A Semantic Approach for Schema Evolution and Versioning in Object-Oriented Databases

Enrico Franconi[1], Fabio Grandi[2], and Federica Mandreoli[2]

[1] Univ. of Manchester, Dept. of Computer Science, Manchester M13 9PL, UK
franconi@cs.man.ac.uk
[2] Univ. di Bologna, Dip. di Elettronica, Informatica e Sistemistica, Bologna, Italy
{fgrandi, fmandreoli}@deis.unibo.it

Abstract. In this paper a semantic approach for the specification and the management of databases with evolving schemata is introduced. It is shown how a general object-oriented model for schema versioning and evolution can be formalized; how the semantics of schema change operations can be defined; how interesting reasoning tasks can be supported, based on an encoding in description logics.

1 Introduction

The problems of schema evolution and versioning arose in the context of long-lived database applications, where stored data were considered worth surviving changes in the database schema [23]. According to a widely accepted terminology [18], a database supports *schema evolution* if it permits modifications of the schema without the loss of extant data; in addition, it supports *schema versioning* if it allows the querying of all data through user-definable version interfaces. For the sake of brevity, schema evolution can be considered as a special case of schema versioning where only the current schema version is retained. With schema versioning, different schemata can be identified and selected by means of a suitable "coordinate system": symbolic labels are often used in design systems to this purpose, whereas proper time values are the elective choice for temporal applications [13, 14].

In this paper, we present and discuss a formal approach, for the specification and management of schema versioning in a very general object-oriented data model. The adoption of an object-oriented data model is the most common choice in the literature concerning schema evolution, though schema versioning in relational databases [10] has also been studied deeply. The approach is based on:

- the definition of an extended object-oriented model supporting evolving schemata (equipped with all the usually adopted schema changes) for which a semantics is provided;
- the formulation of interesting reasoning tasks, in order to support the design and the management of an evolving schema;

– an encoding, which has been proved correct, as inclusion dependencies in a suitable Description Logic, which can then be used to solve the tasks defined for the schema versioning.

Within such a framework, the main problems connected with schema versioning support will be formally characterised, both from a logical and computational viewpoint, leading to the following enhancements.

– The complexity of schema changes becomes potentially unlimited: in addition to the classical schema change primitives (a well-known comprehensive taxonomy can be found in [3]), our approach enables the definition of complex and articulated schema changes.
– We define different notions of consistency, related to the existence of a legal database for the global schema or for a single schema version, or related to the consistency of single classes within a consistent schema (version). Classification tasks we define include the discovery of implicit inclusion/inheritance relationships between classes ([4]). Decidability and complexity results are available for the above mentioned tasks in our framework; tools based on Description Logics can be used for solving these tasks.
– The process of schema transformation can be formally checked. The provided semantics of the various schema change operations makes it possible to reduce the correctness proof of complex sequences of schema changes to solvable reasoning tasks.

However, our semantic approach has not thoroughly addressed the so-called *change propagation* problem yet, which concerns the effects of schema changes on the underlying data instances. In general, change propagation can be accomplished by populating the new schema version with the results of queries involving extant data connected to previous schema versions. In Section 6, our proposal will be reviewed in the light of previous approaches involving query languages (e.g. [1, 9, 17, 19]), and directions for future developments will also be sketched.

The paper is organised as follows. After a survey of the current status of the field, Section 3 first introduces the object-oriented model for evolving schemata, and then formally defines the relevant reasoning problems supporting the design and the management of an evolving schema. Section 5 introduces a provably correct encoding of the model into a Description Logic, so that theoretical, computational and practical results can be proved. A critical discussion (Sec. 6) about the proposed approach precedes the conclusions (Sec. 7).

2 Related Work

The problems of schema evolution and schema versioning support have been diffusively studied in relational and object-oriented database papers: [23] provides an excellent survey on the main issues concerned. The introduction of schema change facilities in a system involves the solution of two fundamental problems:

the *semantics of change*, which refers to the effects of the change on the schema itself, and the *change propagation*, which refers to the effects on the underlying data instances. The former problem involves the checking and maintenance of schema consistency after changes, whereas the latter involves the consistency of extant data with the modified schema.

In the object-oriented field, two main approaches were followed to ensure consistency in pursuing the "semantics of change" problem. The first approach is based on the adoption of *invariants* and *rules*, and has been used, for instance, in the ORION [3] and O_2 [11] systems. The second approach, which was proposed in [22], is based on the introduction of *axioms*. In the former approach, the invariants define the consistency of a schema, and definite rules must be followed to maintain the invariants satisfied after each schema change. In the latter approach, a sound and complete set of axioms (provided with an inference mechanism) formalises the *dynamic schema evolution*, which is the actual management of schema changes in a system in operation. The compliance of the available primitive schema changes with the axioms automatically ensures schema consistency, without need for explicit checking, as incorrect schema versions cannot actually be generated.

For the "change propagation" problem, several solutions have been proposed and implemented in real systems [3, 11, 21]. In most cases, simple *default* mechanisms can be used or user-supplied conversion functions must be defined for non-trivial extant object updates. A notable exception is [19], where a formal notion of logical consistency of the global approach is devised and proved decidable, in the context of a simple object-oriented data model. This work is different from the previous solutions in that there is no automatic reorganisation of the data after the schema update, but only a consistency check of the resulting database.

As far as complex schema changes are concerned, [20] considered sequences of schema change primitives to make up high-level useful changes, solving the propagation to objects problem with simple schema integration techniques. However, with this approach, the consistency of the resulting database is not guaranteed nor checked. In [5], high-level primitives are defined as *well-ordered* sets of primitive schema changes. Consistency of the resulting schema is ensured by the use of invariants' preserving elementary steps and by *ad-hoc* constraints imposed on their application order. In other words, consistency preservation is dependent on an accurate design of high-level schema changes and, thus, still relies on the designer's skills.

3 An Object-Oriented Data Model for Evolving Schemata

The object-oriented model we propose allows for the representation of multiple schema versions. It is based on an expressive version of the "snapshot" – i.e., single-schema – object-oriented model introduced by [1] and further extended and elaborated in its relationships with Description Logics by [7, 8]; in this paper we borrow the notation from [7]. The language embodies the features of the static parts of UML/OMT and ODMG and, therefore, it does not take into account those aspects related to the definition of methods.

