A Principled Approach to Context Schema Evolution in a Data Management Perspective

Elisa Quintarelli, Emanuele Rabosio*, Letizia Tanca

Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, via Ponzio 34/5, 20133 Milano, Italy

Abstract

Context-aware data tailoring studies the means for the system to furnish the users, at any moment, only with the set of data which is relevant for their current context. These data may be from traditional databases, sensor readings, environmental information, close-by people, points of interest etc. To implement context-awareness, we use a formal representation of a conceptual *context model*, used to design the *context schema*, which intensionally represents all the contexts in which the user may be involved in the considered application scenario.

Following this line of thought, in this paper we develop a formal approach and the corresponding strategy to manage the evolution of the context schema of a given context-aware application, when the context perspectives initially envisaged by the system designer are not applicable any more and unexpected contexts are to be activated. Accordingly, when the context schema evolves also the evolution of the corresponding context-aware data portions must be taken care of. The aim of this article is thus to provide the necessary conceptual and formal notions to manage the evolution of a context schema in the perspective of data tailoring: after introducing a set of operators to manage evolution and proving their soundness and completeness, we analyze the impact that context evolution has on the context-based data tailoring process. We then study how sequences of operator applications can be optimized and finally present a prototype validating the feasibility of the approach.

Keywords:
context-awareness, schema evolution, evolution operators

1. Introduction

The technological scenario of our era has enabled an extremely large variety of information sources to become available even to casual users: all kinds of organizations collect, maintain and use terabytes of information about their customers, suppliers and operations, while, with the help of the widespread use of mobile terminals, the WWW is becoming day by day more friendly to any kind of users. The contribution of the two recent phenomena Internet of Things and Social Networking further enriches and complicates the overall scenario.

Such an extensive repository constitutes an unprecedented opportunity for users, but at the same time risks to overwhelm them; often the only way to get the gist of the available information requires that the users know exactly how to formulate the query, a difficult task when the dataset structure and meaning are not known a priori. Moreover, really large data collections simply cannot be stored in the increasingly popular portable devices, still characterized by a relatively limited amount of memory.

The literature has coped with this research challenge by proposing techniques for summarizing [1], compressing [2] and analyzing Big Data [3]. In our opinion, this problem can be cleverly solved also by applying *personalization*, so that the information provided to a user is reduced on the basis of the user's personal preferences [4], on the user's current situation [5] – i.e., her *context* –, or even on both aspects [6]. Note that, even with amounts of data well below our current idea of "big", personalization constitutes an important contribution to data usability.

This work considers context-based personalization. In order to reduce a large dataset on the basis of the context, conceptual *context models* have been introduced (see [7, 8, 9, 10, 11] for surveys), allowing to represent the context through some perspectives (dimensions): typical such dimensions are, for example, the user's current role and her location. Also more sophisticated context parameters can be introduced, like the current activity of the user, or her main interest topic. We call *data tailoring* [12] the activity of selecting, for each specific context, the relevant information: *data tailoring* refers to the capability of the system to provide the users only with the view (over an overall data representation, like for instance a global schema) that is relevant for their current context.

We use the expression *context model* to indicate the

*Corresponding author. Tel.: +39 0223993482
Email addresses: elisa.quintarelli@polimi.it (Elisa Quintarelli), emanuele.rabosio@polimi.it (Emanuele Rabosio), letizia.tanca@polimi.it (Letizia Tanca)

set of constructs and constraints that allow us to represent the dimensions of context and their values at a conceptual level. The activity of designing a context-aware database requires to produce a *context schema*, which exploits the constructs provided by the context model to describe the set of *dimensions* and their *values* relevant for a certain application scenario. A *context instance*, or simply a *context*, represents a particular situation, described according to a context schema.

A context schema thus represents synthetically the *structure* of the context, and as such is useful when reasoning about the assignment of a data portion (*contextual view*) to each context. For example, the work [13] presents a very effective and efficient method to automatically assign data views to all the contexts represented by the schema, only by associating one view with each context dimension value.

Throughout the paper we use a running example in the movie domain: we consider a company offering services of video on demand and reservation of cinema tickets. In this scenario, possible perspectives useful to tailor the data are the kind of user (e.g., adult, teenager or family with children), the interest topic (e.g., cinemas or movies), the situation (e.g., alone or with friends), the time (e.g., daytime or night) and the zone. The company uses context-awareness to suggest to its customers the movie(s) which are most appropriate to their current context.

The useful dimensions for data tailoring depend on the application requirements, that in current systems are intrinsically dynamic and thus can evolve. Just as in the case of database schema evolution, requirement changes can be due to various reasons, including changing business needs or application and technology developments [14]: the context representation used to perform the tailoring process should thus be smoothly adapted to the evolution of requirements over time.

Consider the movie example above. The company might change its business policy, deciding to remove the distinction between daytime and evening schedule; this would lead to removing the *time* dimension from the context representation. Moreover, at a certain point marketing researches might reveal that adult customers and teenagers show the same behavior, thus making it useless to distinguish between the two user groups: then the designer might simplify the representation of the user type, merging the two categories *adult* and *teenager*. In addition, if the impact of technological changes on the considered application grows, the designer might deem it appropriate to tailor the data also on the basis of the kind of device used for the access, thus inserting a *device* dimension in the context representation. The above changes then become out-of-sync w.r.t. the previously envisaged contexts, known by the user (and by the context-aware application) at a given moment; thus the system must be able to respond to queries and applications in a seamless way, that is, a way as similar as possible to the context-aware behavior the users and applications expect. Note that studying context schema evolution is also preliminary to understanding context sharing among different users, a need that may arise in P2P scenarios [15].

As remarked already, this problem is similar to a problem of database schema evolution, where the queries designed to run on old schema versions should be maintained in order to be still applicable in the face of database schema changes. After having studied the literature on schema evolution in various fields, we propose strategies to flexibly manage the evolution of context schemas *in a data management perspective*, i.e. keeping in mind that the context is used to perform data tailoring; the context, in fact, has been employed in the literature to manage not only data, but also many other kinds of entities, including mobile sessions [16], services [17], intelligent spaces [18], etc.

The basic idea of our approach is to introduce a set of *evolution operators* to be used by the designer for modifying the conceptual context schema when necessary. These operators are so conceived as to support the evolution of the contextual (tailored) views as well. We will show in this paper that specific problems encountered in the scenario of context schema evolution cannot be solved using the techniques proposed by database schema evolution while, conversely, the special characteristics of context schema evolution make some problems of the database case trivial.

Indeed, our context model provides intuitive constructs and operations that afford a high level of abstraction with respect to the application scenario and to the employed technologies, and permit the management of (hierarchical) context information of various types, possibly coming from diverse sources. As pointed out in [19, 20], these are fundamental features of the modern context models, thus, we believe that the results and techniques that we present in this paper are easily generalizable to other context models and implementable by means of different programming languages and logical data models, e.g. XML or some object-oriented language. Such implementations may take advantage of previous schema evolution proposals, however this operation must be performed with some caution because the semantics do not immediately correspond to each other, as highlighted in Section 9.

Goal and contributions. Built on research already published in [21] – which provides the initial formalization of schema evolution for context-aware data tailoring – the main original contributions of this paper are:

- The consolidation of the formal basis for context schema evolution.
- An extension of the set of evolution operators introduced in [21], along with their declarative and operational semantics. This is the basis of our principled approach to the overall evolution process.
- The proof of important properties of the operators.

- The study of the impact of context schema evolution on the lifecycle of context-aware data management systems.
- The optimization of sequences of operator applications.
- An engine, implementing the operators, which allowed us to verify experimentally the feasibility and effectiveness of the approach.

The paper is organized as follows. Section 2 recalls the background on the employed context model and its usage for defining contextual views, while Section 3 examines the literature on evolution. Section 4 outlines our framework for managing context schema evolution. Section 5 introduces a formal description of the evolution operators, Section 6 explains how the contextual views are affected by the evolution process, and Section 7 deals with the optimization of sequences of operators. Section 8 provides a full example of the usage of context to tailor data and of the application of the operators, while Section 9 exploits this example for a detailed comparison of our approach with interesting proposals from the literature and explains what are the fundamental differences with the other schema evolution proposals. Section 10 describes the implementation of the context schema evolution system and, finally, Section 11 draws the conclusions.

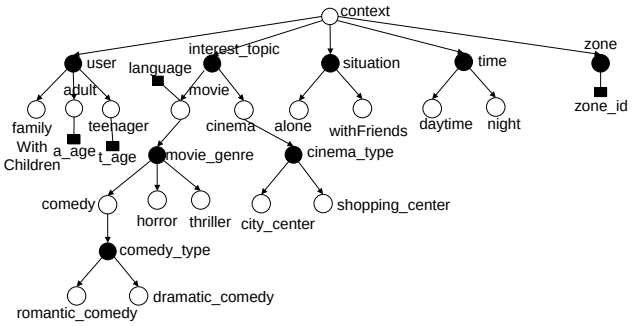


Figure 1: A context schema

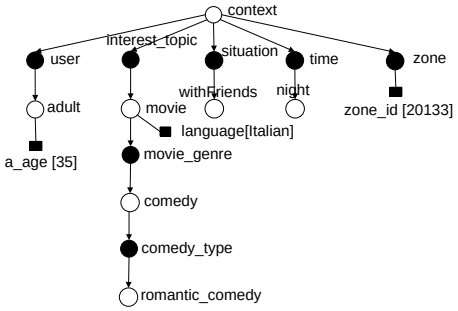


Figure 2: A context instance of the context schema in Figure 1

2. Background: The Context Dimension Model

In this section we present the Context Dimension Model (CDM) [13] and an overview of contextual data tailoring [5].

2.1. Informal Overview of the CDM

The CDM allows to represent context schemas (a.k.a. Context Dimension Trees - CDTs) as trees with nodes of two kinds: *dimensions* and *dimension values* (or *concepts*). An example of context schema for the movie scenario of the running example is shown in Figure 1. Dimension nodes are graphically represented as black nodes, while concepts are drawn as white nodes. Dimension nodes represent the different perspectives describing a context (e.g. the type of `user` and the `situation`), while concepts constitute the admissible values of each dimension (e.g., the concepts `adult` and `teenager` are values of the `user` dimension). The root is a special concept node representing the most general context (capturing all the data), and its children are the main analysis dimensions. Every dimension has only concept children, and each concept has only dimension children; the latter are subdimensions further specifying the concept. Dimensions and concepts can be endowed with attributes, that are parameters whose values can be dynamically derived from the environment or provided by the users themselves at execution time. A dimension can be connected to at most one attribute, used to replace a huge number of concepts when it is impractical to enumerate them all (e.g., a GPS location). If this is the case, the dimension does not have any concept child; e.g., the `zone` dimension has as child the attribute `zone_id`. One or more attributes can be added to concepts too; in such a situation they are used to select specific instances from the set of values represented by a concept node (e.g., the age of an adult customer). Attributes are graphically represented in Figure 1 by square nodes.

An instance of a context schema is a set of dimension-value pairs, and can also be drawn as a tree whose leaves represent the values taken by the correspondng (sub)dimensions. Figure 2 shows an instance of the context schema in Figure 1: a thirty-five-year-old adult, who is located in the zone with identifier 20133, is interested in romantic comedies, and is going to the cinema at night with friends. Note also the attribute values, written in Figure 2 within square brackets.

Note that in real applications context schemas are usually quite small: an analysis carried out in [5] on several application scenarios has shown that it is very unlikely that context schemas exceed 50 nodes. Context instances are even smaller: in fact, a context instance contains a subset of the dimensions of the corresponding context schema, and each of them is associated with just one concept node or an attribute.

2.2. A Formal Definition of the CDM

Let $\mathcal{N}$ be the set of node identifiers. $\mathcal{N}$ is partitioned into two subsets: *concept* node identifiers ($\mathcal{N}^\circ$) and *dimension* node identifiers ($\mathcal{N}^\bullet$). Let $\mathcal{A}$ be the set of attribute identifiers and $\mathcal{L}$ a set of strings. All these sets are pairwise disjoint. We start by defining the *context semischema*, that is a context schema which does not impose the existence of a concept root:

Definition 1 (Context semischema)**.** A context semischema is a tuple $S = (N, E, r, Att, \alpha, \lambda)$ such that:

(i) $N \subseteq \mathcal{N}$ is a set of node identifiers, $N = N^\circ \cup N^\bullet$; $N^\circ \subseteq \mathcal{N}^\circ$ is the set of concept node identifiers and $N^\bullet \subseteq \mathcal{N}^\bullet$ is the set of dimension node identifiers. $E \subseteq N \times N$ is a set of directed edges, $r \in N$ is a node identifier such that (N, E) is a tree with root r, $Att \subseteq \mathcal{A}$ is a set of attribute identifiers.

(ii) Every generation contains node identifiers of the same type and this type is different from that of the immediately previous and following generations, i.e., for each $(n_1, n_2) \in E$, $n_1 \in N^\circ \Rightarrow n_2 \in N^\bullet$ and $n_1 \in N^\bullet \Rightarrow n_2 \in N^\circ$.

(iii) $\alpha : Att \to N$ is a function assigning a node identifier to each attribute. If $n \in N^\bullet$ is a leaf, i.e. if $(\nexists n_1 \in N)((n, n_1) \in E)$, then $|\alpha^{-1}(n)| = 1$; otherwise, $\alpha^{-1}(n) = \varnothing$.

(iv) $\lambda : N \cup Att \to \mathcal{L}$ is an injective function assigning a unique label to node and attribute identifiers.

The set E of edges constitutes a binary relation on the set of node identifiers; its transitive closure is indicated by E^+. Moreover, the following sets are defined: leaf dimensions $\bar{N}^\bullet = \{n_1 \in N^\bullet : (\nexists n_2 \in N)((n_1, n_2) \in E)\}$, leaf concepts $\bar{N}^\circ = \{n_1 \in N^\circ : (\nexists n_2 \in N)((n_1, n_2) \in E)\}$ and leaves $\bar{N} = \bar{N}^\bullet \cup \bar{N}^\circ$.

The Context Dimension Tree is a specialization of the semischema, as follows:

Definition 2 (Context schema, or CDT)**.** A context schema is a context semischema $(N, E, r, Att, \alpha, \lambda)$ in which r is a concept node and $\lambda(r) = \texttt{context}$.

Remark 1. A context schema can be represented as an XML document, where N is the set of XML elements and E describes how they are nested; the set Att is represented by means of XML attributes, associated with the elements by α. Since the attributes in a context schema are not associated with any value, each corresponding XML attribute takes as value a default placeholder (e.g., `age` = "`$age`").

Let $\mathcal{V} = \mathcal{L} \cup \{ALL\}$, where ALL is a special string indicating that no values have been provided. Context semi-instances are defined as follows:

Definition 3 (Context semi-instance)**.** A context semi-instance is a pair $I = (S_I, \rho_I)$ where:

(i) $S_I = (N_I, E_I, r_I, Att_I, \alpha_I, \lambda_I)$ is a context semischema such that every dimension node identifier n with no attributes has exactly one child, i.e., for each $n \in N_I^\bullet : \alpha_I^{-1}(n) = \varnothing$ there is exactly one $n' \in N_I$ such that $(n, n') \in E_I$.

(ii) $\rho_I : Att \to \mathcal{V}$ is a function assigning a string to each attribute identifier.

A context instance is a semi-instance of a CDT:

Definition 4 (Context instance)**.** A context instance is a context semi-instance $I = (S_I, \rho_I)$ such that S_I is a context schema.

The next definition formalizes the relationship between a context schema and a context instance:

Definition 5 (Schema-instance relationship)**.** Let $I = (S_I, \rho_I)$ be a context instance, where $S_I = (N_I, E_I, r_I, Att_I, \alpha_I, \lambda_I)$, and $S = (N_S, E_S, r_S, Att_S, \alpha_S, \lambda_S)$ be a context schema. I is said to be an *instance of* S if there exist an injective function $h_N : N_I \to N_S$ between instance and schema node identifiers, and an injective function $h_A : Att_I \to Att_S$ between instance and schema attribute identifiers satisfying the following conditions:

(i) $h_N(r_I) = r_S$

(ii) for all $(n_1, n_2) \in E_I$, $(h_N(n_1), h_N(n_2)) \in E_S$

(iii) for all $n \in N_I$, $a \in Att_S$, if $\alpha_S(a) = h_N(n)$ then there exists $a_1 \in \alpha_I^{-1}(n)$ such that $a = h_A(a_1)$

(iv) for all $n \in N_I$ it holds that $\lambda_I(n) = \lambda_S(h_N(n))$, and for all $a \in Att_I$ it holds that $\lambda_I(a) = \lambda_S(h_A(a))$

Remark 2. Also an instance can be represented as an XML document, containing a subset of the elements of the document associated with the related schema, in which the placeholders of the attribute values are replaced by actual values.

As in database schema evolution, in this work we need to make instances of a certain schema "evolve losslessly", becoming instances of a different one. The problem of determining whether a transformation is information-preserving has been studied in the database literature, see for example [22, 23]. In particular, in this work we are interested in comparing the quantity of information contained in instances produced by different transformations. Intuitively, in our case a context instance I can be considered more informative than another one I' if it allows to perform a more precise data tailoring, that is, if I contains more dimension nodes (or more concept attributes a such that $\rho(a) \neq ALL$) than I'. The reason is that each dimension (or attribute of a concept node) contained in an instance represents a perspective that contributes to the tailoring process, and thus to refining the context-aware view.

Let $\hat{Att}^{\circ}_I$ be the set of attributes in an instance I associated with concept nodes and whose value is specified, i.e. $\hat{Att}^{\circ}_I = \{a \in Att_I : \alpha_I(a) \in N^{\circ}_I \wedge \rho_I(a) \neq ALL\}$. The *information level* of a context instance is defined as follows:

Definition 6 (Information level)**.** Given a context instance I, its *information level* is defined as $IL(I) = |N^{\bullet}_I| + |\hat{Att}^{\circ}_I|$.

Given a context semischema $S = (N, E, r, Att, \alpha, \lambda)$ and a node identifier $n \in N$, $parent(S, n)$ indicates a node $n' \in N$ such that $(n', n) \in E$. In addition, Table 1 contains some shorthands denoting useful sets.

2.3. Using Context for Data Tailoring

Context-based data tailoring can be performed on various data models (e.g., relational [13], XML [24], ontologies [25]), provided that the employed data model grants: (i) a mechanism to specify views over a dataset; (ii) a containment relation $\trianglelefteq$ between views, formalizing the fact that a view "contains less information" than another one (see [26] for the XML model); (iii) an intersection operation $\barwedge$ between views, used to identify the portion of information that is common between the input views. Optionally, the data model can also provide a union operation $\uplus$ between views, with the intuitive meaning.

Data tailoring is a responsibility of the designer, and consists in associating every context instance with the view representing the data relevant for that context. Given a CDT, the great number of possible context instances makes the manual association of a view with each of them impractical even at design time. In our solution, the designer only associates a view with each concept node and with each dimension node with attributes; then, the system automatically generates one view for each context by combining these views in a suitable way [13].

Formally, let $S = (N, E, r, Att, \alpha, \lambda)$ be a CDT and let $\mathcal{VIEWS}$ be the set of possible views over the dataset of interest. A function $\mathcal{R}el : N^{\circ} \cup \bar{N}^{\bullet} \rightarrow \mathcal{VIEWS}$ must be defined by the designer. The view definitions assigned by $\mathcal{R}el$ to nodes with attributes transform these attributes into parameters of the view. For example, in the context schema in Figure 1 the `movie` interest topic features the attribute `language`, which becomes the parameter *language* of the associated view. Given a node n, the definition of $\mathcal{R}el(n)$ may contain all the attributes associated with the nodes belonging to $\widetilde{asc}(n)$. The attribute values are defined by the function ρ_I of Definition 3.

Let $I = (S_I, \rho_I)$, with $S_I = (N_I, E_I, r_I, \alpha_I, \lambda_I)$, be an instance of the context schema S. The view associated with I is obtained as the intersection of all the views associated with the leaves of the tree (N_I, E_I). Let us introduce the function $View(I) : \mathcal{I} \rightarrow \mathcal{VIEWS}$:

$$View(I) = \bigbarwedge_{n \in \bar{N}_I} \mathcal{R}el_{\rho_I}(h_N(n)) \qquad (1)$$

where $\mathcal{R}el_{\rho_I}(h_N(n))$ is the view in which the attributes take the values dictated by ρ_I.

This approach affords a high degree of flexibility also with respect to evolution; indeed the designer, after the evolution from a context schema to a new one, has only to revise or add the (limited number of) views associated with the nodes that have been modified by the evolution operation, and the combination of these new views with the other ones to form the new context-related views is performed automatically.

Note that in this scenario the definition of the $\mathcal{R}el$ function is manually performed by the designer respecting the hierarchical structure of the context schema [5]. This means that, quite naturally, the nodes situated in the lower parts of the tree are associated with "more detailed" views than their ancestors, i.e. $n_i \in desc(n_j) \Rightarrow \mathcal{R}el(n_i) \trianglelefteq \mathcal{R}el(n_j)$. If the data model also defines a union operation, the designer effort can be further reduced by automatically computing bottom-up the views related to the internal nodes, as unions of the ones associated with the first generation of their concept descendants and their dimension children with attributes:

$$\mathcal{R}el(n) = \biguplus_{\substack{\{n_i \in N^{\circ} : n = parent(S, parent(S, n_i))\} \cup \\ \{n_i \in \bar{N}^{\bullet} : n = parent(S, n_i)\}}} \mathcal{R}el(n_i) \qquad (2)$$

In this case, the designer task becomes even lighter, since she only has to define the views associated with the leaf nodes. Only the view associated with the root is not computed by union: by definition, it corresponds to the whole database.

In a more autonomic scenario [27], the context-aware views can be *learned* by the system, which analyzes the behavior of the various users in the different contexts. Accordingly, when the designer decides some CDT change, the context instances are also changed following the policy we define in this paper, and the associated views are adapted automatically as the system learns from the user behaviors in the newly defined contexts.

3. A Brief Account of the Research on Schema Evolution

The need of managing schema modifications to make applications resilient to changing requirements has grown in the 80s within object-oriented databases, in the scope of CAD/CAM systems [28, 29], and periodically becomes relevant again when new data models or IT components arise in the technological scenario. More recently, the problem has been widely studied for relational databases [30, 31, 32], ontologies [33, 34, 35], XML [36, 37, 38], and in the web domain [39, 40]. Historically, two different approaches have been used to cope with schema modifications [41]: *evolution* and *versioning*. When schema evolution is considered, the old schema version is replaced by the new one, and techniques to keep on dealing with all the entities associated with the old schemas have to be provided. In the literature, these entities have been

Table 1: Shorthands denoting useful sets

- $children(S, n) = \{n' \in N : (n, n') \in E\}$
- $siblings(S, n) = \{n' \in N : n' \neq n \wedge parent(S, n) = parent(S, n')\}$
- $desc(S, n) = \{n' \in N : (n, n') \in E^+\}$
- $\widetilde{desc}(S, n) = desc(S, n) \cup \{n\}$
- $\widetilde{asc}(S, n) = \{n' \in N : (n', n) \in E^+\} \cup \{n\}$

data instances [28], queries [31], mappings [42], other related schemas [43]; [44] have considered transformations between data models, and studied how to update schemas defined in those data models. On the contrary, in the versioning approach all the past schemas are retained and kept operating, with no adaptation needs.

We are interested in the evolution problem. When a database schema changes, in fact, the new schema is often just a different way of organizing the same data instances, and thus it is reasonable to keep the old schemas active. By contrast, context schema modifications reflect changes in the modeled reality: the old context schemas describe context instances now obsolete and no more applicable, and keeping them operating makes little sense.

To the best of our knowledge, the only existing proposal about context schema evolution is that of De Virgilio and Torlone [45], which defines a general framework to support the representation and management of a large variety of context information. They introduce the notion of *profile schemes* as trees composed of sets of dimensions, associated with attributes; *profile instances* are defined assigning values to the attributes. In their framework, the designer should produce mappings associating the old schema versions with the current one; then, starting from these mappings, the paper mentions also a translation function for the instances. This methodology is very general, but only sketched; moreover, it requires a very onerous work for the designer.

