A Formal Model for Temporal Schema Versioning in Object-Oriented Databases

Fabio Grandi [a,*], Federica Mandreoli [b]

[a] *CSITE-CNR and Dipartimento di Elettronica, Informatica e Sistemistica, Alma Mater Studiorum - Università di Bologna, Viale Risorgimento 2, I-40136, Bologna, Italy*

[b] *Dipartimento di Ingegneria dell'Informazione, Università di Modena e Reggio Emilia, Via Vignolese 905, I-41100, Modena, Italy*

Abstract

In this paper we present a formal model for the support of temporal schema versions in object-oriented databases. Its definition is partially based on a generic (ODMG compatible) object model and partially introduces new concepts. The proposed model supports all the schema changes which are usually considered in the OODB literature, for which an operational semantics and a formal analysis of their correct behaviour is provided. Semantic issues arising from the introduction of temporal schema versioning in a conventional or temporal database (concerning the interaction between the intensional and extensional levels of versioning and the management of data in the presence of multiple schema versions) are also considered.

Key words: schema versioning, schema evolution, temporal versioning, temporal database

1 Introduction

Databases usually embody the core of costly, strategic and long-lived information systems. However careful and accurate the initial design may have been, a database schema is likely to undergo changes and revisions after implementation. In order to avoid the loss of data after schema changes, many object-oriented systems like O_2 [1], ORION (ITASCA) [2], and GemStone [3] support **schema evolution**, which provides (partial) automatic recovery of the extant data by adapting them to the new schema. However, if only the updated schema is retained, all the applications

* Corresponding author. Tel.: +39-051-2093555, fax: +39-051-2093540.
Email addresses: fgrandi@deis.unibo.it (Fabio Grandi), mandreoli.federica@unimo.it (Federica Mandreoli).

compiled with the past schema may cease to work. In order to let applications work on multiple schemata, schema evolution is not sufficient and the maintenance of more than one schema is required. This leads to the notion of **schema versioning** and **schema version** which is the persistent outcome of the application of schema modifications.

In OODBMSs used in design environments, schema versioning was introduced to support different users/teams concurrently working on parallel schema versions [4–6]. In this framework, schema versions are hierarchically organized as a DAG, where version derivation lines can be branching (and even merging) and no temporal aspects are considered at all.

More recently, the adoption of object-oriented models has become a common choice in novel application domains, like GIS and spatio-temporal databases, biomedical and multimedia databases, where temporal requirements play a great role (e.g. [7–10]). In this context, if the adoption of a temporal database allows one to represent and manage the history of data objects (extensional properties), the introduction of **temporal schema versioning** enables one to represent and manage the history of the *structure* of data objects (intensional properties). Whereas a great deal of research work was done on temporal OODBs [11–15], temporal schema versioning has been deeply investigated so far only for the relational model [16,17].

In this paper, we deal with the introduction of temporal schema versioning in an object-oriented database, also taking into account formal aspects. Within the object-oriented framework, theoretical work has been done in the field of programming languages [18,19] and databases [20–22], also including temporal ones [11], whereas thorough studies concerning schema versioning are still lacking. To this purpose, we will define $OODM|_{\mathcal{SV}}$ (Object-Oriented Data Model contextualized to $\mathcal{S}$chema $\mathcal{V}$ersions), that is a formal model for the management of *temporal schema versioning* in object-oriented databases. $OODM|_{\mathcal{SV}}$ focuses on the operational aspects of schema versioning in order to provide users with a provably correct evolving schema whose data can then be manipulated through different schema versions. In general, the schema change support implies the adoption of statements for schema modification defined by means of primitives (LLPs - Low Level Primitives) acting on "atomic" elements of the underlying data model [2]. Starting from a general object-oriented model, we will first formally define the semantics at schema and data level of a well-understood taxonomy of schema changes. Then, we will study the impact of schema changes on an evolving schema supporting one or two temporal dimensions and on the underlying database. As to the underlying database, we will show how in our model an evolving schema can interact with a conventional database (denoted as *snapshot* database in the temporal context) as well as with a temporal database with valid and/or transaction time support.

The proposed formalization is aimed at guaranteeing a well-founded implementation of temporal schema versioning in an object-oriented database system. This is

achieved by characterizing two issues related to the temporal schema versioning problem, for which the proposed solutions constitute the main novel contributions of this work. Such issues are the formal correctness of the schema transformation process (and of the resulting evolving schema) and the possibility of accessing data through different schema versions.

- As far as the former issue is concerned, its purpose is to ensure high availability and quality of the information in a system in operation which is subject to schema changes. We will show that the way the operational semantics for schema changes is defined ensures the correctness of the schema modification process and, in general, of the interaction between an evolving schema and the underlying database.
- As far as the latter issue is concerned, since multiple schema versions can coexist on top of a database, the usefulness of accessing data through schema versions different from the current one becomes apparent (e.g. for the reuse of applications compiled with previous schema versions). $OODM|_{\mathcal{SV}}$ ensures full reversibility at schema level of the update process in order to allow data instances to be queried through schema versions different from their own (presumably defining the format in which their are currently stored).

The rest of the paper is organized as follows. Section 2 introduces the basic $OODM|_{\mathcal{SV}}$ definitions of (well-formed) schema version and (legal) database instance. Section 3 presents the semantics of schema changes considering their "local" effects, at schema and data instance level. Section 4 analyzes the interaction between intensional and extensional versioning, when temporal schema versioning has to be supported on a possibly temporal database. The notions of evolving schema and legal database are defined to behave as mutually consistent collections of schema versions and database instances, respectively. The "global" effects of schema changes in $OODM|_{\mathcal{SV}}$ are described in Sec. 5 by formalizing their action at evolving schema and database level. Section 6 is devoted to a brief review of related works and to the discussion of the $OODM|_{\mathcal{SV}}$ approach in such a context. Conclusions can finally be found in Sec. 7.

2 Basic Definitions: Schema Versions and Database Instance

$OODM|_{\mathcal{SV}}$ is a temporal schema versioning model based on a generic object-oriented data model, first introduced in [20], which is general enough to represent the static parts of UML and ODMG [1] . In the following, starting from the definition proposed in [20,24], we formally define the $OODM|_{\mathcal{SV}}$ basic elements.

[1] For the sake of simplicity, we do not take into account in this paper aspects related to the definition of methods. The $OODM|_{\mathcal{SV}}$ model could be extended with methods as shown in [23].

2.1 Types, Subtyping, and Schema Versions

In $OODM|_{\mathcal{SV}}$ a database represents the structural evolution of an enterprise through the temporal versioning of its schema. More properly, an *evolving schema* consists in a collection of schema versions. It is based on a set of class names $\mathcal{CN}$ and each schema version includes definitions for some class names belonging to $\mathcal{CN}$. We assume a class name version $\mathcal{CNV} \subseteq \mathcal{CN}$ to be the set of class names for a specific schema version. Distinct class name versions can also overlap, as happens when the same class is defined in different schema versions. In particular, any schema version specifies the type corresponding to each class in $\mathcal{CNV}$. The *contextualization* of types with respect to a schema version denotes the $OODM|_{\mathcal{SV}}$ types defined on the class names available in $\mathcal{CNV}$. The family of types $\mathcal{T}$ contextualized to $\mathcal{CNV}$, $\mathcal{T}|_{\mathcal{CNV}}$, is defined so that legal $\mathcal{T}|_{\mathcal{CNV}}$ types are:

- the literal types in $\mathcal{LT}$, that is *integer*, *float*, *boolean*, *char*, *string*;
- the class names in a class name version $\mathcal{CNV}$;
- the special type **any**;
- the set $\{\tau\}$, the bag $\{\!|\tau|\!\}$, and the list type $[\tau]$, if $\tau \in \mathcal{T}|_{\mathcal{CNV}}$;
- the record type $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$,
  if $A_1, \ldots, A_n$ are attribute names and $\tau_1, \ldots, \tau_n \in \mathcal{T}|_{\mathcal{CNV}}$.

Since we only consider the static part of schema definitions, a schema version only defines an inheritance hierarchy among classes, as it introduces the specifications (types) associated with classes and the inheritance relationships valid at schema version level. More formally a *schema version* $\mathcal{SV}$ is a tuple $(\mathcal{CNV}, \sigma, \preceq)$ made up of a class name version $\mathcal{CNV}$, a function $\sigma : \mathcal{CNV} \rightarrow \mathcal{T}|_{\mathcal{CNV}}$, associating class names with their types in $\mathcal{T}|_{\mathcal{CNV}}$, and a set of inheritance relationships $\preceq \subseteq \mathcal{CNV} \times \mathcal{CNV}$, which is is a *partial order* on $\mathcal{CNV}$.

In a schema version, the type associated with each class must be a refinement of the types of all its superclasses. To represent this notion, starting from the class hierarchy defined by the inheritance relation $\preceq$, we can introduce a subtyping relation ($\leq$) that specifies when one type refines another.

Definition 1 (Subtyping Relation) *Let $(\mathcal{CNV}, \sigma, \preceq)$ be a schema version. The subtyping relation $\leq \subseteq \mathcal{T}|_{\mathcal{CNV}} \times \mathcal{T}|_{\mathcal{CNV}}$ is the smallest partial order over $\mathcal{T}|_{\mathcal{CNV}}$ satisfying the following conditions:*

(1) $\forall C, C' \in \mathcal{CNV}$*: if* $C \preceq C'$ *then* $C \leq C'$*;*
(2) if $\tau_k, \tau'_k \in \mathcal{T}|_{\mathcal{CNV}}$*,* $\tau_k \leq \tau'_k$*, for each* $k \in [1, n]$ *and* $n \leq m$*, then*
$\langle A_1 : \tau_1, \ldots, A_n : \tau_n, \ldots, A_m : \tau_m \rangle \leq \langle A_1 : \tau'_1, \ldots, A_n : \tau'_n \rangle$*;*
(3) if $\tau, \tau' \in \mathcal{T}|_{\mathcal{CNV}}$*,* $\tau \leq \tau'$*, then* $\{\tau\} \leq \{\tau'\}$*,* $\{\!|\tau|\!\} \leq \{\!|\tau'|\!\}$*,* $[\tau] \leq [\tau']$*;*
(4) $\forall \tau \in \mathcal{T}|_{\mathcal{CNV}}$*:* $\tau \leq$ **any** *(i.e.* **any** *is the top of the type hierarchy).*

Given a subtyping relation $\leq$, let

- *$<$ be its corresponding strict partial order, that is $\tau < \tau'$ iff $\tau \leq \tau' \wedge \tau \neq \tau'$;*
- *$\leq_\tau = \{\tau' | \tau' \leq \tau\}$ be all τ direct and indirect subtypes (also $<_\tau$ can be defined in a similar way);*
- *$\geq_\tau = \{\tau' | \tau \leq \tau'\}$ be all τ direct and indirect supertypes (also $>_\tau$ can be defined in a similar way).*

Definition 2 (Well-formedness) *A schema version $(\mathcal{CNV}, \sigma, \preceq)$ is* well-formed *if for each pair C, C' of class names, $C \preceq C'$ implies $\sigma(C) \leq \sigma(C')$.*

Example 1 *Let $\mathcal{CN} = \{$employee, professor, course, activity$\}$. We introduce the well-formed schema version $\mathcal{SV}_1 = (\mathcal{CNV}_1, \sigma_1, \preceq_1)$, where:*

$$\mathcal{CNV}_1 = \{employee, professor\}$$
$$\sigma_1 : employee \mapsto \langle name : string, ssn : integer\rangle$$
$$\sigma_1 : professor \mapsto \langle name : string, ssn : integer, deg : \{string\}\rangle$$
$$professor \preceq_1 employee$$

which models employees with a name and a social security number, and professors as employees with one or more degrees.

Now we consider a second schema version, which could be derived from the previous one with the addition of two new classes: activity*, which defines activities with attributes* name *and* has_prereq *denoting prerequisites), and* course*, which is defined as a subclass of* activity *with the refinement of* has_prereq*. The corresponding well-formed schema version $\mathcal{SV}_2 = (\mathcal{CNV}_2, \sigma_2, \preceq_2)$ could be:*

$$\mathcal{CNV}_2 = \{employee, professor, activity, course\}$$
$$\sigma_2 : employee \mapsto \langle name : string, ssn : integer\rangle$$
$$\sigma_2 : professor \mapsto \langle name : string, ssn : integer, deg : \{string\}\rangle$$
$$\sigma_2 : activity \mapsto \langle name : string, has\_prereq : \{activity\}\rangle$$
$$\sigma_2 : course \mapsto \langle name : string, has\_prereq : \{course\}\rangle$$
$$professor \preceq_2 employee,\ course \preceq_2 activity$$

With respect to such a schema version, the following subtyping relationships hold:

$$professor \leq employee, \{course\} \leq \{activity\}$$
$$\langle name : string, taught\_by : professor, has\_prereq : \{course\}\rangle \leq \langle name : string, has\_prereq : \{activity\}\rangle$$

2.2 Objects, Values and Instances

Usually, a schema defines a set of constraints which a database must satisfy in order to be legal. In particular an instance assigns object identifiers to classes, and values

to object identifiers. The set $\mathcal{V}$ of $OODM|_{\mathcal{SV}}$ values is based on a set of object identifiers $\mathcal{OI}$ and contains:

- each element in $\mathcal{OI}$, each integer, float, boolean, char and string value;
- each record $\langle A_1 : v_1, \ldots, A_n : v_n \rangle$, set $\{v_1, \ldots, v_n\}$, bag $\{\!|v_1, \ldots, v_n|\!\}$ and list value $[v_1, \ldots, v_n]$, where $v_1, \ldots, v_n$ are values in $\mathcal{V}$ and A_1,...,A_n are attribute names.

Given a set $\mathcal{OI}$ of object identifiers and a set $\mathcal{CN}$ of class names, we define an *instance* $\mathcal{I}$ as a tuple (π, ν) where:

- $\pi : \mathcal{CN} \rightarrow 2^{\mathcal{OI}}$ is an OID assignment, mapping class names to disjoint finite sets of OIDs;
- $\nu : \mathcal{OI} \rightarrow \mathcal{V}$ is a value assignment, mapping OIDs to values.

Given the OID assignment π and a schema version $(\mathcal{CNV}, \sigma, \preceq)$, the proper extension [2] of $C \in \mathcal{CNV}$ is $\pi(C)$, and the extension of C is the set of OIDs which are instances of C and all its subclasses, that is $\pi^*(C) = \bigcup_{C' \in \mathcal{CNV}, C' \preceq C} \pi(C')$, where $\pi^*(C') \subseteq \pi^*(C)$ whenever $C' \preceq C$ (an object of a class C' may also be viewed as an object of a superclass C of C' [24]).

Moreover, values can be associated with types. The semantics of types can be defined as the association between each type and the set of legal values for that type. We assume that the usual interpretation is associated to each literal type belonging to $\mathcal{LT}$ (e.g. the domain of the literal type *integer* is the set $\mathbb{Z}$ of integer numbers and the domain of *void* is the empty set).

Definition 3 (Type Legal Values) *Let π be an OID-assignment, $(\mathcal{CNV}, \sigma, \preceq)$ a schema version and $OID = \{\pi(C) \mid C \in \mathcal{CNV}\}$. The legal extension for each $\tau \in \mathcal{T}|_{\mathcal{CNV}}$, denoted as $dom(\tau)$, is given by:*

(1) $\forall \tau \in \mathcal{T}|_{\mathcal{CNV}} : null \in dom\,(\tau)$
(2) $\forall \tau \in \mathcal{LT} : dom\,(\tau)$ *is the usual interpretation of that type;*
(3) $\forall \tau \in \mathcal{CNV} : dom(\tau) = \pi^*(\tau) \cup \{null\}$*;*
(4) $\forall \tau \in \mathcal{T}|_{\mathcal{CNV}} : dom(\{\tau\}) = \{\{v_1, \ldots, v_n\} \mid n \geq 0, v_i \in \tau,$ *for* $i \in [1, n]\}$*;*
(5) $\forall \tau \in \mathcal{T}|_{\mathcal{CNV}} : dom(\{\!|\tau|\!\}) = \{\{\!|v_1, \ldots, v_n|\!\} \mid n \geq 0, v_i \in \tau,$ *for* $i \in [1, n]\}$ [3] *;*
(6) $\forall \tau \in \mathcal{T}|_{\mathcal{CNV}} : dom([\tau]) = \{[v_1, \ldots v_n] \mid n \geq 0, v_i \in dom(\tau),$ *for* $i \in [1, n]\}$*;*
(7) $\forall \{\tau_1, \ldots, \tau_n\} \in 2^{\mathcal{T}|_{\mathcal{CNV}}} : \forall n > 0 :\ dom(\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle)$
$= \{\langle A_1 : v_1, \ldots, A_n : v_n \rangle \mid v_i \in dom(\tau_i),$ *for* $i \in [1, n]\}$*;*
(8) $dom(\textbf{any}) = \bigcup_{\tau \in \mathcal{T}|_{\mathcal{CNV}}} dom(\tau)$.

[2] Notice that the extension is always defined for every class name in every schema version, as π is a total function defined on $\mathcal{CN}$. If a class C has an empty extension, $\pi(C) = \emptyset$.
[3] With respect to the set type domain, the bag type domain admits repeated elements.

Given a schema version $(\mathcal{CNV}, \sigma, \preceq)$, the set of $OODM|_{\mathcal{SV}}$ legal values contextualized to $\mathcal{CNV}$ is $\mathcal{V}|_{\mathcal{CNV}} = \bigcup_{\tau \in \mathcal{T}|_{\mathcal{CNV}}} dom(\tau)$.

In the following, we introduce the notion of legal instance for a schema version, which will represent the basic element for the definition of the legal database component of an evolving schema.

Definition 4 (Legal Instance) *An instance $\mathcal{I} = (\pi, \nu)$ is* legal *for a schema version $(\mathcal{CNV}, \sigma, \preceq)$ if it satisfies all the constraints implied by the schema version definition: given the OID assignment π, the value associated with each object by means of ν must be legal for the type $\tau|_{\mathcal{CNV}}$ of the class of which the object is instance (i.e. $\forall C \in \mathcal{CNV} : \forall o \in \pi(C) : \nu(o) \in dom(\sigma(C))$).*

Notice that an instance can be legal for more than one schema versions.

Example 2 *Let $\mathcal{OI} = \{oi_1, oi_2, \ldots, oi_{10}\}$. Given the OID assignment:*

$$\pi : professor \mapsto \{oi_1, oi_2\}$$
$$\pi : employee \mapsto \{oi_3\}$$
$$\pi : course \mapsto \{oi_4, oi_7\}$$
$$\pi : activity \mapsto \{oi_6\}$$

and the schema versions of Ex. 1, the following are legal values for types belonging to $\mathcal{T}|_{\mathcal{CNV}_1}$ and $\mathcal{T}|_{\mathcal{CNV}_2}$:

$$\text{“}Brown\text{”} \in dom(string);$$
$$oi_2 \in dom(professor);$$
$$\langle name : \text{“}Brown\text{”}, tutor : oi_1 \rangle \in dom(\langle name : string, tutor : professor \rangle)$$

The following are instead legal values only for types belonging to $\mathcal{T}|_{\mathcal{CNV}_2}$:

$$oi_4 \in dom(course);$$
$$\langle name : \text{“}electronics\text{”}, has\_prereq : \{oi_4\} \rangle \in dom(\langle name : string, has\_prereq : \{course\} \rangle).$$

Given the following value assignment:

$$\nu : oi_1 \mapsto \langle name : \text{“}Smith\text{”}, ssn : 101, deg : \{\text{“}MSc\text{”}\} \rangle$$
$$\nu : oi_2 \mapsto \langle name : \text{“}Jones\text{”}, ssn : 237, deg : \{\text{“}MSc\text{”}, \text{“}PhD\text{”}\} \rangle$$
$$\nu : oi_3 \mapsto \langle name : \text{“}Ford\text{”}, ssn : 154 \rangle$$

the tuple (π, ν) is a legal instance for $\mathcal{SV}_1$.

Category	Primitive Change	Meaning
Changes to a class type	**AddAttribute**	Add an attribute to a record type
	DeleteAttribute	Delete an attribute from a record type
	ChangeAttrName	Change the name of an attribute
	ChangeAttrType	Change the type of an attribute
	ChangeClassType	Change the type of a class
Changes to the class collection	**AddSuperclass**	Make an existing class a superclass
	DeleteSuperclass	Delete a class from the superclasses
	AddClass	Add a new empty isolated class
	DeleteClass	Delete an isolated class
	ChangeClassName	Change the name of a class

Table 1
List of primitive schema changes.

3 Action of Schema Changes at Schema and Instance Level (Local Effects)

$OODM|_{\mathcal{SV}}$ provides a complete collection of *primitive changes* [2] applicable to a schema. They are listed in Tab. 1 where they are classified in two categories: *Changes to a class type* and *Changes to the class collection*. The application of the supported schema changes represents the only way of adding new schema versions starting from existing ones. In other words, each schema change when applied to an existing schema version, leads to the creation of a new schema version. In most cases, such changes have to be propagated to instances in order to ensure their consistency with respect to the new schema version.

We introduce the operational semantics of each schema change which defines the composition of the new schema version w.r.t. the class definitions and also specifies how a database that is legal with respect to a schema version should be modified in order to ensure consistency after the schema version modification. To this purpose, we introduce two functions, the first called $\mathcal{SVU}$ (*Schema Version Update*), whose behaviour specifies how each schema change creates a new schema version, and the second called $\mathcal{IU}$ (*Instance Update*), whose behaviour specifies how instances have to be modified in order to become consistent with the schema version resulting from the application of a schema change.