The definition of an evolving schema $\mathcal{S}$ is based on a set of class and attribute names ($\mathcal{C}_\mathcal{S}$ and $\mathcal{A}_\mathcal{S}$ respectively) and includes a partially ordered set of schema versions. The initial schema version of $\mathcal{S}$ contains a set of class definitions having one of the following forms:

$$\underline{\text{Class}}\ C\ \underline{\text{is-a}}\ C_1, \ldots ,C_h\ \underline{\text{disjoint}}\ C_{h+1}, \ldots ,C_k\ \underline{\text{type-is}}\ T.$$
$$\underline{\text{View-class}}\ C\ \underline{\text{is-a}}\ C_1, \ldots ,C_h\ \underline{\text{disjoint}}\ C_{h+1}, \ldots ,C_k\ \underline{\text{type-is}}\ T.$$

A class definition introduces just necessary conditions regarding the type of the class – this is the standard case in object-oriented data models – while views are defined by means of both necessary and sufficient conditions. The symbol T denotes a type expression built according to the following syntax:

$$\begin{array}{lll} T \rightarrow C \ \mid & & \\ \quad \underline{\text{Union}}\ T_1, \ldots ,T_k\ \underline{\text{End}} \ \mid & & \text{(union type)} \\ \quad \underline{\text{Set-of}}\ [\text{m,n}]\ T \ \mid & & \text{(set type)} \\ \quad \underline{\text{Record}}\ A_1{:}T_1, \ldots ,A_k{:}T_k\ \underline{\text{End}}\ . & & \text{(record type)} \end{array}$$

where $C \in \mathcal{C}_\mathcal{S}$, $A_i \in \mathcal{A}_\mathcal{S}$, and [m,n] denotes an optional cardinality constraint.

A schema version in $\mathcal{S}$ is defined by the application of a sequence of schema changes to a preceding schema version. The schema change taxonomy is built by combining the model elements which are subject to change with the elementary modifications, add, drop and change, they undergo. In this paper only a basic set of elementary schema change operators will be introduced; it includes the standard ones found in the literature (e.g., [3]); however, it is not difficult to consider the complete set of operators with respect to the constructs of the data model.

$$\begin{array}{l} M \rightarrow \underline{\text{Add-attribute}}\ C,\ A,\ T\ \underline{\text{End}} \ \mid \\ \quad \underline{\text{Drop-attribute}}\ C,\ A\ \underline{\text{End}} \ \mid \\ \quad \underline{\text{Change-attr-name}}\ C,\ A,\ A'\ \underline{\text{End}} \ \mid \\ \quad \underline{\text{Change-attr-type}}\ C,\ A,\ T'\ \underline{\text{End}} \ \mid \\ \quad \underline{\text{Add-class}}\ C,\ T\ \underline{\text{End}} \ \mid \\ \quad \underline{\text{Drop-class}}\ C\ \underline{\text{End}} \ \mid \\ \quad \underline{\text{Change-class-name}}\ C,\ C'\ \underline{\text{End}} \ \mid \\ \quad \underline{\text{Change-class-type}}\ C,\ T'\ \underline{\text{End}} \ \mid \\ \quad \underline{\text{Add-is-a}}\ C,\ C'\ \underline{\text{End}} \ \mid \\ \quad \underline{\text{Drop-is-a}}\ C,\ C'\ \underline{\text{End}}\ . \end{array}$$

In this paper, we omit the definition of a schema version coordinate mechanism and simply reference distinct schema versions by means of different subscripts. Any kind of versioning dimension usually considered in the literature could actually be employed – such as transaction time, valid time and symbolic labels – provided that a suitable mapping between version coordinates and index values is defined.

Definition 1. *An evolving object-oriented schema is a tuple* $\mathcal{S} = (\mathcal{C}_\mathcal{S}, \mathcal{A}_\mathcal{S}, \mathcal{SV}_0, \mathcal{M}_\mathcal{S})$, *where:*

- $\mathcal{C}_\mathcal{S}$ *is a finite set of class names;*
- $\mathcal{A}_\mathcal{S}$ *is a finite set of attribute names;*
- $\mathcal{SV}_0$ *is the initial schema version, which includes class and view definitions for some* $C \in \mathcal{C}_\mathcal{S}$*;*
- $\mathcal{M}_\mathcal{S}$ *is a set of modifications* $\mathcal{M}_{ij}$*, where* i, j *denote a pair of version coordinates. Each modification is a finite sequence of elementary schema changes.*

The set $\mathcal{M}_\mathcal{S}$ induces a partial order $\mathcal{SV}$ over a finite and discrete set of schema versions with minimal element $\mathcal{SV}_0$. Hence $\mathcal{SV}_0$ precedes every other schema version and the schema version $\mathcal{SV}_j$ represents the outcome of the application of $\mathcal{M}_{ij}$ to $\mathcal{SV}_i$. $\mathcal{S}$ is called *elementary* if every $\mathcal{M}_{ij}$ in $\mathcal{M}_\mathcal{S}$ contains only one elementary modification, and every schema version $\mathcal{SV}_i$ has at most one immediate predecessor. In the following we will consider only elementary evolving schemata.

Let us now introduce the meaning of an evolving object-oriented schema $\mathcal{S}$. Informally, the semantics is given by assigning to each schema version a possible legal database state – i.e., a legal instance of the schema version – conforming to the constraints imposed by the sequence of schema changes starting from the initial schema version.

Formally, an instance $\mathcal{I}$ of $\mathcal{S}$ is a tuple $\mathcal{I} = (\mathcal{O}^\mathcal{I}, \rho^\mathcal{I}, (\mathcal{I}_0, \ldots, \mathcal{I}_n))$, consisting of a finite set $\mathcal{O}^\mathcal{I}$ of object identifiers, a function $\rho^\mathcal{I} : \mathcal{O}^\mathcal{I} \mapsto \mathcal{V}_{\mathcal{O}^\mathcal{I}}$ giving a value to object identifiers, and a sequence of version instances $\mathcal{I}_i$, one for each schema version $\mathcal{SV}_i$ in $\mathcal{S}$. The set $\mathcal{V}_{\mathcal{O}^\mathcal{I}}$ of values is defined by induction as the smallest set including the union of $\mathcal{O}^\mathcal{I}$ with all possible "sets" of values and with all possible "records" of values. Although the set $\mathcal{V}_{\mathcal{O}^\mathcal{I}}$ is infinite, we consider for an instance $\mathcal{I}$ the finite set $\mathcal{V}_\mathcal{I}$ of *active values*, which is the subset of $\mathcal{V}_{\mathcal{O}^\mathcal{I}}$ formed by the union of $\mathcal{O}^\mathcal{I}$ and the set of values assigned by $\rho^\mathcal{I}$ ([7]).