This paper presents the first full-fledged proposal for context schema evolution, also taking into account ideas from the existing evolution methods in other fields. The literature on the evolution problem is extensive (see [46] for a recent survey, or [47] for a complete list of references) and cannot be exhaustively analyzed here, so we first summarize the main approaches proposed for the relational, object-oriented and ontological data models (Section 3.1). Then, the proposals related to XML, which are the most relevant to our work because of the hierarchical nature of our context model, are presented in detail, also analyzing which of the features that we considered important for our aims are studied in the various techniques (Section 3.2). Finally, we describe a very interesting approach which is not bound to any specific data model (Section 3.3). The detailed comparison of our methodology with the most interesting ones from the literature is postponed to the end of the paper, in Section 9, where we will explain why we chose to develop a framework from scratch rather than trying to apply one of existing approaches to our scenario.

3.1. Schema Evolution in Relational Databases, Object-Oriented Databases and Ontologies

Several schema evolution methodologies have employed predefined evolution operators. A very cited system dealing with object-oriented schema evolution is Orion [28], exploiting a set of operators to modify the schemas and to migrate instance data accordingly; each schema has to fulfill a set of invariants, and a set of rules to resolve possible invariant violations caused by the application of the operators is provided. Operators are used also in the relational setting in the Prism framework [31]; each operator is mapped to a logical representation in terms of disjunctive embedded dependencies, that are used to rewrite queries. Within ontologies, the Kaon system [33] provides a graphical user interface allowing to apply sixteen predefined changes, divided in elementary and composite ones; the changes are then propagated to the instances of the modified ontology as well as to related ontologies.

Other works, like [48] in the relational setting and [35] in the ontological one, allow the designer to apply arbitrary changes and then try to infer which of the changes in a predefined set have been applied. The inference may be approximate and require interaction with the designer. Since ours is the first attempt to manage the evolution of context schemas, we suppose that the changes to be applied are explicitly specified, leaving the change inference problem as a future work.

Finally, it is worth mentioning that [49] suggests the applicability of mapping composition techniques to manage relational schema evolution: the designer defines the mappings between the old schema and the new one, and these mappings are used to migrate instances. However, proceeding this way the designer has to specify complex mappings between schemas and, similarly to [45], this may result in a very hard task.

3.2. XML Schema Evolution

Approaches to define mappings between schemas have been defined in the scope of XML [50] too, and such tech-

niques could be adopted to manage schema evolution. However, similarly to the relational case, the mapping definition may be an onerous task for the designer.

Paper [51] defines some evolution primitives for XML documents, and the corresponding XQuery extensions supporting evolution features; however, problems related to schemas are not considered at all. Other works deal with the update of XML documents to support their efficient incremental validation with respect to a fixed schema: [52, 53, 54] consider an XML document as a tree and define some updating primitives. These approaches consider changing instances w.r.t. to fixed schemas, whereas we need to study instance adaptation as consequence of a schema change.

Some strategies exploit the theory of formal languages: they represent XML schemas through grammars, and their evolutions by evolving such grammars. Hashimoto et al. [55] propose update operations to modify schemas, proving their soundness and completeness; however, the corresponding document adaptation is not considered. Chabin et al. [56] represent DTDs by means of local tree grammars, and to update the schema the designer has to provide a new grammar; the authors propose to cope with the evolution by finding the least local tree grammar able to generate the union of the languages associated with the old and the new one, thus encompassing the old instances as well as the new ones. Shoaran and Thomo [57] introduce insert and delete operations to add or remove substrings from the language associated with the schema with the purpose of making the schema more tolerant; techniques to find the automaton recognizing the language connected to the new schema are proposed. The latter two strategies tackle the evolution problem with the aim of building a schema more "tolerant" than the original one, while in our scenario we also have to consider that the evolution might invalidate some instances.

Other works have considered schema evolution along with the associated instance adaptation. Tan and Goh [58] define operators to evolve XSDs, while Klettke [59] and Dominguez et al. [60] propose to apply the modifications to conceptual models, and to propagate them first to XML schemas and then to the documents; these proposals describe systems only under a practical point of view. Su et al. [36], instead, propose operators to evolve DTDs, providing detailed descriptions of preconditions and effects of their applications similar to ours; however, instance update is covered in a shallow way. An extension of this framework with high-level operators is given by Prashant and Kumar [61]. A formal characterization of operators is introduced by Guerrini et al. [37, 38]. They propose atomic and high-level modification primitives for XSD describing their preconditions and semantics; algorithms for partial revalidation and adaptation of documents dependent on the applied primitives are also defined. This approach is the most similar to ours, and we will compare it with our framework in Section 9.

The primary goal of our research is to provide a formal framework which is sufficiently general to be easily applicable to other context models, with proofs of soundness and completeness. As a second objective, we want to propose a usable evolution infrastructure, so we choose to employ intuitive operators, also providing high-level ones expressing common evolution needs. Moreover, in our framework it is necessary not only to adapt the instances of the most recent schema, but also to deal with older ones by applying sequences of operators; we will show that in certain cases these sequences need to be optimized.

Table 2 classifies the described operator-based approaches to XML schema evolution w.r.t. the above-cited needs; ~ indicates that the problem is addressed only partially, or that not enough details are given in the paper. In particular, Guerrini et al. provide an optimization technique in a subsequent paper [62], but considering a different (and simpler) set of operators. As noted already, in the case of context schema evolution we also have to consider how changes are propagated to the associations between contexts and views; obviously the works about XML schema evolution do not deal with such problems.

Table 2: Operator-based proposals for XML schema evolution

	Instance update	Formalization	Sound./Compl. proofs	Optimization	High-level operators
Hashimoto et al. [55]		✓	✓		
Shoaran and Thomo [57]		✓			
Tan and Goh [58]	~				
Klettke [59]	✓			~	
Dominguez et al. [60]	✓				
Su et al. [36]	✓	~			
Prashant and Kumar [61]	✓				✓
Guerrini et al. [37, 38]	✓	✓		~	✓

All the methodologies listed in Table 2 propose a set of operators deemed useful to evolve XML schemas, expressed through DTDs, XSDs or conceptual schemas. The table highlights how all of them do not take into account some aspects that we believe important for the development of a context schema evolution approach. However, we could choose one of them, represent our context schemas as needed by the chosen methodology (e.g., with DTD or XSD), and then modify and extend the methodology in order to deal with context schema evolution. In Section 9 our techniques are compared in detail with the strategy of Guerrini et al. [37, 38], that according to Table 2 is the approach that provides some operators and conditions similar to ours, to show the points in which this approach cannot represent fully the context modification operators, since their pre- and post-conditions fall short of representing the ones that are needed.

3.3. Schema Evolution at a Conceptual Level

Poulovassilis and McBrien [63] propose a formalism to define schemas as hypergraphs constituted by nodes, edges and constraints. This formalism can be used to provide a common representation for schemas defined through different data models. Instances are described as sets of sets, with a function putting them in relationship with their schema. The schema transformation and the consequent instance adaptation problems are tackled by introducing primitive transformations allowing insertions and deletions of nodes, edges and constraints. The authors show how the ER model can be defined in terms of their hypergraphs, and how intuitive ER transformations are expressible by composing the primitives proposed in this paper.

The framework presented by this paper is very general and interesting, thus in Section 9 we will propose a detailed comparison of our approach also with this methodology.

4. A Framework for Context Schema Evolution

Figure 3 gives an overview of the framework we conceived to manage context schema evolution, supposing w.l.o.g. that all the data resides in a global database[1] on a central server. There are three actors: the server, the user device and the designer. The server stores the global database and performs all the activities needed to manage the context, including evolution management and data tailoring. At every context change, the user's device sends to the server the context instance describing her new situation and requests the related data. The *tailoring* module associates each context instance with a view over the global database, by combining the views related to nodes as described in Section 2.3[2].

During system life-time, the designer may modify the context schema, but only by using a set of predefined evolution operators. The sequence of the modifications performed by the designer is logged by the *history of the applied operators* component, making it possible to reconstruct which operators have led from a version of the context schema to another one.

The application of the evolution operators modifies the context schema, possibly obliging the designer to redefine the views related to some nodes. As will be clear in the rest of the paper, the operators have been defined in such a way as to immediately identify the parts of the CDT that have been modified, and consequently the views that need revision or must be added from scratch. In general, the user device is unaware of the schema evolution, thus might, at a certain point, communicate an outdated context instance to the server. Then, the *instance update* module converts it into an instance of the current CDT, named *evolved instance* in Figure 3. This transformation relies on the modification log.

Let us explain in detail the interaction between the user device and the system through an example. Consider a user Alice, who is in the context described by the instance of Figure 2. Suppose that, as assumed in our introductory example, the company has decided that the data should not be tailored any more on the basis of the kind of user; this entails the deletion of the `user` dimension from the context schema of Figure 1. Moreover, imagine that Alice's client application, obviously not aware of this change, starts the interaction with the system sending to the server the context instance in Figure 2 (coherent with the old schema, Figure 1), and expecting to obtain the related context-dependent view. Together with the instance, Alice's device also sends the CDT of Figure 1; this allows the system to understand whether the received instance is expressed in terms of the current context schema or of an obsolete one. Since Alice's CDT is indeed obsolete, the server computes the sequence of operators that have led from the schema in Figure 1 to the current one, and the instance in Figure 2 is transformed into an instance of the new schema. In the example this trivially means eliminating the `user` dimension and (consequently) the view associated with the `adult` node from the intersection which computes the contextual view. If Alice's client application is flexible enough to update its internal context representation, the new schema is now sent to Alice's device; otherwise, the client application will continue to use the old schema, and the server will evolve the context instances produced by the client each time this operation is required. Note that, in both cases, Alice herself is completely unaware of the evolution management process.

5. Evolution Operators

In this section we define (i) the declarative semantics of the evolution operators used by the designer to update the context schema and (ii) the transformations that the instances undergo after the application of each operator.

An evolution operator *op* describes the features of the resulting schema in terms of the source one and of some other parameters. Each operator *op* is also associated with a function IE_{op} specifying the effects induced on the instances; this function adapts the instances to the new schema, *preserving as much information as possible*. Each operator is characterized by a set of preconditions, imposing some restrictions on the schemas to which they are applicable; preconditions are expressed through first-order formulas.

Given a context schema S_S, the execution of an operator *op* with input parameters $p_1, \ldots, p_n$ produces – if the preconditions are fulfilled – a new schema $S_T = op(S_S, p_1, \ldots, p_n)$. The effect of this transformation on a legal instance I_S of S_S is the production of a new instance $I_T = \mathsf{IE}_{op}\ (S_S, S_T, I_S, p_1, \ldots, p_n)$.

[1] Note that the fact that the database itself be centralized or not is irrelevant w.r.t. this discussion. What is important is that the views of $\mathcal{VIEWS}$ be defined over some global schema.

[2] In the view-learning scenario of [27], the tailoring module does nothing more than assigning periodically the view definitions computed by the learning system to the current context instance.

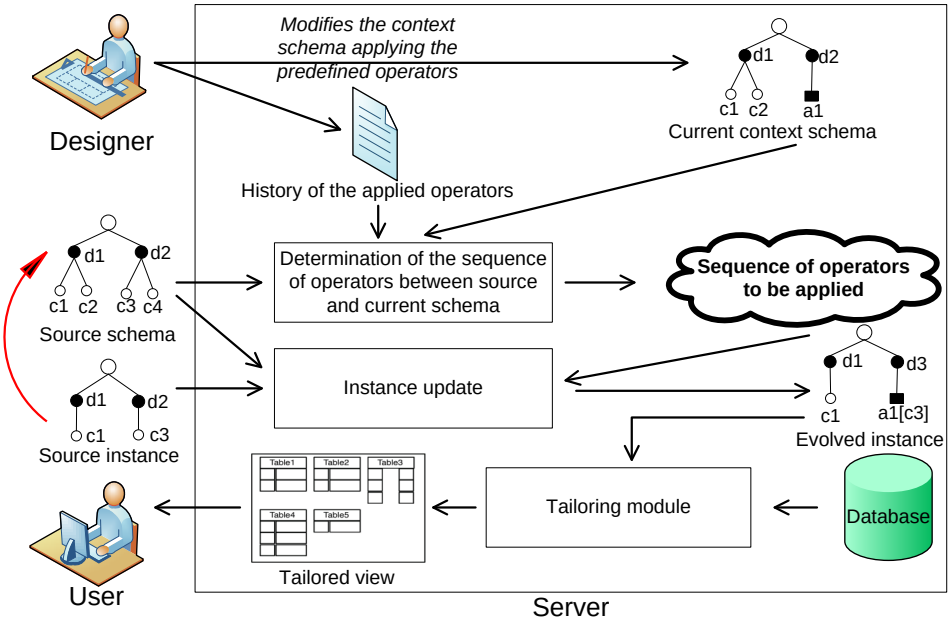


Figure 3: Framework for context schema evolution

First, we define the set of the *atomic evolution operators*, which has the following features: i) **completeness**: this set of evolution operators is sufficient to define the evolution to any valid target context schema; ii) **soundness**: any sequence of these operators is guaranteed to produce a context schema complying with Definition 2.

Do note that the application of these operators on a schema triggers a sound instance adaptation, i.e. the update induced on the instances produces context instances complying with Definition 4, and maintains the consistency between the resulting instances and the corresponding (evolved) schema. We remark that, by contrast, it makes no sense to require that the instance adaptation be complete: in fact, instance adaptation is conceived to define how the instances have to be modified as a consequence of a schema evolution, and not to transform an instance into another, arbitrary one.

Finally, the atomic operators along with their effects on the instances are **minimal**: for each atomic operator, no combination of other atomic operators exists that can produce the same evolved schema and effects on the instances.

Later, we introduce *high-level evolution operators*, which can be expressed as sequences of atomic ones, producing the same effects on both schemas and instances; they represent common evolution needs in a more compact way.

The set of atomic operators is indicated by $\mathbb{OP}_{AT}$, and the set of high-level operators by $\mathbb{OP}_{HL}$; $\mathbb{OP} = \mathbb{OP}_{AT} \cup \mathbb{OP}_{HL}$.

5.1. Basic Atomic Evolution Operators and their Completeness

In this subsection we describe the first atomic operators, *Delete* and *Insert*. Their preconditions and semantics, along with their effect on instances, are formally described in the appendix (Table A.1). A textual description follows.

Delete: the *Delete* operator eliminates the subtree rooted in a node n from the source schema S_S. As preconditions, since a dimension must have either an attribute (when it is a leaf) or at least a concept child, *Delete* can only be applied to remove either a dimension node and its subtree, or a concept node – if it has at least another sibling – and its subtree. The effect of *Delete* on an instance of the input schema eliminates from the instance the subtree rooted in the node with identifier $h_N(n)$, if such a node is present; moreover, if n is a concept, in order not to produce a leaf dimension, in the instance also the image through h_N of the parent of n must be eliminated, together with the edge connecting it with $h_N(n)$. In Table A.1 this instance transformation is called IE_{Delete}.

Insert: given a source schema S_S, the operator *Insert* inserts a semischema R as a child of a specified node identified by n. The identifiers contained in the semischema must be different from the ones in S_S, and the correct type alternation between pairs of node generations must be preserved; moreover, the labels of the nodes and the attributes in the semischema must be different from those already used in S_S, in order to not introduce label conflicts. Since *Insert* does not alter the existing nodes and attributes, the instances are not affected at all (see IE_{Insert}

in Table A.1).

In the figures that follow, for simplicity, nodes and attributes are identified through their labels.

Example 1 (Basic atomic operators)**.** Figure 4 shows the application of the described operators to the CDT illustrated in Figure 1 (left hand side), and the resulting modifications of the instance depicted in Figure 2 (right hand side). The changes are briefly described in the following:

1. The dimension labeled `time` is eliminated; the deletion is applied to the instance too.
2. The subdimension labeled `time` is added under the `movie` node; according to the IE_{Insert} semantics, the insertion in the CDT does not affect the instance.

The set of basic atomic operators is sufficient to express all the possible schema modifications, i.e., it is *complete*. Our notion of completeness is similar to that of [28, 36, 37].

Theorem 1 (Completeness)**.** *Given two arbitrary CDTs S_1 and S_2, it is possible to find a finite sequence of operators belonging to {Insert, Delete} that transforms S_1 into S_2.*

Proof. Let us consider two context schemas S_S and S_T. Let $c_{S1}, \ldots, c_{Sn}$ be the children of r_S and $c_{T1}, \ldots, c_{Tm}$ the children of r_T. Moreover, let $T_{T1}, \ldots, T_{Tm}$ be the subtrees rooted in $c_{T1}, \ldots, c_{Tm}$.

The following sequence of *Delete* and *Insert* operators builds S_T starting from S_S:

- $S_0 = S_S$
- for i: $1, \ldots, n$, $S_i = Delete(S_{i-1}, c_{S_i})$
- $S_0 = S_n$
- for i: $1, \ldots, m$, $S_i = Insert(S_{i-1}, T_{Ti}, r_{S_i})$
- $S_T = S_m$

□

5.2. Methodological Considerations and Further Atomic Operators

Let us analyze Example 1 in more detail: at step 1, the subtree rooted in the dimension named `time` is eliminated from the schema, thus causing the same deletion from all the instances that contain it. At step 2 a subtree identical – syntactically and semantically – to the one that has been deleted is inserted under the concept node `movie`; however, according to the IE_{Insert} semantics, the instance remains unaltered when new information is added to the schema. Nevertheless, such a sequence of changes might intuitively represent a "*move*" operation, that is, in the designer's intention, the time information has probably become relevant only for those users who are interested in movies. The initial instance indicates an interest both in the movies and in time but, when the time information is deleted, this aspect is completely lost in the context instance and does not influence the subsequent insertion. Therefore the (elsewhere reasonable) effects of the *Delete* and *Insert* operators on the instances in this case result in the loss of the information related to time, due to the fact that the evolution process "forgets" the deleted subtree, taking care only of the information contained in the most recent schema and instances. A similar issue has been considered also by Lerner [48], that in the scenario of the evolution of object-oriented types illustrates a similar need to move an attribute from a type to another one.

The problem can be solved by storing the eliminated subtrees in order to facilitate later reintegration, if necessary. We enrich *Delete* with two functions: the *schema cache* SC_{Delete} and the *instance cache* IC_{Delete} (Table A.2 of the appendix). The former returns the cached content after its deletion from the schema, while the latter does the same after the deletion from the instance. The new *Delete* entails the caching of the semischema $S_M = \mathsf{SC}_{Delete}(S_S, S_T, n)$, while IE_{Delete} is associated with the caching of the semi-instance $I_M = \mathsf{IC}_{Delete}(S_S, S_T, S_M, I_S, I_T, n)$; for reasons that will be clear later in the paper, the node n_M with which S_M was connected in the source schema is cached too. Note that SC_{Delete} and IC_{Delete} do not implement any schema or instance modification: they only define the information to be cached after a deletion. The stored content is available to be used by an insertion if this is executed immediately after the deletion, then it is purged. Note that now, in order to exploit the content stored during the deletion, a "memory-aware" insertion operator will be necessary.

Consider now the schema and the instance obtained after the evolution described in Example 1. Suppose first that the designer deletes from the CDT the nodes `adult` and `teenager` – children of `user` –, triggering the removal of the `user` dimension from the associated instance; then, she inserts a *new* node `person` under the same dimension `user`. According to the IE_{Insert} semantics, the instance remains unaltered, because the instance where IE_{Insert} is applied now does not carry any information about the new node. Nevertheless, such a sequence of changes might intuitively represent a "merge" operation, that is, in the designer's aims, the added node is meant as a substitute for both `adult` and `teenager`; the sequence of deletions and insertions, though able to modify the schema according to the designer's intentions, did not modify the instance as intended. Again, it turns out that, even if *Insert* and *Delete* are enough to achieve schema update completeness, the designer might need more atomic operators inducing useful behaviors *on the instances* and not obtainable as combinations of insertions and deletions.

To satisfy this need we add two atomic operators to $\mathbb{OP}_{AT}$: a "memory-aware" insertion *InsertFromMemory*, and then a *Merge* operator. We also add a further operator whose need we have noticed: the *ReplaceSubtreesWithAttribute*. This is useful when the designer deems the hierarchy underlying the dimension node no more interest-

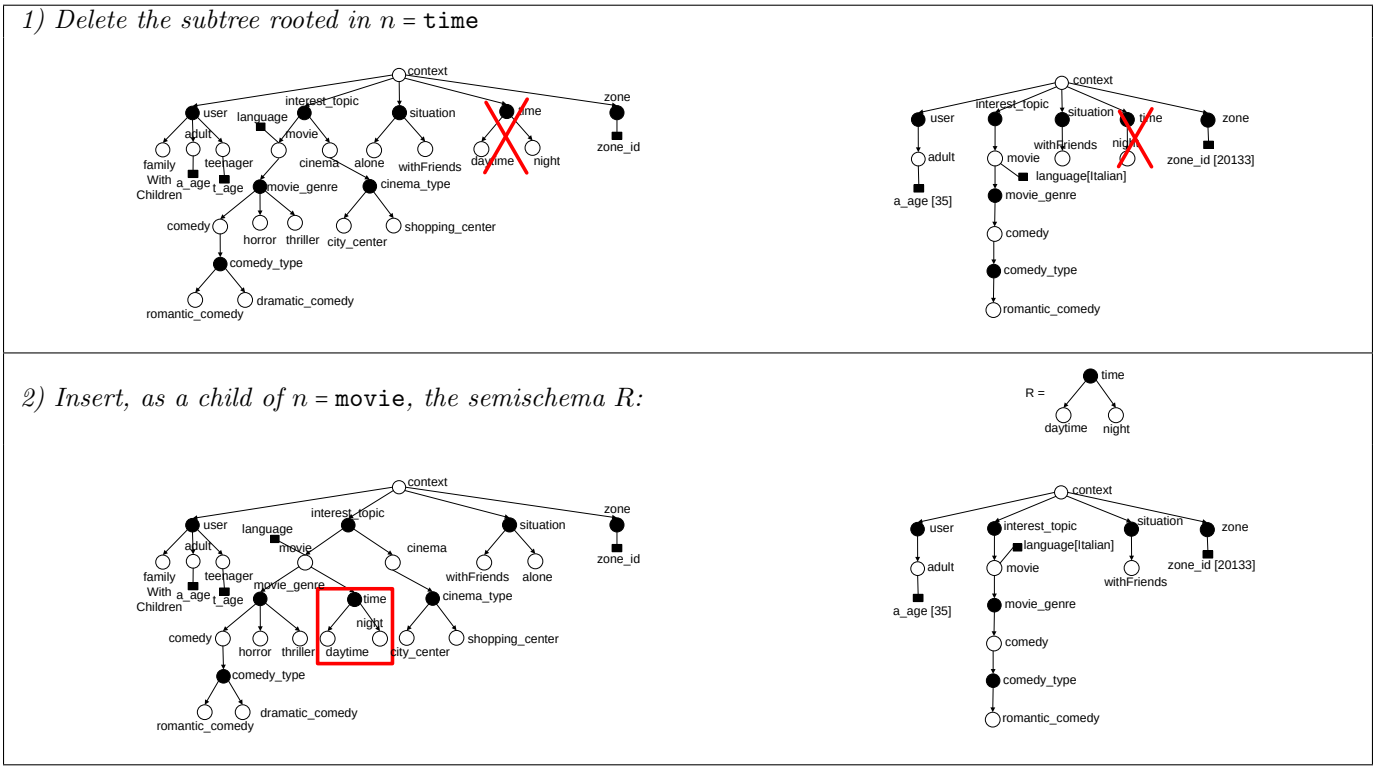


Figure 4: Application of the basic atomic operators described in Example 1

ing. Example 3 shows an instance of this case. Remark that this set of new operators has been determined according to our intuition about the possible changes that may take place as consequences of the dynamism of the application requirements; therefore, it may be further widened. Moreover, note also that despite the memory-aware insertion *InsertFromMemory* has been defined, we still retain the standard insertion operator *Insert*, which is necessary when the designer wants to insert in the context schema a completely new semischema, not resulting from a deletion.

A textual description of the new operators follows, while their preconditions and semantics are shown in Tables A.3 and A.4 of the appendix.