Definition 5 (Schema Version Update) *Let $\mathcal{SC}$ be the set of all possible schema changes, and $\mathcal{SV}s$ a set of schema versions $\mathcal{SV}$. The Schema Version Update is a function*

$$\mathcal{SVU} : \mathcal{SC} \rightarrow (\mathcal{SV}s \rightarrow \mathcal{SV}s)$$

Definition 6 (Instance Update) *Let $\mathcal{SC}$ be the set of all possible schema changes, and $\mathcal{I}s$ a set of instance versions $\mathcal{I}$. The Instance Update is a function*

$$\mathcal{IU} : \mathcal{SC} \rightarrow (\mathcal{I}s \rightarrow \mathcal{I}s)$$

In the following, we define the behaviour of $\mathcal{SVU}$ and $\mathcal{IU}$ for all the schema changes listed in Tab. 1. The formalization we propose allows the schema update process to be formally checked and some desirable properties to be ensured. These aspects will be investigated in Sec. 3.3.

We first introduce two associative operations $\oplus_\tau$ (concatenation) and $\ominus_\tau$ (elimination) defined on the record type. More specifically,

$$\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle \oplus_\tau \langle A : \tau \rangle = \begin{cases} \langle A_1 : \tau_1, \ldots, A_n : \tau_n, A : \tau \rangle & \text{if } \nexists i : A_i = A \\ \bot & \text{otherwise} \end{cases}$$

$$\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle \ominus_\tau \langle A : \tau \rangle = \begin{cases} \langle A_1 : \tau_1, \ldots, A_{k-1} : \tau_{k-1}, A_{k+1} : \tau_{k+1}, \ldots, A_n : \tau_n \rangle & \\ \qquad \text{if } \exists k : A_k = A, \tau_k = \tau & \\ \langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle \qquad \text{otherwise} & \end{cases}$$

Notice that the concatenation of record types is possible only if the attributes involved have unique names, otherwise the operation is rejected. In general, we will adopt the $\bot$ symbol to denote the outcome of a rejected operation.

3.1 Changes to a Class Type

When applying a *change to a class type* operation, the effect at intensional level is the update of the types of the class having the property concerned and of its subclasses. In particular, at schema level, the application of any of such modifications to the class $\overline{C}$ is articulated in two steps:

- an intermediate step consisting of the local modification of the type of $\overline{C}$ and, in some cases, of its subclasses;
- the recalculation of the type of all the classes in the class hierarchy rooted on $\overline{C}$.

The intermediate step is realized by the definition of the $\overline{\sigma}$ function. The final step must also deal with problems related to multiple inheritance, which are solved by rejecting operations involving the integration of incompatible types. Notice that the application of a change to the class type is not always possible. In particular, in order to delete an attribute or to change its name, the attribute must have been locally defined in the class, rather than simply being inherited by any of its superclasses, where it has been defined. Moreover, in order to change the name of an attribute,

Schema Change	Semantics
AddAttribute$_{A:\tau,\overline{C}}$	$\sigma'(C) = \begin{cases} \tau_C(C,(\mathcal{CNV},\overline{\sigma},\preceq)) & \text{if } C \in <_{\overline{C}} \\ \overline{\sigma}(C) & \text{otherwise} \end{cases}$ and $\overline{\sigma}(C) = \begin{cases} \sigma(C) \oplus_\tau \langle A:\tau\rangle & \text{if } C = \overline{C} \\ \sigma(C) & \text{otherwise} \end{cases}$
	$\nu'(oi) = \begin{cases} \mathcal{I}_C(\nu(oi),\sigma(C),\sigma'(C),(\mathcal{CNV},\sigma',\preceq)) & \text{if } oi \in \pi(C), C \in \leq_{\overline{C}} \\ \nu(oi) & \text{otherwise} \end{cases}$
DeleteAttribute$_{A:\tau,\overline{C}}$	$\sigma'(C) = \begin{cases} \tau_C(C,(\mathcal{CNV},\overline{\sigma},\preceq)) & \text{if } C \in \leq_{\overline{C}} \\ \overline{\sigma}(C) & \text{otherwise} \end{cases}$ and $\overline{\sigma}(C) = \begin{cases} \sigma(C) \ominus_\tau \langle A:\tau\rangle & \text{if } C \in \leq_{\overline{C}} \\ \sigma(C) & \text{otherwise} \end{cases}$
	$\nu'(oi) = \begin{cases} \mathcal{I}_C(\nu(oi),\sigma(C),\sigma'(C),(\mathcal{CNV},\sigma',\preceq)) & \text{if } oi \in \pi(C), C \in \leq_{\overline{C}} \\ \nu(oi) & \text{otherwise} \end{cases}$
ChangeAttrName$_{A,A',\overline{C}}$	$\sigma'(C) = \begin{cases} \tau_C(C,(\mathcal{CNV},\overline{\sigma},\preceq)) & \text{if } C \in \leq_{\overline{C}} \\ \overline{\sigma}(C) & \text{otherwise} \end{cases}$ and $\overline{\sigma}(C) = \begin{cases} \langle A_1:\tau_1,\ldots,A':\tau,\ldots,A_n:\tau_n\rangle & \text{if } C \in \leq_{\overline{C}}, \\ \qquad \sigma(C) = \langle A_1:\tau_1,\ldots,A:\tau,\ldots,A_n:\tau_n\rangle & \\ \sigma(C) & \text{otherwise} \end{cases}$
	$\nu'(oi) = \begin{cases} \mathcal{I}_C(\overline{\nu}(oi),\overline{\sigma}(C),\sigma'(C),(\mathcal{CNV},\sigma',\preceq)) & \text{if } oi \in \pi(C), C \in \leq_{\overline{C}} \\ \overline{\nu}(oi) & \text{otherwise} \end{cases}$ and $\overline{\nu}(oi) = \begin{cases} \langle A_1:v_1,\ldots,A':v,\ldots,A_n:v_n\rangle & \text{if } oi \in \pi(C), C \in \leq_{\overline{C}}, \\ \qquad \nu(oi) = \langle A_1:v_1,\ldots,A:v,\ldots,A_n:v_n\rangle & \\ \nu(oi) & \text{otherwise} \end{cases}$
ChangeAttrType$_{A,\tau,\tau',\overline{C}}$	$\sigma'(C) = \begin{cases} \tau_C(C,(\mathcal{CNV},\overline{\sigma},\preceq)) & \text{if } C \in <_{\overline{C}} \\ \overline{\sigma}(C) & \text{otherwise} \end{cases}$ and $\overline{\sigma}(C) = \begin{cases} \langle A_1:\tau_1,\ldots,A:\tau',\ldots,A_n:\tau_n\rangle & \\ \qquad \text{if } C = \overline{C}, \sigma(C) = \langle A_1:\tau_1,\ldots,A:\tau,\ldots,A_n:\tau_n\rangle, & \\ \qquad \forall C' \in >_C\text{: } \langle A_1:\tau_1,\ldots,A:\tau',\ldots,A_n:\tau_n\rangle \leq \sigma(C') & \\ \bot \quad \text{if } C = \overline{C}, \sigma(C) = \langle A_1:\tau_1,\ldots,A:\tau,\ldots,A_n:\tau_n\rangle, & \\ \qquad \exists C' \in >_C\text{: } \langle A_1:\tau_1,\ldots,A:\tau',\ldots,A_n:\tau_n\rangle \not\leq \sigma(C') & \\ \langle A_1:\tau_1,\ldots,A:\tau',\ldots,A_n:\tau_n\rangle & \\ \qquad \text{if } C \in <_{\overline{C}}, \sigma(C) = \langle A_1:\tau_1,\ldots,A:\tau,\ldots,A_n:\tau_n\rangle & \\ \sigma(C) & \text{otherwise} \end{cases}$
	$\nu'(oi) = \begin{cases} \mathcal{I}_C(\nu(oi),\sigma(C),\sigma'(C),(\mathcal{CNV},\sigma',\preceq)) & \text{if } oi \in \pi(C), C \in \leq_{\overline{C}} \\ \nu(oi) & \text{otherwise} \end{cases}$

continued on next page

Schema Change	Semantics
ChangeClassType$_{\overline{C},\tau,\tau'}$	$\sigma'(C) = \begin{cases} \tau_C(C,(\mathcal{CNV},\overline{\sigma},\preceq)) & \text{if } C \in <_{\overline{C}} \\ \overline{\sigma}(C) & \text{otherwise} \end{cases}$ and $\overline{\sigma}(C) = \begin{cases} \tau' & \text{if } C = \overline{C}, \forall C' \in >_C : \tau' \leq \sigma(C') \\ \bot & \text{if } C = \overline{C}, \exists C' \in >_C : \tau' \not\leq \sigma(C') \\ \sigma(C) & \text{otherwise} \end{cases}$
	$\nu'(oi) = \begin{cases} \mathcal{I}_C(\nu(oi),\sigma(C),\sigma'(C),(\mathcal{CNV},\sigma',\preceq)) & \text{if } oi \in \pi(C), C \in \leq_{\overline{C}} \\ \nu(oi) & \text{otherwise} \end{cases}$

Table 2
Semantics of the changes to a class type

no other property in the type of $\overline{C}$ and of all its (direct or indirect) subclasses must have the same name as the target property name [4] .

More formally, given a schema version $\mathcal{SV} = (\mathcal{CNV},\sigma,\preceq)$ and an instance $\mathcal{I} = (\pi,\nu)$ legal for $\mathcal{SV}$, the execution of a schema change m produces a new schema version $\mathcal{SV}'$ and a new instance $\mathcal{I}'$ defined as $(\mathcal{CNV},\sigma',\preceq) = \mathcal{SVU}(m)(\mathcal{SV})$ and $(\pi,\nu') = \mathcal{IU}(m)(\mathcal{I})$, respectively. The semantics of these operations, for each m belonging to the *changes to a class type* category, is shown in Tab. 2 as the definition of the modified components σ' and ν'. For instance, when an attribute A with type τ is added to the class $\overline{C}$ (via $m =$ **AddAttribute**$_{A:\tau,\overline{C}}$), we first add the new attribute A to the type of $\overline{C}$ by means of the $\overline{\sigma}$ function. Then all its subclasses must inherit the new attribute. If no subclass of $\overline{C}$ contains an attribute with the same name, the new attribute is simply propagated. Otherwise, if a subclass of $\overline{C}$ already contains a proper or inherited attribute with the same name and a type compatible with the type of A, the attribute with the most specific type is selected. If the types of the two properties with the same name A are incompatible, the operation is rejected. To this end, the type conversion function τ_C infers the type of each $\overline{C}$ subclass and is defined in order to reject (with outcome $\bot$) the operation when two properties with the same name but with incompatible types are found. For each class type to be calculated, $\tau_C(C,(\mathcal{CNV},\sigma,\preceq))$ is a shorthand for $\sigma(C) \sqcap \sigma(C_1) \sqcap \cdots \sqcap \sigma(C_n)$ (where $\{C_1,\ldots,C_n\} = >_C$) which merges the class type with the types of its superclasses. On the other hand, the instance conversion function $\mathcal{I}_C$ adapts the values of the objects, which are instances of the modified types, to the new type versions, trying to preserve as much information as possible. The complete definitions of τ_C and $\mathcal{I}_C$ can be found in Appendix A.

[4] An alternative approach would consist in recalculating the type of the classes included in $\leq_{\overline{C}}$ by choosing the most specific type when two properties with the same name occur. However, in this case, a problem arises during the propagation of types to objects due to the semantic interpretation of data, as both properties with the same name are already populated with meaningful values.

Schema Change	Semantics
AddSuperclass$_{\overline{C},C'}$	$\preceq' = \begin{cases} \preceq \cup\{(C', \overline{C})\} & \text{if it is a } \textit{partial order} \text{ relation} \\ \text{rejected} & \text{otherwise} \end{cases}$
	$\sigma'(C) = \begin{cases} \tau_C(C, (\mathcal{CNV}, \sigma, \preceq')) & \text{if } C \in \leq'_{C'} \\ \sigma(C) & \text{otherwise} \end{cases}$
	$\nu'(oi) = \begin{cases} \mathcal{I}_C(\nu(oi), \sigma(C), \sigma'(C), (\mathcal{CNV}, \sigma', \preceq')) & \text{if } oi \in \pi(C), C \in \leq'_{C'} \\ \nu(oi) & \text{otherwise} \end{cases}$
DeleteSuperclass$_{\overline{C},C'}$	$\preceq' = \preceq \setminus\{(C', \overline{C})\}$
	$\sigma'(C) = \tau_C(C, (\mathcal{CNV}, \sigma, \preceq'))$ for each $C \in \mathcal{CNV}$
	$\nu'(oi) = \mathcal{I}_C(\nu(oi), \sigma(C), \sigma'(C), (\mathcal{CNV}, \sigma', \preceq'))$
AddClass$_{\overline{C}}$	$\mathcal{CNV}' = \mathcal{CNV} \cup \{\overline{C}\}$ (we may assume $\overline{C}$ already present in $\mathcal{CN}$)
	$\sigma' = \sigma \cup \{\overline{C} \mapsto$ **any**$\}$
	$\pi' = \pi \cup \{\overline{C} \mapsto \emptyset\}$
DeleteClass$_{\overline{C}}$	$\mathcal{CNV}' = \mathcal{CNV} \setminus \{\overline{C}\}$
	$\sigma'(C) = \begin{cases} \bot & \text{if } \overline{\sigma}(C) \text{ contains references to } \overline{C} \\ \overline{\sigma}(C) & \text{otherwise} \end{cases}$ and $\overline{\sigma} = \sigma \setminus \{\overline{C} \mapsto \sigma(\overline{C})\}$
	$\pi' = \pi \setminus \{\overline{C} \mapsto \pi(\overline{C})\} \cup \{\overline{C} \mapsto \emptyset\}$
	$\nu' = \nu \setminus \{oi \mapsto \nu(oi) \mid oi \in \pi(\overline{C})\}$
ChangeClassName$_{\overline{C},C'}$	$\mathcal{CNV}' = \mathcal{CNV} \setminus \{\overline{C}\} \cup \{C'\}$
	$\sigma'(C) = \{C \mapsto \tau' \mid (C \mapsto \tau) \in \sigma, C \neq \overline{C}\} \cup \{C' \mapsto \tau' \mid (\overline{C} \mapsto \tau) \in \sigma, \}$ and τ' is obtained from τ by substituting all $\overline{C}$ occurrences with C'
	$\preceq' = \{(C', C) \mid (\overline{C}, C) \in \preceq\} \cup \{(C, C') \mid (C, \overline{C}) \in \preceq\}$ $\cup\{(C, \tilde{C}) \mid (C, \tilde{C}) \in \preceq, C \neq \overline{C}, \tilde{C} \neq \overline{C}\}$
	$\pi' = \{C \mapsto OIs \mid (C \mapsto OIs) \in \pi, C \neq \overline{C}\}$ $\cup\{C' \mapsto OIs \mid (\overline{C} \mapsto OIs) \in \pi\}$

Table 3
Semantics of the changes to the class collection

3.2 Changes to the Class Collection

In this subsection, we consider the behaviour of the *changes to the class collection*. Let us start from modifications to hierarchical relationships. To add or delete a supertype, respectively, means to make an existing class $\overline{C}$ a supertype of a class C' or to remove a class $\overline{C}$ from the supertypes of a class C'. In both cases, two steps

are required:

- the update of the inheritance relation with the proviso that it remains a partial order;
- the inference of the type of the affected class C' and of its subclasses.

When $\overline{C}$ becomes a C' superclass, the extension π^* of $\overline{C}$ must be changed so that it also contains all C' instances. Since π^* is defined by means of the inheritance relationship $\preceq$ and the proper extension π, the π^* update is automatically done by updating the $\preceq$ component (π remains unchanged).

More formally, given a schema version $\mathcal{SV} = (\mathcal{CNV}, \sigma, \preceq)$ and an instance $\mathcal{I} = (\pi, \nu)$ legal for $\mathcal{SV}$, the outcomes of the application of the $\mathcal{SVU}$ and $\mathcal{IU}$ functions for each of the *changes to the class collection* m — $(\mathcal{CNV}, \sigma', \preceq') = \mathcal{SVU}(m)(\mathcal{SV})$ and $(\pi, \nu') = \mathcal{IU}(m)(\mathcal{I})$ — are shown in Tab. 3, where $\leq'$ is the subtyping relation based on the class hierarchy in $(\mathcal{CNV}, \sigma, \preceq')$. Notice that the addition of a pair $(C', \overline{C})$ to the partial order $\preceq$ could also cause such a relation to lose the property of being a partial order if, for instance, the antisymmetry is broken or a cycle is introduced. In such a case, the operation is rejected.

In the addition of a new superclass, since the inheritance relation is augmented with a new relationship, all subtyping relationships which were valid in the previous schema version continue to be valid.

The deletion of a superclass, due to the "reduction" of the subtyping relation, is actually the only schema modification which requires the re-computation of the type of all the classes of the lattice. Notice that, when removing the relationship $C' \preceq \overline{C}$, some subtyping relationships which were valid in the previous schema versions may become invalid. Suppose, for instance, that the type of a class D is simply a record only containing an attribute A with type $\overline{C}$ and that the subclass D' of D redefines the attribute A with type C'. Before the deletion of the connection between C' and $\overline{C}$, the type of D' is certainly a subtype of the D type, whereas, after the deletion, this relationship is no longer ensured. Starting from the type associated to each class by σ, the τ_C function verifies if either the subtyping relationships between the types of classes and the corresponding superclasses continue to exist or not. In fact, in its base cases, the τ_C function produces the $\perp$ result if a subtyping relationship is not defined between two classes. Notice also that for all subclasses C of C', the relationship $C \preceq' \overline{C}$ continues to exist since the inheritance relation is a partial order. These relationships can be deleted through the removal of all the hierarchical relationships between C' and its subclasses.

For all the other schema modifications we propose local solutions which only involve the classes included in the lattice rooted on the modified class.

The class addition or deletion, instead, involve a single class. The primitive change **AddClass** produces an empty class with type **any**. This new class is isolated in

the class hierarchy, that is it has neither superclasses nor subclasses. More complex changes, like "add a class with attributes" in the middle of the class hierarchy, could be effected by combining primitive changes, for example **AddClass** followed by **AddAttribute** and **AddSuperclass**. In the same way, also **DeleteClass** requires that the class to be deleted be isolated. Any complex change, like "delete a class in the middle of the class hierarchy", could be effected by making **DeleteClass** follow all the necessary **DeleteSuperclass** that must be executed in order to isolate the class before deletion. In any case, deleting a class may give rise to a new problem since all references (by means of attributes) to such a class become dangling. To solve this problem, various approaches have been proposed [2,25,26]. The two mainstream solutions are the following:

(1) transform any reference to the deleted class into **any** and any reference to its corresponding objects into $null$ (i.e. this is the solution adopted by O_2 [25]),
(2) forbid schema versions giving rise to referential integrity problems that could be avoided by executing appropriate schema changes before the **DeleteClass** operation.

Since the first solution can be obtained by means of specific schema and object changes (already provided by our model), we continue to follow the LLP approach by adopting the second solution. Obviously, in an implementation phase, the schema changes to be placed before the **DeleteClass** operation can (semi)automatically be applied by the system. The components of $\mathcal{SV}$ and $\mathcal{I}$ modified by the application of the **AddClass** and **DeleteClass** operations are shown in Tab. 3. The last schema change is the "change the name of a class" operation which leads to a new schema version where all its elements have been modified in order to replace all occurrences of $\overline{C}$ with C'.

3.3 Correctness of the Schema Update Process

The formalization of the schema update and propagation mechanism allows the evolving schema process to be formally checked. The correctness of the process is carried out according to the following two requirements.

- The schema update function $\mathcal{SVU}$ must be correct, that is it must respect the semantics of types and inheritance.
- The instance update function $\mathcal{IU}$ must be correct as well, that is it should ensure that the mapping associated with each schema change produces a legal instance for the corresponding transformed schema version.

First, we consider the semantics of types and inheritance during the schema update process. The $\mathcal{SVU}$ function defined above always applies the τ_C function which is devoted to the computation of the class types after each schema change by means of the *meet* operator ($\sqcap$) applied to a given set of types. The following Lemma states

that the $\sqcap$ operation is monotonic.

Lemma 1 *Let $(\mathcal{CNV}, \sigma, \preceq)$ be a schema version. The meet operation $\sqcap$ is* monotonic*, that is given $\tau, \tau', \alpha, \alpha' \in \mathcal{T}|_{\mathcal{CNV}}$ where $\tau \leq \tau'$, $\alpha \leq \alpha'$, $\tau \sqcap \alpha \neq \bot$ and $\tau' \sqcap \alpha' \neq \bot$ then:*

$$\tau \sqcap \alpha \leq \tau' \sqcap \alpha'$$

Given the above result, the following Corollary can easily be proved (all the proofs can be found in Appendix B).

Corollary 1 *Let $(\mathcal{CNV}, \sigma, \preceq)$ be a schema version and $C \in \mathcal{CNV}$. For each $C_i \in >_C$:*

(1) $\tau_C(C, (\mathcal{CNV}, \sigma, \preceq)) \leq \sigma(C_i)$
(2) $\tau_C(C, (\mathcal{CNV}, \sigma, \preceq)) \leq \tau_C(C_i, (\mathcal{CNV}, \sigma, \preceq))$

Now, we can show that the schema update process preserves the class hierarchy semantics. In other words, each schema change transforms well-formed schema versions into well-formed schema versions.

Theorem 1 *Let $(\mathcal{CNV}, \sigma, \preceq)$ be a well-formed schema version. For each schema modification m, $(\mathcal{CNV}', \sigma', \preceq') = \mathcal{SVU}(m)(\mathcal{CNV}, \sigma, \preceq)$ is a well-formed schema version.*

Given a schema modification, the $\mathcal{IU}$ function transforms legal instances of a schema version into legal instances of the corresponding transformed one, obtained by means of the $\mathcal{SVU}$ function application. The proof of this property first requires the proof of the Lemma below which states that the $\mathcal{I}_C$ function, employed by $\mathcal{IU}$, is a well-formed value transformation.

Lemma 2 *Let $(\mathcal{CNV}, \sigma, \preceq)$ be a well-formed schema version, $\tau, \tau' \in \mathcal{T}|_{\mathcal{CNV}}$. If $v \in dom(\tau)$ then $\mathcal{I}_C(v, \tau, \tau', (\mathcal{CNV}, \sigma, \preceq)) \in dom(\tau')$.*

Thence, the propagation process only produces legal instances for the transformed schema version.