A version instance $\mathcal{I}_i = (\pi^{\mathcal{I}_i}, \cdot^{\mathcal{I}_i})$ consists of a total function $\pi^{\mathcal{I}_i} : \mathcal{C}_\mathcal{S} \mapsto 2^{\mathcal{O}^\mathcal{I}}$, giving the set of object identifiers in the extension of each class $C \in C_\mathcal{S}$ for that version, and of a function $\cdot^{\mathcal{I}_i}$ (the *interpretation* function) mapping type expressions to sets of values, such that the following is satisfied:

$$
\begin{aligned}
C^{\mathcal{I}_i} &= \pi^{\mathcal{I}_i}(C) \\
(\underline{\text{Union}}\ T_1, \ldots, T_k\ \underline{\text{End}})^{\mathcal{I}_i} &= T_1^{\mathcal{I}_i} \cup \ldots \cup T_k^{\mathcal{I}_i} \\
(\underline{\text{Set-of}}\ [\text{m,n}]\ T)^{\mathcal{I}_i} &= \{\{\!|\ v_1, \ldots, v_k\ |\!\}\mid m \leq k \leq n, v_j \in T^{\mathcal{I}_i}, \\
&\quad\ \text{for } j \in \{1, \ldots, k\}\} \\
(\underline{\text{Record}}\ A_1{:}T_1, \ldots, A_k{:}T_k\ \underline{\text{End}})^{\mathcal{I}_i} &= \{[\![A_1 : v_1, \ldots, A_k : v_k, \ldots, A_h : v_h]\!] \mid \\
&\quad\ \text{for some } h \geq k, \\
&\quad\ v_j \in T_j^{\mathcal{I}_i}, \text{for } j \in \{1, \ldots, k\}, \\
&\quad\ v_j \in \mathcal{V}_{\mathcal{O}^\mathcal{I}}, \text{for } j \in \{k+1, \ldots, h\}\}
\end{aligned}
$$

Add-attribute $\mathbf{C}$, $\mathbf{A}$, $\mathbf{T}$	$\pi^{\mathcal{I}_j}(\mathbf{C}) = \pi^{\mathcal{I}_i}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\![\ldots, \mathbf{A}: v, \ldots]\!] \wedge v \in \mathbf{T}^{\mathcal{I}_j}\}$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}$
Drop-attribute $\mathbf{C}$, $\mathbf{A}$	$\pi^{\mathcal{I}_i}(\mathbf{C}) = \pi^{\mathcal{I}_j}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\![\ldots, \mathbf{A}: v, \ldots]\!]\}$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}$
Change-attr-name $\mathbf{C}$, $\mathbf{A}$, $\mathbf{A}'$	$\pi^{\mathcal{I}_i}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\![\ldots, \mathbf{A}: v, \ldots]\!]\} = \pi^{\mathcal{I}_j}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\![\ldots, \mathbf{A}': v, \ldots]\!]\}$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}$
Change-attr-type $\mathbf{C}$, $\mathbf{A}$, $\mathbf{T}'$	$\pi^{\mathcal{I}_i}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\![\ldots, \mathbf{A}: v, \ldots]\!] \wedge v \in \mathbf{T}'^{\mathcal{I}_j}\} = \pi^{\mathcal{I}_j}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\![\ldots, \mathbf{A}: v, \ldots]\!]\}$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}$
Add-class $\mathbf{C}$, $\mathbf{T}$	$\pi^{\mathcal{I}_i}(\mathbf{C}) = \emptyset$, $\rho^{\mathcal{I}}(\pi^{\mathcal{I}_i}(\mathbf{C})) \subseteq \mathbf{T}^{\mathcal{I}_j}$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}$
Drop-class $\mathbf{C}$	$\pi^{\mathcal{I}_j}(\mathbf{C}) = \emptyset$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}$
Change-class-name $\mathbf{C}$, $\mathbf{C}'$	$\pi^{\mathcal{I}_i}(\mathbf{C}) = \pi^{\mathcal{I}_j}(\mathbf{C}')$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}, \mathbf{C}'$
Change-class-type $\mathbf{C}$, $\mathbf{T}'$	$\pi^{\mathcal{I}_j}(\mathbf{C}) = \pi^{\mathcal{I}_i}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) \in \mathbf{T}'^{\mathcal{I}_j}\}$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}$
Add-is-a $\mathbf{C}$, $\mathbf{C}'$	$\pi^{\mathcal{I}_j}(\mathbf{C}) = \pi^{\mathcal{I}_i}(\mathbf{C}) \cap \pi^{\mathcal{I}_i}(\mathbf{C}')$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}$
Drop-is-a $\mathbf{C}$, $\mathbf{C}'$	$\pi^{\mathcal{I}_i}(\mathbf{C}) = \pi^{\mathcal{I}_j}(\mathbf{C}) \cap \pi^{\mathcal{I}_j}(\mathbf{C}')$, $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D)$ for all $D \neq \mathbf{C}$

Fig. 1. Semantics of the schema changes.

where an open semantics for records is adopted (called *-interpretation in [1]) in order to give the right semantics to inheritance. In a set constructor if the minimum or the maximum cardinalities are not explicitly specified, they are assumed to be zero and infinite, respectively.

The semantics of schema changes is shown in Fig. 1. For each schema change $\mathcal{M}_{ij}$, it defines a relationship between the instances of the involved schema versions.

A *legal* instance $\mathcal{I}$ of a schema $\mathcal{S}$ should satisfy the constraints imposed by the class definitions in the initial schema version and by the schema changes between schema versions.