InsertFromMemory: *InsertFromMemory* is similar to *Insert* but takes as additional the cached semischema S_M; the associated function $\mathsf{IE}_{InsertFM}$ takes as input also the semi-instance I_M. Therefore $S_T = InsertFM(S_S, S_M, R, n)$ and $I_T = \mathsf{IE}_{InsertFM}(S_S, S_T, S_M, I_S, I_M, R, n)$. The operator *InsertFromMemory* behaves exactly as *Insert* does, but the semischema R inserted under the node n is retrieved from the cache (if any); it is allowed to modify attributes and labels, while nodes and edges have to remain the same of the semischema stored in the memory. If n is a concept node, the memory contains a semi-instance, and the source instance contains the node corresponding to n, then $\mathsf{IE}_{InsertFM}$ reinserts the stored semi-instance. On the contrary, if n is a dimension node, the semi-instance is reintegrated only if the semischema has been reinserted exactly in the same position, thus simply rolling back the previous deletion; in fact, if the stored semischema were moved, the reinsertion would cause the presence of white siblings in the instance (forbidden by Definition 3). Note that *InsertFromMemory* must necessarily follow a *Delete* operation. A specific precondition guarantees this fact, by requiring the presence of a non-empty cached semischema (see the appendix for details), which can only be produced by a *Delete* operation. Note also that *InsertFromMemory* allows only to reinsert a previously eliminated subtree, and not to duplicate a subtree in a different position; copying a subtree would produce a context schema inconsistent with Definition 1, since duplicated identifiers are not allowed.

Merge: the *Merge* operator merges a set of concept siblings $\{m_1, \ldots, m_p\}$ into a unique node labeled ℓ; the new node will have all the attributes previously connected to the replaced nodes. The root cannot be involved in a merging. The label ℓ must be different from the labels already in use in the schema, with the exception of those of the nodes $\{m_1, \ldots, m_p\}$ that are being removed. If an instance contains a node corresponding to one of the merged ones, IE_{Merge} substitutes it with a new node labeled ℓ. It is immediate to see that this operator is atomic, because its effect on instances IE_{Merge} cannot be obtained by combining IE_{Delete} and IE_{Insert}.

ReplaceSubtreesWithAttribute: *ReplaceSubtrees-WithAttribute (RSWA)* replaces all the subtrees rooted in the (concept) children of a dimension node identified by n with an attribute labeled ℓ; the label ℓ cannot be among those associated with nodes and attributes of the source

schema that are not part of the replaced subtrees. The effect IE_{RSWA} updates an instance if it contains a node identifier k corresponding to one of the children of n; in such a situation the subtree rooted in k is replaced by the new attribute, whose value will be the label of k. This operator is atomic for the same reason as the previous one.

The following theorem states that after the application of *Merge* the information level of the instances is greater or equal than after the application of a sequence of *Delete* and *Insert*.

Theorem 2. *Given the context schemas S_S and $S_T = Merge(S_S, \{m_1, \ldots, m_p\}, \ell)$, and the context instance I_S of S_S, let $I_M = \mathsf{IE}_{Merge}(S_S, S_T, I_S, \{m_1, \ldots, m_p\}, \ell)$. For each instance I_D obtained as the effect of a sequence of Delete and Insert producing the same results as Merge on the schema, it holds that:*

(i) $IL(I_D) \leq IL(I_M)$

(ii) If there exists a "witness node" n_1 in I_S such that $h_N(n_1) \in \{m_1, \ldots, m_p\}$, then $IL(I_D) < IL(I_M)$

Proof. Suppose that there exists a witness node $n_1 \in N_{IS}$ such that $h_N(n_1) \in \{m_1, \ldots, m_p\}$, and consider the semantics of IE_{Merge}. By definition of N_{IT}, variations to the set of nodes concern only concept nodes, therefore $N^{\bullet}_{IM} = N^{\bullet}_{IS}$. Moreover, according to the definitions of Att_{IM} and ρ_{IM} some attributes are added to Att_{IS}, but all of them take the value ALL. Therefore also $\hat{Att}^{\circ}_{IM} = \hat{Att}^{\circ}_{IS}$, and $IL(I_M) = IL(I_S)$. Suppose now that there is no $n_1 \in N_{IS}$ such that $h_N(n_1) \in \{m_1, \ldots, m_p\}$; according to the IE_{Merge} semantics $I_M = I_S$, and so $IL(I_M) = IL(I_S)$. Thus, in any case, after the application of Merge, $IL(I_M) = IL(I_S)$.

Let us consider a sequence of insertions and deletions producing the same schema obtained with Merge, and an instance I_D; such a sequence exists due to Theorem 1. IE_{Insert} leaves the instances unchanged, so the information level of I_D is determined exclusively by the deletions included in the sequence. IE_{Delete} can only remove nodes and attributes from the instance, without adding anything, so $IL(I_D) \leq IL(I_S)$. Given that $IL(I_M) = IL(I_S)$, we have $IL(I_D) \leq IL(I_M)$, that is (i).

To prove (ii), consider the situation in which there exists the witness $n_1 \in N_{IS}$ such that $h_N(n_1) \in \{m_1, \ldots, m_p\}$. Let $l \in N_{IS}$ be such that $h_N(l) \in \{m_1, \ldots, m_p\}$. The sequence of Insert and Delete required to obtain S_T needs necessarily to eliminate all the subtrees rooted in $m_1, \ldots, m_p$ from S_S, in order to add the new node n as a child of $parent(S_S, m_1)$. The corresponding applications of IE_{Delete} cannot add new nodes or attributes, but according to the IE_{Delete} semantics it will surely remove the node $parent(I_S, l)$ from the instance. As a consequence, $|N^{\bullet}_{ID}| < |N^{\bullet}_{IS}|$ and $|\hat{Att}^{\circ}_{ID}| \leq |\hat{Att}^{\circ}_{IS}|$, and thus $IL(I_D) < IL(I_S) = IL(I_M)$. Therefore (ii) holds. □

Theorems 3 and 4 provides results analogous to Theorem 2 for *ReplaceSubtreesWithAttribute* and *InsertFromMemory*. Their proofs are similar to that of Theorem 2, and are omitted for brevity.

Theorem 3. *Given the context schemas S_S and $S_T = RSWA(S_S, a, \ell)$, and the context instance I_S of S_S, let $I_R = \mathsf{IE}_{RSWA}(S_S, S_T, I_S, a, \ell)$. For each instance I_D obtained as the effect of a sequence of Delete and Insert producing the same results as ReplaceSubtreesWithAttribute on the schema, it holds that:*

(i) $IL(I_D) \leq IL(I_R)$

(ii) If there exists a node n_1 in I_S such that $h_N(n_1) = \alpha_S(a)$, then $IL(I_D) < IL(I_M)$

Theorem 4. *Given the context schemas S_S and $S_T = InsertFM(S_S, S_M, R, n)$, and the context instance I_S of S_S, let $I_I = \mathsf{IE}_{InsertFM}(S_S, S_T, I_S, S_M, R, n)$. For each instance I_D obtained as the effect of a sequence of Delete and Insert producing the same results as InsertFromMemory on the schema, it holds that:*

(i) $IL(I_D) \leq IL(I_R)$

(ii) If the instance cache is not empty and either of the following is true:

(a) n is a white node and there exists a node n_1 in I_S such that $h_N(n_1) = n$

(b) $h_{NM}(r_{IM}) = n$

then $IL(I_D) < IL(I_M)$.

Note that a consequence of Theorem 4 is that the information level of the instance produced as an effect of an *Insert* is less than or equal to that of the instance obtained after *InsertFromMemory*. Intuitively, IE_{Insert} does not alter the source instance at all, while $\mathsf{IE}_{InsertFM}$ may recover previously cached dimensions and attributes.

Example 2 (InsertFromMemory)**.** Figure 5 shows two examples that revise steps 1 and 2 of Example 1 taking into account the cache functionality: step 1' considers the extension of the *Delete* operation with the cache, while Step 2' mimics step 2 of Example 1, but applying *InsertFromMemory* instead of *Insert*.

Note that, in the instance obtained after the application of *Insert* in Example 1, $|N^{\bullet}| = 6$ and $|\hat{Att}^{\circ}| = 2$, while in the instance derived applying *InsertFromMemory* it holds that $|N^{\bullet}| = 7$ and $|\hat{Att}^{\circ}| = 2$; therefore the former instance is less informative than the latter one.

Example 3 (Merge and RSWA)**.** Figure 6 shows the application of the *Merge* and *ReplaceSubtreesWithAttribute* operators to the context schema obtained after the operations described in Example 1, and the resulting modifications of the instance. The changes are briefly described in the following:

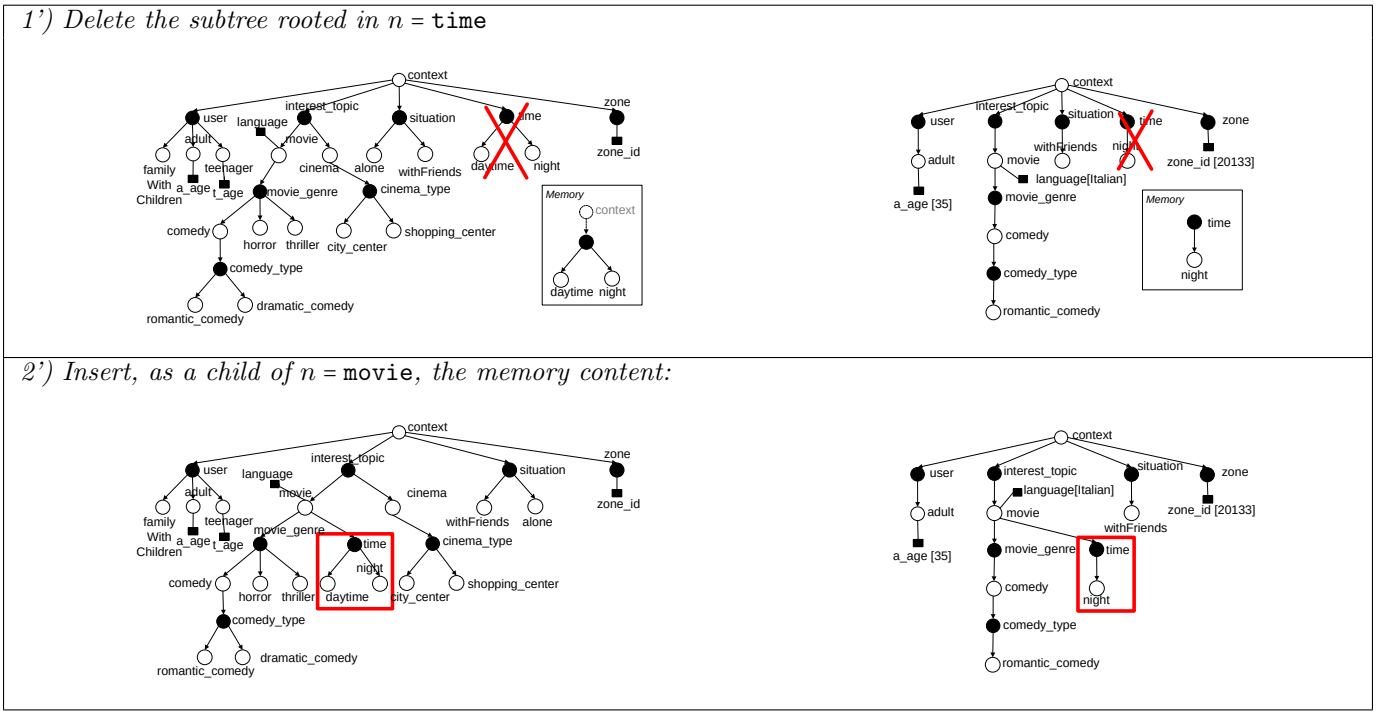


Figure 5: Application of *Delete* and *InsertFromMemory*, as described in Example 2

3. The nodes labeled `adult` and `teenager` are transformed into a unique node labeled `person`; in the instance, the node labeled `adult` is replaced by a node labeled `person`. Note that, in the instance obtained after *Merge*, $|N^\bullet| = 6$ and $|A\hat{t}t^\circ| = 2$; if a sequence of deletions and insertions had been employed, the instance would not have contained the `user` dimension, thus having $|N^\bullet| = 5$ and $|A\hat{t}t^\circ| = 1$. As a consequence the instance obtained with *Merge* is more informative.

4. The subtrees rooted in the dimension labeled `movie_genre` are replaced by a new attribute labeled `genre`; this change means that the classification among various kinds of comedies is considered as no more useful for the application, and that movies with further and unpredictable genres are expected to come into the catalog. In the instance the subtree rooted in `comedy` is removed, and the new attribute assumes the value `comedy`. Note that in the instance obtained after *RSWA* $|N^\bullet| = 5$ and $|A\hat{t}t^\circ| = 2$; if a sequence of deletions and insertions had been employed, the instance would have not contained the `movie_genre` dimension, thus having $|N^\bullet| = 4$ and $|A\hat{t}t^\circ| = 2$. As a consequence the instance obtained with *ReplaceSubtreesWithAttribute* is more informative.

5.3. Further Fundamental Properties of the Atomic Operators

In Section 5.1 the completeness of the atomic operators has been proven. In this section, we study two more fundamental properties of our evolution framework: soundness and minimality.

The following two theorems deal with the soundness of the evolution process. If an evolution operator is applied to a legal context schema fulfilling the preconditions, it should produce a legal context schema according to Definition 2. Moreover, the adaptation of an instance as effect of the evolution must be such that its outcome is: (i) a legal context instance according to Definition 4 and (ii) an instance of the schema produced by the evolution operator, according to Definition 5 [64].

Theorem 5 (Soundness of the schema evolution)**.** *Let S_S be a context schema, $op \in \mathbb{OP}_{AT}$ and $p_1, \ldots, p_n$ the additional parameters required by op, then $op(S_S, p_1, \ldots, p_n)$ gives as result a context schema S_T.*

Proof. The soundness should be proven separately for each operator. We choose to show the proof for Merge: the others follow a similar pattern. Note that in the proof we use the definitions of the components of the target schema and the preconditions of Merge, whose formal details can be found in the appendix.

Let $\{m_1, \ldots, m_p\} \subset N_S$ and $\ell \in \mathcal{L}$ be the input parameters of Merge, and let n be the identifier of the new node labeled ℓ inserted into the new schema. S_T is obtained from S_S by eliminating the concept siblings (preconditions 2 and 3, and definition of N_T), the edges involving them (definition of E_T) and inserting the node identifier n; such a node identifier is connected with the parent of $\{m_1, \ldots, m_p\}$ and to all their children (definition of E_T). It is easy to prove that S_T is a context schema. Indeed:

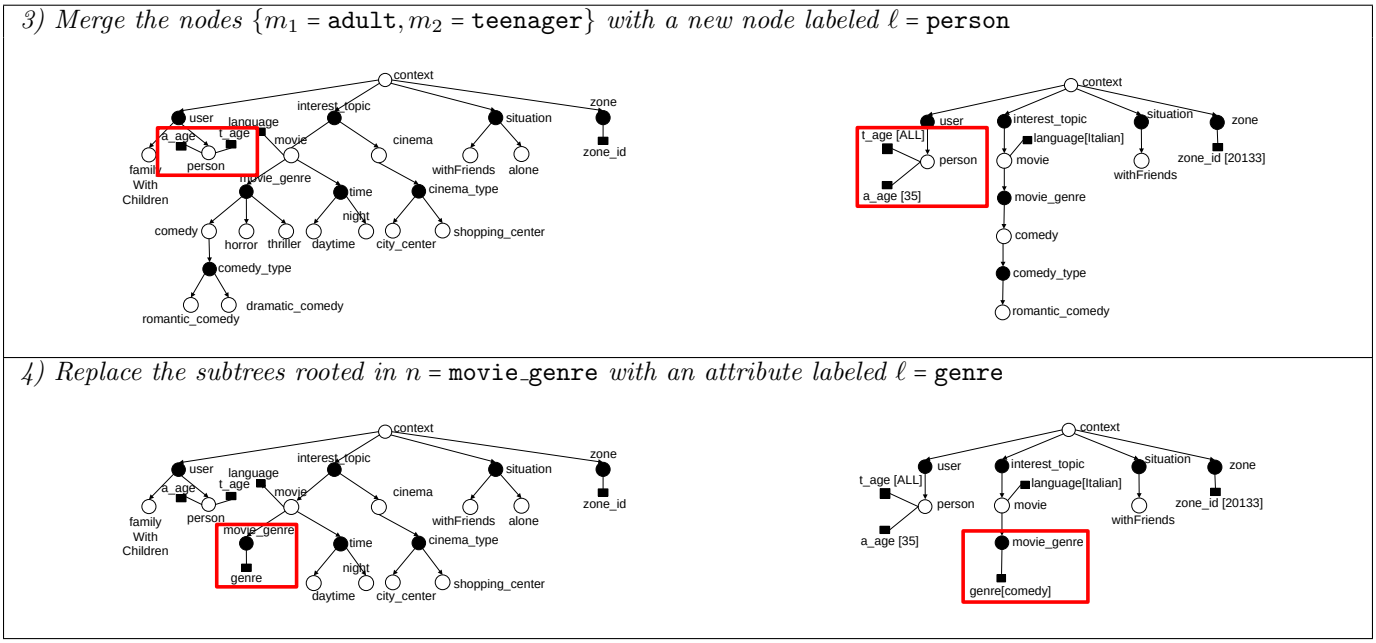


Figure 6: Application of *Merge* and *ReplaceSubtreesWithAttribute* operators, described in Example 3

- By definition of N_T and E_T, no dangling edges are present in S_T, and all the nodes are reachable starting from r_T. Therefore, condition (i) of Def. 1 is satisfied.
- Being n a concept node identifier (precondition 1) as $m_1, \ldots, m_p$, the fact that every generation contains node identifiers of the same type is preserved (condition (ii) of Def. 1).
- By definition of Att_T and α_T, each attribute is connected to one and only one node in N_T; moreover, the removed edges are replaced by a new node, therefore it is impossible to have a dimension lacking both concept children and an attribute (condition (iii) of Def. 1).
- Precondition 4 prevents label conflicts to arise (condition (iv) of Def. 1).
- The root, not modified according to precondition 2, is labeled `context` and is a concept node (Def. 2).

□

Theorem 6 (Soundness of the instance adaptation)**.** *Let S_S be a context schema, I_S an instance of S_S, $op \in \mathbb{OP}_{AT}$, $p_1, \ldots, p_n$ the additional parameters required by op, S_T the context schema result of $op(S_S, p_1, \ldots, p_n)$. The result I_T of $\mathsf{IE}_{op}(S_S, S_T, I_S, p_1, \ldots, p_n)$ is an instance of S_T.*

Proof. Again, we prove the theorem only for the Merge operator. Let $\{m_1, \ldots, m_p\} \subset \mathcal{N}$ and $\ell \in \mathcal{L}$ be the input parameters of Merge and IE_{Merge}.

Two facts have to be proven:

1. $I_T = (S_{IT}, \rho_{IT})$ is a context instance: S_{IT} is obtained by S_{IS} by only replacing, if present, a concept node $n_1 : h_N(n_1) \in \{m_1, \ldots, m_p\}$ with another concept node; therefore, condition (i) of Def. 3 is satisfied. Moreover, ρ_{IT} correctly assigns a value to all the attributes in Att_{IT} (condition (ii) of Def. 3).
2. $I_T = (S_{IT}, \rho_{IT})$ is an instance of S_T: Suppose that h_N and h_A be the functions relating I_S and S_S. Let $b_1 \ldots b_k$, $d_1 \ldots d_k$ and l be defined as in the semantics of the instance update (see the appendix). The functions between S_{IT} and S_T are defined as follows:
$$h'_N(n_1) = \begin{cases} n & \text{if } n_1 = l \\ h_N(n_1) & \text{otherwise} \end{cases}$$
$$h'_A(n_1) = \begin{cases} d_i & \text{if } n_1 = b_i \\ h_A(n_1) & \text{otherwise} \end{cases}$$
h'_N and h'_A are obtained considering the old node identifiers as before, and associating the possible new node identifier in S_{IT} with the new node identifier in S_T; the new attribute identifiers that the new node identifier takes from the siblings of $h_N(l)$ in the source schema are associated with the corresponding ones in the target schema. The four conditions of Def. 5 are therefore satisfied.

□

The following theorem states that the set of atomic operators is minimal:

Theorem 7 (Minimality)**.** *Given an atomic evolution operator Op and a schema S, there is no sequence of atomic evolution operators different from Op producing on S and on all its instances the same result as the application of Op.*

Proof. To prove the minimality of the atomic operators, we need to build, for each of them, an example of an evolution that can be performed using that operator, and cannot be obtained by applying only other ones. Examples of this kind have already been shown in the paper. Here we propose a formal proof for Merge; the other proofs proceed similarly.

For conciseness, let us represent a semischema through the identifier of its root and the list of semischemas – if any – that are children of the root, like this: *a[b[c], d[e]]*. Labels, attributes and attribute values, when relevant, can be indicated by explicitly specifying the functions λ, α and ρ.

We need Merge to evolve $S_1 = a[b[c, d]]$ to $S_2 = Merge(S_1, \{c, d\}, \ell) = a[b[e]]$ with $\lambda_2(e) = \ell$, updating the instance $I_1 = f[g[i]]$ of S_1 with $h_N(f) = a$, $h_N(g) = b$ and $h_N(i) = c$ to the instance $I_2 = \mathsf{IE}_{Merge}(I_1, S_1, S_2, \{c, d\}, \ell) = f[g[m]]$ with $h_N(m) = e$ and $\lambda_{I2}(m) = \ell$. This evolution could not be performed without involving Merge, because Merge is the only operation allowing to add to the target instance a node (like i) that was not present in the source instance or in previous versions of the instance. □

5.4. High-Level Evolution Operators

In this subsection we define four high-level schema evolution operators, allowing to move subtrees, rename nodes or attributes, insert and delete attributes. High-level operators are shortcuts for sequences of atomic ones: a high-level schema operator modifies the schema in the same way as the corresponding sequence of atomic ones, and triggers an identical update on the instances. Preconditions and semantics of the high-level operators, along with the semantics of the corresponding effects on the instances, are formally described in Table A.5 of the appendix. Here follows a textual description, also reporting the sequences of atomic operators needed to obtain the effects of the high-level ones.

Move: the operator *Move* moves the subtree rooted in the dimension node n as a child of the concept node identified by m; the latter cannot be a descendant of the former. If the moved subtree is also partially present in an instance, IE_{Move} keeps and moves it only if its new parent is contained too. If (a part of) the moved subtree is contained in the instance but its new parent is not present, the subtree is eliminated. Note that only movements of subtrees rooted in dimension nodes are considered; though in rare cases it could be sensible, e.g. to make `romantic_comedy` a real movie genre and not only a type of comedy, usually the movement of concepts is not useful, and therefore we have decided to ignore it in order to keep the semantics of *Move* simpler. *Move* can be expressed with atomic operators firstly by deleting the subtree rooted in n ($S_1 = Delete(S_S, n)$), and then by reinserting it – with no modifications – under m with $InsertFM(S_1, S_M, S_M, m)$.

Rename: the operator *Rename* renames a node or attribute, with identifier n, assigning ℓ as new label. It is necessary that the new label be not already in use. If necessary, IE_{Rename} performs the same renaming on the instances. Let R be the semischema obtained modifying the one rooted in n by changing to ℓ the label of n; *Rename* can be realized by using only atomic operators deleting the subtree rooted in n ($S_1 = Delete(S_S, n)$), and then reinserting its updated version with $InsertFM(S_1, S_M, R, parent(S_S, n))$.

InsertAttribute: the operator *InsertAttribute* inserts a new attribute labeled ℓ, associating it with a concept node identifier n. Moreover, the label ℓ must be different from the labels already defined in the source schema. If an instance contains a node identifier corresponding to n, $\mathsf{IE}_{InsertAttribute}$ adds the new attribute to the instance with the value *ALL*. Let R be the semischema obtained modifying the one rooted in n by adding an attribute labeled ℓ associated with the node n; *InsertAttribute* can be mimicked with atomic operators by deleting the subtree rooted in n ($S_1 = Delete(S_S, n)$) and then reinserting it incorporating the new attribute, with $InsertFM(S_1, S_M, R, parent(S_S, n))$.

DeleteAttribute: the operator *DeleteAttribute* deletes the attribute identified by a. $\mathsf{IE}_{DeleteAttribute}$ updates the instances eliminating the attribute whose identifier corresponds to a through the function h_A, if present. Let R be the semischema obtained modifying the one rooted in n, which has a as attribute, by removing the attribute a; *DeleteAttribute* can be simulated with atomic operators by deleting the subtree rooted in n ($S_1 = Delete(S_S, n)$) and then reinserting it excluding a, with $InsertFM(S_1, S_M, R, parent(S_S, n))$.

Example 4 (High-level operators)**.** Figure 7 shows the application of the high-level operators, starting from the context schema obtained after the operators in Examples 1 and 3; the resulting modifications of the instances are shown too. The changes are briefly described in the following:

5. The dimension labeled `situation` is moved as a subdimension of `person`; being both `situation` and `person` included in the instance, the movement is performed there too.
6. The node labeled `zone` changes its name into `place`; the renaming is performed on the instance too.
7. The new attribute labeled `num_friends` is added to the node labeled `withFriends`; the attribute is also added in the instance, where takes the value *ALL*.
8. The attributes labeled `t_age` and `a_age` are removed; the instance undergoes the same modifications.