Theorem 2 *Let (π, ν) be a legal instance for the schema version $(\mathcal{CNV}, \sigma, \preceq)$. For each schema modification m, $(\pi', \nu') = \mathcal{IU}(m)(\pi, \nu)$ is a legal instance for the schema version $\mathcal{SVU}(m)(\mathcal{CNV}, \sigma, \preceq)$.*

As a final remark, it should be noted that the problem of the correctness of a schema under modifications has already been investigated in the schema evolution context. The main approach (adopted for instance by ORION [2], COCOON [27], O_2 [1]) consists in defining and enforcing a number of *invariants*:

Closure Invariants All types in the type lattice have supertypes.

Acyclicity Invariant There are no cycles in the type lattice.
Rootedness Invariant There is a single type which is the supertype of all types.
Full Inheritance Invariant The type of a class consists of the native (or proper) type and inherited types.
Axiom of Unique Naming Classes must have unique names in the schema version; attributes must have different names within their domain class types.

Notice that all of them hold in our definition of well-formed schema version. Therefore, Theorem 1 ensures that such invariants are preserved throughout the schema modification process.

4 Interaction between Intensional and Extensional Versioning: a Temporal Approach

A schema describes the structure of the data that are stored in a database. $OODM|_{SV}$ introduces the notion of *evolving schema*, which is the core of the temporal schema versioning support. As to the underlying database, we impose no restriction on the temporal dimensions they support. In the temporal database literature, two time dimensions are usually considered: **valid time** (concerning the real world) and **transaction time** (concerning the database life) [28]. In fact, we will show how an evolving schema can interact with snapshot databases as well as with temporal databases with valid and/or transaction time support.

4.1 Temporal Schema Versioning

As far as the temporal dimensions involved in schema versioning are concerned, versioning along one time dimension gives rise to transaction- or valid-time schema versioning and versioning along both time dimensions produces bitemporal schema versioning [16].

Transaction-time schema versioning Transaction-time schema versioning allows *on-time* schema changes, that is schema changes that are effective when applied. In this case, the management of time is completely transparent to the user: only the current schema can be modified and schema changes are effected in the usual way, without any reference to time. However, support of past schema versions is granted by the system non-deletion policy, so that the user can always *rollback* the full database to a past state of its life.
Valid-time schema versioning It has long been debated whether valid time is eligible for schema versioning, and the answer depends on application requirements: valid-time schema versioning is necessary when *retroactive* or *proactive* schema modifications have to be supported [16] and it is useful to assign a tem-

poral *validity* to schema versions. With valid-time schema versioning, multiple schema versions, valid at different times, are all available to access and manipulate data and also for further modifications. The newly created schema version can be assigned any validity by the designer (also in the past or future to effect resp. retro- or pro-active schema modifications). The (portions of) existing schema versions overlapped by the validity of the new schema version are overwritten.

Bitemporal schema versioning In this case both time dimensions are used. In addition to transaction-time schema versioning, retro- and pro-active schema updates are supported. With respect to valid-time schema versioning, the complete history of schema changes is maintained as no schema version is ever discarded (overlapped portions are "archived" rather than deleted). In a system where full auditing/traceability of the maintenance process is required, only bitemporal schema versioning allows verifying whether a schema version was created by a retro- or pro-active schema change.

In $OODM|_{\mathcal{SV}}$ we consider all the three kinds of temporal schema versioning.

4.2 Evolving Schema and Temporal Databases

An evolving schema consists of a collection of schema versions defined over a set of class names and attribute names. In particular, an evolving schema associates each schema version with its temporal pertinence, which is defined as a subset of the *time domain*.

In accordance with the BCDM model [29], the notion of time is represented in $OODM|_{\mathcal{SV}}$ as a discrete set $\{0, 1, \ldots, now, \ldots, \infty\}$ of *chronons*. The symbol '∞' is used to timestamp a still current fact in transaction time and represents the maximum time value in valid time. The symbol now denotes the current transaction time and the present valid time. Notice that the time points between 'now' and '∞' in transaction time are purely conventional, as they represent system events that have not happened yet.

Since valid and transaction time are orthogonal dimensions (with a different meaning), we will use subscripts to distinguish between them. Hence, the time domains of interest for valid-time, transaction-time and bitemporal schema versioning in $OODM|_{\mathcal{SV}}$ are $\mathcal{TIME}_t$, $\mathcal{TIME}_v$ and the Cartesian product $\mathcal{TIME}_t \times \mathcal{TIME}_v$, respectively. The temporal pertinence (*timestamp*) of a schema version can always be represented by means of a disjoint union of intervals in the first two cases and of rectangles in the last case, where each rectangle is the product of a transaction- and a valid-time interval.

In $OODM|_{\mathcal{SV}}$, one of the conditions which an evolving schema must satisfy is that temporal pertinences of different schema versions are disjoint, that is at most

one schema version is associated to each temporal chronon as the one "active" (i.e. current and/or valid) at that time.

Definition 7 (Evolving Schema) *Let $\mathcal{CN}$ be a set of class names, $\mathcal{AN}$ a set of attribute names, $\mathcal{SV}s$ a set of schema versions and M a set of schema modifications. An evolving schema $\mathcal{ES}$ is a tuple $(\mathcal{CN}, \mathcal{AN}, \mathcal{SV}s, \delta, \psi)$ where $\psi \subseteq M \times \mathcal{SV}s \times \mathcal{SV}s$ is a relation such that $(m, \mathcal{SV}_i, \mathcal{SV}_j) \in \psi$ iff $\mathcal{SV}_j$ is the outcome of the application of m to $\mathcal{SV}_i$ ($\mathcal{SV}_j = \mathcal{SVU}(m)(\mathcal{SV}_i)$) and δ is a* timestamping *function*

$$\delta : \mathcal{TIME} \rightarrow \mathcal{SV}s \cup \{\emptyset\}$$

which associates each temporal chronon with the corresponding active schema version $\mathcal{SV}_i$, if it exists, or with an empty set otherwise. In particular:

- *if transaction time schema versioning is supported, $\mathcal{TIME} = \mathcal{TIME}_t$ and*

$$\delta(tt) = \begin{cases} \mathcal{SV}_i \text{ if } \exists \mathcal{SV}_i \text{ active (current) in } tt \\ \emptyset \quad \text{otherwise} \end{cases}$$

  where $tt \in \mathcal{TIME}_t$ and for all $tt_1, tt_2 \in [now_t, \infty]_t$: $\delta(tt_1) = \delta(tt_2) \neq \emptyset$;
- *if valid time schema versioning is supported, $\mathcal{TIME} = \mathcal{TIME}_v$ and*

$$\delta(vt) = \begin{cases} \mathcal{SV}_i \text{ if } \exists \mathcal{SV}_i \text{ active (valid) in } vt \\ \emptyset \quad \text{otherwise} \end{cases}$$

  where $vt \in \mathcal{TIME}_v$;
- *if bitemporal schema versioning is supported, $\mathcal{TIME} = \mathcal{TIME}_t \times \mathcal{TIME}_v$ – with a little abuse of notation we will always write $\delta(tt, tv)$ instead of $\delta((tt, tv))$– and*

$$\delta(tt, vt) = \begin{cases} \mathcal{SV}_i \text{ if } \exists \mathcal{SV}_i \text{ active in } (tt, vt) \\ \emptyset \quad \text{otherwise} \end{cases}$$

  where $(tt, vt) \in \mathcal{TIME}$ and for each $vt \in \mathcal{TIME}_v$ and for all $tt_1, tt_2 \in [now_t, \infty]_t$: $\delta(vt, tt_1) = \delta(vt, tt_2) \neq \emptyset$.

If $\mathcal{SV} \neq \emptyset$, we will denote as $\delta^{-1}(\mathcal{SV})$ the the set of time points that are associated by the δ function to $\mathcal{SV}$, that is $\delta^{-1}(\mathcal{SV})$ represents the *temporal pertinence* (timestamp) of $\mathcal{SV}$.

Example 3 *Let us consider an evolving schema consisting in the collection $\{\mathcal{SV}_1, \mathcal{SV}_2, \mathcal{SV}_3\}$ of schema versions which have been generated by the following transactions:*

t1 *Definition of a schema made up of two classes:* employee *and* professor*;* employee *contains two attributes,* name *and* ssn*,* professor *is an* employee *subclass that also contains the attribute* deg *(its definition is $\mathcal{SV}_1$ of Ex. 1).*

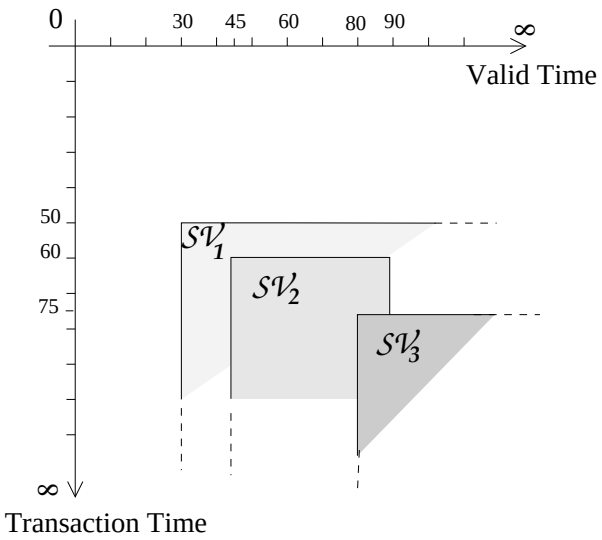


Fig. 1. Temporal representation of an evolving schema

t2 *Addition of a new attribute* badge_no *to the* employee *class in the schema version* $\mathcal{SV}_1$*; the outcome is:* $\mathcal{SV}_2 = \mathcal{SVU}(\textbf{\textit{AddAttribute}}_{badge\_no:integer,employee})(\mathcal{SV}_1) = (\mathcal{CNV}_1, \sigma_2, \preceq_1)$*, where:*

$$\begin{aligned} \sigma_2 : employee &\mapsto \langle name : string, ssn : integer, badge\_no : integer\rangle, \\ \sigma_2 : professor &\mapsto \langle name : string, ssn : integer, deg : \{string\}, \\ &\quad badge\_no : integer\rangle; \end{aligned}$$

t3 *Addition of a new class* course *to the schema version* $\mathcal{SV}_1$*; the outcome is:* $\mathcal{SV}_3 = \mathcal{SVU}(\textbf{\textit{AddClass}}_{course})(\mathcal{SV}_1) = (\mathcal{CNV}_1 \cup \{course\}, \sigma_1 \cup \{course \mapsto \textbf{\textit{any}}\}, \preceq_1)$*.*

The evolving schema after transaction ***t3*** *is:*

$$(\{employee, professor, course\}, \{name, ssn, deg, badge\_no\}, \{\mathcal{SV}_1, \mathcal{SV}_2, \mathcal{SV}_3\}, \delta, \psi)$$

where:

$$\psi = \{(\textbf{\textit{AddAttribute}}_{badge\_no:int,employee}, \mathcal{SV}_1, \mathcal{SV}_2), (\textbf{\textit{AddClass}}_{course}, \mathcal{SV}_1, \mathcal{SV}_3)\},$$

and Fig. 1 shows a graphical representation of a possible definition of $\delta : \mathcal{TIME}_t \times \mathcal{TIME}_v \rightarrow \mathcal{SV}s \cup \{\emptyset\}$*. In fact, it defines the following pertinences:* $\delta^{-1}(\mathcal{SV}_1) = [50, 59]_t \times [30, \infty]_v \cup [60, \infty]_t \times [30, 44]_v \cup [60, 74]_t \times [91, \infty]_v$*,* $\delta^{-1}(\mathcal{SV}_2) = [60, 74]_t \times [45, 90]_v \cup [75, \infty]_t \times [45, 79]_v$*,* $\delta^{-1}(\mathcal{SV}_3) = [75, \infty]_t \times [80, \infty]_v$*.*

In the following we consider the interaction of an evolving schema with an underlying database $\mathcal{DB}$, which could be a snapshot database or also a (bi)temporal database. In the former case, a $\mathcal{DB}$ is simply an instance $\mathcal{I}$ as introduced in Subsec. 2.2; in the latter case, a *temporal database* can be considered as a function mapping

↓ Type of Database — Type of Schema Versioning →		**(tS)**	**(vS)**	**(tvS)**
(D)	$\mathcal{DB}$	$\delta(now_t)$	$\delta(now_v)$	$\delta(now_t, now_v)$
(tD)	$\forall tt \in \mathcal{TIME}_t : \mathcal{DB}(tt)$	$\delta(tt)$	$\delta(tt)$	$\delta(tt, tt)$
(vD)	$\forall vt \in \mathcal{TIME}_v : \mathcal{DB}(vt)$	$\delta(now_t)$	$\delta(vt)$	$\delta(now_t, vt)$
(tvD)	$\forall tt \in \mathcal{TIME}_t : \forall vt \in \mathcal{TIME}_v : \mathcal{DB}(tt, vt)$	$\delta(tt)$	$\delta(vt)$	$\delta(tt, vt)$

To be read as: "The instance(s) above... ...must be legal wrt schema version above".

Table 4
Conditions for the consistency of a database with evolving schema

the time domain to database instances:

$$\mathcal{DB} : \mathcal{TIME} \rightarrow \mathcal{I}s \cup \{\emptyset\}$$

where $\mathcal{I}s$ is a set of (snapshot) database instances and $\mathcal{TIME}$ is $\mathcal{TIME}_t$, $\mathcal{TIME}_v$ or $\mathcal{TIME}_t \times \mathcal{TIME}_v$, respectively, if a valid-time, transaction-time or bitemporal database is considered. Notice that we are not interested here in a detailed modeling of temporal databases but rather in the investigation of the interaction between an evolving schema and its extensional component. For this reason we have adopted an abstract (functional) generic approach to temporal databases; however, any specific temporal data model (e.g.[11]) can be put in this form.

4.3 Definition of Legal Databases

In this subsection we introduce the notion of legal database by considering all the combinations of the two orthogonal levels of versioning, intensional and extensional. Notice that an evolving schema is made up of one or more schema versions, whereas a database consists of one or more instances. Hence, in all the cases we will consider, consistency will always be based on the notion of legal instance for a schema version (Def. 4); the cases will simply differ on the instances and schema versions to be considered. As far as the interaction between the temporal dimensions involved in intensional and extensional versioning, we follow here a **Synchronous Management** approach [16], according to which *temporal data are always stored, retrieved and updated through the schema version having the same temporal pertinence*. Notice that the synchronous management is the usual choice for schema versioning support in object-oriented databases, where data objects are normally only visible (and updatable) through the schema version they belong to, that is the one in which they were created [17]. We will consider, at intensional level: transaction-time (**tS**), valid-time (**vS**) and bitemporal schema versioning (**tvS**), and, at extensional level: snapshot (**D**), transaction-time (**tD**), valid-time (**vD**) and bitemporal databases (**tvD**).

Let us start with the interaction between an evolving schema and a snapshot database

(**D**), which is simply made up of one instance. We must introduce different consistency conditions according to the semantics of the time dimensions supported by schema versioning. In fact, a snapshot database only represents the current state of the modeled enterprise; it maintains no history, either from the system or from the real world viewpoints. For this reason, we require a snapshot database to be legal with respect to the current (with transaction time equal to now_t) and present (with valid time equal to now_v) schema version [5] . As a consequence, when an evolving schema interacts with a snapshot database, a current and present schema version must always exist.

A transaction-time database (**tD**) can only "mimic" the evolution of the modeled real world evolution by means of its own modification history. In this perspective, the transaction-time pertinence of the stored data gives also the best possible approximation of their validity [31]. For this reason, when valid-time schema versioning is supported, transaction-time of data has to coincide with the valid-time of the schema to comply with a synchronous management. On the other hand, when transaction-time schema versioning is supported, a "pure" full synchronous approach can be followed.

As to a valid-time database (**vD**), it allows one to maintain the most accurate history (as known by users) of the modeled real world in the current state of the system. For this reason, when transaction-time schema versioning is supported, we require all instances in the valid-time database to be legal for the current schema version. When valid-time schema versioning is supported, a synchronous management forces us to consider the valid-time adopted for intensional and extensional versioning to be exactly the same. The case of bitemporal schema versioning can be simply managed as the combination of the previous two.

Finally, for a bitemporal database (**tvD**) we require full compliance with the adopted synchronous management approach.

All the resulting consistency constraints are summarized in Tab. 4, showing the conditions a database $\mathcal{DB}$ must satisfy in order to be legal with respect to an evolving schema $\mathcal{ES} = (\mathcal{CN}, \mathcal{AN}, \mathcal{SV}s, \delta, \psi)$. The columns on the right correspond to the three different kinds of intensional versioning whereas rows correspond to the four different kinds of extensional versioning. For each kind of database and of evolving schema, the table shows which instance(s) must be legal with respect to which schema version(s).

[5] Notice that, as time goes by, the value of now grows. For transaction time, this is not a problem since now_t is equivalent to $[now_t, \infty]_t$, whereas the semantics of valid time allows different schema versions (created via proactive schema changes) to be associated to different points in $[now_v, \infty]_v$. However, in order to enforce the consistency rules, we can always consider an *anchored* value of now_v, that coincides with the present time. This corresponds to consider the denotation of the "valid-time variable" now_v at the reference time of evaluation [30].

The evolving schema definition given in Def. 7 states that each schema version is the outcome of the application of one or more schema changes to another schema version. In other words, if $\mathcal{SV}_j$ derives from the application of the schema modification m to $\mathcal{SV}_i$, then m represents a "direct connection" between $\mathcal{SV}_i$ and $\mathcal{SV}_j$ ($(m, \mathcal{SV}_i, \mathcal{SV}_j) \in \psi$). In this perspective, an evolving schema can be interpreted as a directed graph, with the collection of schema versions representing nodes and the applied schema modifications representing edges. In fact, given an evolving schema $\mathcal{ES} = (\mathcal{CN}, \mathcal{AN}, \mathcal{SV}s, \delta, \psi)$, ψ can be interpreted as a *schema graph*, where each element $(m, \mathcal{SV}_i, \mathcal{SV}_j)$ represents an edge labeled with m connecting node $\mathcal{SV}_i$ to node $\mathcal{SV}_j$. More precisely, the graph is a tree where each schema version has at most one predecessor since $\mathcal{SVU}$ always generates a new schema version.

In a system with schema versioning support, the opportunity to access instances which are legal for a schema version through other schema versions will arguably be seized (e.g. to run compiled applications against data conforming to a subsequently modified schema). The schema graph can represent a way for understanding how to transform data such that they become consistent with respect to the target schema version. This will be possible only if each edge of the schema graph can be followed in both directions. Our first aim, thus, is to show that each schema modification m has an inverse m^{-1} in the supported collection. In some cases, reversibility of schema modification requires the introduction of some applicability conditions.

Example 4 *Consider the evolving schema of Ex. 3. Given an instance $\mathcal{I}$ which is legal for a schema version, $\mathcal{SV}_2$ for instance, and assuming that we are interested in transforming $\mathcal{I}$ into an instance which is legal for another schema version, say $\mathcal{SV}_3$, then we could apply the following modifications to $\mathcal{I}$:*

$$\mathcal{I}' = \mathcal{IU}(\textbf{\textit{AddClass}}_{course})(\mathcal{IU}(\textbf{\textit{AddAttribute}}^{-1}_{badge\_no:int,employee})(\mathcal{I}))$$

which, if the schema modification corresponding to $\textbf{\textit{AddAttribute}}^{-1}_{badge\_no:int,employee}$ is known to produce legal instances, ensures that $\mathcal{I}'$ is legal for $\mathcal{SV}_3$ by Theorem 2.

Tab. 5 lists the modifications to be applied when inverses are required. The following theorem ensures that, under some conditions on the source schema, the given changes are actually the inverse modifications of the supported schema changes. For instance, the applicability condition of **AddAttribute** specifies that if any subclass of $\overline{C}$ already contains an attribute with name A then its type must be a subtype of the one associated with the attribute A to be added to $\overline{C}$.

Theorem 3 *With reference to Tab. 5, for each schema modification m applied on a source schema $\mathcal{SV}$ (on which the evidenced preconditions hold) the schema modification m^{-1} of Tab. 1 is its inverse, that is: if $\mathcal{SVU}(m)(\mathcal{SV}) = \mathcal{SV}'$ where $\mathcal{SV}' \neq \perp$,*

m	m^{-1}
AddAttribute$_{A:\tau,\overline{C}}$ $\forall C \in <_{\overline{C}}$: if $\exists k : \sigma(C) = \langle \ldots, A : \tau_k, \ldots \rangle$ then $\tau_k \leq \tau$	**DeleteAttribute** $_{A:\tau,\overline{C}}$
DeleteAttribute$_{A:\tau,\overline{C}}$	**AddAttribute** $_{A:\tau,\overline{C}}$
ChangeAttrName$_{A,A',\overline{C}}$ $\forall C \in <_{\overline{C}} : \forall D \in >_C \setminus \leq_{\overline{C}}$: if $\nexists k : \sigma(D) = \langle \ldots, A : \tau_k, \ldots \rangle$	**ChangeAttrName**$_{A',A,\overline{C}}$
ChangeAttrType$_{A,\tau,\tau',\overline{C}}$ $\tau \leq \tau'$, $\forall C \in <_{\overline{C}} : \forall D \in >_C$: if $\exists k : \sigma(D) = \langle \ldots, A : \tau_k, \ldots \rangle$ then $\tau' \leq \tau_k$	**ChangeAttrType**$_{A,\tau',\tau,\overline{C}}$
ChangeClassType$_{\overline{C},\tau,\tau'}$ $\tau \leq \tau'$	**ChangeClassType**$_{\overline{C},\tau,\tau'}$
AddSuperclass$_{\overline{C},C'}$ $\nexists C'' \in \mathcal{CNV} : C'' \preceq C'$	**DeleteSuperclass**$_{\overline{C},C'}$
DeleteSuperclass$_{\overline{C},C'}$	**AddSuperclass**$_{\overline{C},C'}$
AddClass$_{\overline{C}}$	**DeleteClass**$_{\overline{C}}$
DeleteClass$_{\overline{C}}$	**AddClass**$_{\overline{C}}$
ChangeClassName$_{\overline{C},C'}$	**ChangeClassName**$_{C',\overline{C}}$

Table 5
List of inverse schema changes

then $\mathcal{SVU}(m^{-1})(\mathcal{SV}') = \mathcal{SV}$.