Definition 2. *An instance $\mathcal{I}$ of a schema $\mathcal{S}$ is said to be legal if*

- *for each class definition in $\mathcal{SV}_0$*
  Class C is-a $C_1, \ldots, C_h$ disjoint $C_{h+1}, \ldots, C_k$ type-is T, it holds that:
  $C^{\mathcal{I}_0} \subseteq C_j^{\mathcal{I}_0}$ *for each* $j \in \{1, \ldots, h\}$,
  $C^{\mathcal{I}_0} \cap C_j^{\mathcal{I}_0} = \emptyset$ *for each* $j \in \{h+1, \ldots, k\}$,
  $\{\rho^{\mathcal{I}}(o) \mid o \in \pi^{\mathcal{I}_0}(C)\} \subseteq T^{\mathcal{I}_0}$;
- *for each view definition in $\mathcal{SV}_0$*
  View-class C is-a $C_1, \ldots, C_h$ disjoint $C_{h+1}, \ldots, C_k$ type-is T, it holds that:
  $C^{\mathcal{I}_0} \subseteq C_j^{\mathcal{I}_0}$ *for each* $j \in \{1, \ldots, h\}$,
  $C^{\mathcal{I}_0} \cap C_j^{\mathcal{I}_0} = \emptyset$ *for each* $j \in \{h+1, \ldots, k\}$,
  $\{\rho^{\mathcal{I}}(o) \mid o \in \pi^{\mathcal{I}_0}(C)\} = T^{\mathcal{I}_0}$;

– *for each schema change $\mathcal{M}_{ij}$ in $\mathcal{M}$, the version instances $\mathcal{I}_i$ and $\mathcal{I}_j$ satisfy the equations of the corresponding schema change type at the right hand side of Tab. 1.*

4 Reasoning Problems

According to the semantic definitions given in the previous section, several reasoning problems can be introduced, in order to support the design and the management of an evolving schema.

Definition 3. *Reasoning problems:*

a. *Global/local Schema Consistency: an evolving schema $\mathcal{S}$ is globally consistent if it admits a legal instance; a schema version $\mathcal{SV}_i$ of $\mathcal{S}$ is locally consistent if the evolving schema $\mathcal{S}_{\downarrow i}$ – obtained from $\mathcal{S}$ by reducing the set of modifications $\mathcal{M}_{\mathcal{S}_{\downarrow i}}$ to the linear sequence of schema changes in $\mathcal{M}_{\mathcal{S}}$ which led to the version $\mathcal{SV}_i$ from $\mathcal{SV}_0$ – admits a legal instance. In the following, a global reasoning problem refers to $\mathcal{S}$, while a local one refers to $\mathcal{S}_{\downarrow i}$.*
b. *Global/local Class Consistency: a class C is globally inconsistent if for every legal instance $\mathcal{I}$ of $\mathcal{S}$ and for every version $\mathcal{SV}_i$ its extension is empty, i.e., $\forall i.\ \pi^{\mathcal{I}_i}(C) = \emptyset$; a class C is locally inconsistent in the version $\mathcal{SV}_i$ if for every legal instance $\mathcal{I}$ of $\mathcal{S}_{\downarrow i}$ its extension is empty, i.e., $\pi^{\mathcal{I}_i}(C) = \emptyset$.*
c. *Global/local Disjoint Classes: two classes C, D are globally disjoint if for every legal instance $\mathcal{I}$ of $\mathcal{S}$ and for every version $\mathcal{SV}_i$ their extensions are disjoint, i.e., $\forall i.\ \pi^{\mathcal{I}_i}(C) \cap \pi^{\mathcal{I}_i}(D) = \emptyset$; two classes C, D are locally disjoint in the version $\mathcal{SV}_i$ if for every legal instance $\mathcal{I}$ of $\mathcal{S}_{\downarrow i}$ their extensions are disjoint, i.e., $\pi^{\mathcal{I}_i}(C) \cap \pi^{\mathcal{I}_i}(D) = \emptyset$.*
d. *Global/local Class Subsumption: a class D globally subsumes a class C if for every legal instance $\mathcal{I}$ of $\mathcal{S}$ and for every version $\mathcal{SV}_i$ the extension of C is included in the extension of D, i.e., $\forall i.\ \pi^{\mathcal{I}_i}(C) \subseteq \pi^{\mathcal{I}_i}(D)$; a class D locally subsumes a class C in the version $\mathcal{SV}_i$ if for every legal instance $\mathcal{I}$ of $\mathcal{S}_{\downarrow i}$ the extension of C is included in the extension of D, i.e., $\pi^{\mathcal{I}_i}(C) \subseteq \pi^{\mathcal{I}_i}(D)$.*
e. *Global/local Class Equivalence: two classes C, D are globally/locally equivalent if C globally/locally subsumes D and viceversa.*

Please note that the classical *subtyping* problem – i.e., finding the explicit representation of the partial order induced on a set of type expressions by the containment between their extensions – is a special case of class subsumption, if we restrict our attention to view definitions.

As to the *change propagation* task, which is one of the fundamental task addressed in the literature (see Sec. 2), it is usually dealt with by populating the classes in the new version with the result of queries over the previous version. The same applies for our framework: a language for the specification of views can be defined for specifying how to populate classes in a version from the previous data. Formally, we require a query language for expressing views providing a mechanism for explicit creation of object identifiers. At present, our

approach includes one single data pool and a set of version instances which can be thought as views over the data pool. Therefore we consider update as a *schema augmentation* problem in the sense of [17], where the original logical schema is augmented and the new data may refer to the input data. The result of applying any view to a source data pool may involve OIDs from the source besides the new required OIDs to be created. The association between the source OIDs and the target ones should not be destroyed, and only the target data pool will be retained. In Section 6 an alternative approach will be discussed.
Of course, at this point the problem of global consistency of an evolving schema $\mathcal{S}$ becomes more complex, since it involves the additional constraints defined by the data conversions: an instance would therefore be legal if it satisfies not only the constraints of Definition 2 but also the constraints specified by the views. Obviously, a schema $\mathcal{S}$ involving a schema change for which the corresponding semantics expressed by the equation in Tab. 1 and the associated data conversions are incompatible would never admit a legal instance. In general, the introduction of data conversion views makes all the reasoning problems defined above more complex.