5.5. Evolution Algorithms

Up to now we have proposed declarative definitions of context schemas, context instances, and evolution operators. In this subsection we give their procedural semantics

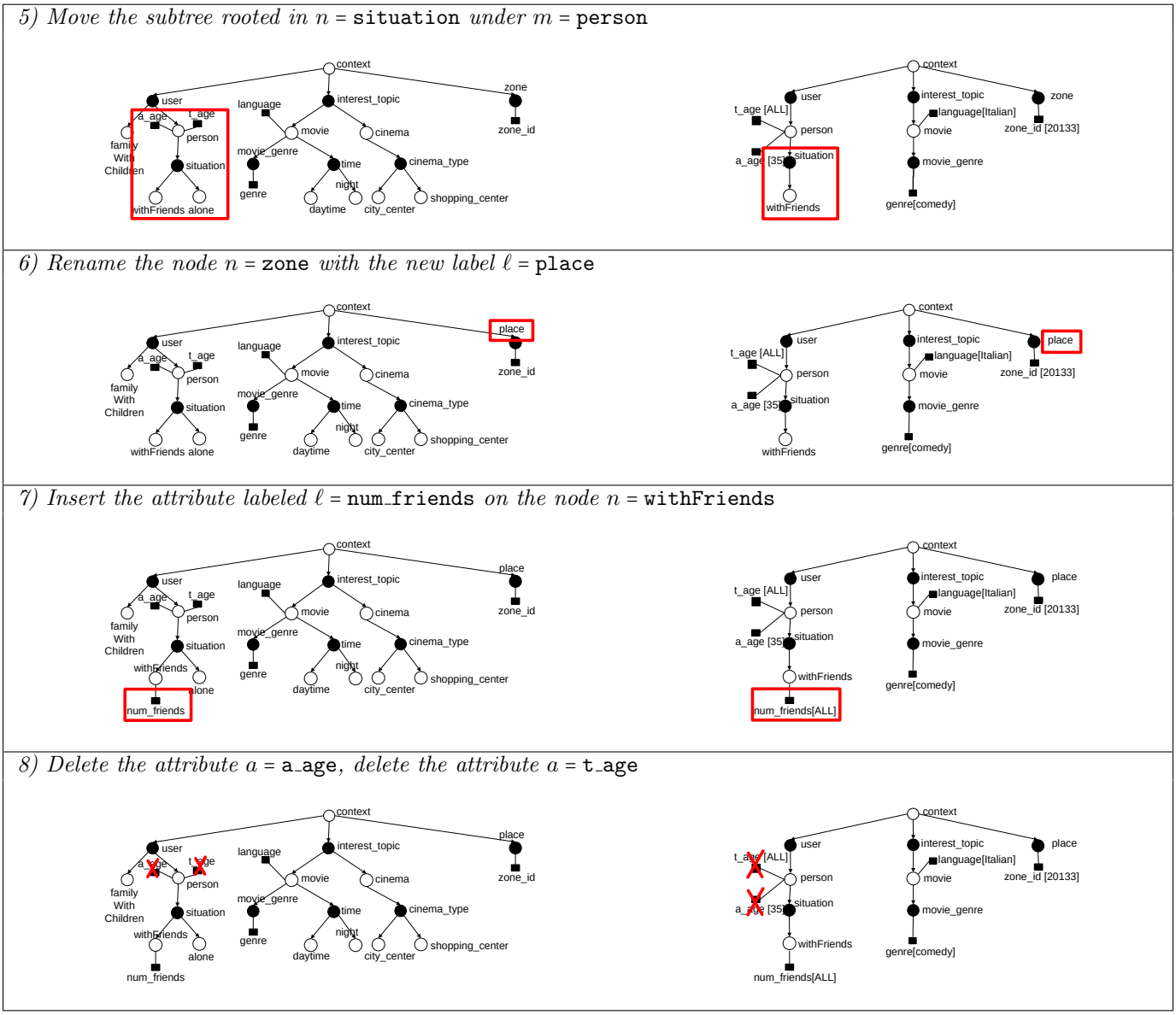


Figure 7: Application of the high-level operators described in Example 4 to the resulting schema of Example 3

providing, as examples, some of the algorithms that compute schema evolution and modify the instances accordingly. For this purpose, we consider a more appropriate representation of context schemas and instances as trees: a node of the schema and instance trees is represented as a data structure *(id, label, type, parent, child, att)*, where *id* is the node identifier, *label* is the node label, *type* may be *concept* or *dimension*, *parent* is the parent node, *child* is the set of children nodes and *att* is the set of attributes associated with the node. An attribute, in its turn, is a data structure *(id, label, value)*, where the *value* field has a value only in context instances. We show the algorithms that compute the *Merge* operator because it is the most complex.

Algorithm 1 takes as input the tree representation of the source schema, the identifiers of the nodes to be merged $m_1, \ldots, m_p$, and the label of the new node; the source schema is updated performing the merging. First of all the node associated with the identifier m_1 is located through the function *findNode*, implementing a standard tree-search algorithm, that in the worst case is linear in the number of nodes; *findNode* returns *not_found* if the requested identifier is not contained in the tree. The siblings of this node are retrieved at Line 3; this step is linear in the maximum number of children of a node, that we indicate by *max_child*. If the preconditions are fulfilled, at Line 5 a new node is created, generating (in constant time) a new identifier with the *genId*() function; then, the nodes identified by $m_1, \ldots, m_p$ are merged into the new one. Function PrecMerge checks the preconditions; the most complex step is that associated with the last one, checked through the *for* loop at Line 21 requiring

Algorithm 1 *Merge*

Require: Source schema $tree$, identifiers $m_1, \ldots, m_p$ of the nodes to be merged, label ℓ of the new node

1: $node\_m_1 = findNode(tree, m_1)$
2: $node\_f = node\_m_1.parent$
3: Locate $node\_m_2, \ldots, node\_m_p$ in $node\_f.child$
4: **if** $precMerge(tree, node\_m_1, \ldots, node\_m_p, \ell)$ **then**
5: $node\_new = newNode(genId(), \ell, concept, node\_f, \bigcup_{i=1}^{p} node\_m_i.child, \bigcup_{i=1}^{p} node\_m_i.att)$
6: $node\_f.child = (node\_f.child \setminus \{node\_m_1, \ldots, node\_m_p, \}) \cup \{node\_new\}$
7: **for all** $node \in node\_new.child$ **do**
8: $node.parent = node\_new$
9: **end for**
10: **end if**

Ensure: $tree$ is updated if the preconditions are fulfilled

11: **function** PRECMERGE($tree, node\_m_1, \ldots, node\_m_p, \ell$)
12: $node\_f = node\_m_1.parent$
13: **for all** $i \in 1 \ldots p$ **do**
14: **if** $node\_m_i = not\_found \vee node\_m_i.type \neq concept \vee node\_m_i.id = tree.id$ **then**
15: **return** false
16: **end if**
17: **if** $node\_m_i.parent \neq node\_f$ **then**
18: **return** false
19: **end if**
20: **end for**
21: **for all** n node or attr. in the tree with root $tree$ **do**
22: **if** $n.id \notin \cup_{i=1}^{p} \{node\_m_i.id\} \wedge node.label = \ell$ **then**
23: **return** false
24: **end if**
25: **end for**
26: **return** true
27: **end function**

$p \cdot (|N_S| + |Att_S|)$ operations. The global complexity of Algorithm 1 is thus $O(max\_child + p \cdot (|N_S| + |Att_S|))$; considering that $max\_child \ll |N_S|$, $p \ll |N_S|$ and $|Att_S| \ll |N_S|$, we can simply write $O(|N_S|)$.

Algorithm 2 supplies an implementation of IE_{Merge}; note that it does not require the target schema as an input. The algorithm looks for each of the merged nodes, and if one of them is present it is substituted with the new node. The function *convert* at Line 2 computes the instance node identifier m_i_i such that $h_N(m_i\_i) = m_i$; we suppose that this can be done in constant time. The function *genIdInstance()* at Line 5 generates in constant time the identifier of the new node on the basis of the corresponding one in the schema. The complexity of Algorithm 2 is determined by the *findNode* function, so it is linear in the number $|N_{IS}|$ of nodes in the instance.

Similarly to *Merge*, also the algorithms associated with the other operators are at most linear in the number of nodes of the input schema or instance.

Algorithm 2 IE_{Merge}

Require: Source instance $inst$, source schema sch, identifiers $m_1, \ldots, m_p$ of the nodes to be merged in the schema, label ℓ of the new node, identifier n of the new node in the schema

1: **for all** $m_i \in \{m_1, \ldots, m_p\}$ **do**
2: $m_i\_i = convert(inst, sch, m_i)$
3: $node\_m_i\_i = findNode(inst, m_i\_i)$
4: **if** $node\_m_i\_i \neq not\_found$ **then**
5: $node\_new = newNode(genIdInstance(n), \ell, concept, node\_m_i\_i.parent, node\_m_i\_i.child, node\_m_i\_i.att)$
6: $node\_m_i\_i.parent.child = \{node\_new\}$
7: **for all** $node \in node\_new.child$ **do**
8: $node.parent = node\_new$
9: **end for**
10: **end if**
11: **end for**

Ensure: $inst$ is updated

6. Context-Aware View Evolution

The node-based view definition approach, introduced in [13] and summarized in Section 2.3, guarantees a high flexibility with respect to evolutions: after a context schema modification has taken place, in order to update the corresponding view definitions the designer has only to revise the views connected with the nodes affected by the change, and subsequently the new views associated with the possible context instances can be automatically determined using Equation (1). The flexibility is even higher if the data model also provides a union operation: in that case, the only views that the designer must redefine are the ones regarding the leaf nodes involved in the schema evolution. Moreover, the node-based view definition approach is consistent with the schema evolution strategy, since the introduced operators allow to easily understand which nodes are affected by an evolution step.

In such a scenario, the system may help the designer identify the nodes whose views have to be adjusted during the evolution process. To give more details we illustrate the case that affords the highest flexibility, in which the underlying data model provides a union operation. Similar considerations hold also when a union operation is not available.

After the application of an evolution operator, two sets of nodes are defined: N_{DEF} and N_{COMP}. The former contains the nodes whose views have to be revised, or defined from scratch, while the latter contains the nodes whose views must be automatically recomposed using Equation (2) of Section 2.3; N_{DEF} contains only leaf nodes, while N_{COMP} contains only internal nodes.

A formal description of the N_{DEF} and N_{COMP} sets after the application of the evolution operators, in terms of their source and target schemas, is given in the appendix (Table B.1). Here follows a textual explanation of the view updates needed for each evolution operator[3].

[3]Note that for the data models that do not provide a union oper-

Delete: it is used to delete a subtree rooted in node n. The parent node $p = parent(S_S, n)$ of n may become a concept leaf, and, if this is the case, it needs an associated view ($p \in N_{DEF}$). Moreover, all the views related to each ancestor a of p have to be recomposed ($a \in N_{COMP}$); if p has not become a leaf, it is internal and thus also its view needs to be recomposed ($p \in N_{COMP}$).

Insert: after the new subtree has been inserted, the views associated with its leaves have to be defined. Also, the views related to nodes with a descendant among the new leaves must be automatically recomposed bottom-up.

InsertFromMemory: when inserting the cached content in the context schema, two cases have to be analyzed: the subtree may be inserted in a different position with respect to the one it had before the deletion, or it may be inserted at the same position. The first case occurs when the node connected with the cached subtree is not the node n under which it is reinserted, i.e. $n_M \neq n$. In this first case *InsertFromMemory* behaves exactly as *Insert* in terms of views: all the views related to the leaf nodes in the inserted subtree need to be redefined. On the contrary, if the stored subtree is reinserted in its original position, the operation results in a simple rollback of the deletion; the effects of the deletion must be discarded, i.e. not considered when computing the union of the sets N_{DEF} and N_{COMP} associated with the operators in the evolution sequence, and the views are updated on the basis of the possible modifications of the *Att* set. Note that if an attribute associated with a node n_1 has been inserted or deleted, the views related to the leaves in the descendants of n_1 have to be revised, because they may exploit the deleted/inserted attribute.

Merge: if the new node n replaces concept leaves, its view must be defined and the ones connected to its ancestors must be recomposed. By contrast, if n replaces internal nodes no new views are defined, and only the view associated with n needs to be recomposed on the basis of those of its descendants.

ReplaceSubtreesWithAttribute: the dimension connected with the new attribute is a leaf, therefore it needs the definition of a view. Moreover, the views related to the ancestors of this node must be automatically recomposed.

Move: after the *Move* of a subtree t rooted in n, the views associated with the leaves of t need revision; moreover, if the old parent of n has become a leaf, a view must be defined for it. In addition, all the views related to the ancestors of both the old parent and the new parent of n have to be automatically recomposed; also the view of the old parent must be recomposed, unless it has been defined from scratch.

Rename: the *Rename* operator affects the context schema only in the labeling function, requiring no views to be revised or recomposed.

InsertAttribute: the insertion of an attribute connected to a node n makes necessary the revision of all the views related to the nodes that may employ the new attribute, i.e. the ones associated with the leaves among the descendants of n. Moreover, the views of the concepts that are ancestors of the nodes whose view is revised must be automatically recomposed.

DeleteAttribute: after the deletion of an attribute associated with a node n, the views of the nodes that may employ that attribute – i.e., the ones associated with the leaves among the descendants of n – must be revised. Moreover, the views of the concepts that are ancestors of the nodes whose view is revised must be automatically recomposed.

Example 5 (View update)**.** Let us consider the evolutions shown in the Examples 1, 2, 3 and 4. The necessary views to be revised or recomposed after the application of each operator are explained in the following. For brevity, in this example nodes are identified by means of their labels.

1. *Delete*: the deleted subtree is rooted in `time`, a child of the root, thus no views need to be redefined nor recomposed.
   $N_{DEF} = \varnothing$, $N_{COMP} = \varnothing$
2. *Insert*: `daytime` and `night` are the inserted leaves, therefore only their views must be defined; moreover, the ones associated with their ancestors must be automatically computed.
   $N_{DEF} = \{\texttt{daytime}, \texttt{night}\}$, $N_{COMP} = \{\texttt{movie}\}$

2'. *InsertFromMemory*: the root `time` of the reinserted subtree is connected to a different node with respect to its previous parent. Therefore, the effect of the previous deletion does not have to be discarded. The view of `time` must be redefined, and the ones of its ancestors must be recomposed.
   $N_{DEF} = \{\texttt{daytime}, \texttt{night}\}$, $N_{COMP} = \{\texttt{movie}\}$

3. *Merge*: the view of the new concept node `person` has to be defined. It does not have concept ancestors different from `context`, therefore no views have to be automatically recomposed.
   $N_{DEF} = \{\texttt{person}\}$, $N_{COMP} = \varnothing$
4. *ReplaceSubtreesWithAttribute*: the view of `movie_genre`, become a leaf, has to be defined. Moreover, the view of its ancestor `movie` has to be automatically recomposed.
   $N_{DEF} = \{\texttt{movie\_genre}\}$, $N_{COMP} = \{\texttt{movie}\}$
5. *Move*: the view of the leaves contained in the moved subtree, rooted in `situation`, has to be redefined, while the one of the ancestor `person` must be recomposed.
   $N_{DEF} = \{\texttt{alone}, \texttt{withFriends}\}$, $N_{COMP} = \{\texttt{person}\}$
6. *Rename*: by definition, this operation does not affect view definitions.
   $N_{DEF} = \varnothing$, $N_{COMP} = \varnothing$

ation the N_{COMP} set is empty and the designer has to redefine the views of all the nodes.

7. *InsertAttribute*: the view of the node `withFriends`, connected to the new attribute, must be redefined; the view of the ancestor `person` has to be recomposed.
   N_{DEF} = {`withFriends`}, N_{COMP} = {`person`}

8. *DeleteAttribute*: the view of the node connected to the deleted attribute is redefined; it has no concept ancestors excluding the root, therefore no views must be automatically recomposed.
   N_{DEF} = {`person`}, $N_{COMP} = \varnothing$

The N_{DEF} sets can be computed for all the operators in linear time with respect to the number of nodes in the schema. The most complex task necessary to evaluate N_{COMP} is the computation of the set of nodes having a descendant in N_{DEF}, often needed to determine N_{COMP}; this step requires $|N_T||N_{DEF}|$ operations. Assuming $|N_{DEF}| << |N_T|$, also N_{COMP} can be determined in linear time.

Note that, in a more autonomic, self-managing scenario, the system can govern view evolution by means of machine learning techniques [27]. In this case all the affected nodes belong to N_{DEF}, and the definition of the associated views is left to the system, which computes and associates new views with the new context instances by automatically detecting new interests of the users in the corresponding contexts.

7. Optimization of Sequences of Operators

So far we have considered the evolution operators from a formal viewpoint. However, we can well imagine that in real life the design task be supported by a GUI where graphical components implement the operators. A generic schema evolution task may involve the application of a sequence of operators where, since design is seldom a straightforward process, the designer might change or even cancel her decisions. For instance, she may decide to rename a certain node, and later delete a subtree including that node; it is obvious that the deletion makes renaming useless, and thus the employed sequence of operators is not optimal. Hence the opportunity to compute a non-redundant sequence of *steps* that evolves from a schema version S_i to obtain S_j, finding the *optimized evolution sequence between the two schema versions.*

The optimization of evolution transformations has been studied in the literature in the scope of object-oriented databases [65, 66], conceptual schemas [67], and XML [68, 62] with the main objective of minimizing execution time. As we will show in the experimental section, due to the small sizes of schemas and instances, these benefits are not particularly relevant in our framework. However, the optimization is still important for two reasons:

- Sequence optimization allows the designer to monitor the system history more effectively: it must be noted that the sequence of the applied operators, if logged, is useful not only to update the instances as illustrated in the architecture of Figure 3, but also because it allows the designer to inspect and check the transformations undergone by the schema; to this aim, it is very important to have a representation of the actual "net effect " of the changes occurred between a schema version and the following one, without redundancy. Moreover, in some situations the optimization of longer sequences is also needed. In fact the designer, in her monitoring activity, could be interested in a synthetic description of the difference between two arbitrary non-consecutive versions S_i and S_{i+k}, for example because many clients are using one of them. This difference is expressed by the optimal evolution sequence between S_i and S_{i+k}. Indeed, clearly, this sequence cannot be computed simply as the concatenation of the optimal sequences defined between each pair of consecutive schemas on the path leading from S_i to S_{i+k}.

- Sequence optimization may have a significant impact on the computation of the schema nodes whose views need revising after evolution: suppose, for instance, that during the latest design session the designer has first added an attribute to node n, and subsequently deleted it. The set N_{DEF} of the nodes whose views have to be redefined is computed by merging those associated with the two operators and, according to the definitions in Table B.1, the system would suggest to the designer to revise the views of the leaf nodes in the subtree rooted in n. This suggestion is not appropriate, since the schema has not changed at all. In addition, after the performed operators also the N_{COMP} set is non-empty, and this entails the useless recomposition of some views. Therefore, to improve the behavior of the view update module, redundancy should be detected and eliminated.

As we will see in the experimental section, the optimization process is rather fast. So, each time the designer updates the schema, we suppose that the evolution sequence she applies is optimized by the system, enabling the view-update module to produce the right suggestions about the nodes whose views need to be redefined or recomposed. In addition, the optimization is also run every time the designer wants to monitor the optimal evolution sequence between two arbitrary schema versions.

Let Δ be a sequence of evolution operators applied to a schema S_1, and Δ^I the corresponding sequence of instance evolution operations. Δ_i indicates the (i-th) operator of Δ that transforms the schema S_i into the schema S_{i+1}, and $\Delta_{i...j}$ is the subsequence of Δ including the operators between i and j (i and j included); l denotes the length of the sequence $|\Delta|$, and the last operator Δ_l transforms S_l into S_{l+1}. Given an instance I_1 of S_1, $\Delta^I(I_1)$ indicates the instance of S_{l+1} obtained applying the instance effect functions corresponding to the operators in Δ to I_1. We define

the notions of *correct* sequence and *equivalent* sequences:

Definition 7 (Correct Sequence)**.** A sequence Δ of evolution operators is *correct* iff the preconditions of each operator in Δ are satisfied.

Definition 8 (Equivalent Sequences)**.** Two sequences Δ, Δ' of schema evolution operators are *equivalent* iff $S_1 = S'_1$, $S_{l+1} = S'_{l'+1}$ and, for each instance I_1 of S_1, $\Delta^I(I_1) = \Delta'^I(I_1)$.

In the rest of this section we propose techniques for the optimization of context schema evolution sequences. Our strategy relies on a sound and minimal set of optimization rules, in the spirit of [62].

7.1. Optimization Rules

Each optimization rule transforms an input evolution sequence by eliminating a pair of operators or replacing them with a unique one, and can be applied if the two operators involved satisfy certain conditions.

One difficulty is that after each deletion the removed semischema is cached, and it can be possibly reinserted by an immediately subsequent application of *InsertFromMemory*. *InsertFromMemory* inserts the content of the semischema specified as a parameter, with the constraint that this semischema has the same nodes and edges as the cached one. However, the application of an optimization rule may change the operator sequence, and therefore the intermediate schemas. Suppose that an optimization rule modifies an intermediate schema altering also some nodes and edges contained in a semischema eliminated by *Delete*: this implies that the semischema cached after the deletion changes. The result is that if *Delete* is immediately followed by an application of *InsertFromMemory*, the latter necessarily violates its preconditions, because the semischema it specifies cannot have the same nodes and edges of the cached one. The solution to this problem varies depending on the applied optimization rule, therefore each rule has also to specify how to deal with such inconsistencies. Note that [62] does not discuss these problems, because their operators do not support cache functionalities.

Let $\mathcal{D}$ be the set of possible evolution sequences. An optimization rule might specify that two operators op_i and op_j are both eliminated, or replaced by another operator either at position i or at position j, or that only one of them is kept.

Definition 9 (Optimization Rule)**.** An *optimization rule* is a function $\mathcal{D} \to \mathcal{D}$ specified by a tuple $(op_i, op_j, op_{new}, C, pol)$, where:

- op_i is the operator at position i in the input sequence
- op_j is the operator at position j in the input sequence
- op_{new} is an operator that replaces op_i and op_j in the output sequence, and has position new, with $new = i$ or $new = j$; the operator may also be undefined for some optimization rules
- C is a set of conditions that must hold in the input sequence for the rule to be applied
- *pol* is a policy to solve the inconsistencies arising with the operator *InsertFromMemory* when the sequence is modified

The conditions in the set C mainly express constraints on the operators that are allowed in the sequence between op_i and op_j in order for the rule to be applicable. In the definition of the conditions we employ the following predicates, formally summarized in Table C.1 of the appendix, referring to the operators between positions i and j in the input sequence:

- $no\_use(n)$ = `true` iff no operators use n as a parameter
- $no\_use\_sub(n)$ = `true` iff no operators use as a parameter any node in the subtree rooted in n
- $no\_ins\_label(\ell)$ = `true` iff no nodes or attributes labeled ℓ are inserted
- $no\_ins\_label\_sub(n)$ = `true` iff no nodes or attributes with label equal to one of the labels used in the subtree rooted in n are inserted
- $no\_unique\_child(n)$ = `true` iff n does not remain a unique child

In the following, with a slight abuse of notation, we apply the operators also to semischemas; we will see that this is needed to modify, during the optimization, the semischemas added through *Insert*. We define the following predicate, which applies to the sequence independently of i and j:

- $p(R, op_x)$ = `true` iff the semischema R satisfies the preconditions of the operator op_x

About the last component of the optimization rule, the policy *pol*, we define three possible policies:

- *recomp*: recompute the semischema inserted by *InsertFromMemory* on the basis of the cached one, modifying labels and attributes according to the information contained in the inserted semischema.
- $apply\_op_j$: apply the operator op_j to the semischema inserted by *InsertFromMemory*.
- $apply\_\varnothing$: do nothing.

In the rest of this section, similarly to [65], we identify three categories of rules: overriding, cancellation and insertion rules. Overriding rules (O) eliminate operators whose effects are erased by those of a subsequent one, cancellation rules (C) eliminate pairs of operators whose effects are one the inverse of the other, while insertion rules (I) collapse operators acting on inserted nodes or attributes together. The optimization rules are formally described in Table C.2 of the appendix, and the following subsections provide a textual explanation of their way of operating.