Therefore, if all schema modifications respected all the applicable preconditions, any edge of a schema graph can be followed in both directions, direct and inverse. Now, we introduce the notions of *minimal path* and *minimal modification* between any pair of schema versions.

Definition 8 (Minimal Path and Modification) *Let* $\mathcal{ES} = (\mathcal{CN}, \mathcal{AN}, \mathcal{SV}s, \delta, \psi)$ *be an evolving schema and* $\mathcal{SV}_i, \mathcal{SV}_j \in \mathcal{SV}s$.

The minimal path from $\mathcal{SV}_i$ *to* $\mathcal{SV}_j$ *is the sequence* $[\mathcal{SV}'_0, \mathcal{SV}'_1 \ldots, \mathcal{SV}'_{n-1}, \mathcal{SV}'_n]$ *of schema versions in* $\mathcal{SV}s$ *in the shortest (undirected) path which connects* $\mathcal{SV}_i$ *and* $\mathcal{SV}_j$ *in the schema graph, where* $\mathcal{SV}'_0 = \mathcal{SV}_i$ *and* $\mathcal{SV}'_n = \mathcal{SV}_j$.

Given the minimal path $[\mathcal{SV}'_0, \mathcal{SV}'_1 \ldots, \mathcal{SV}'_{n-1}, \mathcal{SV}'_n]$ *from* $\mathcal{SV}_i$ *to* $\mathcal{SV}_j$*, the corresponding minimal modification denoted as* $\mathcal{SV}_i \mapsto \mathcal{SV}_j$ *is the sequence of schema*

modifications $[m'_1, \ldots, m'_n]$ *where for each* $k \in [1, n]$

$$m'_k = \begin{cases} m_k & \text{if } (m_k, \mathcal{SV}'_{k-1}, \mathcal{SV}'_k) \in \psi \\ m_k^{-1} & \text{if } (m_k, \mathcal{SV}'_k, \mathcal{SV}'_{k-1}) \in \psi \end{cases}$$

Corollary 2 *Let* $G = (\mathcal{SV}s, M, \psi)$ *be a schema graph. For any pair* $(\mathcal{SV}_i, \mathcal{SV}_j)$ *of schema versions, a minimal path and a minimal modification from* $\mathcal{SV}_i$ *to* $\mathcal{SV}_j$ *always exists.*

It directly follows from the fact that ψ defines a tree and that, for each schema modification, $OODM|_{\mathcal{SV}}$ supports its inverse. Actually, since the schema graph is a tree, if SV_ℓ is the "youngest" common ancestor of SV_i and SV_j (i.e. the root of the minimal subtree containing either SV_i and SV_j), the minimal modification is made up of all inverse modifications to go up the subtree from SV_i to SV_ℓ and all direct modifications to go down the subtree from SV_ℓ to SV_j.

Finally, we can introduce a way for modifying an instance legal for a schema version such that it becomes legal for any other schema version in the schema graph.

Lemma 3 *Let* $G = (\mathcal{SV}s, M, \psi)$ *be a schema graph,* $\mathcal{SV}_i, \mathcal{SV}_j \in \mathcal{SV}s$*,* $[m'_1, \ldots, m'_n]$ *the minimal modification* $\mathcal{SV}_i \mapsto \mathcal{SV}_j$ *and* $\mathcal{I}$ *an instance legal for* $\mathcal{SV}_i$*. The instance*
$\mathcal{IU}(m'_n) \circ \mathcal{IU}(m'_{n-1}) \circ \ldots \circ \mathcal{IU}(m'_1)(\mathcal{I})$ *denoted as* $\mathcal{I}_{\mathcal{SV}_i \mapsto \mathcal{SV}_j}$ *is a legal instance for* $\mathcal{SV}_j$*.*

The proof of this Lemma directly follows from Lemma 4 and Theorem 2. Notice that, although $\mathcal{I}_{\mathcal{SV}_i \mapsto \mathcal{SV}_j}$ is legal for $\mathcal{SV}_j$, it could miss some (extensional) information with respect to $\mathcal{I}$. Lemma 4 operates at schema level, stating that it is possible to undo the effects of the application of any schema modification by applying its inverse. At instance level instead, undoing is not always possible. Consider, for example, the deletion of an attribute and its inverse, the addition: in this case, starting from an instance which is legal for a schema containing the attribute to be deleted (which can have non-null values), the deletion of such an attribute and then its re-addition will cause the introduction of all null values for the attribute.

5 Action of Schema Changes at Evolving Schema and Database Level (Global Effects)

When schema versioning is supported, schema changes allow database administrators to add new schema versions to the evolving schema, to reflect structural modifications to the modeled real world. In $OODM|_{\mathcal{SV}}$ each schema change affects one schema version and generates a new one. In the temporal context, where schema versions are "distributed" along the supported time dimensions, two aspects have

to be considered:

- which schema version is subject to change,
- which is the temporal pertinence of the new schema version resulting from the schema change.

When transaction time is supported, its particular semantics forces the current schema version(s) to be exclusively considered for changes and the interval $[now_t, \infty]_t$ to be assigned as the transaction-time pertinence to the resulting schema version. On the other hand, the management of valid time is only under the users' responsibility. For this reason, when valid time is supported, we require users to specify which schema version has to be modified and which is the valid-time pertinence of the newly introduced schema version. This is accomplished by means of two parameters to be added to the schema change specification: the *schema selection validity* and the *schema change validity*. The former ($\overline{ss}$) is a valid-time chronon and is used to select the schema version; the latter ($\overline{sc}$) is a valid-time element, that is a disjoint union of valid-time intervals, representing the validity to be assigned to the schema change result.

Therefore, when one or more schema changes in Tab. 1 have to be applied to the schema, one schema version is selected: the current one, for transaction-time schema versioning, the one which satisfies the schema selection validity (i.e. valid at $\overline{ss}$) for valid-time schema versioning and, among the current schema versions, the one which satisfies the schema selection validity for bitemporal schema versioning. Then a new schema version is generated as the outcome of the schema change(s) effected on the selected schema version. The temporal pertinence of the new schema version is $[now, \infty]_t$ for transaction-time schema versioning, the schema change validity $\overline{sc}$ for valid-time schema versioning and the product $[now, \infty]_t \times \overline{sc}$ for bitemporal schema versioning. As a consequence, all the schema versions whose temporal pertinence is overlapped by the new schema version have their overlapped part overwritten (or "archived"). When transaction time is supported, this means that they quit the *current* part of the database and can no longer be modified. Such a process is formalized by the schema update function $\mathcal{SU}$. The $\mathcal{SU}$ function transforms the evolving schema into a new one and, obviously, its signature depends on the supported temporal schema versioning (additional parameters are required only if valid-time is supported) and it makes use of the schema version update function $\mathcal{SVU}$ for the derivation of the new schema version.

We first introduce a function, named $place$, which associates a schema version with its pertinence as follows ($\mathcal{SV}s$ is the set of all possible schema versions and Δs is the set of all possible timestamping functions):

$$place : \mathcal{SV}s \times 2^{\mathcal{TIME}} \times \Delta s \rightarrow \Delta s$$

if $\mathcal{SV} \in \mathcal{SV}s$ is a schema version, $T_P \in 2^{\mathcal{TIME}}$ is the "new" pertinence to be assigned to $\mathcal{SV}$ and $\delta \in \Delta s$, then $place(\mathcal{SV}, T_P, \delta) = \delta'$ where for each $t \in$

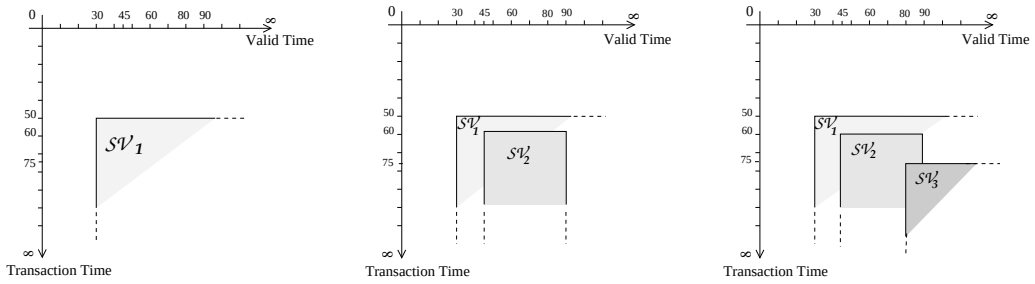


Fig. 2. Example of evolution of a database schema

$\mathcal{TIME}$:

$$\delta'(t) = \begin{cases} \mathcal{SV} & \text{if } t \in T_P \\ \delta(t) & \text{otherwise} \end{cases}$$

Definition 9 (Schema Update) *Let $\mathcal{SC}$ be the set of all possible schema changes, and $\mathcal{ES}s$ the set of all possible evolving schemata of the same type (valid-time, transaction-time or bitemporal schemata). The Schema Update is a function*

$$\mathcal{SU} : \mathcal{SC} \rightarrow (\mathcal{ES}s[\times \mathcal{TIME}_v \times 2^{\mathcal{TIME}_v}] \rightarrow \mathcal{ES}s)$$

where the part of the signature in square brackets is only required if the evolving schema is valid-time or bitemporal.

If $\mathcal{ES} = (\mathcal{CN}, \mathcal{AN}, \delta, \psi)$ is an evolving schema, m a schema change [and, if valid-time is supported, $\overline{ss} \in \mathcal{TIME}_v$ is the schema selection validity and $\overline{sc} \subseteq \mathcal{TIME}_v$ is the schema change validity], then

$$\mathcal{SU}(m)(\mathcal{ES}[, \overline{ss}, \overline{sc}]) = \mathcal{ES}' = (\mathcal{CN}', \mathcal{AN}', \delta', \psi')$$

where $\psi' = \psi \cup \{(m, \mathcal{SV}_i, \mathcal{SV}_j)\}$ and

- *if transaction time schema versioning is supported, then $\mathcal{SV}_i = \delta(now_t) \neq \emptyset$, $\mathcal{SV}_j = \mathcal{SVU}(m)(\mathcal{SV}_i)$, and $\delta' = place(\mathcal{SV}_j, [now_t, \infty]_t, \delta)$*
- *if valid time schema versioning is supported, $\mathcal{SV}_i = \delta(\overline{ss}) \neq \emptyset$, $\mathcal{SV}_j = \mathcal{SVU}(m)(\mathcal{SV}_i)$, and $\delta' = place(\mathcal{SV}_j, \overline{sc}, \delta)$*
- *if bitemporal schema versioning is supported, $\mathcal{SV}_i = \delta(now_t, \overline{ss}) \neq \emptyset$, $\mathcal{SV}_j = \mathcal{SVU}(m)(\mathcal{SV}_i)$, and $\delta' = place(\mathcal{SV}_j, [now_t, \infty]_t \times \overline{sc}, \delta)$*

Example 5 *The following table defines a schema update process based on the transactions described in Ex. 3: the corresponding transaction time (tt), the schema selection validity ($\overline{ss}$) and the schema change validity ($\overline{sc}$) are listed below.*

Transaction	*tt*	$\overline{ss}$	$\overline{sc}$
t1	50	–	$[30, \infty]$
t2	60	35	$[45, 90]$
t3	75	30	$[75, \infty]$

The evolution of the δ function after each transaction commit is shown in Fig. 2.

The first transaction result (not a schema change*) is:*

$$\mathcal{ES} = (\{employee, professor\}, \{name, ssn, deg\}, \psi, \delta)$$

where $\psi = \emptyset$ and $\delta : [50, \infty]_t \times [30, \infty]_v \mapsto \mathcal{SV}_1$ as displayed in Fig. 2(a).

For the second transaction, the Schema Update is:

$$\mathcal{SU}(\textbf{\textit{AddAttribute}}_{badge\_no:integer,employee})(\mathcal{ES}, 35, [45, 90])$$

where $\delta(now_t = 60, 35) = \mathcal{SV}_1$. The resulting evolving schema is:

$$\mathcal{ES}' = (\{employee, professor\}, \{name, ssn, deg, badge\_no\}, \psi', \delta')$$

where $\psi' = \{(\textbf{\textit{AddAttribute}}_{badge\_no:int,employee}, \mathcal{SV}_1, \mathcal{SV}_2)\}$ and (see Fig. 2(b)):

$$\delta' : [50, 59]_t \times [30, \infty]_v \cup [60, \infty]_t \times [30, 44]_v \cup [30, \infty]_t \times [91, \infty]_v \mapsto \mathcal{SV}_1$$
$$\delta' : [60, \infty]_t \times [45, 90]_v \mapsto \mathcal{SV}_2$$

For the third transaction, the Schema Update is

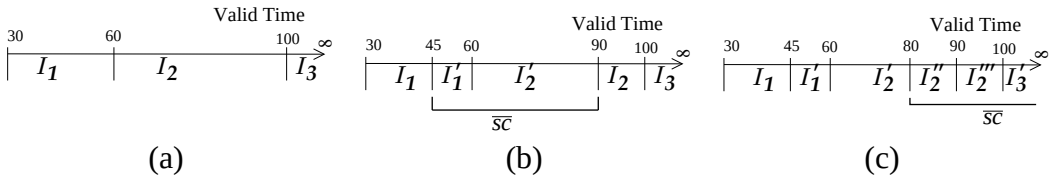
$$\mathcal{SU}(\textbf{\textit{AddClass}}_{course})(\mathcal{ES}', 30, [80, \infty])$$

where $\delta'(now_t = 75, 30) = \mathcal{SV}_1$ (see Fig. 2(c)). The resulting evolving schema was described in Ex. 3.

In a database management system, an evolving schema must be associated with a legal database. When it undergoes changes, its database must be modified too in order to become legal for the new evolving schema. The problem can be solved according to the following steps:

(1) locate the instance(s) of the database to be modified;
(2) transform those instances such that the entire database becomes legal for the transformed evolving schema.

As to the first point, let us compare an evolving schema before and after any schema change: the association of the time domain with schema versions which is encoded by δ and δ' is only modified in the portion overlapping the temporal pertinence



(a) (b) (c)

Fig. 3. Example of a valid-time database with evolving schema

of the newly introduced schema version (*modified portion* in the following, that is $[now_t, \infty]_t$ if transaction-time is supported times $\overline{sc}$ if valid-time is supported). In fact, these time points only are associated by δ' with the new schema version. Hence, the instance(s) of the database to be modified are all those which were legal for the schema versions $\mathcal{SV}_k$ previously associated with the modified portion. As to the second point, since the newly introduced schema version $\mathcal{SV}_j$ is the outcome of the application of the schema change m to the selected schema version $\mathcal{SV}_i$, we can transform the database instance underlying $\mathcal{SV}_k$ into a legal instance for $\mathcal{SV}_i$, $\mathcal{I}_{\mathcal{SV}_k \mapsto \mathcal{SV}_i}$, before the application of $\mathcal{IU}$, eventually yielding $\mathcal{IU}(m)(\mathcal{I}_{\mathcal{SV}_k \mapsto \mathcal{SV}_i})$, which is a legal instance for the new schema version $\mathcal{SV}_j$ as ensured by Lemma 2. This is globally equivalent to the evaluation of the formula $\mathcal{I}_{\mathcal{SV}_k \mapsto \mathcal{SV}_j}$, since $\mathcal{SV}_j$ is uniquely and directly reachable from $\mathcal{SV}_i$ through the applied schema change. The database update function $\mathcal{DBU}$ formalizes these steps: it receives the applied schema modification m, a database $\mathcal{DB}$ legal for the evolving schema $\mathcal{ES}$, the two parameters $\overline{ss}$ (schema selection validity) and $\overline{sc}$ (schema change validity), if required, and produces a database $\mathcal{DB}'$ which is legal for the evolving schema $\mathcal{ES}' = \mathcal{SU}(m)(\mathcal{ES}[, \overline{ss}, \overline{sc}])$:

$$\mathcal{DB}' = \mathcal{DBU}(m)(\mathcal{ES}, \mathcal{DB}[, \overline{ss}, \overline{sc}])$$

the composition of the new database $\mathcal{DB}'$ depends both on the kind of evolving schema $\mathcal{ES}$ and the kind of database $\mathcal{DB}$.

Example 6 *Let us assume the bitemporal evolving schema of Ex. 3 to interact with a* valid-time *database. We assume that, after the transaction* ***t1*** *which created the initial schema $\mathcal{SV}_1$, an initial database composed of three database instances, say $\mathcal{I}_1, \mathcal{I}_2$ and $\mathcal{I}_3$, has also been created. We assume the initial database to be defined as follows (see also Fig. 3(a)):*

$$\mathcal{DB} : [30, 59]_v \mapsto \mathcal{I}_1$$
$$\mathcal{DB} : [60, 99]_v \mapsto \mathcal{I}_2$$
$$\mathcal{DB} : [100, \infty]_v \mapsto \mathcal{I}_3$$

(for instance, it is sufficient that some data object has been inserted with validity $[30, 99]_v$ and some other object has been inserted with validity $[60, \infty]_v$). We further assume, for the sake of simplicity, that no modifications are made to the data instances after their creation.

We saw in Ex. 3 that transaction ***t2*** *selects $\mathcal{SV}_1$ for modification and produces $\mathcal{SV}_3$*

with validity $\overline{sc} = [45, 90]_v$. Owing to synchronous management, instances $\mathcal{I}_1$ and $\mathcal{I}_2$ have a modified portion overlapping $[45, 59]_v$ and $[60, 90]_v$, respectively, which has to adapted to the new schema $\mathcal{SV}_3$ valid in $[45, 90]_v$. The instance $\mathcal{I}_3$ and the unaffected portions of $\mathcal{I}_1$ and $\mathcal{I}_2$ remain unchanged. The result of the application of the $\mathcal{DBU}$ function is, in this case (see Fig. 3(b)):

$$\begin{aligned}
&\mathcal{DB}' : [30, 44]_v \mapsto \mathcal{I}_1 \\
&\mathcal{DB}' : [45, 59]_v \mapsto \mathcal{I}'_1 = (\mathcal{I}_1)_{\mathcal{SV}_1 \mapsto \mathcal{SV}_2} \\
&\mathcal{DB}' : [60, 90]_v \mapsto \mathcal{I}'_2 = (\mathcal{I}_2)_{\mathcal{SV}_1 \mapsto \mathcal{SV}_2} \\
&\mathcal{DB}' : [91, 99]_v \mapsto \mathcal{I}_2 \\
&\mathcal{DB}' : [100, \infty]_v \mapsto \mathcal{I}_3
\end{aligned}$$

The applied transformation which makes the modified portions of $\mathcal{I}_1$ and $\mathcal{I}_2$ legal with respect to their "new" schema is, in both cases: $\mathcal{I}'_i = (\mathcal{I}_i)_{\mathcal{SV}_1 \mapsto \mathcal{SV}_2} = \mathcal{IU}(\textbf{\textit{AddAttribute}}_{badge\_no:integer,employee})(\mathcal{I}_i)$ (for $i = 1, 2$).

*Transaction **t3** still selects $\mathcal{SV}_1$ for modification and produces $\mathcal{SV}_3$ with validity $\overline{sc} = [80, \infty]_v$. The synchronous management requires that the modified portion of all instances whose validity overlaps $\overline{sc}$ (i.e. $\mathcal{I}'_2, \mathcal{I}_2$ and $\mathcal{I}_3$) be made legal with respect to their "new" schema $\mathcal{SV}_3$. The effects of $\mathcal{DBU}$ are, thus (see Fig. 3(c)):*

$$\begin{aligned}
&\mathcal{DB} : [30, 44]_v \mapsto \mathcal{I}_1 \\
&\mathcal{DB} : [45, 59]_v \mapsto \mathcal{I}'_1 \\
&\mathcal{DB} : [60, 79]_v \mapsto \mathcal{I}'_2 \\
&\mathcal{DB} : [80, 90]_v \mapsto \mathcal{I}''_2 = (\mathcal{I}'_2)_{\mathcal{SV}_2 \mapsto \mathcal{SV}_3} \\
&\mathcal{DB} : [91, 99]_v \mapsto \mathcal{I}'''_2 = (\mathcal{I}_2)_{\mathcal{SV}_1 \mapsto \mathcal{SV}_3} \\
&\mathcal{DB} : [100, \infty]_v \mapsto \mathcal{I}'_3 = (\mathcal{I}_3)_{\mathcal{SV}_1 \mapsto \mathcal{SV}_3}
\end{aligned}$$

where the transformation applied to $\mathcal{I}'_2$ is the same as considered in Ex. 4, whereas $(\mathcal{I}_i)_{\mathcal{SV}_1 \mapsto \mathcal{SV}_3} = \mathcal{IU}(\textbf{\textit{AddClass}}_{course})(\mathcal{IU}(\textbf{\textit{AddAttribute}}_{badge\_no:integer,employee})(\mathcal{I}_i))$ (for $i = 2, 3$).

For a complete definition of the $\mathcal{DBU}$ behaviour, all the alternatives can be easily derived from the definition of legal database (see Tab. 4) and lead to a definition of $\mathcal{DBU}(m)(\mathcal{ES}, \mathcal{DB}[, \overline{ss}, \overline{sc}])$ as listed below:

(D) $\mathcal{DB}$ **is a snapshot database** In this case, $\mathcal{DB}$ has to be consistent with the current and present schema version. Therefore, $\mathcal{DB}$ is modified if and only if such a schema version is affected by the applied schema change m.