We will try to explain the application of the reasoning problems through an example. Let us consider an evolving schema $\mathcal{S}$ describing the employees of a company. The schema includes an initial schema version $\mathcal{SV}_0$ defined as follows:

Class `Employee` type-is Union `Manager`, `Secretary`, `Worker` End;
Class `Manager` is-a `Employee` disjoint `Secretary`, `Worker` ;
Class `Secretary` is-a `Employee` disjoint `Worker` ;
Class `Worker` is-a `Employee`;
View-class `Senior` type-is Record `has_staff`: Set-of [2,n] `Worker` End;
View-class `Junior` type-is Record `has_staff`: Set-of [0,1] `Worker` End;
Class `Executive` disjoint `Secretary`, `Worker`;
View-class `Everybody` type-is Union `Senior`, `Junior` End End;

Figure 2 shows the UML-like representation induced by the initial schema $\mathcal{SV}_0$; note that classes with names prefixed by a slash represent the views. The evolving schema $\mathcal{S}$ includes a set of schema modifications $\mathcal{M}_{\mathcal{S}}$ defined as follows:

($\mathcal{M}_{01}$)	Add-is-a `Secretary`, `Manager` End;
($\mathcal{M}_{02}$)	Add-is-a `Everybody`, `Manager` End;
($\mathcal{M}_{23}$)	Add-is-a `Everybody`, `Secretary` End;
($\mathcal{M}_{04}$)	Add-is-a `Executive`, `Employee` End;
($\mathcal{M}_{45}$)	Add-attribute `Manager`, `IdNum`, `Number` End;
($\mathcal{M}_{56}$)	Change-attr-type `Manager`, `IdNum`, `Integer` End;
($\mathcal{M}_{67}$)	Change-attr-type `Manager`, `IdNum`, `String` End;
($\mathcal{M}_{68}$)	Drop-class `Employee` End;

Let us analyse the effect of each schema change $\mathcal{M}_{ij}$ by considering the schema version $\mathcal{SV}_j$ it produces.

First of all, it can be noticed that in $\mathcal{SV}_0$ the `Junior` and `Senior` classes are disjoint classes and that `Everybody` contains all the possible instances of the

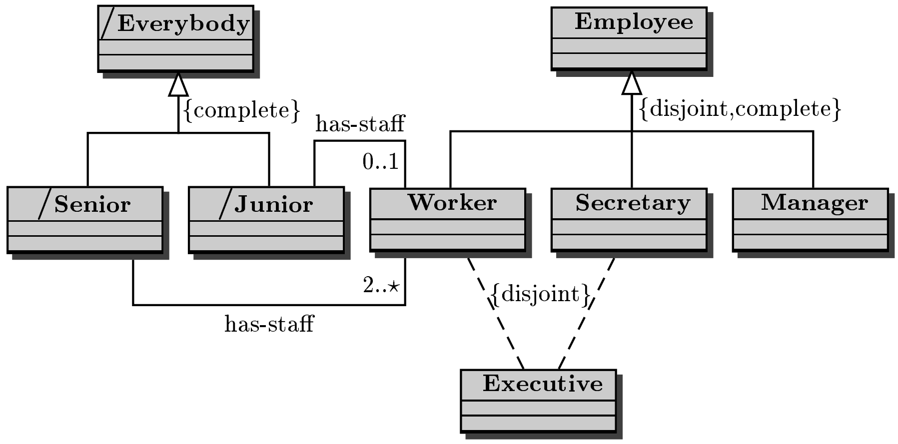

Fig. 2. The Employee initial schema version in UML notation.

record type. In fact, `Everybody` is defined as the union of view classes which are complementary with respect to the record type: any possible record instance is the value of an object belonging either to `Senior` or `Junior`.

`Secretary` is inconsistent in $\mathcal{SV}_1$ since `Secretary` and `Manager` are disjoint: its extension is included in the `Manager` extension only if it is empty (for each version instance $\mathcal{I}_1$, $\texttt{Secretary}^{\mathcal{I}_1} = \emptyset$). Therefore, `Secretary` is *locally inconsistent*, as it is inconsistent in $\mathcal{SV}_1$ but not in $\mathcal{SV}_0$.

The schema version $\mathcal{SV}_3$ is inconsistent because `Secretary` and `Manager`, which are both superclasses of `Everybody`, are disjoint and the intersection of their extensions is empty: no version instance $\mathcal{I}_3$ exists such that $\texttt{Everybody}^{\mathcal{I}_3} \subseteq \emptyset$. It follows that $\mathcal{S}$ is locally inconsistent with respect to $\mathcal{SV}_3$ and, thus, globally inconsistent (although is locally consistent wrt the other schema versions).

In $\mathcal{SV}_4$, it can be derived that `Executive` is locally subsumed by `Manager`, since it is a subclass of `Employee` disjoint from `Secretary` and `Worker` (`Manager`, `Secretary` and `Worker` are a partition of `Employee`).

The schema version $\mathcal{SV}_5$ exemplifies a case of attribute inheritance. The attribute `IdNum` which has been added to the `Manager` class is inherited by the `Executive` class. This means that *every* legal instance of $\mathcal{S}$ should be such that every instance of `Executive` in $\mathcal{SV}_5$ has an attribute `IdNum` of type `Number`, i.e., $\texttt{Executive}^{\mathcal{I}_5} \subseteq \{o \mid \rho^{\mathcal{I}}(o) = [\![\ldots, \texttt{IdNum} : v, \ldots]\!] \wedge v \in \texttt{Number}^{\mathcal{I}_5}\}$. Of course, there is no restriction on the way classes are related via subsumption, and multiple inheritance is allowed as soon as it does not generate an inconsistency.

The Change-attr-type elementary schema change allows for the modification of the type of an attribute with the proviso that the new type is not incompatible with the old one, like in $\mathcal{M}_{56}$. In fact, the semantics of elementary schema changes as defined in Tab. 1 is based on the assumption that the updated view should coexist with the starting data, since we are in the context of update as *schema augmentation*. If an object changes its value, then its object identifier should change, too. Notice that, for this reason, $\mathcal{M}_{67}$ leads to an inconsistent version if `Number` and `String` are defined to be non-empty disjoint classes. Since the only elementary change that can refer to *new* objects is Add-class, in order to specify a schema change involving a restructuring of the data and the creation of new objects – like in the case of the change of the type of an attribute with an incompatible new type – a sequence of Drop-class and Add-class should be specified, together with a data conversion view specifying how the data is converted from one version to the other.