7.1.1. Overriding Rules

An operator is overridden by another one if the latter erases the effect of the former, making it redundant in the sequence. This usually happens when the schema components involved by an operator are successively deleted. Overriding rules eliminate such operators, and are formally described in the upper part of Table C.2 of the appendix; all of them eliminate op_i and keep op_j without modifications, so $new = j$ and $op_{new} = op_j$.

Rules O1-O7 remove the operators involving nodes or attributes in the subtree eliminated by a *Delete* or by a *ReplaceSubtreesWithAttribute*. If op_i inserted nodes or attributes, they must not be referred by the operators between i and j. Moreover, rule O1 removes the insertion of a subtree potentially rooted in a concept node; in the last case, to keep the sequence correct, it must not happen that all its siblings are deleted. If an operator that removes some nodes/attributes is eliminated, the components that were removed remain in the schema, so it is necessary to guarantee that they cause no label conflicts. Finally, if op_j is a *ReplaceSubtreesWithAttribute*, the children of the node whose subtrees are eliminated must not be altered, because their labels might be used during the instance update to assign a value to the new attribute.

Rules O8-O10 remove renamings of nodes/attributes later replaced, deleted or further renamed, taking care of possible label conflicts.

Rule O11 eliminates a *Move* that shifts a semischema S_1 located in the semischema S_2 under a different node in S_2, when S_2 is successively deleted. Nothing must happen within S_1 between i and j, because the same operations might not be valid in the new position; moreover, label conflicts have to be avoided.

Recall that the *InsertFromMemory* operator must be handled with care: rules O1-O4 and O11 act on the tree structure, thus it is necessary to recompute the semischema to be reinserted. On the contrary, rules O5-O10 affect only attributes or labels, therefore the inserted semischema is different from the cached one only in terms of attributes and labels; such modifications are allowed by the semantics of *InsertFromMemory*, and are anyway voided by the deletion at position j. As a consequence, no actions need to be taken.

Example 6 (Overriding Rules)**.** Consider the following sequence Δ of operators applied to the schema in Figure 1, referred to as S_1, where for simplicity nodes and attributes are identified through their labels:

1. $S_2 = Insert(S_1, R_1, \texttt{daytime})$
2. $S_3 = InsertAttribute(S_2, \texttt{night}, \texttt{hour})$
3. $S_4 = Merge(S_3, \{\texttt{adult}, \texttt{teenager}\}, \texttt{person})$
4. $S_5 = Rename(S_4, \texttt{person}, \texttt{individual})$
5. $S_6 = Delete(S_5, \texttt{user})$

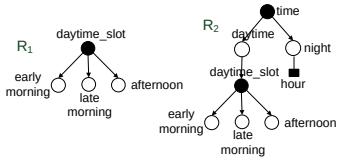


Figure 8: Semischemas inserted by the operators Δ_1 and Δ_7 of Example 6

6. $S_7 = Delete(S_6, \texttt{time})$
7. $S_8 = InsertFM(S_7, R_2, \texttt{movie})$
8. $S_9 = DeleteAttribute(S_8, \texttt{hour})$
9. $S_{10} = Merge(S_9, \{\texttt{early\_morning}, \texttt{late\_morning}\}, \texttt{morning})$

Figure 8 reports the semischemas R_1 and R_2 inserted at steps 1 and 7 .

The *Rename* at position 4 renames the node `person`, and can be removed according to rule O7, because at position 5 the subtree rooted in `user`, that is the parent of `person`, is deleted. The *Merge* at position 3, which merges the existing concepts `adult` and `teenager` into `person`, can be eliminated too for the same reason using rule O2. However, the removal of the *Merge* can take place only after that of the *Rename*: indeed, the *Rename* modifies `person`, and the removal of *Merge* would make it refer to a nonexistent node.

7.1.2. Cancellation Rules

Cancellation rules eliminate pairs of operators such that the latter undoes the modifications due to the former. The rules are formally described in the middle part of Table C.2 of the appendix; all of them eliminate both the involved operators, therefore op_{new} is not defined.

Rules C1 and C2 eliminate a pair insertion/deletion of subtrees or attributes, if no operators use the inserted components. Rule C2 also requires that no dimensions remain without children nor attributes between positions i and j. Rule C3 removes a couple of renamings when the second restores the name changed by the first, if no label conflicts arise.

Rule C1 influences nodes and edges, therefore it is necessary to recompute the semischemas inserted by *InsertFromMemory*. Rules C2-C3 affect only attributes and labels; to avoid that an *InsertFromMemory* restores the insertion/renaming performed by op_i, op_j has to be applied also on the inserted semischema.

Example 7 (Cancellation Rules)**.** Consider the operators Δ_2 and Δ_8 of the evolution sequence Δ in Example 6; the latter deletes the attribute `hour`, inserted by the former. The two operators can be eliminated according to rule C2. The semischema inserted by *InsertFromMemory* at position 7 has to be modified removing the attribute, as shown in Figure 9.

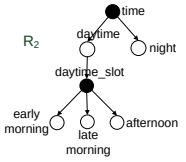


Figure 9: Semischema inserted by the operator Δ_7 after the application of the rules of Example 7

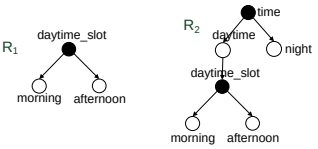


Figure 10: Semischemas inserted by the operators Δ_1 and Δ_7 after the application of the rules of Example 8

7.1.3. Insertion Rules

Insertion rules merge an *Insert* or *InsertAttribute* operator with subsequent ones affecting the inserted subtree or attribute. In fact, for example, it is useless to insert a subtree and then merge a couple of nodes belonging to it in a new node: it is worth inserting directly a subtree already carrying the new node.

Insertion rules – formally described in the lower part of Table C.2 of the appendix – eliminate the second operator, executing the required evolution on the inserted semischema. As a consequence, $new = i$. All the rules can be applied if no label conflicts arise; in addition, rules I1-I7 and I9 also require that the inserted semischema satisfy the preconditions of op_j.

Rules I1-I4, I9 affect nodes and edges, making the recomputation of the semischema inserted by *InsertFromMemory* necessary. Rules I5-I8, instead, affect labels and attributes, and the modifications caused by op_j have to be applied also to the semischema inserted by *InsertFromMemory*.

Example 8 (Insertion Rules)**.** Consider the operators Δ_1 and Δ_9 of the evolution sequence Δ of Example 6. The latter merges the nodes {`early_morning`, `late_morning`}, located in the subtree R_1 inserted by Δ_1. Rule I2 eliminates Δ_9 performing the merge directly on the inserted semischema R_1. Moreover, the semischema R_2 inserted by the operator *InsertFromMemory* at position 7 has to be recomputed according to the policy *recomp*. The new R_1 and R_2, also considering the modifications introduced in Example 7, are shown in Figure 10.

7.2. Properties and Application of the Optimization Rules

An optimization rule is sound if it transforms a correct input sequence into a correct, equivalent one.

Theorem 8 (Soundness of the optimization rules)**.** *Given an optimization rule applied to a correct sequence Δ, the resulting sequence Δ' is:*

(i) correct

(ii) equivalent to Δ

Proof. A different proof should be provided for each single optimization rule. We choose to show the proof for rule O2, the others follow a similar pattern.

Let n be the identifier of the new node replacing $\{m_1, \ldots, m_p\}$.

The sequence Δ' differs from Δ only for the lack of Merge at position i. Therefore, the schemas $S_1, \ldots, S_i$ are identical to $S'_1, \ldots, S'_i$. Moreover, if the schemas $S'_{i+1}, \ldots, S'_{j-1}$ are well-defined (i.e., if the sequence $\Delta'_{i+1\ldots j-1}$ is correct), they differ from $S_{i+2}, \ldots, S_j$ only by the possible presence of $\{m_1, \ldots, m_p\}$ in the place of n; since n is anyway deleted by Δ_j and condition $\Delta_{j+1} \neq InsertFM(\ldots)$ prevents its reinsertion, $S_{j+1}, \ldots, S_{n+1}$ is identical to $S'_j, \ldots, S'_{l'+1}$.

Similarly, given an instance I_1 of S_1, the instances $\Delta^I_{1\ldots1}(I_1)$, …, $\Delta^I_{1\ldots i-1}(I_1)$ are identical to $\Delta'^I_{1\ldots1}(I_1)$, …, $\Delta'^I_{1\ldots i-1}(I_1)$. Moreover, if it is not true that $\{m_1, \ldots, m_p\} \subseteq children(S_j, n')$ with $op_j = RSWA(S_j, n', \ell)$ – which would cause the label of n to be used after the position j – the fact that $\Delta'_{i\ldots j-1}$ is correct guarantees also that $\Delta^I_{1\ldots j}(I_1)$, …, $\Delta^I_{1\ldots l}(I_1)$ are identical to $\Delta'^I_{1\ldots j-1}$, …, $\Delta'^I_{1\ldots l'}(I_1)$.

According to the previous considerations, and because the applicability conditions ensure that if $op_j = RSWA$ then $\{m_1, \ldots, m_p\} \notin children(S_j, n')$, to prove both (i) and (ii) we have to show that $\Delta'_{i\ldots j-1}$ is correct. To this aim, we prove the following five facts:

1. *All the operators act on nodes/attributes of the source schema*: indeed, the only node that is not present any more after the application of O2 is n. The condition $no\_use(n)$ ensures that no operators between i and j use n.

2. *The correct alternation of concepts and dimensions is preserved*: this is true because n and $m_1, \ldots, m_p$ are all concepts.

3. *No label conflicts arise*: indeed, label conflicts may arise if a node or an attribute with the same label as one of $m_1, \ldots, m_p$ were inserted between positions i and j; the conditions *no_ins_label* prevent such a possibility.

4. *The operator InsertFromMemory inserts a semischema with the same nodes/edges of the cached one*: in fact, the elimination of Merge may change the tree nodes and edges, but the policy *recomp* ensures that the constraint on InsertFromMemory is satisfied.

5. *Each InsertFromMemory is preceded by a Delete*: rule O2 does not eliminate any Delete operator.

□

The proposed set of optimization rules is minimal:

Theorem 9 (Minimality of the optimization rules)**.** *Each optimization rule produces an effect on the evolution sequences that is not obtainable using the other ones.*

Proof. To show the minimality of our optimization rule set it is needed to provide, for each of the 23 rules, an example of optimization that can be obtained using that rule but not using other ones. To prove that a rule r is necessary, we show a sequence of two operators where we can easily verify, by trying to apply each of the other 22 rules, that only rule r can produce the desired optimization. We show a sample optimization for one rule of each category; the proof for the other rules follows a similar pattern. Context semischemas are synthetically represented as in the proof of Theorem 7.

Given the sequence $(Insert(a[b[c]], d, b))$, $Delete(a[b[c, d]], b))$, it can be transformed into $(Delete(a[b[c, d]], b))$ only by using rule O1. In fact, no other rule can remove an insertion followed by the deletion of a semischema containing the inserted nodes.

Given the sequence $(Insert(a, b[c], a)$, $Delete(a[b[c]], b))$, it can be transformed into the empty sequence only by using rule C1. In fact, no other rule can eliminate both an insertion and a following deletion when the latter removes exactly the same semischema introduced by the former.

Given the sequence $(Insert(a, b[c], a)$, $Insert(a[b[c]], d[e], c))$, it can be transformed into $(Insert(a, b[c[d[e]]], a))$ only by using rule I1. In fact, no other rule is able to merge two insertions. □

We now propose an algorithm to apply the optimization rules. We consider the operator sequence from the beginning to the end, and for each operator op_j perform a scan of the sequence backwards looking for another operator op_i such that an optimization rule can be applied to the pair (op_i, op_j). If a suitable rule is found, the reduction is executed. The pseudocode of the procedure is shown in Algorithm 3.

Algorithm 3 Rule application

```
Require: Sequence of operators Δ, initial context schema S_1
 1: Build the sequence of context schemas S_1, ..., S_{j+1}
 2: repeat
 3:     for j = 1 ... n do
 4:         for i = j ... 1 do
 5:             for all optimization rule do
 6:                 if the rule is applicable to Δ_i, Δ_j then
 7:                     Update Δ according to the rule
 8:                     Recompute the context schemas between
                            S_i and S_j
 9:                 end if
10:             end for
11:         end for
12:     end for
13: until some optimization is applied
Ensure: Δ is optimized
```

Note that Algorithm 3 imposes a precise ordering in rule application, and that in some cases different rule orderings may lead to different (maybe more) optimized sequences. Since we work with short evolution sequences and the execution time is in general limited (see Section 10), for the time being we do not consider this as a crucial issue. However, a deeper investigation of the problems related to the rule ordering is part of our future work.

Let N be the set of nodes in the schema. The initialization step at Line 1 has to apply l operators, so it is performed in $O(l|N|)$ time. At most l rules may be executed, and for each rule the sequence of schemas must be updated recomputing up to l schemas; the cost of rule application is $O(l^2|N|)$. In a single execution of the algorithm, the applicability of the rules can be verified $O(l^2)$ times, and the check requires to scan the sequences of rules and schemas between positions i and j; the check of the conditions of a rule on a schema is $O(|N|)$. The algorithm can be executed up to l times, so the complexity of the applicability checks, and of the whole procedure, is $O(l^4|N|)$. Consider that N is not very large, in general no more than 50 nodes [5], and that, as a consequence, also the length l of the evolution sequence is usually reasonable. In addition, recall that, as we have explained in detail at the beginning of Section 7, it is worth optimizing a redundant evolution sequence independently of the length of the sequence itself, since (i) the optimized sequences provide a better representation of the changes that the schema have undergone between two different versions, (ii) the optimized sequences can avoid the useless redefinition of some views after a schema modification session.

8. An Example: Context-Aware Data Tailoring over XML Data

In this section we exemplify the data tailoring process of an XML document D. Informally, a view over D is defined by an XQuery expression, which identifies a subset of the elements contained in D. A view is contained ($\Subset$) into another one if the latter contains all the XML elements present in the former. According to the XQuery semantics, the intersection ($\Cap$) among views keeps the common elements, while the union ($\Cup$) contains all the information present in at least one of them.

Let us consider as global dataset D an XML document storing information in the movie domain; Figure 11 contains an intuitive graphical representation of (a portion of) the schema of this document. The *show* elements describe which movies are programmed in which cinemas, with date and time.

The views associated with concept and leaf dimension nodes are defined as sets of XQuery expressions; since the language provides a union operation, the views for internal concept nodes need not be specified and can be derived by composition. We report the associations for the context schema in Figure 1, considering only the nodes involved in the context instance in Figure 2. For simplicity, the dimensions `situation` and `zone` are ignored. For brevity, nodes are identified by their labels.

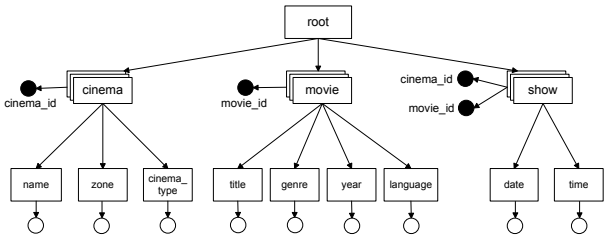


Figure 11: Schema of a portion of the XML movie database

- Adults are potentially interested in all movies and cinemas, thus $\mathcal{R}el$(`adult`) contains the whole database: `doc("movies.xml")/root/*`
- $\mathcal{R}el$(`romantic_comedy`) is:

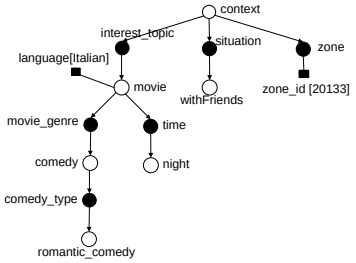
  ```
  doc("movies.xml")/root/movie[language = "$language"
   and genre = "romantic_comedy"]
  ```

  The views related to `dramatic_comedy`, `horror` and `thriller` are defined similarly. Moreover:
- $\mathcal{R}el$(`comedy`) = $\mathcal{R}el$(`romantic_comedy`) $\uplus$ $\mathcal{R}el$(`dramatic_comedy`).
- $\mathcal{R}el$(`movie`)=$\mathcal{R}el$(`thriller`) $\uplus$ $\mathcal{R}el$(`horror`) $\uplus$ $\mathcal{R}el$(`comedy`).
- $\mathcal{R}el$(`night`) includes the movies screened later than a certain time, and does not apply any filter on cinemas. The view is specified as:

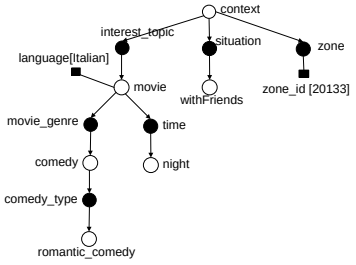
  ```
  doc("movies.xml")/root/cinema union

  doc("movies.xml")/root/show[time > 20] union(

  let $Movies:=
  doc("movies.xml")/root/show[time>20]/@movie_id
  for $m in doc("movies.xml")/root/movie
  where some $s in $Movies satisfies $s=$m/@movie_id
  return $m )
  ```

The context instance I in Figure 2 (excluding the dimensions `situation` and `zone`) is therefore associated with $View(I)$ = $\mathcal{R}el_{\rho_I}$(`adult`) $\barwedge$ $\mathcal{R}el_{\rho_I}$(`romantic_comedy`) $\barwedge$ $\mathcal{R}el_{\rho_I}$(`night`). This view can be expressed by means of a unique XQuery, and contains the romantic comedies programmed during the night.

Suppose now that the designer revises the schema deciding first to merge the concepts `adult` and `teenager` into a unique node `person`, and then to eliminate the whole subtree rooted in `user`; she also performs another change, moving the `time` dimension as a child of `movie`. The resulting schema is shown in Figure 12(a). When the designer has finished with the modifications, the system optimizes the sequence removing the merge, overridden by the subsequent deletion. Then, the system alerts the designer signaling some views that have to be redefined: the deletion of `user` does not need any view redefinition, while the movement of `time`, according to Table B.1, requires the redefinition of the views connected with `daytime` and `night`.



(a) Context schema



(b) Context instance

Figure 12: Context schema and an instance obtained after the evolutions described in Section 8

A possible redefinition for the view associated with `night` in the new position excludes the *show* and *cinema* elements, keeping only the ones associated with the movies described by the third expression of the previous definition.

Then, suppose that the user, whose context has not changed and is still described in terms of the old context schema, nevertheless requires an update of her data. The system applies IE_{Delete} and then IE_{Move}, obtaining the context instance in Figure 12(b). As a consequence, the new portion of data provided to the user, excluding the dimensions `zone` and `situation`, is $View(I)$ = $\mathcal{R}el_{\rho_I}$(`romantic_comedy`) $\barwedge$ $\mathcal{R}el_{\rho_I}$(`night`). Therefore, the movies are not filtered any more on the basis of the user information; note also that the data contained in $\mathcal{R}el_{\rho_I}$(`night`) is changed.

9. Comparisons

Now we exploit the example developed in the previous section to compare our approach with other evolution methodologies presented in the literature, in order to motivate why we have not applied one of them to solve the context schema evolution problem. In particular, we consider the general technique of Poulovassilis and McBrien [63] and the proposal by Guerrini et al. [37, 38], that have been identified as the most interesting in Section 3.

To solve the context schema evolution problem with a methodology of the literature, not explicitly conceived for context schemas, two steps are required: 1) provide a representation of context schemas and instances in the data model employed by the specific methodology; 2) provide a

$Nodes = \{context,\ user,\ familyWithChildren,\ adult,\ a\_age,$
$\quad teenager,\ t\_age,\ \ldots\}$
$Edges = \{<context,\ user>,\ <user,\ adult>,\ <adult,\ a\_age>,\ \ldots\}$
$Constraints = \{|context| = 1,\ |user| \leq 1,\ |adult| \leq 1,\ |a\_age| \leq 1,$
$\quad |user| = 1 \Rightarrow (|adult| + |teenager| + |familyWithChildren| = 1),$
$\quad |a\_age| = 1 \Rightarrow |adult| = 1,\ |movie\_genre| = 1 \Rightarrow |movie| = 1,\ \ldots\}$

Figure 13: Representation of the context schema in Figure 1 according to the model of [63]

$Ext_{S,I}(context) = \{context\}$
$Ext_{S,I}(user) = \{user\}$
$Ext_{S,I}(a\_age) = \{35\}$
$Ext_{S,I}(teenager) = \varnothing$
$Ext_{S,I}(< context, user >) =< context, user >$
$\ldots$

Figure 14: Representation of the context instance in Figure 2 according to the model of Poulovassilis and McBrien [63]

$S_2 = Delete(S_1, \texttt{user})$
$DelEdge(<adult,\ a\_age>)$
$DelEdge(<user,\ adult>)$
$DelNode(adult)$
$DelEdge(<teenager,\ t\_age>)$
$DelEdge(<user,\ teenager>)$
$DelNode(teenager)$
$DelEdge(<context,\ user>)$
$DelNode(user)$
$DelEdge(<user,\ familyWithChildren>)$
$DelNode(familyWithChildren)$

$S_3 = Move(S_2, \texttt{time}, \texttt{movie})$
$AddEdge(<movie,\ time>,\ \{x : x \in\!< context, time >\ \ if\ |movie| > 0,$
$\quad \varnothing\ otherwise\})$
$DelEdge(<context,\ time>)$

Figure 15: Evolution of Section 8 using the transformations of Poulovassilis and McBrien [63]

representation of our operators, that we have identified as useful and necessary for context schema evolution, in terms of the transformations defined by the specific methodology. In the following these two aspects are analyzed for the proposals of [63] and [37, 38].

Let us start by representing context schemas and instances with the formalism of Poulovassilis and McBrien [63]. In this work schemas are represented as hypergraphs with nodes, edges and constraints. We need a hypergraph node for each node of the context schema, and one for each attribute. Moreover, edges can connect nodes with other nodes or attributes. Finally, we need some constraints to restrict the possible instances: the root of the context schema must be associated with one and only one element in an instance, each node and attribute of the context schema must have at most one element associated in an instance, sibling concept nodes cannot be part of an instance, an attribute cannot appear if the corresponding node is not part of the instance, and a node cannot appear if its parent is not part of the instance. [63] does not distinguish labels and identifiers, so we identify the nodes through their labels. Figure 13 contains an excerpt from such a representation for the context schema in Figure 1. Note that the schema representation is not very intuitive, especially because dimension nodes, concept nodes and attributes are represented through the same construct. In [63] instances are sets of sets, with a function $Ext_{S,I}$ connecting the schema elements with their extension in the instance. An excerpt from the representation of the context instance in Figure 2 is in Figure 14.

We now express the evolution in Section 8 by using the transformations of Poulovassilis and McBrien [63]. For brevity, we consider only the deletion of `user` and the movement of `time` as a child of `movie`. The transformations are in Figure 15. The methodology of Poulovassilis and McBrien provides only the possibility of eliminating single nodes and edges, so the deletion of the subtree rooted in `user` is translated in a sequence of such basic deletions; these basic deletions correctly remove the associated nodes and edges from the instances too. In order to move `time`, it is necessary to delete the old edge connecting `time` to `context` and insert a new edge connecting `time` to `movie`. Note that the second parameter of *addEdge* is a query, specified in a language of choice, that allows to select which elements of the instances have to be associated with the new edge. Thanks to this query, it is possible to obtain on the instances the behavior specified by the semantics of our *Move*. It is possible to represent with the formalism of [63] also the other operators that we have proposed, even if the representations of *Merge* and *InsertFromMemory* are quite involved. Similarly to our operators, also the transformations of [63] envisage preconditions. However, such preconditions are referred to general hypergraphs, and are not suitable for the semantics of context schemas. For instance, no one would prevent us for moving the subtree rooted in `time` as a child of another dimension node, or even as a child of an attribute, since in the hypergraph nodes and attributes of the context schema are represented in the same way.

As a conclusion, the work [63] proposes a general, formal framework at the same level of abstraction as ours, but representing the primitives needed for context evolution implies modifications to their operators.

Guerrini et al. [37, 38] is a methodology for XML schema evolution, so we need to provide an XSD representation of context schemas. This can be achieved by representing both concepts and dimensions with complex types, while the context schema attributes are represented with XML attributes. Also in this case there is no distinction between identifiers and labels. An excerpt from an XSD representation of the context schema in Figure 1 is in Figure 16, while the XML document in Figure 17 is a part of the representation of the context instance in Figure 2. Again, note that this kind of representation is by far less intuitive than the one we have discussed in Section 2.