- **(tS)** $\mathcal{ES}$ is a transaction-time schema: $\mathcal{DB}' = \mathcal{IU}(m)(\mathcal{DB})$;
- **(vS)** $\mathcal{ES}$ is a valid-time schema: $\mathcal{DB}' = \begin{cases} \mathcal{DB}_{\mathcal{SV}_k \mapsto \mathcal{SV}_j} & \text{if } now_v \in \overline{sc},\ \mathcal{SV}_k = \delta(now_v) \\ \mathcal{DB} & \text{otherwise} \end{cases}$

  where $\mathcal{SV}_j = \mathcal{SVU}(m)(\delta(\overline{ss}))$;

- **(tvS)** $\mathcal{ES}$ is a bitemporal schema: $\mathcal{DB}' = \begin{cases} \mathcal{DB}_{\mathcal{SV}_k \mapsto \mathcal{SV}_j} & \text{if } now_v \in \overline{sc}, \\ & \mathcal{SV}_k = \delta(now_t, now_v) \\ \mathcal{DB} & \text{otherwise} \end{cases}$

  where $\mathcal{SV}_j = \mathcal{SVU}(m)(\delta(now_t, \overline{ss}))$;

(tD) $\mathcal{DB}$ **is a transaction-time database** In this case, we consider all transaction-time chronons and modify only those instances whose corresponding schema version has been modified by the application of m. For each $tt \in \mathcal{TIME}_t$, $\mathcal{I} = \mathcal{DB}(tt)$ and:

- **(tS)** $\mathcal{ES}$ is a transaction-time schema: $\mathcal{DB}'(tt) = \begin{cases} \mathcal{IU}(m)(\mathcal{I}) & \text{if } tt \geq now_t \\ \mathcal{I} & \text{otherwise} \end{cases}$
- **(vS)** $\mathcal{ES}$ is a valid-time schema: $\mathcal{DB}'(tt) = \begin{cases} \mathcal{I}_{\mathcal{SV}_k \mapsto \mathcal{SV}_j} & \text{if } tt \in \overline{sc}, \mathcal{SV}_k = \delta(tt) \\ \mathcal{I} & \text{otherwise} \end{cases}$

  where $\mathcal{SV}_j = \mathcal{SVU}(m)(\delta(\overline{ss}))$;
- **(tvS)** $\mathcal{ES}$ is a bitemporal schema: $\mathcal{DB}'(tt) = \begin{cases} \mathcal{I}_{\mathcal{SV}_k \mapsto \mathcal{SV}_j} & \text{if } tt \geq now_t, tt \in \overline{sc}, \\ & \mathcal{SV}_k = \delta(tt, tt) \\ \mathcal{I} & \text{otherwise} \end{cases}$

  where $\mathcal{SV}_j = \mathcal{SVU}(m)(\delta(now_t, \overline{ss}))$;

(vD) $\mathcal{DB}$ **is a valid-time database** In this case, we consider all valid-time chronons and modify only those instances whose corresponding schema version has been modified by the application of m. In particular, if transaction time is supported we consider now_t as parameter for the schema version selection whereas if valid time is supported we follow the synchronous approach. For each $vt \in \mathcal{TIME}_v$, $\mathcal{I} = \mathcal{DB}(vt)$ and:

- **(tS)** $\mathcal{ES}$ is a transaction-time schema: $\mathcal{DB}'(vt) = \mathcal{IU}(m)(\mathcal{I})$
- **(vS)** $\mathcal{ES}$ is a valid-time schema: $\mathcal{DB}'(vt) = \begin{cases} \mathcal{I}_{\mathcal{SV}_k \mapsto \mathcal{SV}_j} & \text{if } vt \in \overline{sc}, \mathcal{SV}_k = \delta(vt) \\ \mathcal{I} & \text{otherwise} \end{cases}$

  where $\mathcal{SV}_j = \mathcal{SVU}(m)(\delta(\overline{ss}))$;
- **(tvS)** $\mathcal{ES}$ is a bitemporal schema: $\mathcal{DB}'(vt) = \begin{cases} \mathcal{I}_{\mathcal{SV}_k \mapsto \mathcal{SV}_j} & \text{if } vt \in \overline{sc}, \\ & \mathcal{SV}_k = \delta(now_t, vt) \\ \mathcal{I} & \text{otherwise} \end{cases}$

  where $\mathcal{SV}_j = \mathcal{SVU}(m)(\delta(now_t, \overline{ss}))$;

(tvD) $\mathcal{DB}$ **is a bitemporal database** In this case, we simply follow a synchronous approach. For each $(tt, vt) \in \mathcal{TIME}$, $\mathcal{I} = \mathcal{DB}(tt, vt)$ and:

- **(tS)** $\mathcal{ES}$ is a transaction-time schema $\mathcal{DB}'(tt, vt) = \begin{cases} \mathcal{IU}(m)(\mathcal{I}) & \text{if } tt \geq now_t \\ \mathcal{I} & \text{otherwise} \end{cases}$
- **(vS)** $\mathcal{ES}$ is a valid-time schema $\mathcal{DB}'(tt, vt) = \begin{cases} \mathcal{I}_{\mathcal{SV}_k \mapsto \mathcal{SV}_j} & \text{if } vt \in \overline{sc}, \mathcal{SV}_k = \delta(vt) \\ \mathcal{I} & \text{otherwise} \end{cases}$

- **(tvS)** $\mathcal{ES}$ is a bitemporal schema $\mathcal{DB}'(tt, vt) = \begin{cases} \mathcal{I}_{\mathcal{SV}_k \mapsto \mathcal{SV}_j} & \text{if } vt \in \overline{sc},\ tt \geq now_t, \\ & \mathcal{SV}_k = \delta(now_t, vt) \\ \mathcal{I} & \text{otherwise} \end{cases}$

  where $\mathcal{SV}_j = \mathcal{SVU}(m)(\delta(now_t, \overline{ss}))$.

In the Theorem which follows, we eventually consider global correctness of an evolving schema. It stems from the fact that, for any kind of evolving schema and database, the application of a schema change leads to a transformed database which continues to be legal for the modified evolving schema.

Theorem 4 *Let $\mathcal{ES}$ be an evolving schema and $\mathcal{DB}$ a database legal for $\mathcal{ES}$. For each schema modification m, and parameters $\overline{ss}$ (schema selection validity) and $\overline{sc}$ (schema change validity) when required, the database $\mathcal{DBU}(m)(\mathcal{ES}, \mathcal{DB}, [, \overline{ss}, \overline{sc}])$ is legal for the new evolving schema $\mathcal{SU}(m)(\mathcal{ES}[, \overline{ss}, \overline{sc}])$.*

The proof directly follows from Lemma 3 and from the definitions of legal database for evolving schema and schema update function $\mathcal{SU}$.

6 Related Work and Discussion

The problems of schema evolution and schema versioning support have been widely studied in relational and object-oriented database papers: [17] 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 [2] and O_2 [1] systems. The second approach, which was proposed in [32] and developed in the context of the TIGUKAT [33], 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, which can be ascribed to four main approaches:

(1) *Immediate conversion (coercion)*: changes are propagated via immediate object conversion – used for instance in GemStone [34];
(2) *Deferred conversion (lazy updates, screening)*: changes are propagated via deferred object conversion – used for instance in ORION [2];
(3) *Filtering*: changes are never propagated; objects are indeed assigned to different schema versions according to their semantics – used for instance in CLOSQL [35];
(4) *Hybrid*: uses or combines two or more of the previous approaches – used for instance in Sherpa [36] and O_2 [1].

In any case, simple default mechanisms can be used, or user-supplied conversion functions must be defined for non-trivial extant object updates. The work [37] introduces an axiomatic model for change propagation which is capable of identifying in a declarative manner the set of objects affected by a schema change, that can serve, for instance, as input to any available object conversion method.

Instead of relying on automatic re-organisation of the data after the schema change, in [38] a declarative *instance update programming language* is proposed to be used in combination with schema updates. A remarkable advantage of this approach is that it is based on a formal notion of consistency, which provides the user with a decidable static consistency checking mechanism to validate the schema and extant data modifications.

A completely different approach is taken in [39], where algorithms were devised to analyse complex type changes by comparing two schema versions and accordingly derive transformation rules that can applied to propagate the changes to extant objects.

The work done by the research group(s) led by Elke A. Rundensteiner deserves a separate mention, as she has been in recent years one of the most active scientists in the schema evolution and versioning field. The primary goal of her research in this context was to develop transparent schema change technology that allows on-line modification of databases without disturbing existing applications [40]. Ongoing projects include the study of an extensible, re-usable and flexible framework based on the integration of a fixed set of invariant-preserving primitive change operations (with the standard object query language OQL as the vehicle for flexible object migration) [41], and the optimization of complex sequences of schema evolution operations [42].

Finally, also our previous work [43,44] concerned a formal characterization of the schema evolution process in an object-oriented database. We formalized the notion

of schema version and the interschema relationships induced by schema changes using an encoding in Description Logics [45]. We introduced interesting reasoning tasks concerning the check of different types of consistency defined at local (i.e. single schema version) or global (i.e. complete database) level, which can be solved using the inference engine of the Description Logic. However, we did not consider the change propagation problem in [43,44]: we actually assumed dealing with a single database instance, compatible with every derivable schema version, as the only way to ensure portability of applications compiled with past schema versions. An extreme consequence of such an approach is the introduction of a strong notion of "monotonicity": all the legal instances of the schema version resulting from a schema change were also legal with respect to the schema version which has been modified, so that there is no need for change propagation at all. Although such a framework is suitable to describe some progressive "schema refinement" process, it is unable to capture what it is usually meant (also in the present paper) by schema evolution and versioning.

With respect to previous work, the present paper deals with the "semantics of change" problem with the definition of a schema update process which has been proved correct (Theorem 1) and is consistent with the invariant-based approach. The proposed solution for the "change propagation" problem relies in our model on a simple coercion mechanism (e.g. based on the introduction of *nulls* for newly added object attributes), which has been proved correct with respect to the schema update process (Theorem 2). Moreover, whereas temporal schema versioning, also in the presence of temporal data, has been previously considered in the relational database field (e.g. in [16], where the principle of *synchronous management* has been introduced), this is the first attempt to address, on a sound formal basis, the problem in the context of the object-oriented data model. To this end, we considered in $OODM|_{\mathcal{SV}}$ the interaction between the intensional and extensional versioning levels, and introduced the notions of evolving schema and of legal database with respect to an evolving schema. We addressed the issue of "reversibility" (at schema level) for schema changes and defined appropriate inverse modifications (Theorem 3) to deal with multiple schema versions in a unified framework. The proposed solutions for managing the global effects of schema changes have been proved consistent with respect to the legality maintenance requirements (Theorem 4).

It should be noted that we considered in this paper only the impact of schema changes on an evolving schema and not the pure management of *extensional* data versions. On the other hand, extensional data versioning has been dealt with formally in several other object models, where different data versions are allowed to coexist in the presence of a fixed schema (e.g. extensional versioning considered in temporal databases [11] or in multiversion databases [46]). If we want to add temporal schema versioning to such models, our approach can be used to enforce consistency constraints between the intensional and extensional levels. In particular, if a temporal database is considered, such constraints are exactly those studied in Section 4; if a non-temporal multiversion database is considered, we can apply

the results of Section 4 for the case of snapshot database, since the semantics of time has no further impact on the interaction between the two levels.

As far as turning our $OODM|_{SV}$ model into a working system is concerned, the semantics proposed for the schema change and propagation of change operations could serve as a guideline for their correct implementation: since semantics is given in a functional form, the operations can be implemented in a straightforward way. Obviously, although such an approach would lead to an effective implementation which is guaranteed consistency-preserving owing to the results proved in this paper, to achieve efficiency is a different kind of problem. As a first step, the semantics of operations has been carefully defined in Section 3 as acting on the minimal required portion of the schema (e.g. only in the **DeleteSuperclass** case the whole class hierarchy has to be recomputed). However, a fully optimized implementation of operations, which is also strictly dependent on an optimized physical design of the data structures of the underlying database, was clearly beyond the scope of this paper and will be addressed in our future work. The main purpose of the present work was to guarantee that any possible implementation of temporal schema versioning were free from malfunctionings due to a non careful specification of operations. Therefore, future implementation of our schema versioning solutions will be grounded on the solid foundations provided by the theoretical framework introduced here.

7 Conclusions

In this paper we presented $OODM|_{SV}$, a formal model for the management of temporal schema versioning in the context of a possibly temporal object-oriented database. An operational semantics for the available schema change primitives has been provided and its correctness property have been investigated. $OODM|_{SV}$ is also provided with an embedded mechanism to translate data from a given schema version into another, which is used to correctly propagate schema changes to extant data after schema changes by preserving global temporal integrity (following a synchronous management approach). The same mechanism can also be used to answer queries in the presence of multiple schemata, which is a major requirement of schema versioning support (e.g. for the reuse of legacy applications or for auditing purposes).

References

[1] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, J. Madec, Database Evolution in the O_2 Object Database System, in: Proc. of the 21st Int. Conf. on Very Large Databases (VLDB), Zurich, Switzerland, 1995, pp. 170–181.

[2] J. Banerjee, W. Kim, H.-J. Kim, H. F. Korth, Semantics and Implementation of Schema Evolution in Object-Oriented Databases, in: Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD), San Francisco, CA, 1987, pp. 311–322.

[3] R. Breitl, The GemStone Data Management System, in: W. Kim, F. Lochovsky (Eds.), Object-Oriented Concepts, Databases, and Applications, Addison-Wesley, Reading MA, 1989, pp. 283–308.

[4] W. Kim, H.-T. Chou, Versions of Schema for Object-Oriented Databases, in: Proc. of the 14th Int. Conf. on Very Large Databases (VLDB), Los Angeles, CA, 1988, pp. 148–159.

[5] K. R. Dittrich, R. A. Lorie, Version Support for Engineering Database Systems, IEEE Transactions on Software Engineering 14 (4) (1988) 429–436.

[6] S.-E. Lautemann, Schema Versioning in Object-Oriented Database Systems, in: Proc. of the 5th Int. Conf. on Database Systems for Advanced Applications (DASFAA), Melbourne, Australia, 1997, pp. 323–332.

[7] R. Newell, D. Theriault, M. Easterfieldy, Temporal GIS - Modeling The Evolution of Spatial Data in Time, Computers and Geosciences 18 (4) (1992) 427–434.

[8] G. Faria, C. Bauzer Medeiros, M. Nascimento, An Extensible Framework for Spatio-Temporal Database Applications, in: Proc. of 10th Int. Conf. on Scientific and Statistical Database Management (SSDBM), IEEE Computer Society, Capri, Italy, 1998, pp. 202–205.

[9] W. W. Chu, I. T. Ieong, R. K. Taira, C. M. Breant, Temporal Evolutionary Object-Oriented Data Models and Its Query Language for Medical Image Management, in: Proc. of the 18th Int. Conf. on Very Large Databases (VLDB), Vancouver, Canada, 1992, pp. 53–64.

[10] Y. Masunaga, An Object-Oriented Apporach to Temporal Multimedia Data Modeling, IEICE Transactions on Information and Systems E78-D (11).

[11] E. Bertino, E. Ferrari, G. Guerrini, T-Chimera: A Temporal Object-Oriented Data Model, Theory And Practice Of Object Systems 3 (2) (1997) 103–125.

[12] I. A. Goralwalla, M. T. Özsu, Temporal Extensions to a Uniform Behavioral Object Model, in: R. A. Elmasri, V. Kuramajian, B. Thalheim (Eds.), Entity-Relationship Approach — ER'93, Springer-Verlag, 1993, pp. 110–121, lNCS No. 823.

[13] W. Käfer, H. Schöning, Realizing a Temporal Complex-Object Data Model, in: Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD), San Diego, CA, 1992, pp. 266–275.

[14] E. Rose, A. Segev, TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints, in: Proc. of the 10th Int. Conf. on Entity-Relationship Approach (ER), San Mateo, CA, 1991, pp. 205–229.

[15] G. T. Wuu, U. Dayal, A Uniform Model for Temporal Object-Oriented Databases, in: Proc. of the 8th IEEE Int. Conf. on Data Engineering (ICDE), Tempe, AZ, 1992, pp. 584–593.

[16] C. De Castro, F. Grandi, M. R. Scalas, Schema Versioning for Multitemporal Relational Databases, Information Systems 22 (5) (1997) 249–290.

[17] J. F. Roddick, A Survey of Schema Versioning Issues for Database Systems, Information and Software Technology 37 (7) (1995) 383–393.

[18] L. Cardelli, A Semantics of Multiple Inheritance, in: Proc. of the Int. Symp. on Semantics of Datatypes, Sophia-Antipolis, France, 1984, pp. 51–67.

[19] C. Gunter, J. Mitchell, Theoretical Aspects of Object-Oriented Programming, MIT Press, Boston, MS, 1994.

[20] S. Abiteboul, P. Kanellakis, Object Identity as a Query Language Primitive, Journal of the ACM 45 (5) (1998) 798–842, a first version appeared in SIGMOD'89 Proceedings.

[21] C. Beeri, Formal Models for Object Oriented Databases, in: Proc. of the 1st Int. Conf. on Deductive and Object-Oriented Databases (DOOD), Kyoto, Japan, 1989, pp. 370–395.

[22] C. Beeri, T. Milo, Subtyping in OODBs, Journal of Computer and System Sciences 51 (2) (1995) 223–243.

[23] F. Mandreoli, Schema Versioning in Object-Oriented Databases, Ph.D. thesis, Department of Electronics Computer Science and Systems, University of Bologna (Mar. 2001).

[24] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, Reading, MA, 1995.

[25] P. Brèche, M. Wörner, How to Remove a Class in an Object Database System, in: Proc. of the 2nd Int. Conf. of Applications of Databases (ADB), San Jose, CA, 1995, pp. 476–495.

[26] S. Scherrer, A. Geppert, K. R. Dittrich, Schema Evolution in NO^2, Tech. Rep. 93.12, Institut für Informatik der Universität Zürich (1993).

[27] M. Tresch, M. H. Scholl, Meta Object Management and its Application to Database Evolution, in: Proc. of the 11th Int. Conf. on the Entity Relationship Approach (ER), Karlsruhe, Germany, 1992, pp. 299–321.

[28] C. S. Jensen, C. E. Dyreson, (Eds.), M. Böhlen, J. Clifford, R. A. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. K. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsoupoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. U. Tansel, P. Tiberio, G. Wiederhold, The Consensus Glossary of Temporal Database Concepts - February 1998 Version, in: O. Etzion, S. Jajodia, S. Sripada (Eds.), Temporal Databases — Research and Practice, Springer-Verlag, 1998, pp. 367–405, lNCS No. 1399.

[29] C. S. Jensen, M. D. Soo, R. T. Snodgrass, Unifying Temporal Data Models via a Conceptual Model, Information Systems 19 (7) (1994) 513–547.

[30] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, R. T. Snodgrass, On the Semantics of "Now" in Databases, ACM Transactions on Database Systems 22 (2) (1997) 171–214.

[31] C. De Castro, F. Grandi, M. R. Scalas, Semantic Interoperability of Multitemporal Relational Data, in: R. A. Elmasri, V. Kuramajian, B. Thalheim (Eds.), Entity-Relationship Approach — ER'93, Springer-Verlag, Berlin, 1993, pp. 463–474, lNCS No. 823.

[32] R. J. Peters, M. T. Özsu, An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems, ACM Transactions on Database Systems 22 (1) (1997) 75–114.

[33] M. T. Özsu, R. J. Peters, D. Szafron, B. Irani, A. Lipka, A. Muñoz, TIGUKAT: A Uniform Behavioral Objectbase Management System, The VLDB Journal 4 (3) (1995) 445–492, special issue on persistent object systems.

[34] D. J. Penney, J. Stein, Class Modification in the GemStone Object-Oriented DBMS, in: Proc. of the Int. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Orlando, FL, 1987, pp. 111–117.

[35] S. Monk, I. Sommerville, A Model for Versioning of Classes in Object-Oriented Databases, in: Proc. of the 10th British Nat. Conf. of Databases (BNCOD), Aberdeen, Scotland, 1992, pp. 42–58.

[36] G. Nguyen, D. Rieu, Schema Evolution in Object-oriented Database Systems, Data and Knowledge Engineering 4 (1989) 43–67.

[37] R. J. Peters, K. Barker, Change propagation in an Axiomatic Model of Schema Evolution for Objectbase Management Systems, in: H. Balsters, B. de Brock, S. Conrad (Eds.), Database Schema Evolution and Meta-Modeling — Proc. Intl' Workshop on Foundations of Models and Languages for Data and Objects, FoMLaDO/DEMM 2000, Selected Papers, no. 2065 in LNCS, Springer-Verlag, 2001, pp. 142–162.

[38] J. Lagorce, A. Stockus, E. Waller, Object-Oriented Database Evolution, in: Proc. of the 6th Int. Conf. on Database Theory (ICDT), Delphi, Greece, 1997, pp. 379–393.

[39] B. Staudt Lerner, A Model for Compound Type Changes Encountered in Schema Evolution, ACM Transactions on Database Systems 25 (1) (2000) 83–127.

[40] Y.-G. Ra, E. A. Rundensteiner, A Transparent Schema-Evolution System Based on Object-Oriented View Technology, IEEE Transactions on Knowledge and Data Engineering 9 (4) (1997) 600–624.

[41] H. Su, K. T. Claypool, E. A. Rundensteiner, Extending the Object Query Language for Transparent Metadata Access, in: H. Balsters, B. de Brock, S. Conrad (Eds.), Database Schema Evolution and Meta-Modeling — Proc. Intl' Workshop on Foundations of Models and Languages for Data and Objects, FoMLaDO/DEMM 2000, Selected Papers, no. 2065 in LNCS, Springer-Verlag, 2001, pp. 68–84.

[42] K. T. Claypool, C. Natarajan, E. A. Rundensteiner, Optimizing Performance of Schema Evolution Sequences, in: Proc. of the Intl' Symposium on Objects and Databases, Sophia Antipolis, France, 2000, pp. 114–127.

[43] E. Franconi, F. Grandi, F. Mandreoli, A Semantic Approach for Schema Evolution and Versioning in Object-Oriented Databases, in: Proc. of the 6th Intl' Conf. on Rules

and Objects in Databases (DOOD) as a stream of the 1st Intl' Conf. on Computational Logic (CL), no. 1861 in LNAI, Springer-Verlag, London, UK, 2000, pp. 1048–1062.