The deletion of the class `Employee` in $\mathcal{SV}_8$ does not cause any inconsistency in the resulting schema version. In $\mathcal{SV}_8$ the `Employee` extension is empty and the former `Employee` subclasses continue to exist (with the constraint that their extensions are subsets of the extension of `Employee` in $\mathcal{SV}_6$). Notice that, in a classical object model where the class hierarchy is explicitly based on a DAG, the deletion of a non-isolated class would require a restructuring of the DAG itself (e.g. to get rid of dangling edges).

5 Reasoning using Description Logics

In this section we establish a relationship between the proposed model for evolving schemata and the $\mathcal{ALCQI}$ description logic. To this end, we provide an encoding from an evolving schema into an $\mathcal{ALCQI}$ knowledge base Σ, such that the reasoning problems mentioned in the previous section can be reduced to corresponding description logics reasoning problems, for which extensive theories and well founded and efficient implemented systems exist. The encoding is grounded on the fact that there is a correspondence between the models of the knowledge base and the legal instances of the evolving schema.

We give here only a very brief introduction to the $\mathcal{ALCQI}$ description logic; for a full account, see, e.g., [6]. The basic types of a description logic are *concepts* and *roles*. The syntax rules at the left hand side of Figure 3 define valid concept and role expressions. Concepts are interpreted as sets of individuals—as for unary predicates—and roles as sets of pairs of individuals—as for binary predicates. Formally, an *interpretation* is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consisting of a set $\Delta^{\mathcal{I}}$ of individuals (the *domain* of $\mathcal{I}$) and a function $\cdot^{\mathcal{I}}$ (the *interpretation function* of $\mathcal{I}$) mapping every concept to a subset of $\Delta^{\mathcal{I}}$ and every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, such that the equations at the right hand side of Figure 3 are satisfied.

A *knowledge base* is a finite set Σ of axioms of the form $C \dot{\sqsubseteq} D$, involving concept expressions C, D; we write $C \equiv D$ as a shortcut for both $C \dot{\sqsubseteq} D$ and $D \dot{\sqsubseteq} C$. An interpretation $\mathcal{I}$ satisfies $C \dot{\sqsubseteq} D$ if and only if the interpretation of C is included in the interpretation of D, i.e., $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; it is said that C is subsumed by D. An interpretation $\mathcal{I}$ is a *model* of a knowledge base Σ iff every axiom of Σ is satisfied by $\mathcal{I}$. If Σ has a model, then it is *satisfiable*. Σ *logically implies* an axiom $C \dot{\sqsubseteq} D$ (written $\Sigma \models C \dot{\sqsubseteq} D$) if $C \dot{\sqsubseteq} D$ is satisfied by every model of Σ. Reasoning in $\mathcal{ALCQI}$ (i.e., deciding knowledge base satisfiability and logical implication) is decidable, and it has been proven to be an EXPTIME-complete problem [6].

$$
\begin{array}{lll}
C, D \rightarrow & A \mid & \\
& \top \mid & \top^{\mathcal{I}} = \Delta^{\mathcal{I}} \\
& \bot \mid & \bot^{\mathcal{I}} = \emptyset \\
& \neg C \mid & (\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
& C \sqcap D \mid & (C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
& C \sqcup D \mid & (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
& \forall R.C \mid & (\forall R.C)^{\mathcal{I}} = \{i \in \Delta^{\mathcal{I}} \mid \forall j.\ R^{\mathcal{I}}(i,j) \Rightarrow C^{\mathcal{I}}(j)\} \\
& \exists R.C \mid & (\exists R.C)^{\mathcal{I}} = \{i \in \Delta^{\mathcal{I}} \mid \exists j.\ R^{\mathcal{I}}(i,j) \wedge C^{\mathcal{I}}(j)\} \\
& \geq nR.C \mid & (\geq nR.C)^{\mathcal{I}} = \{i \in \Delta^{\mathcal{I}} \mid \sharp\{j \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(i,j) \wedge C^{\mathcal{I}}(j)\} \geq n\} \\
& \leq nR.C & (\leq nR.C)^{\mathcal{I}} = \{i \in \Delta^{\mathcal{I}} \mid \sharp\{j \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(i,j) \wedge C^{\mathcal{I}}(j)\} \leq n\} \\
 & & \\
R, S \rightarrow & P \mid & \\
& R^{-} & (R^{-})^{\mathcal{I}} = \{(i,j) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(j,i)\}
\end{array}
$$

Fig. 3. $\mathcal{ALCQI}$ concept and role expressions and their semantics.

As in [7], the encoding of an object-oriented schema in an $\mathcal{ALCQI}$ knowledge base is based on the *reification* of type expressions – i.e., explicit individuals exist to denote values of complex types. We introduce the concept $\mathsf{AbstractClass}$ to represent the classes, the concepts $\mathsf{RecType}$, $\mathsf{SetType}$ to represent types, the role value to model the association between classes and types, and the role member to specify the type of the elements of a set. In particular, a record is represented as an individual connected by means of (functional) roles – corresponding to attributes – to the fillers of its attributes. The mapping function ψ_i translates type expressions into $\mathcal{ALCQI}$ concepts as follows:

$$
\begin{aligned}
\psi_i(\mathrm{C}) &= C_i \\
\psi_i(\underline{\mathrm{Union}}\ T_1, \ldots, T_k\ \underline{\mathrm{End}}) &= \psi_i(T_1) \sqcup \ldots \sqcup \psi_i(T_k) \\
\psi_i(\underline{\mathrm{Set\text{-}of}}\ [\mathrm{m,n}]\ T) &= \mathsf{SetType} \sqcap \forall \mathsf{member}.\psi_i(T) \sqcap \\
&\quad\ \geq m \mathsf{member}.\top \sqcap \leq n \mathsf{member}.\top \\
\psi_i(\underline{\mathrm{Record}}\ A_1{:}T_1, \ldots, A_k{:}T_k\ \underline{\mathrm{End}}) &= \mathsf{RecType} \sqcap \exists A_1.\psi_i(T_1) \sqcap \ldots \sqcap \exists A_k.\psi_i(T_k)
\end{aligned}
$$

The translation function ψ_i is contextualised to the ith schema version, since a class in different schema version may have different extensions, and it is mapped into distinct concepts.