Figure 18 shows the same evolution executed with the XML schema modification primitives described in [37, 38]. The deletion of the subtree rooted in `user` is implemented through the *remove_substructure* primitive, that removes the subtree also from the instances; after the deletion of the substructure, the corresponding complex type may be eliminated too. The only way to alter the schema moving the subtree rooted in `time` as a child of `movie` exploiting

```
<xsd:schema>
   <xsd:complexType name="typeAdult">
      <xsd:attribute name="a_age"/>
   </xsd:complexType>
   ...
   <xsd:complexType name="typeUser">
      <xsd:choice>
         <xsd:element name="familyWithChildren"
            type="typeFamilyWithChildren"/>
         <xsd:element name="adult" type="typeAdult"/>
         <xsd:element name="teenager" type="typeTeenager"/>
      </xsd:choice>
   </xsd:complexType>
   ...
   <xsd:complexType name="typeContext">
      <xsd:all>
         <xsd:element name="user" type="typeUser" minOccurs="0"/>
         <xsd:element name="interest_topic"
            type="typeInterest_topic" minOccurs="0"/>
         ...
      </xsd:all>
   </xsd:complexType>
   <xsd:element name="context" type="typeContext"/>
</xsd:schema>
```

Figure 16: XSD representation of the context schema in Figure 1

```
<context>
   <user>
      <adult a_age="35"/>
   </user>
   <interest_topic>
      <movie language="Italian"/>
         <movie_genre>
            <comedy>
               <comedy_type>
                  <romantic_comedy/>
               </comedy_type>
            </comedy>
         </movie_genre>
      </movie>
   </interest_topic>
   ...
</context>
```

Figure 17: XML representation of the context instance in Figure 2

the primitives of Guerrini et al. [37, 38] is deleting it using *remove_substructure*, and then reinserting it in the new position with *insert_local_element*. Operating in this way, however, the algorithms of [37, 38] modify the instances removing the subtree rooted in `time`, but then they do not reinsert it in the new position. Therefore, the desired behavior of *Move* on the instances cannot be achieved. It is not possible to correctly simulate even *Merge*, *ReplaceSubtreesWithAttribute* and *InsertFromMemory*. To apply the methodology of [37, 38] in our framework, therefore, the set of their primitives should be extended and modified. Moreover, the preconditions envisaged by the primitives of [37, 38] deal with general XML schemas, thus being not suitable for context schemas. Therefore for example, similarly to [63], no one would prevent us to move the subtree rooted in `time` as a child of another dimension, or to leave dangling dimensions, and so on.

As we have seen, also this approach falls short of expressing the operators and conditions that are needed to specify the evolution fully. Nevertheless, it can be used to implement our operators adapting its data model and

$S_2 = Delete(S_1, \texttt{user})$
remove_substructure(2, context)
remove_type(typeUser)

$S_3 = Move(S_2, \texttt{time}, \texttt{movie})$
remove_substructure(5, context)
insert_local_element(time, (0,1), typeTime, 3, movie)

Figure 18: Evolution of Section 8 using the primitives of Guerrini et al. [37, 38]

applying the necessary modifications to the framework.

This is true also for the other schema evolution frameworks: for instance, we might provide a relational representation of context schemas and then found our work on the framework of Curino et al. [31], introducing the necessary modifications and extensions.

One could argue that an intermediate representation based on a previous work could have been anyway adopted at least in the implemented software. However, our engine, that will be described in Section 10, is rather simple and fast, and at the design time we have not seen any advantage in integrating it in a more complex system envisaging an additional layer to manage an intermediate representation for context schemas.

10. System Implementation and Experiments

We have implemented a Java engine to validate the effectiveness of our approach.

The *schema update module* is used by the designer to modify the context schema in the design phase by means of the atomic and high-level evolution operators. The module checks the preconditions, alters the schema, and evaluates the sets of views that have to be redefined or recomposed. We have adopted a tree-based representation of schemas, and implemented the algorithms for all the operators. These algorithms are in the same style as the one shown in Section 5.5 for *Merge*.

The *instance update module* receives a context instance from a client device and applies to the instance (online) a sequence of modifications corresponding to the effects of a schema evolution. The client sends the instance to the server in XML format. This XML document is loaded into the server memory, and then the modifications are applied. The instance adaptation algorithms are like the one proposed in Section 5.5 for IE_{Merge}. Moreover, since in the tool the instances are expressed in XML, we also provide an alternative implementation of the instance algorithms relying on the XQuery Update Facility [69], and we compare it with our procedures.

The *optimization module*, finally, takes as input sequences of evolution operators, and minimizes them.

Note that we are reasoning in terms of milliseconds, and thus possible performance gains would have no impact on the usage of the tool. We just provide an alternative implementation of the instance update module using the XQuery Update Facility, because such an implementation proved to be rather immediate and intuitive.

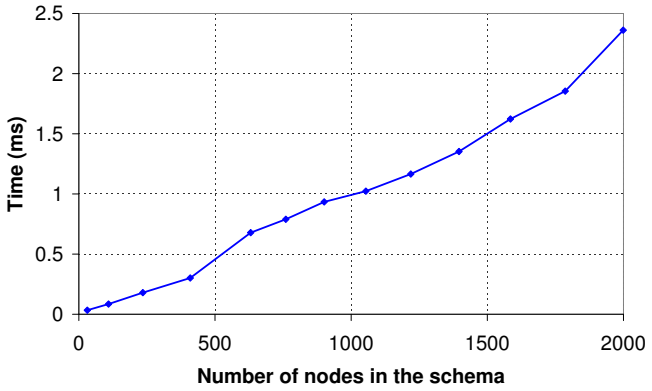


Figure 19: Execution time of the application of the schema operators against schema size

10.1. Experiments

We have measured the execution time associated with the three tasks of our engine with respect to the sizes of input schemas and instances and, where appropriate, with respect to the length of the operator sequences. Schemas and instances of various sizes have been randomly generated; to check the performance of our system in all situations, huge context schemas have been employed too, considering even (unrealistic) sizes up to 2000 nodes. We have also generated uniformly distributed random sequences of operators. Again, we have produced also very long evolution sequences, up to 1000 operators; we remark that in general extremely long sequences make no sense, since the schemas are usually quite small.

All the experiments have been performed on a 2.50 GHz Intel Core 2 Duo machine with 3 GB main memory, running Windows Vista. To implement the instance adaptation algorithms with the XQuery Update Facility we have adopted Saxon EE 9.3.0.5 [70], a popular XQuery processor implemented in Java. All the experiments have been repeated a hundred times, and the graphs have been built computing the median of the results obtained in the repetitions. The various repetitions have led to very similar measurements, with the exception of a few outliers whose effect has been neutralized through the computation of the median.

Schema Update. Figure 19 represents the average time taken to apply our operators as a function of the number of nodes. The growth is linear, and this is the expected result since both the schema modification and the computation of the nodes whose views need update are linear. Moreover, the computation time is always very low, under 2.5 ms also for huge schemas, thus allowing online execution.

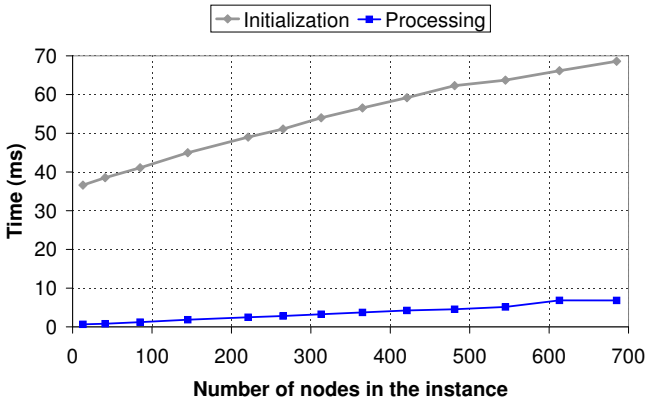
Instance Update. Figure 20(a) shows how the time to apply a realistic sequence of 90 operators to a context instance grows with respect to the number of nodes. In order to evaluate the impact of the initializations on the time required to update an instance, in Figure 20(a) we draw two lines: one represents the time required to load in memory the XML file describing the instance, and the other represents the processing time to perform the adaptation. Figure 20(b), on the contrary, describes the execution time to update an instance of 61 nodes with operator sequences of increasing length.

First of all we observe that, as expected, the processing time follows a trend which is linear in the number of nodes of the initial schema, and that the initialization time is linear too. By contrast, the growth of the processing time with respect to the number of operators in the sequence is linear as expected in the first part, but then seems to grow slowly; this happens because when the sequence is long, the presence of several *Delete* and *ReplaceSubtreesWithAttribute* operators may significantly reduce the size of both schema and instances. Moreover, all the evaluated computations are very fast also for big schemas or long sequences, and therefore they can be run on the fly. Finally, the figures suggest that, even for long evolution sequences, the processing time is dominated by the time required for the initialization.

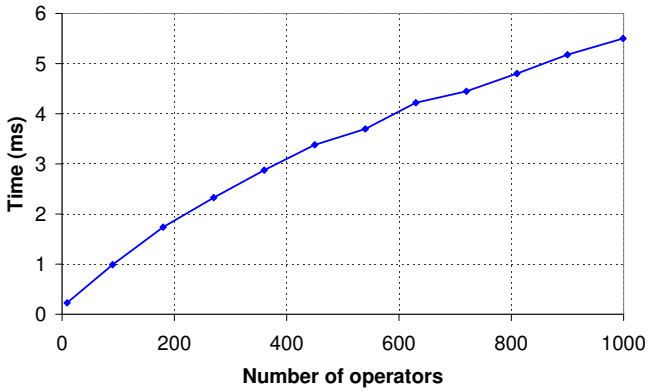
Figures 20(c) and 20(d) present the execution time when using the alternative implementation exploiting XQuery Update. We observe that the results seem to not depend on the number of nodes of the tree, at least for the tested sizes. However, the measured values are much higher than those obtained employing the algorithms which directly manipulate the trees. In fact, XQuery Update is designed to be efficient also for XML documents of several MBs, and so it relies on internal data structures that are complex to initialize and process. Even the biggest (and unrealistic) context schemas that we have tested do not exceed 60 KBs, and in these circumstances simple algorithms like the one for IE_{Merge} described in Section 5.5 are definitely more convenient.

Optimization. Figures 21(a) and 21(b) report the time to optimize sequences of operators, with respect to the number of nodes and the number of operators in the initial sequence; in the first case we have used a fixed length of 90 operators for the sequences, and in the second a fixed size of 166 nodes for the schemas. As estimated, the execution time is linear in the initial number of nodes. On the other hand, the complexity evaluated for Algorithm 3 in terms of the length of the sequence suggests a polynomial trend with degree greater than one, and the graph in Figure 21(b) behaves coherently with the theoretical result in the initial portion; then it slows down, again because of the reduction of the size of the schema that occurs when many operators are applied.

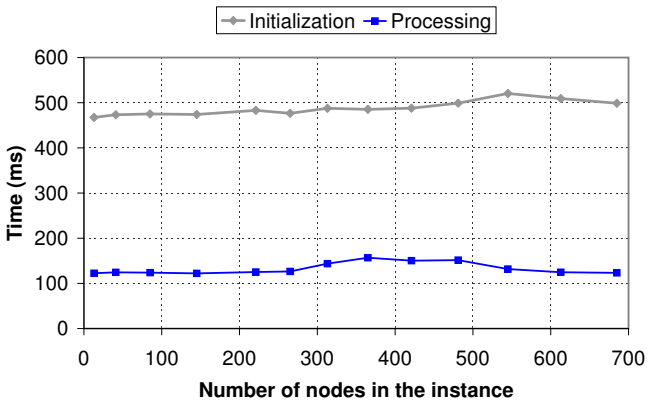
The graphs prove the effectiveness of the approach: the designer can obtain the optimal sequence of operators joining two versions in some milliseconds if the initial sequence is short, and just waiting a few seconds even if the initial sequence is long. Moreover, the employed random sequences have been reduced remarkably in our experiments: for example, a sequence of 270 operators is reduced, on



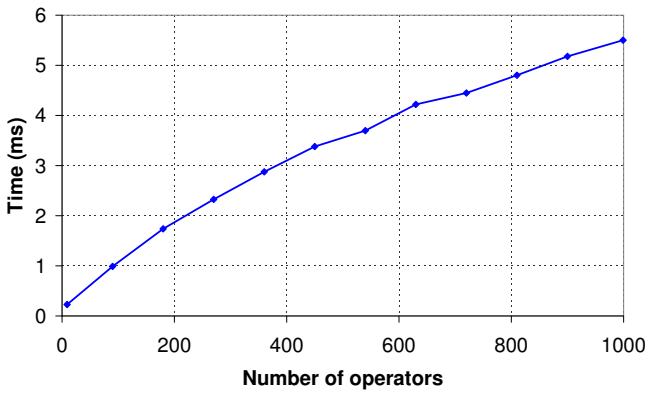
(a) Execution time against instance size



(b) Execution time against sequence length

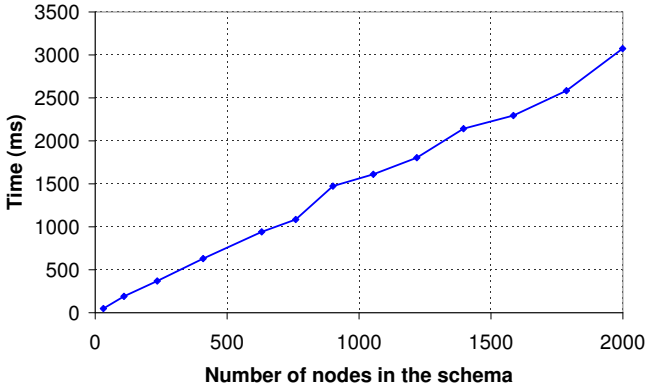


(c) Execution time against instance size, using XQuery Update

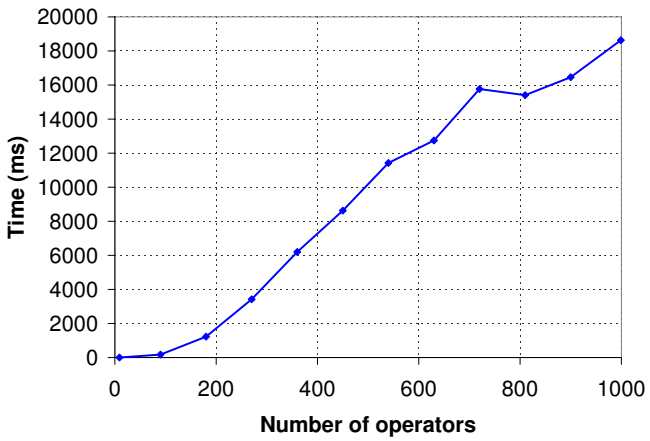


(d) Execution time against sequence length, using XQuery Update

Figure 20: Performance of the instance update



(a) Execution time against schema size



(b) Execution time against sequence length

Figure 21: Performance of the optimization methodology

the average, of 51%. In a real situation, of course, the extent of the reduction heavily depends on the amount of redundancy introduced by the designer during the design process.

To conclude we observe that, if optimized sequences have been computed, it is worth using them also for instance adaptation. However, Figures 20(a) and 20(b) have shown that the time necessary for instance update is determined mainly by the initializations. Since the initialization time is independent of the length of the evolution sequence, as anticipated in Section 7, the performance gain brought by the optimization may be moderate.

11. Conclusions and Future Work

In this paper we have investigated the problem of context schema evolution, paying special attention to its impact on context-aware data management. A set of evolution operators has been introduced, and their semantics has been formally defined; we have proven the soundness and the completeness of the evolution process, and explained how context schema evolution affects data tailoring. We have also proposed a technique to optimize sequences of evolution operators. Finally, a prototype system has been implemented, showing the effectiveness of our strategies.

In the future, we intend to thoroughly study the behavior of the optimization strategy when the application order of the rules varies. Another research that we are going to carry out is related to the application of the techniques developed in this paper to the automatic learning of contextual views introduced in [27], in order to perform the inference correctly even when the context schema evolves over time. Finally, we plan also to enrich our prototype by developing a full-fledged user interface that the designer can use to apply the operators.

Acknowledgments

This research has been partially funded by the Politecnico di Milano Polisocial Award 2013 project ObiGame, by the Italian MIUR-PRIN project GenData 2020, by the Italian project MOTUS of the program "Industria 2015", and by the Italian project SHELL CTN01_00128_111357 of the program "Cluster Tecnologici nazionali".

The authors wish to thank Jan Hidders for the helpful discussions on the formalization of the context model.

Appendix A. Preconditions and Semantics of the Operators

Notation We indicate by $\mathcal{S}$ the set of all possible context schemas, by $\mathcal{SS}$ the set of all possible semischemas, by $\mathcal{I}$ the set of all possible context instances and by $\mathcal{SI}$ the set of all possible semi-instances.

Table A.1: Preconditions and declarative semantics of the basic atomic evolution operators

Delete operator

$Delete : \mathcal{S} \times \mathcal{N} \to \mathcal{S}$
$Delete(S_S, n) = S_T$

- $N_T = N_S \setminus \widetilde{desc}(S_S, n)$
- $E_T = E_S \setminus \{(n_1, n_2) \in E_S : n_2 \in \widetilde{desc}(S_S, n)\}$
- $r_T = r_S$
- $Att_T = Att_S \setminus \{a_1 \in Att_S : \alpha_S(a_1) \in \widetilde{desc}(S_S, n)\}$
- $\alpha_T(a_1) = \alpha_S(a_1)$ if $a_1 \in Att_T$
- $\lambda_T(n_1) = \lambda_S(n_1)$ if $n_1 \in N_T \cup Att_T$

Preconditions
1) $n \in N_S$, 2) $n \in N_S^\circ \Rightarrow siblings(S_S, n) \neq \emptyset$, 3) $n \neq r_S$

Modifications induced on the instances I_S of S_S

$\mathsf{IE}_{Delete} : \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \mathcal{N} \to \mathcal{I}$
$\mathsf{IE}_{Delete}(S_S, S_T, I_S, n) = I_T$

- $N_{IT} = \{n_1 \in N_{IS} : h_N(n_1) \in N_T \wedge (n_1 \in N_{IS}^\bullet \Rightarrow (\nexists n_2 \in children(S_{IS}, n_1))(h_N(n_2) = n))$
- $E_{IT} = \{(n_1, n_2) \in E_{IS} : (h_N(n_1), h_N(n_2)) \in E_T \wedge (n_2 \in N_{IS}^\bullet \Rightarrow (\nexists n_3 \in children(S_{IS}, n_2))(h_N(n_3) = n))\}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = \{a_1 \in Att_{IS} : h_A(a_1) \in Att_T\}$
- $\alpha_{IT}(a_1) = \alpha_{IS}(a_1)$ if $n_1 \in Att_{IT}$
- $\lambda_{IT}(n_1) = \lambda_{IS}(n_1)$ if $n_1 \in N_{IT} \cup Att_{IT}$
- $\rho_{IT}(a_1) = \rho_{IS}(a_1)$ if $a_1 \in Att_{IT}$

Insert operator Given $R = (N, E, r, Att, \alpha, \lambda)$

$Insert : \mathcal{S} \times \mathcal{SS} \times \mathcal{N} \to \mathcal{S}$
$Insert(S_S, R, n) = S_T$

- $N_T = N_S \cup N$
- $E_T = E_S \cup E \cup \{(n, r)\}$
- $r_T = r_S$
- $Att_T = Att_S \cup Att$
- $\alpha_T(a_1) = \begin{cases} \alpha_S(a_1) & \text{if } a_1 \in Att_S \\ \alpha(a_1) & \text{otherwise} \end{cases}$
- $\lambda_T(n_1) = \begin{cases} \lambda_S(n_1) & \text{if } n_1 \in N_S \cup Att_S \\ \lambda(n_1) & \text{otherwise} \end{cases}$

Modifications induced on the instances I_S of S_S

$\mathsf{IE}_{Insert} : \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \mathcal{SS} \times \mathcal{N} \to \mathcal{I}$
$\mathsf{IE}_{Insert}(S_S, S_T, I_S, R, n) = I_T$

- $N_{IT} = N_{IS}$
- $E_{IT} = E_{IS}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = Att_{IS}$
- $\alpha_{IT} = \alpha_{IS}$
- $\lambda_{IT} = \lambda_{IS}$
- $\rho_{IT} = \rho_{IS}$

Preconditions
1) $n \in N_S$, 2) $N \cap N_S = \emptyset$, 3) $Att \cap Att_S = \emptyset$, 4) $(\forall n_1 \in (N_S \cup Att_S))(\forall (n_2 \in N \cup Att))(\lambda_S(n_1) \neq \lambda(n_2))$, 5) $n \in N_S^\circ \Rightarrow r \in N^\bullet$, 6) $n \in N_S^\bullet \Rightarrow r \in N^\circ$, 7) $n \in N_S^\bullet \Rightarrow \alpha^{-1}(n) = \emptyset$

Table A.2: Schema and instance cache

Delete schema cache

$\mathsf{SC}_{Delete} : \mathcal{S} \times \mathcal{S} \times \mathcal{N} \to \mathcal{SS}$
$\mathsf{SC}_{Delete}(S_S, S_T, n) = S_M$

- $N_M = N_S \setminus N_T$
- $E_M = E_S \setminus (E_T \cup \{(parent(S_S, n), n)\})$
- $r_M = n$
- $Att_M = Att_S \setminus Att_T$
- $\alpha_M(a_1) = \alpha_S(a_1)$ if $a_1 \in Att_M$
- $\lambda_M(n_1) = \lambda_S(n_1)$ if $n_1 \in N_M \cup Att_M$
- $n_M = parent(S_S, n)$

Delete instance cache

$\mathsf{IC}_{Delete} : \mathcal{S} \times \mathcal{S} \times \mathcal{SS} \times \mathcal{I} \times \mathcal{I} \times \mathcal{N} \to \mathcal{SI}$
$\mathsf{IC}_{Delete}(S_S, S_T, S_M, I_S, I_T, n) = I_M$

- $N_{IM} = N_{IS} \setminus N_{IT}$
- $E_{IM} = \begin{cases} E_{IS} \setminus (E_{IT} \cup \{(n_1, n_2) : n_1 = h_N(parent(S_S, parent(S_S, n))) \wedge n_2 = h_N(parent(S_S, n))\}) & \text{if } (\exists n_1 \in N_{IS}^\circ)(h_N(n_1) = n) \\ E_{IS} \setminus (E_{IT} \cup \{(n_1, n_2) : n_1 = h_N(parent(S_S, n)) \wedge n_2 = h_N(n)\}) & \text{otherwise} \end{cases}$
- $r_{IM} = \begin{cases} n_1 \in N_{IS} : (\exists n_2 \in N_{IS})(n_1 = parent(S_{IS}, n_2) \wedge n_2 = h_N(n)) & \text{if } (\exists n_1 \in N_{IS}^\circ)(h_N(n_1) = n) \\ n_1 \in N_{IS} : h_N(n_1) = n & \text{if } (\exists n_1 \in N_{IS}^\bullet)(h_N(n_1) = n) \\ \text{undefined} & \text{otherwise} \end{cases}$
- $Att_{IM} = Att_{IS} \setminus Att_{IT}$
- $\alpha_{IM}(a_1) = \alpha_{IS}(a_1)$ if $a_1 \in Att_{IM}$
- $\lambda_{IM}(n_1) = \lambda_{IS}(n_1)$ if $n_1 \in N_{IM} \cup Att_{IM}$
- $\rho_{IM}(a_1) = \rho_{IS}(a_1)$ if $a_1 \in Att_{IM}$

Table A.3: Preconditions and declarative semantics of *InsertFromMemory*

InsertFromMemory operator Given $R = (N, E, r, Att, \alpha, \lambda)$

$\mathsf{SU}_{InsertFM} : \mathcal{S} \times \mathcal{SS} \times \mathcal{SS} \times \mathcal{N} \to \mathcal{S}$
$\mathsf{SU}_{InsertFM}(S_S, S_M, R, n) = S_T$

- $N_T = N_S \cup N$
- $E_T = E_S \cup E \cup \{(n, r)\}$
- $r_T = r_S$
- $Att_T = Att_S \cup Att$
- $\alpha_T(a_1) = \begin{cases} \alpha_S(a_1) & \text{if } a_1 \in Att_S \\ \alpha(a_1) & \text{otherwise} \end{cases}$
- $\lambda_T(n_1) = \begin{cases} \lambda_S(n_1) & \text{if } n_1 \in N_S \cup Att_S \\ \lambda(n_1) & \text{otherwise} \end{cases}$