[44] E. Franconi, F. Grandi, F. Mandreoli, Schema Evolution and Versioning: a Logical and Computational Characterisation, in: S. C. Herman Balsters, Bert de Brock (Ed.), Database Schema Evolution and Meta-Modeling — Proc. Intl' Workshop on Foundations of Models and Languages for Data and Objects, FoMLaDO/DEMM 2000, Selected Papers, no. 2065 in LNCS, Springer-Verlag, 2001, pp. 85–99.

[45] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, Reasoning in Expressive Description Logics, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning – Vol. II, Elsevier, 2001, pp. 1581–1634.

[46] S. Gançarski, G. Jomier, A Framework for Programming Multiversion Databases, dke 36 (1) (2001) 29–53.

A Conversion Functions for Types and Values

A.1 The Type Conversion Function τ_C

The τ_C function calculates the type of the class C based on the class hierarchy in $(\mathcal{CNV}, \sigma, \preceq)$:

$$\tau_C(C, (\mathcal{CNV}, \sigma, \preceq)) = \sigma(C) \sqcap \sigma(C_1) \sqcap \cdots \sqcap \sigma(C_n)$$

where $\{C_1, \ldots, C_n\} = >_C$.

The symbol $\sqcap$ denotes a partial function which is defined as follows:

(1) if $\tau, \tau' \in \mathcal{LT}$ or $\tau, \tau' \in \mathcal{CNV}$:

$$\tau \sqcap \tau' = \begin{cases} \tau & \text{if } \tau \leq \tau' \\ \tau' & \text{if } \tau' \leq \tau \\ \bot & \text{otherwise} \end{cases}$$

(2) if $\tau = \{\tau_i\}$ and $\tau' = \{\tau'_i\}$:

$$\tau \sqcap \tau' = \begin{cases} \{\tau_i \sqcap \tau'_i\} & \text{if } \tau \sqcap \tau' \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

The bag and list cases can be treated analogously;

(3) if $\tau = \langle A_i : \tau_i \mid i \in I \rangle$ and $\tau' = \langle A_i : \tau'_i \mid i \in I' \rangle$:

- $\tau \sqcap \tau' = \langle A_i : \tau \mid (i \in (I \setminus I') \wedge \tau = \tau_i) \vee (i \in (I' \setminus I) \wedge \tau = \tau'_i)$
$\vee (i \in (I \cap I') \wedge \tau = \tau_i \sqcap \tau'_i)\rangle$
if for each $i \in I \cap I'$: $\tau_i \sqcap \tau'_i \neq \bot$;
- $\tau \sqcap \tau' = \bot$, otherwise

(4) if $\tau = \mathbf{any}$ then $\tau \sqcap \tau' = \tau'$; if $\tau' = \mathbf{any}$ then $\tau \sqcap \tau' = \tau$;
(5) otherwise, $\tau \sqcap \tau' = \bot$.

Notice that, since $\sqcap$ is commutative and associative, we can simply write $C_1 \sqcap \ldots \sqcap C_n$ as $\sqcap\{C_1, \ldots, C_n\}$.

A.2 The Instance Conversion Function $\mathcal{I}_C$

The $\mathcal{I}_C$ function, given a schema version $(\mathcal{CNV}, \sigma, \preceq)$, transforms the value v which is legal for the type τ into a new value v' legal for τ'. It can be recursively defined as follows:

$$\mathcal{I}_C(v, \tau, \tau', (\mathcal{CNV}, \sigma, \preceq)) = v'$$

where

(1) if $\tau = \tau'$ then $v' = v$;
(2) if $\tau' < \tau$ then
- if $\tau, \tau' \in \mathcal{CNV}$ then $v' = null$;
- if $\tau, \tau' \in \mathcal{LT}$ then $v' = v$;
- if $\tau = \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle$, $\tau' = \langle A_1 : \tau'_1, \ldots, A_n : \tau'_n, \ldots, A_m : \tau'_m\rangle$ with $\tau'_k \leq \tau_k$, for $k \in [1, n]$ and $v = \langle A_1 : v_1, \ldots, A_n : v_n\rangle$ then $v' = \langle A_1 : v'_1, \ldots, A_n : v'_n, A_{n+1} : null, \ldots, A_m : null\rangle$, where $v'_k = \mathcal{I}_C(v_k, \tau_k, \tau'_k, (\mathcal{CNV}, \sigma, \preceq))$, for $k \in [1, n]$;
- if $\tau = \{\overline{\tau}\}$, $\tau' = \{\overline{\tau}'\}$ and $v = \{v_1, \ldots, v_n\}$, then $v' = \{v'_1, \ldots, v'_n\}$ where $v'_k = \mathcal{I}_C(v_k, \overline{\tau}, \overline{\tau}', (\mathcal{CNV}, \sigma, \preceq))$, for $k \in [1, n]$ and (v' is similarly defined also for bag and list types);

(3) if $\tau' > \tau$ then
- if $\tau, \tau' \in \mathcal{CNV}$ then $v' = v$;
- if $\tau, \tau' \in \mathcal{LT}$ then $v' = v$;
- if $\tau' = \langle A_1 : \tau'_1, \ldots, A_n : \tau'_n\rangle$, $\tau = \langle A_1 : \tau_1, \ldots, A_n : \tau_n, \ldots, A_m : \tau_m\rangle$ with $\tau_k \leq \tau'_k$, for $k \in [1, n]$ and $v = \langle A_1 : v_1, \ldots, A_n : v_n, \ldots, A_m : v_m\rangle$, then $v' = \langle A_1 : v'_1, \ldots, A_n : v'_n\rangle$, where $v'_k = \mathcal{I}_C(v_k, \tau_k, \tau'_k, (\mathcal{CNV}, \sigma, \preceq))$, for $k \in [1, n]$;
- if $\tau = \{\overline{\tau}\}$, $\tau' = \{\overline{\tau}'\}$ and $v = \{v_1, \ldots, v_n\}$, then $v' = \{v'_1, \ldots, v'_n\}$ where $v'_k = \mathcal{I}_C(v_k, \overline{\tau}, \overline{\tau}', (\mathcal{CNV}, \sigma, \preceq))$, for $k \in [1, n]$ (v' is similarly defined also for bag and list types);

(4) if $\tau \sqcap \tau' = \bot$ then $v' = null$.

In particular, if τ, τ' are both record types and τ' contains more attributes than τ (as it happens when adding new properties), then v' contains a null value for each new attribute, otherwise if τ' contains less attributes than τ (as it happens when deleting properties), then v' no longer contains the values of the attributes which are not present in τ'. Finally, since the extension of a class contains the extension of all its subclasses, if τ, τ' are both object types (i.e. classes) and $\tau' > \tau$ then v' is equal to v.

B Proofs

Proof of Lemma 1. Let us consider all the cases in the $\sqcap$ operation definition which do not originate $\bot$ as outcome. The proof is by induction on the structure of types.

Base cases: Let $\tau, \alpha \in \mathcal{LT}$ and $\tau \leq \tau'$, $\alpha \leq \alpha'$, then $\tau = \tau' \in \mathcal{LT}$ and $\alpha = \alpha' \in \mathcal{LT}$. Moreover $\tau \sqcap \alpha \neq \bot$ if $\tau = \alpha$ (i.e. $\tau \sqcap \alpha = \tau$). Analogously, $\tau' \sqcap \alpha' \neq \bot$ if $\tau' = \alpha'$. In this case $\tau \sqcap \alpha \leq \tau' \sqcap \alpha'$ since $\tau \sqcap \alpha = \tau = \tau' = \tau' \sqcap \alpha'$.

Let $\tau, \alpha \in \mathcal{CNV}$ and $\tau \leq \tau'$, $\alpha \leq \alpha'$, then $\tau = C, \tau' = C', \alpha = D, \alpha' = D' \in \mathcal{CNV}$ and $C \preceq C'$ and $D \preceq D'$. Then $C \sqcap D \neq \bot$ in the two following cases:

$$C \sqcap D = \begin{cases} C & \text{if } C \preceq D \\ D & \text{if } D \preceq C \ . \end{cases}$$

Analogously,

$$C' \sqcap D' = \begin{cases} C' & \text{if } C' \preceq D' \\ D' & \text{if } D' \preceq C' \ . \end{cases}$$

Let us consider the four non-bottom outcomes:

- if $C \sqcap D = C$ and $C' \sqcap D' = C'$, then $C \preceq D$ and $C' \preceq D'$ and $C \sqcap D \leq C' \sqcap D'$ since $C \sqcap D = C \preceq C' = C' \sqcap D'$ by hypothesis;
- if $C \sqcap D = C$ and $C' \sqcap D' = D'$, then $C \preceq D$ and $D' \preceq C'$ and $C \sqcap D \leq C' \sqcap D'$ since $C \sqcap D = C \preceq D \preceq D' = C' \sqcap D'$ by hypothesis;
- if $C \sqcap D = D$ and $C' \sqcap D' = C'$, then $D \preceq C$ and $C' \preceq D'$ and $C \sqcap D \leq C' \sqcap D'$ since $C \sqcap D = D \preceq C \preceq C' = C' \sqcap D'$ by hypothesis;
- if $C \sqcap D = D$ and $C' \sqcap D' = D'$, then $D \preceq C$ and $D' \preceq C'$ and $C \sqcap D \leq C' \sqcap D'$ since $C \sqcap D = D \preceq D' = C' \sqcap D'$ by hypothesis.

Induction cases: Let $\tau = \{\overline{\tau}\}$, $\alpha = \{\overline{\alpha}\}$, $\tau' = \{\overline{\tau'}\}$ and $\alpha' = \{\overline{\alpha'}\}$. Suppose that $\tau \leq \tau'$ and $\alpha \leq \alpha'$ and that if $\overline{\tau} \leq \overline{\tau'}$ and $\overline{\alpha} \leq \overline{\alpha'}$ then $\overline{\tau} \sqcap \overline{\alpha} \leq \overline{\tau'} \sqcap \overline{\alpha'}$. We have to show that $\{\overline{\tau}\} \sqcap \{\overline{\alpha}\} \leq \{\overline{\tau'}\} \sqcap \{\overline{\alpha'}\}$.
The expression $\{\overline{\tau}\} \sqcap \{\overline{\alpha}\}$ produces non-bottom outcome $\{\overline{\tau} \sqcap \overline{\alpha}\}$ if $\overline{\tau} \sqcap \overline{\alpha} \neq \bot$, in the same way the expression $\{\overline{\tau'}\} \sqcap \{\overline{\alpha'}\}$ produces non-bottom outcome $\{\overline{\tau'} \sqcap \overline{\alpha'}\}$ if $\overline{\tau'} \sqcap \overline{\alpha'} \neq \bot$. Then $\{\overline{\tau}\} \sqcap \{\overline{\alpha}\} = \{\overline{\tau} \sqcap \overline{\alpha}\} \leq \{\overline{\tau'} \sqcap \overline{\alpha'}\} = \{\overline{\tau'}\} \sqcap \{\overline{\alpha'}\}$. In

fact, $\{\overline{\tau}\} \leq \{\overline{\tau'}\}$ if $\overline{\tau} \leq \overline{\tau}'$ and $\{\overline{\alpha}\} \leq \{\overline{\alpha'}\}$ if $\overline{\alpha} \leq \overline{\alpha}'$. Therefore by induction hypothesis $\overline{\tau} \sqcap \overline{\alpha} \leq \overline{\tau'} \sqcap \overline{\alpha'}$. From the definition of subtyping relationship, it follows that $\{\overline{\tau} \sqcap \overline{\alpha}\} \leq \{\overline{\tau'} \sqcap \overline{\alpha'}\}$.

Let $\tau = \langle A_i : \tau_i \mid i \in I\rangle$, $\alpha = \langle A_j : \alpha_j \mid j \in J\rangle$, $\tau' = \langle A_i : \tau'_i \mid i \in I'\rangle$, $\alpha' = \langle A_j : \alpha'_j \mid j \in J'\rangle$. Suppose that $\tau \leq \tau'$ and $\alpha \leq \alpha'$ and that if $\tau_i \leq \tau'_j$ and $\alpha_k \leq \alpha'_h$ ($i \in I$, $j \in J$, $k \in I'$, $h \in J'$) then $\tau_i \sqcap \alpha_k \leq \tau'_j \sqcap \alpha'_h$. We have to show that $\tau \sqcap \alpha \leq \tau' \sqcap \alpha'$.
From the first assertion it follows that

$$I' \subseteq I \tag{B.1}$$
$$\tau_i \leq \tau'_i \qquad \text{for each } i \in I' \tag{B.2}$$
$$J' \subseteq J \tag{B.3}$$
$$\alpha_j \leq \alpha'_j \qquad \text{for each } j \in J' \ . \tag{B.4}$$

The expression $\tau \sqcap \alpha$ coincides with

$$\langle A_i : \beta_i \mid (i \in I \setminus J \wedge \beta_i = \tau_i) \vee (i \in J \setminus I \wedge \beta_i = \alpha_i) \vee (i \in I \cap J \wedge \beta_i = \tau_i \sqcap \alpha_i)\rangle.$$

Analogously,

$$\tau' \sqcap \alpha' = \langle A_i : \beta'_i \mid (i \in I' \setminus J' \wedge \beta'_i = \tau'_i) \vee (i \in J' \setminus I' \wedge \beta'_i = \alpha'_i) \vee (i \in I' \cap J' \wedge \beta'_i = \tau'_i \sqcap \alpha'_i)\rangle.$$

Then $\tau \sqcap \alpha \leq \tau' \sqcap \alpha'$ if

(1) $I' \cup J' \subseteq I \cup J$,
(2) $\forall i \in I' \cup J'$: $\beta_i \leq \beta'_i$.

As to condition 1, it follows from the fact that $I' \subseteq I$ and $J' \subseteq J$.
As to condition 2, we consider three options about $i \in I' \cup J'$:

(1) if $i \in I' \setminus J'$, then $\beta'_i = \tau'_i$;
(2) if $i \in J' \setminus I'$, then $\beta'_i = \alpha'_i$;
(3) if $i \in I' \cap J'$, then $\beta'_i = \tau'_i \sqcap \alpha'_i$.

If condition 1 is verified, then $i \in I$ by equation B.1. If $i \in I \setminus J$ then $\beta_i = \tau_i$ and $\beta_i = \tau_i \leq \tau'_i = \beta'_i$ by equation B.2. Otherwise, if $i \in I \cap J$ then $\beta_i = \tau_i \sqcap \alpha_i$. Since $\tau_i \leq \tau'_i$ by equation B.2 and $\alpha_i \leq \mathbf{any}$, by induction hypothesis $\beta_i = \tau_i \sqcap \alpha_i \leq \tau'_i \sqcap \mathbf{any} = \tau'_i = \beta'_i$. Condition 2 can be treated analogously.
If condition 3 is verified, then $i \in I \cap J$ by equations B.1 and B.3 and $\beta_i = \tau_i \sqcap \alpha_i$. Since $\tau_i \leq \tau'_i$ by equation B.2 and $\alpha_i \leq \alpha'_i$ by equation B.4, by induction hypothesis $\beta_i = \tau_i \sqcap \alpha_i \leq \tau'_i \sqcap \alpha'_i = \beta'_i$. □

Proof of Corollary 1. Let $(\mathcal{CNV}, \sigma, \preceq)$ be a schema version, $C \in \mathcal{CNV}$, and $C_i \in >_C$. Then $\tau_C(C, (\mathcal{CNV}, \sigma, \preceq)) = \sqcap\{\sigma(C), \sigma(C_1), \ldots, \sigma(C_n)\}$ where $\{C_1, \ldots, C_n\} =$

$>_C$ and $\sqcap\{\sigma(C), \sigma(C_1), \ldots, \sigma(C_n)\} = \sigma(C) \sqcap \sigma(C_1) \sqcap \ldots \sqcap \sigma(C_n)$.
As to assertion 1, it follows from the fact that $C \leq C_i$ and $C_j \leq$ **any** for each $C_j \in >_C$. In fact, by Lemma 1 which states that the $\sqcap$ operator is monotonic and since $\sqcap$ is also associative $\sigma(C) \sqcap \sigma(C_1) \sqcap \ldots \sqcap \sigma(C_n) \leq \sigma(C_i) \sqcap$ **any** $\sqcap \ldots \sqcap$ **any** $= \sigma(C_i)$.
As to assertion 2, it follows from the fact that $>_{C_i} \subset >_C$, since the relation $\leq$ is transitive, and from Lemma 1. In fact, let $\tau_C(C_i, (\mathcal{CNV}, \sigma, \preceq)) = \sqcap\{\sigma(C_i), \sigma(D_1), \ldots, \sigma(D_m)\}$ where $\{D_1, \ldots, D_m\} = >_{C_i}$, then $\sigma(C) \leq \sigma(C_i)$; for each $D_k \in >_{C_i}$, exists $C_h \in >_C$ such that $D_k = C_h$, that is $\sigma(D_k) = \sigma(C_h)$, and $\sigma(C_j) \leq$ **any** for each $C_j \in (>_C \setminus >_{C_i})$. It follows that $\tau_C(C, (\mathcal{CNV}, \sigma, \preceq)) = \sigma(C) \sqcap \{\sigma(C_h) \mid \exists D_k \in >_{C_i}\colon D_k = C_h\} \sqcap \{\sigma(C_j) \mid C_j \in (>_C \setminus >_{C_i})\} \leq \sigma(C_i) \sqcap \{\sigma(D_k) \mid D_k \in >_{C_i}\} \sqcap$ **any** $= \tau_C(C_i, (\mathcal{CNV}, \sigma, \preceq))$. □

Proof of Theorem 1. For each schema modification m, we have to show that for each pair of classes $C_i, C_j \in \mathcal{CNV}'$ such that $C_i \preceq' C_j$ then $\sigma'(C_i) \leq' \sigma'(C_j)$.

Let us first consider the *changes to a class type*. In this case a single class $\overline{C}$ is subject to changes and $\mathcal{CNV}' = \mathcal{CNV}$, $\preceq' = \preceq$, therefore $\leq'=\leq$. Regardless of the specific schema change, the semantics in Tab. 2 ensures that $\sigma'(C) \neq \sigma(C)$ only for $\overline{C}$ and the classes C which are subclasses of $\overline{C}$. We therefore consider the following alternatives deriving from the combinations of the relationships $\preceq$, $\not\preceq$ and = between C_i, C_j and $\overline{C}$:

$$C_i \not\preceq \overline{C} \text{ and } C_j \not\preceq \overline{C} \tag{B.1}$$
$$C_i \preceq \overline{C} \text{ and } C_j \not\preceq \overline{C} \tag{B.2}$$
$$C_i \preceq \overline{C} \text{ and } C_j \preceq \overline{C} \tag{B.3}$$
$$C_i = \overline{C} \tag{B.4}$$
$$C_j = \overline{C}. \tag{B.5}$$

Notice that the combination $C_i \not\preceq \overline{C}$ and $C_j \preceq \overline{C}$ is not possible, since $C_i \preceq C_j$, $C_j \preceq \overline{C}$ and $\preceq$ is a transitive relation. Moreover, if $C_i = \overline{C}$, the relationship between C_j and $\overline{C}$ is uniquely defined since $C_i \preceq C_j$. The same happens when $C_j = \overline{C}$.

AddAttribute: If condition B.1 holds, then $\sigma'(C_i) \overset{C_i \not\preceq \overline{C}}{=} \overline{\sigma}(C_i) \overset{C_i \not\preceq \overline{C}}{=} \sigma(C_i) \leq \sigma(C_j) = \overline{\sigma}(C_j) \overset{C_j \not\preceq \overline{C}}{=} \sigma'(C_j)$.
If condition B.2 holds, then $\sigma'(C_i) \overset{C_i \preceq \overline{C}}{=} \tau_C(C_i, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \overset{C_j \in >_{C_i}}{\leq} \overline{\sigma}(C_j) \overset{C_j \not\preceq \overline{C}}{=} \sigma'(C_j)$. Notice that $\tau_C(C_i, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \leq \overline{\sigma}(C_j)$ thanks to Corollary 1.
If condition B.3 holds, then $\sigma'(C_i) \overset{C_i \preceq \overline{C}}{=} \tau_C(C_i, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \overset{C_j \in >_{C_i}}{\leq} \tau_C(C_j, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \overset{C_j \preceq \overline{C}}{=} \sigma'(C_j)$.
If condition B.4 holds, then $\sigma'(C_i) = \sigma'(\overline{C}) = \overline{\sigma}(\overline{C}) = \sigma(\overline{C}) \oplus_\tau \langle A : \tau \rangle \leq \sigma(\overline{C}) \leq \sigma(C_j) \overset{\overline{C} \preceq C_j}{=} \overline{\sigma}(C_j) = \sigma'(C_j)$.
If condition B.5 holds, then $\sigma'(C_i) \overset{C_i \preceq \overline{C}}{=} \tau_C(C_i, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \overset{\overline{C} \in >_{C_i}}{\leq} \overline{\sigma}(\overline{C}) =$

$\sigma'(\overline{C})$.

For all the other schema modifications, conditions B.1, B.2, B.3 are treated in the same way as the **AddAttribute** case.
DeleteAttribute, **ChangAttributeName**: As to condition B.4
$\sigma'(C_i) = \sigma'(\overline{C}) = \tau_C(\overline{C}, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \overset{C_j \in >_{\overline{C}}}{\leq} \overline{\sigma}(C_j) = \sigma'(C_j)$.
As to condition B.5 $\sigma'(C_i) \overset{C_i \preceq \overline{C}}{=} \tau_C(C_i, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \overset{\overline{C} \in >_{C_i}}{\leq} \tau_C(\overline{C}, (\mathcal{CNV}, \overline{\sigma}, \preceq)) = \sigma'(\overline{C})$.
ChangeAttributeType: As to condition B.4
$\sigma'(C_i) = \sigma'(\overline{C}) = \overline{\sigma}(\overline{C}) = \langle A_1 : \tau_1, \ldots, A : \tau', \ldots, A_n : \tau_n \rangle \overset{by\ def}{\leq} \sigma(C_j) \overset{\overline{C} \preceq C_j}{=} \overline{\sigma}(C_j) = \sigma'(C_j)$ where $\sigma(\overline{C}) = \langle A_1 : \tau_1, \ldots, A : \tau, \ldots, A_n : \tau_n \rangle$.
If condition B.5 holds, then $\sigma'(C_i) \overset{C_i \preceq \overline{C}}{=} \tau_C(C_i, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \overset{\overline{C} \in >_{C_i}}{\leq} \overline{\sigma}(\overline{C}) = \sigma'(\overline{C})$. The **ChangeClassType** case can be treated analogously.