Definition 4. *The $\mathcal{ALCQI}$ knowledge base $\Sigma = \psi(\mathcal{S})$ corresponding to the object-oriented evolving schema $\mathcal{S} = (\mathcal{C_S}, \mathcal{A_S}, \mathcal{SV}_0, \mathcal{M_S})$ is composed by the following axioms:*

- *Axioms on basic types:*
  $\mathsf{AbstractClass} \sqsubseteq \exists \mathsf{value}.\top \sqcap \leq 1 \mathsf{value}.\top$
  $\mathsf{RecType} \sqsubseteq \forall \mathsf{value}.\bot$
  $\mathsf{SetType} \sqsubseteq \forall \mathsf{value}.\bot \sqcap \neg \mathsf{RecType}$
- *For each class definition*
  Class C is-a $C_1, \ldots, C_h$ disjoint $C_{h+1}, \ldots, C_k$ type-is T in $\mathcal{SV}_0$:
  $\psi_0(C) \sqsubseteq \mathsf{AbstractClass} \sqcap \psi_0(C_1) \sqcap \ldots \sqcap \psi_0(C_h) \sqcap \forall \mathsf{value}.\psi_0(T)$
  $\psi_0(C) \sqsubseteq \neg\psi_0(C_{h+1}) \sqcap \ldots \sqcap \neg\psi_0(C_k)$

Add-attribute $\mathbf{C}$, $\mathbf{A}$, $\mathbf{T}$	$\psi_j(\mathbf{C}) \equiv \psi_i(\mathbf{C}) \sqcap \forall \mathsf{value}.(\mathsf{RecType} \sqcap \exists \mathbf{A}.\psi_j(\mathbf{T}))$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}$
Drop-attribute $\mathbf{C}$, $\mathbf{A}$	$\psi_i(\mathbf{C}) \equiv \psi_j(\mathbf{C}) \sqcap \forall \mathsf{value}.(\mathsf{RecType} \sqcap \exists \mathbf{A}.\top)$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}$
Change-attr-name $\mathbf{C}$, $\mathbf{A}$, $\mathbf{A}'$	$\psi_i(\mathbf{C}) \sqcap \forall \mathsf{value}.(\mathsf{RecType} \sqcap \exists \mathbf{A}.\top) \equiv$ $\psi_j(\mathbf{C}) \sqcap \forall \mathsf{value}.(\mathsf{RecType} \sqcap \exists \mathbf{A}'.\top)$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}$
Change-attr-type $\mathbf{C}$, $\mathbf{A}$, $\mathbf{T}'$	$\psi_i(\mathbf{C}) \sqcap \forall \mathsf{value}.(\mathsf{RecType} \sqcap \exists \mathbf{A}.\psi_j(\mathbf{T}')) \equiv$ $\psi_j(\mathbf{C}) \sqcap \forall \mathsf{value}.(\mathsf{RecType} \sqcap \exists \mathbf{A}.\top)$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}$
Add-class $\mathbf{C}$, $\mathbf{T}$	$\psi_i(\mathbf{C}) \equiv \bot$, $\psi_j(\mathbf{C}) \sqsubseteq \mathsf{AbstractClass} \sqcap \forall \mathsf{value}.\psi_j(\mathbf{T})$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}$
Drop-class $\mathbf{C}$	$\psi_j(\mathbf{C}) \equiv \bot$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}$
Change-class-name $\mathbf{C}$, $\mathbf{C}'$	$\psi_i(\mathbf{C}) \equiv \psi_j(\mathbf{C}')$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}, \mathbf{C}'$
Change-class-type $\mathbf{C}$, $\mathbf{T}'$	$\psi_j(\mathbf{C}) \equiv \psi_i(\mathbf{C}) \sqcap \forall \mathsf{value}.\psi_j(\mathbf{T}')$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}$
Add-is-a $\mathbf{C}$, $\mathbf{C}'$	$\psi_j(\mathbf{C}) \equiv \psi_i(\mathbf{C}) \sqcap \psi_i(\mathbf{C}')$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}$
Drop-is-a $\mathbf{C}$, $\mathbf{C}'$	$\psi_i(\mathbf{C}) \equiv \psi_j(\mathbf{C}) \sqcap \psi_j(\mathbf{C}')$, $\psi_i(D) \equiv \psi_j(D)$ for all $D \neq \mathbf{C}$

Fig. 4. The axioms induced by the schema changes.

- *For each view definition*
  View-class C is-a $C_1, \ldots, C_h$ disjoint $C_{h+1}, \ldots, C_k$ type-is T in $\mathcal{SV}_0$:
  $\psi_0(C) \sqsubseteq \mathsf{AbstractClass} \sqcap \psi_0(C_1) \sqcap \ldots \sqcap \psi_0(C_h)$
  $\psi_0(C) \sqsubseteq \neg\psi_0(C_{h+1}) \sqcap \ldots \sqcap \neg\psi_0(C_k)$
  $\psi_0(C) \equiv \forall \mathsf{value}.\psi_0(T)$
- *For each attribute in $\mathcal{A}_\mathcal{S}$:*
  $\exists A_i.\top \sqsubseteq \leq 1 A_i.\top$
- *For each schema modification $\mathcal{M}_{ij} \in \mathcal{M}_\mathcal{S}$ a corresponding axiom from Tab. 4.*

Based on the results of [8], we have proved in [12] that the encoding is correct, in the sense that there is a correspondence between the models of the knowledge base and the legal instances of the evolving schema. The semantic correspondence is exploited to devise a correspondence between reasoning problems at the level of evolving schemata and reasoning problems at the level of the description logic.

Theorem 1. *Given an evolving schema $\mathcal{S}$, the reasoning problems defined in the previous section are all decidable in EXPTIME with a PSPACE lower bound. The reasoning problems can be reduced to corresponding satisfiability problems in the $\mathcal{ALCQI}$ Description Logic.*

Please note that the worst case complexity between PSPACE and EXPTIME does not imply bad practical computational behaviour in the real cases: in fact, a preliminary experimentation with the Description Logic system FaCT [16] shows that reasoning problems in realistic scenarios of evolving schemata are solved very efficiently.