Modifications induced on the instances I_S of S_S

$\mathsf{IE}_{InsertFM} : \mathcal{S} \times \mathcal{S} \times \mathcal{SS} \times \mathcal{I} \times \mathcal{SI} \times \mathcal{SS} \times \mathcal{N} \to \mathcal{I}$
$\mathsf{IE}_{InsertFM}(S_S, S_T, S_M, I_S, I_M, R, n) = I_T$
Let $e_1, \ldots, e_k$ be the system-assigned identifiers for the attributes $f_1, \ldots, f_k$ of the inserted semischema defined as follows: $f_i \in Att : (\exists n_1 \in N_{IM})(\alpha(f_i) = h_{NM}(n_1) \wedge (\nexists a_1 \in Att_{IM})(f_i = h_{AM}(a_1)))$

- $N_{IT} = \begin{cases} N_{IS} \cup N_{IM} & \text{if } N_{IM} \neq \varnothing \wedge ((n \in N_T^\circ \wedge (\exists n_1 \in N_{IS})(h_N(n_1) = n)) \vee h_{NM}(r_{IM}) = n) \\ N_{IS} & \text{otherwise} \end{cases}$
- $E_{IT} = \begin{cases} E_{IS} \cup E_{IM} \cup \{(n_1, r_{IM}) : h_N(n_1) = n\} & \text{if } N_{IM} \neq \varnothing \wedge n \in N_T^\circ \wedge (\exists n_1 \in N_{IS})(h_N(n_1) = n) \\ E_{IS} \cup E_{IM} \cup \{(n_1, n_2) : h_{NM}(n_2) = n \wedge h_N(n_1) = parent(S_S, n)\} & \text{if } N_{IM} \neq \varnothing \wedge h_{NM}(r_{IM}) = n \\ E_{IS} & \text{otherwise} \end{cases}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = \begin{cases} Att_{IS} \cup \{a_1 \in Att_{IM} : (\exists n_1 \in N, a_2 \in Att)(a_2 = h_{AM}(a_1) \wedge \alpha_M(a_1) = \alpha(a_1) = n_1)\} \cup \{e_1, \ldots, e_k\} & \text{if } N_{IM} \neq \varnothing \wedge ((n \in N_T^\circ \wedge (\exists n_1 \in N_{IS}) (h_N(n_1) = n)) \vee h_{NM}(r_{IM}) = n)) \\ Att_{IS} & \text{otherwise} \end{cases}$
- $\alpha_{IT}(a_1) = \begin{cases} \alpha_{IM}(a_1) & \text{if } a_1 \in Att_{IM} \\ \{n_1 \in N_{IM} : \alpha(f_i) = h_{NM}(n_1)\} & \text{if } a_1 = e_i,\ e_i \in \{e_1, \ldots, e_k\} \\ \alpha_{IS}(a_1) & \text{otherwise} \end{cases}$
- $\lambda_{IT}(n_1) = \begin{cases} \lambda_{IM}(n_1) & \text{if } n_1 \in N_{IM} \cup Att_{IM} \\ \lambda(f_i) & \text{if } n_1 = e_i, e_i \in \{e_1, \ldots, e_k\} \\ \lambda_{IS}(n_1) & \text{otherwise} \end{cases}$
- $\rho_{IT}(a_1) = \begin{cases} \rho_{IM}(a_1) & \text{if } a_1 \in Att_{IM} \\ ALL & \text{if } a_1 \in \{e_1, \ldots, e_k\} \\ \rho_{IS}(a_1) & \text{otherwise} \end{cases}$

Preconditions
1) $n \in N_S$, 2) $N = N_M$, 3) $E = E_M$, 4) $Att \cap Att_S = \varnothing$, 5) $(\forall n_1 \in (N_S \cup Att_S))(\forall n_2 \in (N \cup Att))(\lambda_S(n_1) \neq \lambda(n_2))$, 6) $n \in N_S^\circ \Rightarrow r \in N^\bullet$, 7) $n \in N_S^\bullet \Rightarrow r \in N^\circ$, 8) $n \in N_S^\bullet \Rightarrow \alpha^{-1}(n) = \varnothing$, 9) $N_M \neq \varnothing$

Table A.4: Preconditions and declarative semantics of the *Merge* and *ReplaceSubtreesWithAttribute* atomic operators

Merge operator

$Merge : \mathcal{S} \times \wp(\mathcal{N}) \times \mathcal{L} \to \mathcal{S}$
$Merge(S_S, \{m_1, \ldots, m_p\}, \ell) = S_T$
Let n the system-generated identifier for the new node labeled ℓ.

- $N_T = (N_S \setminus \{m_1, \ldots, m_p\}) \cup \{n\}$
- $E_T = (E_S \setminus (\{(parent(S_S, m_i), m_i) : m_i \in \{m_1, \ldots, m_p\}\} \cup \{(m_i, n_1) \in E_S : m_i \in \{m_1, \ldots, m_p\} \wedge n_1 \in N_S\})) \cup \{(parent(S_S, m_1), n)\} \cup \{(n, n_1) : (\exists m_i \in \{m_1, \ldots, m_p\})((m_i, n_1) \in E_S)\}$
- $r_T = r_S$
- $Att_T = Att_S$
- $\alpha_T(a_1) = \begin{cases} n & \text{if } \alpha_S(a_1) \in \{m_1, \ldots, m_p\} \\ \alpha_S(a_1) & \text{otherwise} \end{cases}$
- $\lambda_T(n_1) = \begin{cases} \ell & \text{if } n_1 = n \\ \lambda_S(n_1) & \text{otherwise} \end{cases}$

Modifications induced on the instances I_S of S_S

$\mathsf{IE}_{Merge} : \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \wp(\mathcal{N}) \times \mathcal{L} \to \mathcal{I}$
$\mathsf{IE}_{Merge}(S_S, S_T, I_S, \{m_1, \ldots, m_p\}, \ell) = I_T$

Let l be the identifier generated, if necessary, by the system for the new node labeled ℓ. Let $b_1, \ldots, b_k$ be the identifiers generated, if necessary, for the attributes $d_1, \ldots, d_k$ of the target schema defined in this way: $d_i \in \alpha_T^{-1}(n) : (\nexists c_i \in Att_{IS})(d_i = h_A(c_i))$. The attributes $d_1, \ldots, d_k$ are the ones associated with the nodes belonging to $\{m_1, \ldots, m_p\}$ that do not have a corresponding node in the instance.

- $N_{IT} = \begin{cases} \left.\begin{array}{l}(N_{IS} \setminus \{n_1 \in N_{IS} : \\ h_N(n_1) \in \{m_1, \ldots, m_p\}\}) \cup \{l\}\end{array}\right\} & \text{if } (\exists n_1 \in N_{IS})(h_N(n_1) \in \{m_1 \ldots m_p\}) \\ N_{IS} & \text{otherwise} \end{cases}$
- $E_{IT} = \begin{cases} \left.\begin{array}{l}\{(n_1, n_2) \in E_{IS} : h_N(n_1), h_N(n_2) \notin \{m_1, \ldots, m_p\}\} \cup \{(l, n_1) : \\ (\exists n_2 \in N_{IS})((n_2, n_1) \in E_{IS} \wedge ((\exists m_i \in \{m_1, \ldots, m_p\})(m_i = \\ h_N(n_2))))\} \cup \{(parent(S_{IS}, n_1), l) : (h_N(n_1) \in \{m_1, \ldots, m_p\})\}\end{array}\right\} & \text{if } (\exists n_1 \in N_{IS})(h_N(n_1) \in \{m_1 \ldots m_p\}) \\ E_{IS} & \text{otherwise} \end{cases}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = \begin{cases} Att_{IS} \cup \{b_1, \ldots, b_k\} & \text{if } (\exists n_1 \in N_{IS})(h_N(n_1) \in \{m_1 \ldots m_p\}) \\ Att_{IS} & \text{otherwise} \end{cases}$
- $\alpha_{IT}(a_1) = \begin{cases} n & \text{if } h_N(\alpha_{IS}(a_1)) \in \{m_1, \ldots, m_p\} \vee a_1 \in \{b_1, \ldots, b_k\} \\ \alpha_{IS}(a_1) & \text{otherwise} \end{cases}$
- $\lambda_{IT}(n_1) = \begin{cases} \ell & \text{if } n_1 = l \\ \lambda_T(d_i) & \text{if } n_1 = b_i, b_i \in \{b_1, \ldots, b_k\} \\ \lambda_{IS}(n_1) & \text{otherwise} \end{cases}$
- $\rho_{IT}(a_1) = \begin{cases} ALL & \text{if } a_1 \in \{b_1, \ldots, b_k\} \\ \rho_{IS}(a_1) & \text{otherwise} \end{cases}$

Preconditions
1) $n \in \mathcal{N}^\circ$, 2) $(\forall m_i \in \{m_1, \ldots, m_p\})(m_i \in N_S^\circ \setminus \{r_S\})$, 3) $(\forall m_i, m_j \in \{m_1 \ldots m_p\})(parent(S_S, m_i) = parent(S_S, m_j))$, 4) $(\forall n_1 \in (N_S \cup Att_S) \setminus \{m_1, \ldots, m_p\})(\lambda_S(n_1) \neq \ell)$

RSWA operator

$RSWA : \mathcal{S} \times \mathcal{N} \times \mathcal{L} \to \mathcal{S}$
$RSWA(S_S, n, \ell) = S_T$
Let a the system-generated id. of the new attribute labeled ℓ.

- $N_T = N_S \setminus desc(S_S, n)$
- $E_T = E_S \setminus \{(n_1, n_2) \in E_{IS} : n_2 \in desc(S_S, n)\}$
- $r_T = r_S$
- $Att_T = (Att_S \cup \{a\}) \setminus \{a_1 \in Att_S : \alpha_S(a_1) \in desc(S_S, n)\}$
- $\alpha_T(a_1) = \begin{cases} n & \text{if } a_1 = a \\ \alpha_S(a_1) & \text{otherwise} \end{cases}$
- $\lambda_T(n_1) = \begin{cases} \ell & \text{if } n_1 = a \\ \lambda_S(n_1) & \text{otherwise} \end{cases}$

Modifications induced on the instances I_S of S_S

$\mathsf{IE}_{RSWA} : \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \mathcal{N} \times \mathcal{L} \to \mathcal{I}$
$\mathsf{IE}_{RSWA}(S_S, S_T, I_S, n, \ell) = I_T$
Let b be the id. generated, if necessary, by the system for the new attribute labeled ℓ.

- $N_{IT} = \{n_1 \in N_{IS} : h_N(n_1) \in N_T\}$
- $E_{IT} = \{(n_1, n_2) \in E_{IS} : (h_N(n_1), h_N(n_2)) \in E_T\}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = \begin{cases} \{a_1 \in Att_{IS} : h_A(a_1) \in Att_T\} \cup \{b\} & \text{if } (\exists n_1 \in N_{IS})(h_N(n_1) = n) \\ Att_{IS} & \text{otherwise} \end{cases}$
- $\alpha_{IT}(a_1) = \begin{cases} n_1 \in N_{IS} : h_N(n_1) = n & \text{if } a_1 = b \\ \alpha_{IS}(a_1) & \text{otherwise} \end{cases}$
- $\lambda_{IT}(n_1) = \begin{cases} \ell & \text{if } n_1 = b \\ \lambda_{IS}(n_1) & \text{otherwise} \end{cases}$
- $\rho_{IT}(a_1) = \begin{cases} \left.\begin{array}{l}\lambda_{IS}(n_2) : \\ (\exists (n_1, n_2) \in E_{IS})(h_N(n_1) = n)\end{array}\right\} & \text{if } (\exists n_1 \in N_{IS}) \\ & (h_N(n_1) = n) \\ \rho_{IS}(a_1) & \text{otherwise} \end{cases}$

Preconditions
1) $n \in N_S^\bullet$, 2) $\alpha_S^{-1}(n) = \varnothing$, 3) $(\forall n_1 \in (N_S \cup Att_S) \setminus (desc(S_S, n) \cup \{a_1 \in Att_S : \alpha_S(a_1) \in desc(S_S, n)\}))(\lambda_S(n_1) \neq \ell)$

Table A.5: Preconditions and declarative semantics of the high-level operators

Move operator

$Move : \mathcal{S} \times \mathcal{N} \times \mathcal{N} \to \mathcal{S}$
$Move(S_S, n, m) = S_T$

- $N_T = N_S$
- $E_T = (E_S \setminus \{(parent(S_S, n), n)\}) \cup \{(m, n)\}$
- $r_T = r_S$
- $Att_T = Att_S$
- $\alpha_T = \alpha_S$
- $\lambda_T = \lambda_S$

Preconditions
1) $n \in N_S^\bullet$, 2) $m \in N_S^\circ$, 3) $m \notin \widetilde{desc}(S_S, n)$

Modifications induced on the instances I_S of S_S

$\mathsf{IU}_{Move} : \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \mathcal{N} \times \mathcal{N} \to \mathcal{I}$
$\mathsf{IU}_{Move}(S_S, S_T, I_S, n, m) = I_T$

- $N_{IT} = \begin{cases} N_{IS} & \text{if } (\exists n_1, n_2 \in N_{IS}) \\ & (h_N(n_1) = n \wedge h_N(n_2) = m) \\ N_{IS} \setminus \{n_1 \in N_{IS} : h_N(n_1) \in \widetilde{desc}(S_T, n)\} & \text{otherwise} \end{cases}$
- $E_{IT} = \begin{cases} (E_{IS} \setminus \{(n_1, n_2) \in E_{IS} : h_N(n_2) = n\}) & \text{if } (\exists n_1, n_2 \in N_{IS})(h_N(n_1) = n \\ \cup \{(n_1, n_2) : h_N(n_1) = m \wedge h_N(n_2) = n\} & \wedge h_N(n_2) = m) \\ E_{IS} \setminus \{(n_1, n_2) \in E_{IS} : h_N(n_2) \in \widetilde{desc}(S_T, n) \wedge n_1 \in N_{IS}\} & \text{otherwise} \end{cases}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = \begin{cases} Att_{IS} & \text{if } (\exists n_1, n_2 \in N_{IS})(h_N(n_1) = n \wedge h_N(n_2) = m) \\ Att_{IS} \setminus \{a_1 \in Att_{IS} : \alpha_T(h_A(a_1)) \in \widetilde{desc}(S_T, n)\} & \text{otherwise} \end{cases}$
- $\alpha_{IT}(a_1) = \alpha_{IS}(a_1)$ if $a_1 \in Att_{IT}$
- $\lambda_{IT}(n_1) = \lambda_{IS}(n_1)$ if $n_1 \in N_{IT} \cup Att_{IT}$
- $\rho_{IT}(a_1) = \rho_{IS}(a_1)$ if $a_1 \in Att_{IT}$

Rename operator

$Rename : \mathcal{S} \times (\mathcal{N} \cup \mathcal{A}) \times \mathcal{L} \to \mathcal{S}$
$Rename(S_S, n, \ell) = S_T$

- $N_T = N_S$
- $E_T = E_S$
- $r_T = r_S$
- $Att_T = Att_S$
- $\alpha_T = \alpha_S$
- $\lambda_T(n_1) = \begin{cases} \ell & \text{if } n_1 = n \\ \lambda_S(n_1) & \text{otherwise} \end{cases}$

Preconditions
1) $n \in N_S \cup Att_S$, 2) $\forall (n_1 \in N_S \cup Att_S)(n_1 \neq n \Rightarrow \lambda_S(n_1) \neq \ell)$

Modifications induced on the instances I_S of S_S

$\mathsf{IE}_{Rename} : \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times (\mathcal{N} \cup \mathcal{A}) \times \mathcal{L} \to \mathcal{I}$
$\mathsf{IE}_{Rename}(S_S, S_T, I_S, n, \ell) = I_T$

- $N_{IT} = N_{IS}$
- $E_{IT} = E_{IS}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = Att_{IS}$
- $\alpha_{IT} = \alpha_{IS}$
- $\lambda_{IT}(n_1) = \begin{cases} \ell & \text{if } h_N(n_1) = n \vee h_A(n_1) = n \\ \lambda_{IS}(n_1) & \text{otherwise} \end{cases}$
- $\rho_{IT} = \rho_{IS}$

InsertAttribute operator

$InsertAttribute : \mathcal{S} \times \mathcal{N} \times \mathcal{L} \to \mathcal{S}$
$InsertAttribute(S_S, n, \ell) = S_T$

Let a the system-generated id. of the new attribute labeled ℓ.

- $N_T = N_S$
- $E_T = E_S$
- $r_T = r_S$
- $Att_T = Att_S \cup \{a\}$
- $\alpha_T(a_1) = \begin{cases} n & \text{if } a_1 = a \\ \alpha_S(a_1) & \text{otherwise} \end{cases}$
- $\lambda_T(n_1) = \begin{cases} \ell & \text{if } n_1 = a \\ \lambda_S(n_1) & \text{otherwise} \end{cases}$

Preconditions
1) $n \in N_S^\circ$, 2) $(\forall n_1 \in N_S \cup Att_S)(\lambda_S(n_1) \neq \ell)$

Modifications induced on the instances I_S of S_S

$\mathsf{IE}_{InsertAttribute} : \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \mathcal{N} \times \mathcal{L} \to \mathcal{I}$
$\mathsf{IE}_{InsertAttribute}(S_S, S_T, I_S, n, \ell) = I_T$

Let b be the id. generated, if necessary, by the system for the new attribute labeled ℓ.

- $N_{IT} = N_{IS}$
- $E_{IT} = E_{IS}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = \begin{cases} Att_{IS} \cup \{b\} & \text{if } (\exists n_1 \in N_{IS})(h_N(n_1) = n) \\ Att_{IS} & \text{otherwise} \end{cases}$
- $\alpha_{IT}(a_1) = \begin{cases} n_1 \in N_{IS} : h_N(n_1) = n & \text{if } a_1 = b \\ \alpha_{IS}(a_1) & \text{otherwise} \end{cases}$
- $\lambda_{IT}(n_1) = \begin{cases} \ell & \text{if } n_1 = b \\ \lambda_{IS}(n_1) & \text{otherwise} \end{cases}$
- $\rho_{IT}(a_1) = \begin{cases} ALL & \text{if } a_1 = b \\ \rho_{IS}(a_1) & \text{otherwise} \end{cases}$

DeleteAttribute operator

$DeleteAttribute : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$
$DeleteAttribute(S_S, a) = S_T$

- $N_T = N_S$
- $E_T = E_S$
- $r_T = r_S$
- $Att_T = Att_S \setminus \{a\}$
- $\alpha_T(a_1) = \alpha_S(a_1)$ if $a_1 \in Att_T$
- $\lambda_T(n_1) = \lambda_S(n_1)$ if $n_1 \in N_T \cup Att_T$

Preconditions
1) $a \in Att_S$, 2) $\alpha_S(a) \in N_S^\circ$

Modifications induced on the instances I_S of S_S

$\mathsf{IU}_{DeleteAttribute} : \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \mathcal{A} \to \mathcal{I}$
$\mathsf{IU}_{DeleteAttribute}(S_S, S_T, I_S, a) = I_T$

- $N_{IT} = N_{IS}$
- $E_{IT} = E_{IS}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = \{a_1 \in Att_{IS} : h_A(a_1) \in Att_T\}$
- $\alpha_{IT}(a_1) = \alpha_S(a_1)$ if $a_1 \in Att_{IT}$
- $\lambda_{IT}(n_1) = \lambda_T(n_1)$ if $n_1 \in N_{IT} \cup Att_{IT}$
- $\rho_{IT}(a_1) = \rho_{IS}(a_1)$ if $a_1 \in Att_{IT}$

Appendix B. Context-Aware View Evolution

Notation The set $\hat{N}_{REC}$ indicates the nodes having a descendant contained in N_{DEF}: $\hat{N}_{REC} = \{n_1 \in N_T^\circ : (\exists n_2 \in N_{DEF})(n_2 \in desc(n_1))\}$.

Table B.1: Definition of the N_{DEF} and N_{COMP} sets associated with the schema evolution operators.

$Delete(S_S, n)$	$N_{DEF} = \{parent(S_S, n)\} \cap \bar{N}_T^\circ$ $N_{COMP} = \begin{cases} \hat{N}_{REC} & \text{if } parent(S_S, n) \in \bar{N}_T^\circ \\ \widetilde{asc}(S_T, parent(S_S, n)) \cap N_T^\circ & \text{otherwise} \end{cases}$
$Insert(S_S, R, n)$	$N_{DEF} = \bar{N}$ $N_{COMP} = \hat{N}_{REC}$
$Merge(S_S, \{m_1, \ldots, m_p\}, \ell)$	Let n be the identifier of the new node labeled ℓ $N_{DEF} = \{n\} \cap \bar{N}_T^\circ$ $N_{COMP} = \begin{cases} \hat{N}_{REC} & \text{if } n \in \bar{N}_T^\circ \\ \{n\} & \text{otherwise} \end{cases}$
$RSWA(S_S, n, \ell)$	$N_{DEF} = \{n\}$ $N_{COMP} = \hat{N}_{REC}$
$InsertFM(S_S, R, n)$	**if** $n_M = n$ *the effects of the previous deletion are discarded, then* $N_{DEF} = \{n_1 \in \bar{N} : (\exists n_2 \in \widetilde{asc}(R, n_1))((\exists a_1 \in \alpha^{-1}(n_2))$ $(a_1 \notin \alpha_M^{-1}(n_2)) \vee (\exists a_1 \in \alpha_M^{-1}(n_2))(a_1 \notin \alpha^{-1}(n_2)))\}$ $N_{COMP} = \hat{N}_{REC}$ **else** $N_{DEF} = \bar{N}$ $N_{COMP} = \hat{N}_{REC}$
$Move(S_S, n, m)$	$N_{DEF} = \begin{cases} \widetilde{desc}(S_T, n) \cap \bar{N}_T & \text{if } parent(S_S, n) \notin \bar{N}_T^\circ \\ (\widetilde{desc}(S_T, n) \cap \bar{N}_T) \cup \{parent(S_S, n)\} & \text{otherwise} \end{cases}$ $N_{COMP} = \begin{cases} \hat{N}_{REC} \cup \widetilde{asc}(S_T, parent(S_S, n)) & \text{if } parent(S_S, n) \notin \bar{N}_T^\circ \\ \hat{N}_{REC} & \text{otherwise} \end{cases}$
$Rename(S_S, n, \ell)$	$N_{DEF} = \varnothing$ $N_{COMP} = \varnothing$
$InsertAttribute(S_S, n, \ell)$	$N_{DEF} = \bar{N}_T \cap \widetilde{desc}(S_T, n)$ $N_{COMP} = \hat{N}_{REC}$
$DeleteAttribute(S_S, a)$	$N_{DEF} = \bar{N}_T \cap \widetilde{desc}(S_T, \alpha_S(a))$ $N_{COMP} = \hat{N}_{REC}$

Appendix C. Optimization Rules

Notation Given an operator op_x, $ref(op_x)$ is the set of nodes and attributes mentioned among its parameters; for instance $ref(Merge(S_S, \{n_1, n_2, n_3\}, \ell)) = \{n_1, n_2, n_3\}$.

Table C.1: Predicates used for the definition of the applicability conditions of the optimization rules, referring to the operators between positions i and j of the input sequence.

