (for more detail see [14]). *extent(c)* derives the actual set of member objects (extent) of a class, i.e. a subset of *pmemb* which elements fulfill the class predicate. The actual derived member type of the class objects is either equal to *mtype(c)*, if a member type is explicitly defined, or otherwise, it must be derived from the member type of *c*'s superclasses and class predicate.

Together with each meta-type, there is a meta-class holding the actual instances of the meta-type.

class *Types* : *type* **some** *Objects* ;
class *Set-Types* : *set-type* **some** *Types* ;
class *Fcn-Types* : *fcn-type* **some** *Types* ;
class *Object-Types* : *object-type* **some** *Types* ;

class *Functions* : *function* **some** *Objects* ;

class *Classes* : *class* **some** *Objects* ;
class *Class-Defs* : *class-def* **some** *Classes* ;
class *View-Defs* : *view-def* **some** *Classes* ;

In addition, the following view collects all classes with implicitly defined extent. That is, the view-defined classes and the class-defined ones with an **all** -selector:

view *Views* **as** *View-Defs* **union select** [*auto(c)*] (*c: Class-Defs*);

Table of Contents

This article was processed using the LaTeX macro package with LLNCS style