Let us consider the changes to the class collection.
AddSuperclass: In this case $\preceq'$ is the new partial order calculated from the addition of the pair $(C', \overline{C})$ to $\preceq$, $\mathcal{CNV}' = \mathcal{CNV}$. Therefore the following statement can be easily proved by induction on the structure of types:

$$\text{for each } \tau, \tau' \in \mathcal{T}|_{\mathcal{CNV}} = \mathcal{T}|_{\mathcal{CNV}'} : \text{ if } \tau \leq \tau' \text{ then } \tau \leq' \tau' \tag{B.6}$$

The set $\leq'_{C'}$ corresponds to $\{C \mid C \in \mathcal{CNV}, C \preceq' C'\}$. For all these classes the following condition also holds: $C \preceq' \overline{C}$ since $C' \preceq' \overline{C}$. We consider the six combinations introduced above with C' in place of $\overline{C}$:

(1) If $C_i \not\preceq C'$ and $C_j \not\preceq C'$, then $\sigma'(C_i) \overset{C_i \not\preceq C'}{=} \sigma(C_i) \leq \sigma(C_j) = \sigma'(C_j)$. From statement B.6 it follows that $\sigma'(C_i) \leq' \sigma'(C_j)$.
(2) If $C_i \preceq C'$ and $C_j \not\preceq C'$, then $\sigma'(C_i) \overset{C_i \preceq C'}{=} \tau_C(C_i, (\mathcal{CNV}, \sigma, \preceq')) \overset{C_i \preceq' C_j}{\leq'} \sigma(C_j) \overset{C_j \not\preceq C'}{=} \sigma'(C_j)$.
(3) If $C_i \preceq C'$ and $C_j \preceq C'$, then
$\sigma'(C_i) \overset{C_i \preceq C'}{=} \tau_C(C_i, (\mathcal{CNV}, \sigma, \preceq')) \overset{C_i \preceq' C_j}{\leq'} \tau_C(C_j, (\mathcal{CNV}, \sigma, \preceq')) = \sigma'(C_j)$.
(4) If $C_i = C'$ then $C' \preceq' C_j$. In this case either $C_j = \overline{C}$ or $C_j \neq \overline{C}$.
In all cases $\sigma'(C_i) = \sigma'(C') = \tau_C(C', (\mathcal{CNV}, \sigma, \preceq')) \overset{C' \preceq' C_j}{\leq'} \sigma'(C_j)$ (if $C_j = \overline{C}$ then $\sigma'(C_j) = \tau_C(\overline{C}, (\mathcal{CNV}, \sigma, \preceq'))$ else $\sigma'(C_j) = \sigma(C_j)$)
(5) If $C_j = C'$ then $C_i \preceq' C'$ then
$\sigma'(C_i) \overset{C_i \preceq C'}{=} \tau_C(C_i, (\mathcal{CNV}, \sigma, \preceq')) \overset{C_i \preceq' C_j}{\leq'} \tau_C(C', (\mathcal{CNV}, \sigma, \preceq')) = \sigma'(C') = \sigma'(C_j)$.

DeleteSuperclass: Due to its particular definition, the statement directly follows from Corollary 1.

AddClass: In this case $\mathcal{CNV}' = \mathcal{CNV} \cup \{\overline{C}\}$, $\sigma' = \sigma \cup \{\overline{C} \mapsto \textbf{any}\}$ and $\preceq'=\preceq$. Therefore $\leq'=\leq$ and the statement is verified. The **DeleteClass** operation can be treated analogously since this operation can be applied iff it causes no referential integrity problems and therefore $(\mathcal{CNV}', \sigma', \preceq)$ is a class version hierarchy, since for each class $C \in \mathcal{CNV}'$, $\sigma'(C) \neq \perp$ if it does not contain any reference to $\overline{C}$.
ChangeClassName: In this case the operation changes neither the class lattice nor the type of classes but it simply replace all occurences of $\overline{C}$ with C'. Therefore we omit the proof that directly follows from the operation definition. □

Proof of Lemma 2. Given $v \in dom(\tau)$ and $\tau' \in \mathcal{T}|_{\mathcal{CNV}}$, we have to show that $v' = \mathcal{I}_C(v, \tau, \tau', (\mathcal{CNV}, \sigma, \preceq)) \in dom(\tau')$. The proof is by induction on the structures of τ and τ'.

If $\tau = \tau'$, then $v' = v$ and the statement is verified.

If $\tau' \leq \tau$ then
Base case: If $\tau, \tau' \in \mathcal{CNV}$ then $v' = null \in dom(\tau')$, whereas if $\tau, \tau' \in \mathcal{LT}$ then $v' = v \in \tau'$ since $\tau' \leq \tau$ iff $\tau' = \tau$.
Inductive case: Let $\tau = \langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$, $\tau' = \langle A_1 : \tau'_1, \ldots, A_n : \tau'_n, \ldots, A_m : \tau'_m \rangle$ with $\tau'_i \leq \tau_i$, for $i \in [1, n]$. Suppose that $v = \langle A_1 : v_1, \ldots, A_n : v_n \rangle$ and $v'_i = \mathcal{I}_C(v_i, \tau_i, \tau'_i) \in dom(\tau'_i)$, for $i \in [1, n]$ by induction hypothesis. We have to show that $v' = \langle A_1 : v'_1, \ldots, A_n : v'_n, A_{n+1} : null, \ldots, A_m : null \rangle \in dom(\tau')$. It follows from the fact that $dom(\tau') = \{\langle A_1 : v_1, \ldots, A_n : v_n, \ldots, A_m : v_m \rangle \mid v_i \in dom(\tau'_i)$ for $i \in [1, m]\}$ and $v'_i \in dom(\tau'_i)$ for $i \in [1, n]$ by induction hypothesis and $null \in dom(\tau'_j)$ for $j \in [n + 1, m]$. The set, bag and list cases can be treated analogously.

If $\tau \leq \tau'$ then
Base case: If $\tau, \tau' \in \mathcal{CNV}$ then $v' = v \in dom(\tau')$ since $dom(\tau') = \pi^*(\tau') \cup \{null\}$, $v \in dom(\tau)$ and $dom(\tau) \subseteq dom(\tau')$, whereas if $\tau, \tau' \in \mathcal{LT}$ then $v' = v \in \tau'$ since $\tau' \leq \tau$ iff $\tau' = \tau$.
Inductive case: Let $\tau' = \langle A_1 : \tau'_1, \ldots, A_n : \tau'_n \rangle$, $\tau = \langle A_1 : \tau_1, \ldots, A_n : \tau_n, \ldots, A_m : \tau_m \rangle$ with $\tau_i \leq \tau'_i$, for $i \in [1, n]$. Suppose that $v = \langle A_1 : v_1, \ldots, A_n : v_n, \ldots, A_m : v_m \rangle$ and $v'_i = \mathcal{I}_C(v_i, \tau_i, \tau'_i) \in dom(\tau'_i)$, for $i \in [1, n]$ by induction hypothesis. We have to show that $v' = \langle A_1 : v'_1 \ldots, A_n : v'_n \rangle \in dom(\tau')$. It follows from the fact that $dom(\tau') = \{\langle A_1 : v_1, \ldots, A_n : v_n \rangle \mid v_i \in dom(\tau'_i)$ for $i \in [1, n]\}$ and $v'_i \in dom(\tau'_i)$ for $i \in [1, n]$ by induction hypothesis. The set, bag and list cases can be treated analogously.

Finally, if $\tau \sqcap \tau' = \perp$ then $v' = null \in dom(\tau')$. □

Proof of Theorem 2. For each schema modification m we have to show that $\mathcal{IU}(m)(\mathcal{I}) = (\pi', \nu')$ is a legal instance for $\mathcal{SVU}(m)(\mathcal{SV}) = (\mathcal{CNV}', \sigma', \preceq')$, that is:

(1) π' is an OID assignment legal for $(\mathcal{CNV}', \sigma', \preceq')$;
(2) for each $C \in \mathcal{CNV}'$ and $oi \in \pi'(C)$, $\nu'(oi) \in dom(\sigma'(C))$.

Let us consider all different schema modifications starting from the changes to the class type.
AddAttribute: In this case $\pi' = \pi$, thus condition 1 follows from the fact that π is a legal OID assignment for $(\mathcal{CNV}, \sigma, \preceq)$ and $\mathcal{CNV}' = \mathcal{CNV}$. As to condition 2, we consider two cases:

- if $C \not\in \leq_{\overline{C}}$, $oi \in \pi'(C) = \pi(C)$ then $\nu'(oi) = \nu(oi) \in dom(\sigma(C)) = dom(\sigma'(C))$.
- if $C \in \leq_{\overline{C}}$, $oi \in \pi'(C) = \pi(C)$ then $\nu'(oi) = \mathcal{I}_C(\nu(oi), \sigma(C), \sigma'(C), (\mathcal{CNV}, \sigma', \preceq)) \in dom(\sigma'(C))$ by Lemma 2, which can be applied since $\sigma(C) \in \mathcal{T}|_{\mathcal{CNV}}$, $\sigma'(C) \in \mathcal{T}|_{\mathcal{CNV}'}$ and $\mathcal{CNV} = \mathcal{CNV}'$.

The **DeleteAttribute**, **ChangeClassType** and **ChangeAttributeType** cases can be treated analogously.
ChangeAttributeName: Also in this case $\pi' = \pi$. As to condition 1, it can be proved as in the add attribute case. As to condition 2, we consider two cases:

- if $C \not\in \leq_{\overline{C}}$, $oi \in \pi'(C) = \pi(C)$ then $\nu'(oi) = \nu(oi) \in dom(\sigma(C)) = dom(\sigma'(C))$.
- if $C \in \leq_{\overline{C}}$, $oi \in \pi'(C) = \pi(C)$ then $\nu'(oi) = \mathcal{I}_C(\overline{\nu}(oi), \overline{\sigma}(C), \sigma'(C), (\mathcal{CNV}, \sigma', \preceq)) \in dom(\sigma'(C))$ by Lemma 2 iff $\overline{\nu}(oi) \in dom(\overline{\sigma}(C))$. But $\overline{\nu}(oi) = \langle A_1 : v_1, \ldots, A' : v, \ldots, A_n : v_n \rangle$ where $\nu(oi) = \langle A_1 : v_1, \ldots, A : v, \ldots, A_n : v_n \rangle \in dom(\sigma(C))$ by hypothesis, $\overline{\sigma}(C) = \langle A_1 : \tau_1, \ldots, A' : \tau, \ldots, A_n : \tau_n \rangle$ and $\sigma(C) = \langle A_1 : \tau_1, \ldots, A : \tau, \ldots, A_n : \tau_n \rangle$. Therefore, $\overline{\nu}(oi) \in dom(\overline{\sigma}(C))$ iff $v_1 \in dom(\tau_1), \ldots, v \in dom(\tau), \ldots, v_n \in dom(\tau_n)$, which directly follows from the fact that $\nu(oi) \in dom(\sigma(C))$.

As to the changes to the class collection, the **AddSuperclass** modification can be treated as the **AddAttribute** case, whereas the proof for the **DeleteSuperclass** modification directly follows from Lemma 2, which can be applied since $\mathcal{T}|_{\mathcal{CNV}} = \mathcal{T}|_{\mathcal{CNV}'}$. The **AddClass** only modifies π to include an empty extension for the newly introduced class $\overline{C}$. Therefore (π', ν) satisfies condition 1 since $\mathcal{CNV}' = \mathcal{CNV} \cup \{\overline{C}\}$ and π' defines an extension for all classes in $\mathcal{CNV}$ and also for $\overline{C}$. It also satisfies condition 2, since ν satisfies such condition for each $C \in \mathcal{CNV}$ and the extension of $\overline{C}$ contains no object. The **DeleteClass** can be treated analogously since referential integrity problems are excluded a priori. Finally, for the **ChangeClassName** operation which simply consists in the substitution of all $\overline{C}$ occurrences with C' the proof of the two conditions directly follows from the definition. □

Proof of Theorem 3. To prove Th. 3, we first require the following Lemma (we omit the proof, as it is obvious in both directions):

Lemma 4 *Let $(\mathcal{CNV}, \sigma, \preceq)$ be a schema version and $C \in \mathcal{CNV}$, then $\tau_C(C, (\mathcal{CNV}, \sigma, \preceq)) = \sigma(C)$ iff $(\mathcal{CNV}, \sigma, \preceq)$ is well-formed.*

Let $\mathcal{SV} = (\mathcal{CNV}, \sigma, \preceq)$ and $\mathcal{SV}' = (\mathcal{CNV}', \sigma', \preceq')$. We have to show that if $\mathcal{SVU}(m)(\mathcal{SV}) = \mathcal{SV}'$ (and the preconditions in Tab. 5 hold), then $\mathcal{SVU}(m^-)(\mathcal{SV}') = \mathcal{SV}$ which is equivalent to show that if $\mathcal{SVU}(m^-)(\mathcal{SVU}(m)(\mathcal{SV})) = \mathcal{SV}''$ then $\mathcal{SV}'' = \mathcal{SV}$.

As to the application of m to $\mathcal{SV}$, $\overline{\sigma}$ will denote the support function of σ', whereas as to the application of m^- to $\mathcal{SV}'$, $\mathcal{SV}'' = (\mathcal{CNV}'', \sigma'', \preceq'')$ where $\overline{\sigma'}$ is the support function of σ'', if exists. We have to show that $\mathcal{CNV}'' = \mathcal{CNV}$, $\sigma'' = \sigma$ and $\preceq''=\preceq$ for all cases in Tab. 5. In doing this we will only consider the modified element of the schema version (see Tabs. 2 and 3 for the $\mathcal{SVU}$ behavior).

(*Case 1:* $m =$ **AddAttribute**$_{A:\tau,\overline{C}}$, $m' =$ **DeleteAttribute**$_{A:\tau,\overline{C}}$) We have to show that $\sigma'' = \sigma$.

- If $C \notin \leq_{\overline{C}}$ then $\sigma''(C) = \overline{\sigma'}(C) = \sigma'(C) = \overline{\sigma}(C) = \sigma(C)$.
- If $C = \overline{C}$,

  $$\sigma''(\overline{C}) = \tau_C(\overline{C}, (\mathcal{CNV}, \overline{\sigma'}, \preceq)) = \sqcap\{\overline{\sigma'}(\overline{C}), \overline{\sigma'}(C_1), \ldots, \overline{\sigma'}(C_n)\}$$

  (where $\{C_1, \ldots, C_n\} = >_{\overline{C}}$. Since $\overline{C} \in \leq_{\overline{C}}$ and $\leq_{\overline{C}} \cap >_{\overline{C}} = \emptyset$, the outcome is the following)

  $$\begin{aligned} &= \sqcap\{\sigma'(\overline{C}) \ominus_\tau \langle A : \tau\rangle, \sigma'(C_1), \ldots, \sigma'(C_n)\} \\ &= \sqcap\{\sigma'(\overline{C}) \ominus_\tau \langle A : \tau\rangle, \sigma(C_1), \ldots, \sigma(C_n)\} \ . \end{aligned}$$

  If we show that $\sigma'(\overline{C}) \ominus_\tau \langle A : \tau\rangle = \sigma(\overline{C})$ then $\sigma''(\overline{C}) = \sqcap\{\sigma(\overline{C}), \sigma(C_1), \ldots, \sigma(C_n)\} = \sigma(\overline{C})$ owing to Lemma 4, which follows from the following equation

  $$\sigma'(\overline{C}) \ominus_\tau \langle A : \tau\rangle = \sigma(\overline{C}) \oplus_\tau \langle A : \tau\rangle \ominus_\tau \langle A : \tau\rangle = \sigma(\overline{C}) \qquad \text{(B.1)}$$

  since $A \notin \sigma(\overline{C})$.
- If $C \in <_{\overline{C}}$

  $$\sigma''(C) = \tau_C(C, (\mathcal{CNV}, \overline{\sigma'}, \preceq)) = \sqcap\{\overline{\sigma'}(C), \overline{\sigma'}(C_1), \ldots, \overline{\sigma'}(C_k)\}$$

  (where $\{C_1, \ldots, C_k\} = >_C$. Rearranging terms, this set can be partitioned into two subsets: the first is $\{C_1, \ldots, C_j\} = >_C \cap \leq_{\overline{C}}$ with $j \geq 1$ and $C_j = \overline{C}$, the second is its complement $\{C_{j+1}, \ldots, C_k\} = >_C \setminus \leq_{\overline{C}}$)

  $$\begin{aligned} = \sqcap \{&\sigma'(C) \ominus_\tau \langle A : \tau\rangle, \sigma'(C_1) \ominus_\tau \langle A : \tau\rangle, \ldots, \sigma'(C_{j-1}) \ominus_\tau \langle A : \tau\rangle, \\ &\sigma'(\overline{C}) \ominus_\tau \langle A : \tau\rangle, \sigma'(C_{j+1}), \ldots, \sigma'(C_k)\} \ . \end{aligned}$$

  Notice that $\sigma'(C_i) = \sigma(C_i)$ for $i \in [j+1, \ldots, k]$ since these C_i's do not belong to $\leq_{\overline{C}}$, and $\sigma'(\overline{C}) \ominus_\tau \langle A : \tau\rangle = \sigma(\overline{C})$ by equation B.1. If we show

that $\sigma'(D) \ominus_\tau \langle A : \tau \rangle = \sigma(D)$ for $D = C$ and $D = C_1, \ldots, C_{j-1}$ then $\sigma''(C) = \sqcap\{\sigma(C), \sigma(C_1), \ldots, \sigma(C_{j-1}), \sigma(\overline{C}), \sigma(C_{j+1}), \ldots, \sigma(C_k)\} = \sigma(C)$ due to Lemma 4:

$$\begin{aligned}\sigma'(D) \ominus_\tau \langle A : \tau \rangle =& \tau_C(D, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \ominus_\tau \langle A : \tau \rangle \\ =& \sqcap \{\overline{\sigma}(D), \overline{\sigma}(D_1), \ldots, \overline{\sigma}(D_k)\} \ominus_\tau \langle A : \tau \rangle\end{aligned}$$

(where $\{D_1, \ldots, D_k\} = >_D$. Let $D_k = \overline{C}$)

$$= \sqcap \{\sigma(D), \sigma(D_1), \ldots, \sigma(D_{k-1}), \sigma(\overline{C}) \oplus_\tau \langle A : \tau \rangle\} \ominus_\tau \langle A : \tau \rangle \ .$$

We distinguish two alternatives:

· if $\sigma(D) = \sqcap\{\sigma(D), \sigma(D_1), \ldots, \sigma(D_{k-1}), \sigma(\overline{C})\} = \langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$ where for each $i \in [1, n]$ $A_i \neq A$ then $\sqcap\{\sigma(D), \sigma(D_1), \ldots, \sigma(D_{k-1}), \sigma(\overline{C}) \oplus_\tau \langle A : \tau \rangle\} = \langle A_1 : \tau_1, \ldots, A_n : \tau_n, A : \tau \rangle$ and

$$\begin{aligned}\sigma'(D) \ominus_\tau \langle A : \tau \rangle &= \sqcap\{\sigma(D), \sigma(D_1), \ldots, \sigma(D_{k-1}), \sigma(\overline{C}) \oplus_\tau \langle A : \tau \rangle\} \ominus_\tau \langle A : \tau \rangle \\ &= \langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle = \sigma(D) \ ;\end{aligned}$$

· if $\sigma(D) = \sqcap\{\sigma(D), \sigma(D_1), \ldots, \sigma(D_{k-1}), \sigma(\overline{C})\} = \langle A_1 : \tau_1, \ldots, A_n : \tau_n, A : \tau' \rangle$ then $\sqcap\{\sigma(D), \sigma(D_1), \ldots, \sigma(D_{k-1}), \sigma(\overline{C}) \oplus_\tau \langle A : \tau \rangle\} = \langle A_1 : \tau_1, \ldots, A_n : \tau_n, A : \tau \sqcap \tau' \rangle$ where $\tau \sqcap \tau' = \tau'$ by condition expressed in Table 5 and

$$\begin{aligned}\sigma'(D) \ominus_\tau \langle A : \tau \rangle &= \sqcap\{\sigma(D), \sigma(D_1), \ldots, \sigma(D_{k-1}), \sigma(\overline{C}) \oplus_\tau \langle A : \tau \rangle\} \ominus_\tau \langle A : \tau \rangle \\ &= \langle A_1 : \tau_1, \ldots, A_n : \tau_n, A : \tau' \rangle = \sigma(D) \ .\end{aligned}$$

(*Case 2:* $m =$ **DeleteAttribute**$_{A:\tau,\overline{C}}$, $m' =$ **AddAttribute**$_{A:\tau,\overline{C}}$) We have to show that $\sigma'' = \sigma$.