As a final remark, it should be noted that the high expressiveness of the Description Logic constructs can capture an extended version of the presented object-oriented model, at no extra cost with respect to the computational complexity, since the target Description Logic in which the problem is encoded does not change. This includes not only taxonomic relationships, but also arbitrary boolean constructs, inverse attributes, n-ary relationships, and a large class of integrity constraints expressed by means of $\mathcal{ALCQI}$ inclusion dependencies [7]. The last point suggests that axioms modeling schema changes can be freely combined in order to transform a schema in a new one. Some combination can be defined at database level by introducing new non-elementary primitives.

6 Discussion

In this paper we have introduced an approach to schema versioning which considers a (conceptual) schema change as a (logical) schema augmentation, in the sense of [17]. In fact, the sequence of schema versions can be seen as an increasing set of constraints, as defined in Table 1; every elementary schema change introduces new constraints over a vocabulary augmented by the classes for the new version. An update of the schema is also reflected by the introduction of materialised views at the level of the data which specify how to populate the classes of the new version from the data of the previous version. Formally, in our approach the materialised views coexist together with the base data in the same pool of data. In some sense, there is no proper evolution of the objects themselves, since the emphasis is given to the evolution of the schema.

More complex is the case when it is needed that a particular object maintains its identity over different version – i.e., the object evolves by varying its structural properties – and it is requested to have an overview of its evolution over the various versions. This is the case when a query – possibly over more than one conceptual schema – requires an answer about an object from more than one version.

In this case an explicit treatment of the partial order over the schema versions induced by the schema changes is required at the level of the semantics. Formally, this partial order defines some sort of "temporal structure" which leads us to consider the evolving data as a (formal) temporal database with a temporally extended conceptual data model [15, 2]. With such an approach, different formal "timestamps" can be associated with different schema versions: all the objects connected with a schema version are assigned the same timestamp, such that each data pool represents a homogeneous state (snapshot) in the database evolution along the formal time axis[1]. Objects belonging to different versions can be distinguished by means of the object's OID and the timestamp.

[1] This case corresponds to the multi-pool solution for temporal schema versioning of snapshot data in the [10] taxonomy.

In such a framework, the (materialised) views expressing the data conversions can be expressed as temporal queries. In some sense, we can say that such a query language operates in a *schema translation* fashion [9] instead of a schema augmentation, where new data are presumed to be independent of the source data and an explicit mapping between them has to be maintained. Multischema queries can be seen as temporal queries involving in their formulation distinct (formal) timestamps. Moreover, in case (bi)temporal schema versioning is adopted, this "formal" temporal dimension has also interesting and non-trivial connections, which deserve further investigation, with the "real" temporal dimension(s) used for versioning.

7 Conclusions

This paper deals with the support of database schema evolution and versioning by introducing a general framework based on a semantic approach. The reducibility of a general Object- Oriented conceptual model to the proposed framework made it possible to provide a sound foundation for the purposes stated in the Introduction. In particular, the adoption of a Description Logic for the framework specification implies the availability of powerful services (like consistency checking and classification) which can be proved decidable.

References

1. S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45(5):798–842, 1998. A first version appeared in SIGMOD'89.
2. Alessandro Artale and Enrico Franconi. Temporal ER modeling with description logics. In *Proc. of the International Conference on Conceptual Modeling (ER'99)*. Springer-Verlag, November 1999.
3. J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM-SIGMOD Annual Conference*, pages 311–322, May 1987.
4. S. Bergamaschi and B. Nebel. Automatic Building and Validation of Multiple Inheritance Complex Object Database Schemata. *International Journal of Applied Intelligence*, 4(2):185–204, 1994.
5. P. Brèche. Advanced Principles of Changing Schema of Object Databases. In *Proc. of the 8th Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*, pages 476–495, May 1996.
6. D. Calvanese, G. De Giacomo, M. Lenzerini, and D. Nardi. Reasoning in expressive description logics. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2000. To appear.
7. D. Calvanese, M. Lenzerini, and D. Nardi. Description logics for conceptual data modeling. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 229–263. Kluwer, 1998.
8. D. Calvanese, M. Lenzerini, and D. Nardi. Unifying class-based representation formalisms. *Journal of Artificial Intelligence Research*, 11:199–240, 1999.
9. Ti-Pin Chang and Richard Hull. Using witness generators to support bi-directional update between object-based databases. In *Proc. of the 1995 ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'95)*, 1995.

10. C. De Castro, F. Grandi, and M. R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.
11. F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and Database Evolution in the O_2 Object Database System. In *Proc. of the 21st Int'l Conf. on Very Large Databases (VLDB)*, pages 170–181, September 1995.
12. Enrico Franconi, Fabio Grandi, and Federica Mandreoli. A semantic approach for schema evolution and versioning in object-oriented databases. Technical report, Department of Computer Science, University of Manchester, UK, 2000.
13. F. Grandi and F. Mandreoli. ODMG Language Extensions for Generalized Schema Versioning Support. In *Proc. of ECDM'99 Workshop (in conj. with ER)*, November 1999.
14. F. Grandi, F. Mandreoli, and M. R. Scalas. A Generalized Modeling Framework for Schema Versioning Support. In *Proc. of 11th Australasian Database Conference (ADC 2000)*, January 2000.
15. H. Gregersen and C. S. Jensen. Temporal Entity-Relationship Models - A Survey. *IEEE Transaction on Knowledge and Data Engineering*, 11(3):464–497, 1999.
16. I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In *Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, pages 161–180, 1999.
17. Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proc. of the 16th VLDB Conference*, 1990.
18. C. S. Jensen, J. Clifford, S. K. Gadia, P. Hayes, and S. Jajodia et al. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases - Research and Practice*, pages 367–405. Springer-Verlag, 1998. LNCS No. 1399.
19. J-B Lagorce, A. Stockus, and E. Waller. Object-oriented database evolution. In *Proc. of ICDT'97*, 1997.
20. S.-E. Lautemann. A Propagation Mechanism for Populated Schema Versions. In *Proc. of the 13th International Conference on Data Engineering (ICDE)*, pages 67–78, April 1997.
21. D. J. Penney and J. Stein. Class Modification in the GemStone object-oriented DBMS. In *Proc. of the Int'l Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 111–117, December 1987.
22. R. J. Peters and M. T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transaction on Database Systems*, 22(1):75–114, 1997.
23. J. F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1996.