Predicate	Description
$no\_use(n)$	$\nexists k : i < k < j,\ n \in ref(\Delta_k)$
$no\_use\_sub(n)$	$(\forall n \in \widetilde{desc}(S_{i+1}, n) \cup \{a \in Att_{i+1} : \alpha_{i+1}(a) \in \widetilde{desc}(S_{i+1}, n)\})(no\_use(n))$
$no\_ins\_label(\ell)$	$\nexists k : i < k < j,\ ((\Delta_k = Insert(S_k, S_X, \_) \vee \Delta_k = InsertFM(S_k, S_X, \_)) \wedge (\exists n_1 \in N_X)(\lambda_X(n_1) = \ell)) \vee (\Delta_k = Merge(S_k, \_, \ell)) \vee (\Delta_k = RSWA(S_k, \_, \ell)) \vee (\Delta_k = Rename(S_k, \_, \ell)) \vee (\Delta_k = InsertAttribute(S_k, \_, \ell))$
$no\_ins\_label\_sub(n)$	$(\forall n_1 \in desc(S_i, n))\ (no\_ins\_label(\lambda_i(n_1)) \wedge (\forall a_1 \in \alpha^{-1}(n_1))(no\_ins\_label(\lambda_i(a_1))))$
$no\_unique\_child(n)$	$\nexists k : 1 < k < j,\ siblings(S_k, n) = \varnothing$

Table C.2: Optimization rules

Rule	op_i	op_j	op_{new}	**Conditions**	pol
O1	$op_i \in \{Insert(S_i, S_X, n), InsertFM(S_i, S_X, n)\}$	$op_j \in \{Delete(S_j, n'), RSWA(S_j, n', \ell')\}$	op_j, $new = j$	$(op_j = Delete(\ldots) \Rightarrow \Delta_{j+1} \neq InsertFM(\ldots))$ $\wedge r_X \in desc(S_j, n') \wedge((op_j = RSWA(\ldots) \wedge op_i = InsertFM(\ldots)) \Rightarrow r_X \notin children(S_j, n'))$ $\wedge no\_use\_sub(r_X) \wedge (r_X \in N_X^\circ \Rightarrow no\_unique\_child(r_X))$	$recomp$
O2	$Merge(S_i, \{m_1, \ldots, m_p\}, \ell)$	$op_j \in \{Delete(S_j, n'), RSWA(S_j, n', \ell')\}$	op_j, $new = j$	Let n be the id. of the new node labeled ℓ $(op_j = Delete(\ldots) \Rightarrow \Delta_{j+1} \neq InsertFM(\ldots))$ $\wedge n \in desc(S_j, n') \wedge(op_j = RSWA(\ldots) \Rightarrow n \notin children(S_j, n')) \wedge no\_use(n) \wedge (\forall m_i \in \{m_1, \ldots, m_p\})$ $(no\_ins\_label(\lambda_i(m_i)))$	$recomp$
O3	$RSWA(S_i, n, \ell)$	$op_j \in \{Delete(S_j, n'), RSWA(S_j, n', \ell')\}$	op_j, $new = j$	Let a be the id. of the new attribute labeled ℓ $(op_j = Delete(\ldots) \Rightarrow \Delta_{j+1} \neq InsertFM(\ldots))$ $\wedge n \in \widetilde{desc}(S_j, n') \wedge no\_use(a) \wedge no\_ins\_label\_sub(n)$	$recomp$
O4	$Delete(S_i, n)$	$op_j \in \{Delete(S_j, n'), RSWA(S_j, n', \ell')\}$	op_j, $new = j$	$(op_j = Delete(\ldots) \Rightarrow \Delta_{j+1} \neq InsertFM(\ldots))$ $\wedge \Delta_{i+1} \neq InsertFM(\ldots) \wedge parent(S_i, n) \in \widetilde{desc}(S_j, n')$ $\wedge no\_ins\_label\_sub(n) \wedge no\_ins\_label(\lambda_i(n)) \wedge(\forall a_1 \in \alpha^{-1}(n))(no\_ins\_label(\lambda_i(a_1)))$	$recomp$
O5	$InsAttr(S_i, n, \ell)$	$op_j \in \{Delete(S_j, n'), RSWA(S_j, n', \ell')\}$	op_j, $new = j$	Let a be the id. of the new attribute labeled ℓ $(op_j = Delete(\ldots) \Rightarrow \Delta_{j+1} \neq InsertFM(\ldots))$ $\wedge \alpha_j(a) \in \widetilde{desc}(S_j, n') \wedge no\_use(a)$	$apply\_\varnothing$
O6	$DelAttr(S_i, a)$	$op_j \in \{Delete(S_j, n'), RSWA(S_j, n', \ell')\}$	op_j, $new = j$	$(op_j = Delete(\ldots) \Rightarrow \Delta_{j+1} \neq InsertFM(\ldots))$ $\wedge \alpha_i(a) \in \widetilde{desc}(S_j, n') \wedge no\_ins\_label(\lambda_i(a))$	$apply\_\varnothing$
O7	$Rename(S_i, n, \ell)$	$op_j \in \{Delete(S_j, n'), RSWA(S_j, n', \ell')\}$	op_j, $new = j$	$(op_j = Delete(\ldots) \Rightarrow \Delta_{j+1} \neq InsertFM(\ldots))$ $\wedge(n \in N_i \wedge op_j = Delete(\ldots) \Rightarrow n \in \widetilde{desc}(S_j, n'))$ $\wedge(n \in N_i \wedge op_j = RSWA(\ldots) \Rightarrow n \in desc(S_j, n'))$ $\wedge(op_j = RSWA(\ldots) \Rightarrow n \notin children(S_j, n'))$ $\wedge no\_ins\_label(\lambda_i(n)) \wedge(n \in Att_i \Rightarrow \alpha_i(n) \in \widetilde{desc}(S_j, n'))$	$apply\_\varnothing$
O8	$Rename(S_i, n, \ell)$	$Merge(S_j, \{m_1, \ldots, m_p\}, \ell')$	op_j, $new = j$	$n \in \{m_1, \ldots, m_p\} \wedge no\_ins\_label(\lambda_i(n))$	$apply\_\varnothing$
O9	$Rename(S_i, n, \ell)$	$Rename(S_j, n, \ell')$	op_j, $new = j$	$no\_ins\_label(\lambda_i(n)) \wedge \ell' \neq \lambda_i(n)$	$apply\_\varnothing$
O10	$Rename(S_i, a, \ell)$	$DelAttr(S_j, a)$	op_j, $new = j$	$a \in Att_i \wedge no\_ins\_label(\lambda_i(a))$	$apply\_\varnothing$
O11	$Move(S_i, n, m)$	$op_j \in \{Delete(S_j, n'), RSWA(S_j, n', \ell')\}$	op_j, $new = j$	$(op_j = Delete(\ldots) \Rightarrow \Delta_{j+1} \neq InsertFM(\ldots))$ $\wedge n, m \in \widetilde{desc}(S_j, n') \wedge parent(S_i, n) \in \widetilde{desc}(S_j, n')$ $\wedge no\_use\_sub(n)$	$recomp$
C1	$op_i \in \{Insert(S_i, S_X, n), InsertFM(S_i, S_X, n)\}$	$Delete(S_j, r_X)$	undef.	$\Delta_{j+1} \neq InsertFM(\ldots) \wedge no\_use\_sub(r_X) \wedge (r_X \in N_X^\circ \Rightarrow no\_unique\_child(r_X))$	$recomp$
C2	$InsAttr(S_i, n, \ell)$	$DelAttr(S_j, a)$	undef.	Let a be the id. of the new attribute labeled ℓ $no\_use(a)$	$apply\_op_j$
C3	$Rename(S_i, n, \ell)$	$Rename(S_j, n, \ell')$	undef.	$\ell' = \lambda_i(n) \wedge no\_ins\_label(\lambda_i(n))$	$apply\_op_j$
I1	$Insert(S_i, S_X, n)$	$Insert(S_j, S_Y, n')$	$Insert(S_i, S_Z, n)$, $new = i$, $S_Z = Insert(S_X, S_Y, n')$	$n' \in N_X \wedge((\forall n_1 \in N_Y \cup Att_Y)$ $(no\_ins\_label(\lambda_Y(n_1)))) \wedge prec(Insert(S_X, S_Y, n'))$	$recomp$
I2	$Insert(S_i, S_X, n)$	$Merge(S_j, \{m_1, \ldots, m_p\}, \ell)$	$Insert(S_i, S_Z, n)$, $new = i$, $S_Z = Merge(S_X, \{m_1, \ldots, m_p\}, n)$	Let m be the new node labeled ℓ $\{m_1, \ldots, m_p\} \subseteq N_X \wedge(\forall m_i \in \{m_1, \ldots, m_p\})(no\_use(m_i)) \wedge no\_ins\_label(\ell)$ $\wedge prec(Merge(S_X, \{m_1, \ldots, m_p\}, \ell))$	$recomp$
I3	$Insert(S_i, S_X, n)$	$RSWA(S_j, m, \ell)$	$Insert(S_i, S_Z, n)$, $new = i$, $S_Z = RSWA(S_X, m, \ell)$	$m \in N_X \wedge no\_use\_sub(m) \wedge no\_ins\_label(\ell)$ $\wedge prec(RSWA(S_X, m, \ell))$	$recomp$
I4	$Insert(S_i, S_X, n)$	$Delete(S_j, m)$	$Insert(S_i, S_Z, n)$, $new = i$, $S_Z = Delete(S_X, m)$	$\Delta_{j+1} \neq InsertFM(\ldots) \wedge m \in N_X \wedge no\_use\_sub(m)$ $\wedge prec(Delete(S_X, m))$	$recomp$
I5	$Insert(S_i, S_X, n)$	$InsAttr(S_j, m, \ell)$	$Insert(S_i, S_Z, n)$, $new = i$, $S_Z = InsAttr(S_X, m, \ell)$	Let a be the new attribute labeled ℓ $m \in N_X \wedge no\_ins\_label(\ell) \wedge prec(InsAttr(S_X, m, \ell))$	$apply\_op_j$
I6	$Insert(S_i, S_X, n)$	$DelAttr(S_j, a)$	$Insert(S_i, S_Z, n)$, $new = i$, $S_Z = DelAttr(S_X, a)$	$a \in Att_X \wedge no\_use(a) \wedge prec(InsAttr(S_X, m, \ell))$	$apply\_op_j$
I7	$Insert(S_i, S_X, n)$	$Rename(S_j, m, \ell)$	$Insert(S_i, S_Z, n)$, $new = i$, $S_Z = Rename(S_X, m, \ell)$	$m \in N_X \cup Att_X \wedge no\_ins\_label(\ell) \wedge prec(Rename(S_X, m, \ell))$	$apply\_op_j$
I8	$InsAttr(S_i, n, \ell)$	$Rename(S_j, n, \ell')$	$InsAttr(S_i, n, \ell')$, $new = i$	$no\_ins\_label(\ell')$	$apply\_op_j$
I9	$Insert(S_i, S_X, n)$	$Move(S_j, p, m)$	$Insert(S_i, S_Z, n)$, $new = i$, $S_Z = Move(S_X, p, m)$	$p, m \in N_X \wedge prec(Move(S_X, p, m)) \wedge no\_use\_sub(p)$	$recomp$

References

[1] M. Mazuran, E. Quintarelli, L. Tanca, Data mining for XML query-answering support, IEEE Transactions on Knowledge and Data Engineering 24 (8) (2012) 1393–1407.

[2] M. Chin, New data compression method reduces big-data bottleneck; outperforms, enhances JPEG, ACM TechNews, December 23th, 2013.

[3] D. Agrawal, P. Bernstein, E. Bertino, S. Davidson, U. Dayal, M. Franklin, J. Gehrke, L. Haas, A. Halevy, J. Han, H. V. Jagadish, A. Labrinidis, S. Madden, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, K. Ross, C. Shahabi, D. Suciu, S. Vaithyanathan, J. Widom, Challenges and opportunities with Big Data [White paper]`http://imsc.usc.edu/research/bigdatawhitepaper.pdf`.

[4] G. Koutrika, Y. E. Ioannidis, Personalizing queries based on networks of composite preferences, ACM Transactions on Database Systems 35 (2) (2010) 13:1–13:50.

[5] C. Bolchini, E. Quintarelli, L. Tanca, Carve: Context-aware automatic view definition over relational databases, Information Systems 38 (1) (2013) 45–67.

[6] K. Stefanidis, E. Pitoura, P. Vassiliadis, Managing contextual preferences, Information Systems 36 (8) (2011) 1158–1180.

[7] M. Baldauf, S. Dustdar, F. Rosenberg, A survey on context-aware systems, International Journal of Ad Hoc and Ubiquitous Computing 2 (4) (2007) 263–277.

[8] C. Bolchini, C. Curino, E. Quintarelli, F. A. Schreiber, L. Tanca, A data-oriented survey of context models, SIGMOD Record 36 (4) (2007) 19–26.

[9] J. Hong, E. Suh, S. Kim, Context-aware systems: A literature review and classification, Expert Systems with Applications 36 (4) (2009) 8509–8522.

[10] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, D. Riboni, A survey of context modelling and reasoning techniques, Pervasive and Mobile Computing 6 (2) (2010) 161–180.

[11] D. Zhang, H. Huang, C.-F. Lai, X. Liang, Q. Zou, M. Guo, Survey on context-awareness in ubiquitous media, Multimedia Tools and Applications 67 (1) (2013) 179–211.

[12] C. Bolchini, C. Curino, G. Orsi, E. Quintarelli, R. Rossato, F. A. Schreiber, L. Tanca, And what can context do for data?, Communications of the ACM 52 (11) (2009) 136–140.

[13] C. Bolchini, E. Quintarelli, R. Rossato, Relational data tailoring through view composition, in: Proceedings of ER 2007, 26th International Conference on Conceptual Modeling, Springer, 2007, pp. 149–164.

[14] M. M. Lehman, Software's future: Managing evolution, IEEE Software 15 (1) (1998) 40–44.

[15] D. Bianchini, S. Montanelli, C. Aiello, R. Baldoni, C. Bolchini, S. Bonomi, S. Castano, T. Catarci, V. De Antonellis, A. Ferrara, M. Melchiori, E. Quintarelli, M. Scannapieco, F. A. Schreiber, L. Tanca, Emergent semantics and cooperation in multi-knowledge communities: the esteem approach, World Wide Web 13 (1-2) (2010) 3–31.

[16] S. Buchholz, T. Hamann, G. Hübsch, Comprehensive structured context profiles (CSCP): Design and experiences, in: Workshop proceedings of PerCom 2004, 2nd Conference on Pervasive Computing and Communications, IEEE Computer Society, 2004, pp. 43–47.

[17] P.-G. Raverdy, O. Riva, A. de La Chapelle, R. Chibout, V. Issarny, Efficient context-aware service discovery in multi-protocol pervasive environments, in: Proceedings of MDM 2006, 7th International Conference on Mobile Data Management, IEEE Computer Society, 2006, p. 3.

[18] H. Chen, T. W. Finin, A. Joshi, An intelligent broker for context-aware systems, in: Adjunct Proceedings of UbiComp 2003, 5th International Conference on Ubiquitous Computing, 2003, pp. 183–184.

[19] J. R. Hoyos, J. García-Molina, J. A. Botía, A domain-specific language for context modeling in context-aware systems, Journal of Systems and Software 86 (11) (2013) 2890 – 2905.

[20] R. De Virgilio, R. Torlone, Modeling heterogeneous context information in adaptive web based applications, in: Proceedings of ICWE 2006, 6th International Conference on Web Engineering, IEEE Computer Society, 2006, pp. 56–63.

[21] E. Quintarelli, E. Rabosio, L. Tanca, Context schema evolution in context-aware data management, in: Proceedings of ER 2011, 30th International Conference on Conceptual Modeling, Springer, 2011, pp. 290–303.

[22] P. Buneman, S. B. Davidson, A. Kosky, Semantics of database transformations, in: Semantics in Databases, Springer, 1995, pp. 55–91.

[23] M. Erwig, Toward the automatic derivation of XML transformations, in: Proceedings of Conceptual Modeling for Novel Application Domains, ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM, Springer, 2003, pp. 342–354.

[24] C. Bolchini, E. Quintarelli, Context-driven data filtering: a methodology, in: Proceedings of OTM 2006, International Workshops on On the Move to Meaningful Internet Systems (Part II), Springer, 2006, pp. 1986–1995.

[25] G. Orsi, Context based querying of dynamic and heterogeneous information sources, Ph.D. thesis, Politecnico di Milano (2010).

[26] T. Schwentick, Xpath query containment, SIGMOD Record 33 (1) (2004) 101–109.

[27] P. Garza, E. Quintarelli, E. Rabosio, L. Tanca, Run-time, adaptive generation of contextual views, Research report, Politecnico di Milano, http://home.deib.polimi.it/rabosio/Papers/GQRT2013.pdf (2013).

[28] J. Banerjee, W. Kim, H.-J. Kim, H. F. Korth, Semantics and implementation of schema evolution in object-oriented databases, in: Proceedings of SIGMOD 1987, International Conference on Management of Data, ACM, 1987, pp. 311–322.

[29] W. Kim, H.-T. Chou, Versions of schema for object-oriented databases, in: Proceedings of VLDB 1988, 14th International Conference on Very Large Databases, Morgan Kaufmann, 1988, pp. 148–159.

[30] C. De Castro, F. Grandi, M. R. Scalas, Schema versioning for multitemporal relational databases, Information Systems 22 (5) (1997) 249–290.

[31] C. Curino, H. J. Moon, C. Zaniolo, Graceful database schema evolution: the PRISM workbench, PVLDB 1 (1) (2008) 761–772.

[32] A. Cleve, A.-F. Brogneaux, J.-L. Hainaut, A conceptual approach to database applications evolution, in: Proceedings of ER 2010, 29th International Conference on Conceptual Modeling, Springer, 2010, pp. 132–145.

[33] L. Stojanovic, A. Maedche, B. Motik, N. Stojanovic, User-driven ontology evolution management, in: Proceedings of EKAW 2002, 13th International Conference on Knowledge Engineering and Knowledge Management, Springer, 2002, pp. 285–300.

[34] P. Plessers, O. De Troyer, Ontology change detection using a version log, in: Proceedings ISWC 2005, 4th International Semantic Web Conference, Springer, 2005, pp. 578–592.

[35] V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, V. Christophides, On detecting high-level changes in RDF/S KBs, in: Proceedings of ISWC 2009, 8th International Semantic Web Conference, Springer, 2009, pp. 473–488.

[36] H. Su, D. Kramer, L. Chen, K. T. Claypool, E. A. Rundensteiner, XEM: Managing the evolution of XML documents, in: Proceedings of RIDE-DM 2001, 11th International Workshop on Research Issues in Data Engineering: Document Management for Data Intensive Business and Scientific Applications, IEEE Computer Society, 2001, pp. 103–110.

[37] G. Guerrini, M. Mesiti, D. Rossi, Impact of XML schema evolution on valid documents, in: Proceedings of WIDM 2005, 7th International Workshop on Web Information and Data Management, ACM, 2005, pp. 39–44.

[38] G. Guerrini, M. Mesiti, M. A. Sorrenti, XML schema evolution: Incremental validation and efficient document adaptation, in: Proceedings of XSym 2007, 5th International XML Database

Symposium, Springer, 2007, pp. 92–106.
[39] A. Wienberg, M. Ernst, A. Gawecki, O. Kummer, F. Wienberg, J. W. Schmidt, Content cchema evolution in the CoreMedia®; content application platform CAP, in: Proceedings of EDBT 2002, 8th International Conference on Extending Database Technology, Springer, 2002, pp. 712–721.
[40] S. Bossung, H.-W. Sehring, P. Hupe, J. W. Schmidt, Open and dynamic schema evolution in content-intensive web applications, in: Proceedings of WEBIST 2006, 2nd International Conference on Web Information Systems and Technologies, INSTICC Press, 2006, pp. 109–116.
[41] J. F. Roddick, A survey of schema versioning issues for database systems, Information and Software Technology 37 (7) (1995) 383 – 393.
[42] Y. Velegrakis, R. J. Miller, L. Popa, Preserving mapping consistency under schema changes, VLDB Journal 13 (3) (2004) 274–293.
[43] S. Melnik, E. Rahm, P. A. Bernstein, Rondo: A programming platform for generic model management, in: Proceedings of SIGMOD 2003, International Conference on Management of Data, ACM, 2003, pp. 193–204.
[44] P. Atzeni, P. Cappellari, R. Torlone, P. A. Bernstein, G. Gianforme, Model-independent schema translation, VLDB Journal 17 (6) (2008) 1347–1370.
[45] R. De Virgilio, R. Torlone, A framework for the management of context data in adaptive web information systems, in: Proceedings of ICWE 2008, 8th International Conference on Web Engineering, IEEE Computer Society, 2008, pp. 261–272.
[46] M. Hartung, J. Terwilliger, E. Rahm, Recent advances in schema and ontology evolution, in: Schema Matching and Mapping, Data-Centric Systems and Applications, Springer, 2011, pp. 149–190.
[47] Schema evolution publication categorizer, URL: `http://se-pubs.dbs.uni-leipzig.de/`, accessed 2014-08-03.
[48] B. Staudt Lerner, A model for compound type changes encountered in schema evolution, ACM Transactions on Database Systems 25 (1) (2000) 83–127.
[49] P. A. Bernstein, T. J. Green, S. Melnik, A. Nash, Implementing mapping composition, VLDB Journal 17 (2) (2008) 333–353.
[50] S. Amano, L. Libkin, F. Murlak, XML schema mappings, in: Proceedings of PODS 2009, 28th Symposium on Principles of Database Systems, ACM, 2009, pp. 33–42.
[51] I. Tatarinov, Z. G. Ives, A. Y. Halevy, D. S. Weld, Updating XML, in: Proceedings of SIGMOD 2001, International Conference on Management of Data, ACM, 2001, pp. 413–424.
[52] A. Balmin, Y. Papakonstantinou, V. Vianu, Incremental validation of XML documents, ACM Transactions on Database Systems 29 (4) (2004) 710–751.
[53] D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, M. Arenas, Efficient incremental validation of XML documents, in: Proceedings of ICDE 2004, 20th International Conference on Data Engineering, IEEE Computer Society, 2004, pp. 671–682.
[54] B. Bouchou, M. Halfeld Ferrari Alves, Updates and incremental validation of XML documents, in: Proceedings of DBPL 2003, 9th International Workshop on Database Programming Languages, Springer, 2003, pp. 216–232.
[55] K. Hashimoto, Y. Ishihara, T. Fujiwara, Schema update operations preserving the expressive power in XML databases, in: Proceedings of ICDE Workshops, IEEE Computer Society, 2005, p. 1229.
[56] J. Chabin, M. Halfeld Ferrari Alves, M. A. Musicante, P. Réty, Minimal tree language extensions: A keystone of XML type compatibility and evolution, in: Proceedings of ICTAC 2010, 7th International Colloquium on Theoretical Aspects of Computing, Springer, 2010, pp. 60–75.
[57] M. Shoaran, A. Thomo, Evolving schemas for streaming XML, Theoretical Computer Science 412 (35) (2011) 4545–4557.
[58] M. B. L. Tan, A. Goh, Keeping pace with evolving XML-based specifications, in: Proceedings of Current Trends in Database Technology - EDBT 2004 Workshops, Springer, 2004, pp. 280–288.
[59] M. Klettke, Conceptual XML schema evolution - the CoDEX approach for design and redesign, in: Proceedings of BTW 2007, Datenbanksysteme in Business, Technologie und Web Workshop, Verlagshaus Mainz, 2007, pp. 53–63.
[60] E. Domínguez, J. Lloret, B. Pérez, Á. Rodríguez, A. L. Rubio, M. A. Zapata, Evolution of XML schemas and documents from stereotyped UML class models: A traceable approach, Information & Software Technology 53 (1) (2011) 34–50.
[61] B. V. N. Prashant, P. S. Kumar, Managing XML data with evolving schema, in: Proceedings of COMAD 2006, 13th International Conference on Management of Data, Tata McGraw-Hill Publishing Company Limited, 2006, pp. 174–177.
[62] F. Cavalieri, G. Guerrini, M. Mesiti, B. Oliboni, On the reduction of sequences of XML document and schema update operations, in: Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, IEEE Computer Society, 2011, pp. 77–86.
[63] A. Poulovassilis, P. McBrien, A general formal framework for schema transformation, Data and Knowledge Engineering 28 (1) (1998) 47–71.
[64] F. Grandi, F. Mandreoli, A formal model for temporal schema versioning in object-oriented databases, Data and Knowledge Engineering 46 (2) (2003) 123–167.
[65] K. T. Claypool, C. Natarajan, E. A. Rundensteiner, Optimizing performance of schema evolution sequences, in: Proceedings of the International Symposium on Objects and Databases, Springer, 2000, pp. 114–127.
[66] T. Zäschke, S. Leone, M. C. Norrie, Optimising schema evolution operation sequences in object databases for data evolution, in: Proceedings of ER 2012, 31st International Conference on Conceptual Modeling, Springer, 2012, pp. 369–382.
[67] N. Tong, Database schema transformation optimisation techniques for the AutoMed system, in: Proceedings of BNCOD 2003, 20th British National Conference on Databases, Springer, 2003, pp. 157–171.
[68] F. Cavalieri, G. Guerrini, M. Mesiti, Dynamic reasoning on XML updates, in: Proceedings of EDBT 2011, 4th International Conference on Extending Database Technology, ACM, 2011, pp. 165–176.
[69] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, J. Simon, XQuery Update Facility 1.0. W3C recommendation, `http://www.w3.org/TR/xquery-update-10/`.
[70] Saxonica, Saxon: the XSLT and XQuery processor `http://saxon.sourceforge.net/`.