- If $C \notin \leq_{\overline{C}}$ then $\sigma''(C) = \overline{\sigma'}(C) = \sigma'(C) = \overline{\sigma}(C) = \sigma(C)$.
- If $C = \overline{C}$ then $\sigma''(\overline{C}) = \sigma'(\overline{C}) \oplus_\tau \langle A : \tau \rangle$ and

$$\begin{aligned}\sigma'(\overline{C}) \oplus_\tau \langle A : \tau \rangle &= \tau_C(\overline{C}, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \oplus_\tau \langle A : \tau \rangle \\ &= \sqcap\{\sigma(\overline{C}) \ominus_\tau \langle A : \tau \rangle, \sigma(C_1), \ldots, \sigma(C_n)\} \oplus_\tau \langle A : \tau \rangle\end{aligned}$$

(since, for hypothesis, $A \in \sigma(\overline{C})$ and $A \notin \sigma(C_i)$ for $i \in [1, n]$, and thanks to Lemma 4 we can apply the following transformations)

$$\begin{aligned}&= \sqcap\{\sigma(\overline{C}), \sigma(C_1), \ldots, \sigma(C_n)\} \ominus_\tau \langle A : \tau \rangle \oplus_\tau \langle A : \tau \rangle \\ &= \sqcap\{\sigma(\overline{C}), \sigma(C_1), \ldots, \sigma(C_n)\} = \sigma(\overline{C}) \ .\end{aligned} \tag{B.2}$$

- If $C \in <_{\overline{C}}$, let us first introduce three remarks:

if $A \notin \sigma(C_i)$ with $C_i \in \mathcal{CNV}$ then

$$\sigma(C_i) \oplus_\tau \langle A : \tau \rangle = \sigma(C_i) \sqcap \langle A : \tau \rangle \tag{B.3}$$
$$\tau \sqcap \tau' = \tau \sqcap \tau' \sqcap \ldots \sqcap \tau' \tag{B.4}$$

if $(\mathcal{CNV}, \sigma, \preceq)$ is well-formed and τ is the type of an attribute $A \in \sigma(C_i')$ and $C_i \preceq C_i'$

$$(\sigma(C_i) \ominus_\tau \langle A : \tau \rangle) \sqcap \langle A : \tau \rangle = \sigma(C_i) \ . \tag{B.5}$$

Hence we can write

$$\sigma''(C) = \tau_C(C, (\mathcal{CNV}, \overline{\sigma'}, \preceq)) = \sqcap\{\overline{\sigma'}(C), \overline{\sigma'}(C_1), \ldots, \overline{\sigma'}(C_k)\}$$

(where $\{C_1, \ldots, C_k\} = >_C$. Rearranging terms, it can be partitioned into two subsets: $\{C_1, \ldots, C_j\} = >_C \cap \ \leq_{\overline{C}}$ with $j \geq 1$ and $C_j = \overline{C}$, and $\{C_{j+1}, \ldots, C_k\} = >_C \setminus \ \leq_{\overline{C}}$)

$$= \sqcap \{\sigma'(C), \sigma'(C_1), \ldots, \sigma'(C_{j-1}), \sigma'(\overline{C}) \oplus_\tau \langle A : \tau \rangle, \sigma'(C_{j+1}), \ldots, \sigma'(C_k)\}$$

(we apply remark B.3 since for hypothesis $A \notin \sigma'(\overline{C})$ and we also apply remark B.4)

$$\begin{aligned} =&\sigma'(C) \sqcap \langle A : \tau \rangle \sqcap \sigma'(C_1) \sqcap \langle A : \tau \rangle \sqcap \ldots \sigma'(C_{j-1}) \sqcap \langle A : \tau \rangle \sqcap \sigma'(\overline{C}) \sqcap \langle A : \tau \rangle \\ &\sqcap \sigma'(C_{j+1}) \sqcap \ldots \sqcap \sigma'(C_k) \ . \end{aligned}$$

Notice that $\sigma'(C_i) = \sigma(C_i)$ for $i \in [j+1, \ldots, k]$ since these C_i's do not belong to $\leq_{\overline{C}}$ and $\sigma'(\overline{C}) \sqcap \langle A : \tau \rangle = \sigma'(\overline{C}) \oplus_\tau \langle A : \tau \rangle = \sigma(\overline{C})$ by equation B.2. If we show that $\sigma'(D) \sqcap \langle A : \tau \rangle = \sigma(D)$ for $D = C$ and $D = C_1, \ldots, C_{j-1}$ then $\sigma''(C) = \sigma(C)$ owing to Lemma 4.

$$\begin{aligned} \sigma'(D) \sqcap \langle A : \tau \rangle =&\tau_C(D, (\mathcal{CNV}, \overline{\sigma}, \preceq)) \sqcap \langle A : \tau \rangle \\ =&\sqcap \{\overline{\sigma}(D), \overline{\sigma}(D_1), \ldots, \overline{\sigma}(D_k)\} \sqcap \langle A : \tau \rangle \end{aligned}$$

(where $\{D_1, \ldots, D_k\} = >_D$. Let $\{D_1, \ldots, D_j\} = >_D \cap \leq_{\overline{C}}$ with $j \geq 1$)

$$\begin{aligned} =&\sqcap \{\sigma(D) \ominus_\tau \langle A : \tau \rangle, \sigma(D_1) \ominus_\tau \langle A : \tau \rangle, \ldots \\ &\ldots, \sigma(D_j) \ominus_\tau \langle A : \tau \rangle, \sigma(D_{j+1}), \ldots \sigma(D_k)\} \sqcap \langle A : \tau \rangle \end{aligned}$$

(we apply remarks B.4 and B.5)

$$= \sqcap \{\sigma(D), \sigma(D_1), \ldots, \sigma(D_j), \sigma(D_{j+1}), \ldots \sigma(D_k)\} = \sigma(D) \ .$$

(*Case 3:* $m =$ **ChangeAttrName**$_{A,A',\overline{C}}$, $m' =$ **ChangeAttrName**$_{A',A,\overline{C}}$) We have to show that $\sigma'' = \sigma$.

- If $C \notin \leq_{\overline{C}}$ then the proof coincides with the two previous ones;
- if $C \in \leq_{\overline{C}}$

$$\sigma''(C) = \tau_C(C, (\mathcal{CNV}, \overline{\sigma'}, \preceq)) = \sqcap\{\overline{\sigma'}(C), \overline{\sigma'}(C_1), \ldots, \overline{\sigma'}(C_k)\}$$

(where $\{C_1, \ldots, C_k\} = >_C$. This set can be partitioned into two subsets: the first is $\{C_1, \ldots, C_j\} = >_C \cap \leq_{\overline{C}}$ with $j \geq 1$ and $C_j = \overline{C}$, the second is its complement)

$$= \sqcap \{\overline{\sigma'}(C), \overline{\sigma'}(C_1), \ldots, \overline{\sigma'}(C_j), \overline{\sigma'}(C_{j+1}), \ldots, \overline{\sigma'}(C_k)\} .$$

Notice that $\overline{\sigma'}(C_i) = \sigma(C_i)$ for $i \in [j+1, \ldots, k]$ since they do not belong to $\leq_{\overline{C}}$. If we show that $\overline{\sigma'}(D) = \sigma(D)$ for $D = C$ and $D = C_1, \ldots, C_j$ then $\sigma''(C) = \sqcap\{\sigma(C), \sigma(C_1), \ldots, \sigma(C_{j-1}), \sigma(\overline{C}), \sigma(C_{j+1}), \ldots, \sigma(C_k)\} = \sigma(C)$ by Lemma 4. Let $\sigma(D) = \langle A_1 : \tau_1, \ldots, A : \tau, \ldots, A_n : \tau_n\rangle$ then $\overline{\sigma}(D) = \langle A_1 : \tau_1, \ldots, A' : \tau, \ldots, A_n : \tau_n\rangle$ and

$$\sigma'(D) = \tau_C(D, (\mathcal{CNV}, \overline{\sigma}, \preceq)) = \sqcap\{\overline{\sigma}(D), \overline{\sigma}(D_1), \ldots, \overline{\sigma}(D_k)\}$$

(where $\{D_1, \ldots, D_k\} = >_D$. Let $\{D_1, \ldots, D_j\} = >_D \cap \leq_{\overline{C}}$)

$$= \sqcap \{\overline{\sigma}(D), \overline{\sigma}(D_1), \ldots, \overline{\sigma}(D_j), \sigma(D_{j+1}), \ldots \sigma(D_k)\} = \overline{\sigma}(D) .$$

In fact $\overline{\sigma}(D) \sqcap \sigma(D_h)$ for $h \in [j+1, k]$ is $\overline{\sigma}(D)$. Indeed for hypothesis $A' \notin \sigma(D)$ and therefore $A' \notin \sigma(D_h)$ since $(\mathcal{CNV}, \sigma, \preceq)$ is well formed and $D \preceq D_h$ and $A \notin \sigma(D_h)$ for the condition expressed in Tab. 5. Therefore the type of A' is not modified and A cannot be inherited again. At the same time $\overline{\sigma}(D) \sqcap \overline{\sigma}(D_h)$ for $h \in [1, j]$ is $\overline{\sigma}(D)$ since $D_h \in \leq_{\overline{C}}$. Similarly, starting from $\sigma'(D) = \overline{\sigma}(D)$, it can be shown $\sigma''(D) = \sigma(D)$. In fact $\sigma''(D) = \sqcap\{\overline{\sigma'}(D), \overline{\sigma'}(D_1), \ldots, \overline{\sigma'}(D_j), \sigma(D_{j+1}), \ldots \sigma(D_k)\}$ where $\overline{\sigma'}(D) = \sigma(D)$ and, for $i \in \{1, \ldots, j\}$, $\overline{\sigma'}(D_i) = \sigma(D)$.

(*Case 4:* $m =$ **ChangeAttrType**$_{A,\tau,\tau',\overline{C}}$, $m' =$ **ChangeAttrType**$_{A,\tau',\tau,\overline{C}}$) We have to show that $\sigma'' = \sigma$.

- If $C \notin \leq_{\overline{C}}$ then the proof coincides with the first two ones;
- If $C = \overline{C}$, let $\sigma(C) = \langle A_1 : \tau_1, \ldots, A : \tau, \ldots, A_n : \tau_n\rangle$. Then $\sigma'(C) = \overline{\sigma}(C) = \langle A_1 : \tau_1, \ldots, A : \tau', \ldots, A_n : \tau_n\rangle$ and $\sigma''(C) = \overline{\sigma'}(C) = \langle A_1 : \tau_1, \ldots, A : \tau, \ldots, A_n : \tau_n\rangle$ if $\langle A_1 : \tau_1, \ldots, A : \tau, \ldots, A_n : \tau_n\rangle \leq \sigma'(C')$ for each $C' \in >_C$ which is verified since $\sigma'(C') = \sigma(C')$ ($C' \notin \leq_C$) and $\langle A_1 : \tau_1, \ldots, A : \tau, \ldots, A_n : \tau_n\rangle = \sigma(C)$ and $(\mathcal{CNV}, \sigma, \preceq)$ is well-formed.
- if $C \in <_{\overline{C}}$

$$\sigma''(C) = \tau_C(C, (\mathcal{CNV}, \overline{\sigma'}, \preceq)) = \sqcap\{\overline{\sigma'}(C), \overline{\sigma'}(C_1), \ldots, \overline{\sigma'}(C_k)\}$$

(where $\{C_1, \ldots, C_k\} = >_C$. This set can be partitioned into two subsets: the first is $\{C_1, \ldots, C_j\} = >_C \cap \leq_{\overline{C}}$ with $j \geq 1$ and $C_j = \overline{C}$, the second is its complement)

$$= \sqcap \{\overline{\sigma'}(C), \overline{\sigma'}(C_1), \ldots, \overline{\sigma'}(C_j), \overline{\sigma'}(C_{j+1}), \ldots, \overline{\sigma'}(C_k)\} .$$

Notice that $\overline{\sigma'}(C_i) = \sigma(C_i)$ for $i \in [j+1, \ldots, k]$ since they do not belong to $\leq_{\overline{C}}$ and that $\overline{\sigma'}(\overline{C}) = \sigma(\overline{C})$ from previous point. If we show that $\overline{\sigma'}(D) = \sigma(D)$ for $D = C$ and $D = C_1, \ldots, C_{j-1}$ then $\sigma''(C) = \sqcap\{\sigma(C), \sigma(C_1), \ldots, \sigma(C_{j-1}), \sigma(\overline{C}), \sigma(C_{j+1}), \ldots, \sigma(C_k)\} = \sigma(C)$ by Lemma 4. Let $\sigma(D) = \langle A_1 : \tau_1, \ldots, A : \tau, \ldots, A_n : \tau_n\rangle$ then $\overline{\sigma}(D) = \langle A_1 : \tau_1, \ldots, A : \tau', \ldots, A_n : \tau_n\rangle$ and $\sigma'(D) = \overline{\sigma}(D)$ due to the condition expressed in Tab. 5 and then $\overline{\sigma}'(D) = \langle A_1 : \tau_1, \ldots, A : \tau, \ldots, A_n : \tau_n\rangle = \sigma(D)$. Let $\sigma(D) = \langle A_1 : \tau_1, \ldots, A : \overline{\tau}, \ldots, A_n : \tau_n\rangle$ with $\overline{\tau} \neq \tau$ then $\overline{\tau} < \tau < \tau'$ since $(\mathcal{CNV}, \sigma, \preceq)$ is well formed and $\langle A : \tau\rangle \in \sigma(C)$, $\langle A : \overline{\tau}\rangle \in \sigma(D)$, and $D \preceq C$. Then $\sigma'(D) = \tau_C(D, (\mathcal{CNV}, \overline{\sigma}, \preceq)) = \tau_C(D, (\mathcal{CNV}, \sigma, \preceq)) = \sigma(D)$ by Lemma 4 and $\overline{\sigma'}(D) = \sigma'(D)$.

(*Case 5:* $m =$ **ChangeClassType**$_{\overline{C},\tau,\tau'}$, $m' =$ **ChangeClassType**$_{\overline{C},\tau,\tau'}$) We have to show that $\sigma'' = \sigma$. The proof is similar to that of Case 4.

(*Case 6:* $m =$ **AddSuperclass**$_{\overline{C},C'}$, $m' =$ **DeleteSuperclass**$_{\overline{C},C'}$) We have to show that $\preceq''=\preceq$ and $\sigma'' = \sigma$. Notice that $\preceq''=\preceq' \setminus\{(C', \overline{C})\} =\preceq \cup\{(C', \overline{C})\} \setminus \{(C', \overline{C})\} =\preceq$ since $\preceq'=\preceq \cup\{(C', \overline{C})\}$ due to condition in Tab. 5. Notice also that

$$\leq''_{C'} =\leq'_{C'}=\leq_{C'}= \{C'\} \tag{B.6}$$
$$>'_{C'} =>_{C'} \cup\{\overline{C}\} \tag{B.7}$$

and that for each $C \in \mathcal{CNV}$ with $C \neq C'$

$$>'_C= \{D \mid (C, D) \in\preceq'\} \overset{C\neq C'}{=} \{D \mid (C, D) \in\preceq\} = >_C \ . \tag{B.8}$$

Let us consider the two cases related to Eq. B.6:

- if $C = C'$ then

$$\begin{aligned}\sigma''(C') =&\tau_C(C', (\mathcal{CNV}, \sigma', \preceq'')) = \tau_C(C', (\mathcal{CNV}, \sigma', \preceq))\\ &= \sqcap \{\sigma'(C'), \sigma'(C_1), \ldots, \sigma'(C_k)\}\end{aligned}$$

(where $\{C_1, \ldots, C_k\} = >_{C'}$. Since $>_{C'} \cap \leq'_{C'} \overset{Eq.\ B.6}{=} >_{C'} \cap \leq_{C'}=>_{C'} \cap \{C'\} = \emptyset$ the outcome is the following)

$$= \sqcap \{\sigma'(C'), \sigma(C_1), \ldots, \sigma(C_k)\} .$$

If we show that $\sigma'(C') = \sigma(C)$ then $\sigma''(C') = \sqcap\{\sigma(C'), \sigma(C_1), \ldots, \sigma(C_k)\} = \sigma(C)$ by Lemma 4.

$$\sigma'(C') = \tau_C(C', (\mathcal{CNV}, \sigma, \preceq'))$$

(due to Eq. B.7 and to Lemma 4)

$$= \tau_C(C', (\mathcal{CNV}, \sigma, \preceq)) \sqcap \sigma(\overline{C}) = \sigma(C') \sqcap \sigma(\overline{C}) = \sigma(C') \ .$$

- if $C \neq C'$

$$\begin{aligned} \sigma''(C) =& \tau_C(C, (\mathcal{CNV}, \sigma', \preceq'')) = \tau_C(C, (\mathcal{CNV}, \sigma', \preceq)) \\ =& \sqcap \{\sigma'(C), \sigma'(C_1), \ldots, \sigma'(C_k)\} \end{aligned}$$

(where $\{C_1, \ldots, C_k\} = >_C$. Notice that $>_C \cap \leq'_{C'} \stackrel{Eq.\ B.6}{=} >_C \cap \leq_{C'} = >_C \cap \{C'\} = \emptyset$. In fact, if exists $C_i \in >_C \cap \{C'\}$ then $C_i = C'$ and $C \preceq C'$ which is impossible due to the precondition in Table 5)

$$= \sqcap \{\sigma(C), \sigma(C_1), \ldots, \sigma(C_k)\} = \sigma(C) \ .$$

(*Case 7:* $m =$ **DeleteSuperclass**$_{\overline{C},C'}$, $m' =$ **AddSuperclass**$_{\overline{C},C'}$) We have to show that $\preceq''=\preceq$ and $\sigma'' = \sigma$. Notice that $\preceq''=$ partial order from $\preceq' \cup \{(C', \overline{C})\} =$ partial order from $\preceq \setminus \{(C', \overline{C})\} \cup \{(C', \overline{C})\} = \preceq$. Notice also that

$$>'_{C'} = >_{C'} \setminus \{\overline{C}\} \tag{B.9}$$

and that for each $C \in \mathcal{CNV}$ with $C \neq C'$

$$>'_C = >_C \ . \tag{B.10}$$

As to σ'', we have to consider three different cases:

- If $C \notin \leq_{\overline{C}}$, then $\sigma''(C) = \sigma'(C) = \tau_C(C, (\mathcal{CNV}, \sigma, \preceq')) \stackrel{Eq.\ B.10}{=} \tau_C(C, (\mathcal{CNV}, \sigma, \preceq)) = \sigma(C)$ since $(\mathcal{CNV}, \sigma, \preceq)$ is well-formed;
- If $C = \overline{C}$, then $\sigma''(C') = \tau_C(C', (\mathcal{CNV}, \sigma, \preceq'')) = \tau_C(C', (\mathcal{CNV}, \sigma, \preceq)) = \sigma(C') \sqcap \sigma(C_1) \sqcap \ldots \sqcap \sigma(C_k) \sqcap \sigma(\overline{C})$ where $\{C_1, \ldots, C_k, \overline{C}\} = >_{C'}$. If we show that $\sigma'(D) = \sigma(D)$ for $D = C'$ and $D = C_1, \ldots, C_k, \overline{C}$ then $\sigma''(C') = \sqcap\{\sigma(C'), \sigma(C_1), \ldots, \sigma(C_k), \sigma(\overline{C})\} = \sigma(C')$ by Lemma 4. Notice that it is verified for $D = C_1, \ldots, C_k, \overline{C}$ due to the previous case and that $\sigma'(C') = \tau_C(C, (\mathcal{CNV}, \sigma, \preceq')) \stackrel{Eq.\ B.9}{=} \sigma(C') \sqcap \sigma(C_1) \sqcap \ldots \sqcap \sigma(C_k)$. Since $(\mathcal{CNV}, \preceq, \sigma)$ is well-formed and $C' \preceq \overline{C}$, we have $\sigma(C') \leq \sigma(\overline{C})$ and, thus, $\sigma(C') = \sigma(C') \sqcap \sigma(\overline{C})$. Hence, $\sigma'(C') = \sigma(C') \sqcap \sigma(\overline{C}) \sqcap \sigma(C_1) \sqcap \ldots \sqcap \sigma(C_k) = \sigma(C)$ since $(\mathcal{CNV}, \sigma, \preceq)$ is well-formed;
- if $C \in <_{\overline{C}}$, then $\sigma''(C) = \tau_C(C, (\mathcal{CNV}, \sigma, \preceq'')) = \tau_C(C, (\mathcal{CNV}, \sigma, \preceq)) = \sigma(C') \sqcap \sigma(C_1) \sqcap \ldots \sqcap \sigma(C_k) \sqcap \sigma(C')$ where $\{C_1, \ldots, C_k, C'\} = >_C$. If we show that $\sigma'(D) = \sigma(D)$ for $D = C$ and $D = C_1, \ldots, C_k, C'$ then $\sigma''(C) = \sqcap\{\sigma(C), \sigma(C_1), \ldots, \sigma(C_k), \sigma(C')\} = \sigma(C)$ by Lemma 4. Notice that it is verified for C' due to the previous case and for all $C_i \in >_C \setminus <''_{C'}$ due to the first case. As to the classes $C_i \in >_C \cap <''_{C'}$, it follows that $\sigma'(C) = \tau_C(C, (\mathcal{CNV}, \sigma, \preceq')) \stackrel{Eq.\ B.10}{=} \tau_C(C, (\mathcal{CNV}, \sigma, \preceq)) = \sigma(C)$.

For the remaining alternatives, the proof directly follows from the specification of the semantics.

□

Fabio Grandi is currently an Associate Professor in the Faculty of Engineering of the University of Bologna, Italy. Since 1989 he has worked at the CSITE (formerly CIOC) center of the Italian National Research Council (CNR) in Bologna in the field of neural networks and temporal databases. For this work he was initially supported by a fellowship from the CNR. In 1993 and 1994 he was an Adjunct Professor at the Universities of Ferrara, Italy, and Bologna. He joined his current department (Dept. of Electronics, Computer Science and Systems) as a Research Associate in 1994. His scientific interests include temporal databases, storage structures, access cost models, WWW extensions. He received a Laurea degree in Electronics Engineering and a PhD in Electronics Engineering and Computer Science from the University of Bologna. Further information can be found at http://www-db.deis.unibo.it/∼fgrandi/.

Federica Mandreoli is currently a Research Associate at the Department of Information Engineering of the University of Modena and Reggio Emilia, Italy. Her research interests include information retrieval, multi-database systems, semantic web, object-oriented databases and schema versioning. She holds a Laurea degree in Computer Science and a PhD in Electronics Engineering and Computer Science from the University of Bologna. She is member of the Association for Computer Machinery (ACM). Further information can be found at http://www.dbgroup.unimo.it/members.html.