A Generalized Modeling Framework for Schema Versioning Support

Fabio Grandi, Federica Mandreoli and Maria Rita Scalas
C.S.I.TE.-C.N.R. - D.E.I.S, University of Bologna, Italy
Viale Risorgimento, 2 - I-40136 Bologna, Italy
e-mail: {fgrandi, fmandreoli, mrscalas}@deis.unibo.it

Abstract

Advanced object-oriented applications require the management of schema versions, in order to cope with changes in the structure of the stored data. Two types of versioning have been separately considered so far: branching and temporal. The former arose in application domains like CAD/CAM and software engineering, where different solutions have been proposed to support design schema versions (consolidated versions). The latter concerns temporal databases, where some works considered temporal schema versioning to fulfil advanced needs of other typical object-oriented applications like GIS and the multimedia ones.

In this work, we propose a general model which integrates the two approaches by supporting both design and temporal schema versions. The model is provided with a complete set of schema change primitives for full-fledged version manipulation whose semantics is described in the paper.

Keywords: *Schema versioning, Schema evolution, OODBMS, Temporal databases*

1 Introduction

In the literature, the need for maintaining data under a schema definition which undergoes changes is not a new issue [Rod96] and *schema versioning* offers a smart solution to the problem. Moreover, advanced application systems can also take benefit of schema versioning to enhance their functionalities. In the object-oriented field, the strictly-related problem of *schema evolution*, concerning the maintenance of extant data in response to schema changes, was considered in several works (see, for instance, [Odb96] for bibliographies and resource links). However, also early requirements for schema versioning arose in application domains like CAD/CAM and software engineering in order to support dynamic aspects of the engineering design process (see [Kat90] for a brief review of different approaches). In this context, schema versions can be organized as a DAG [KC88, Lau97], where version derivation lines can be branching, in order to represent alternatives, and also, sometimes, merging. In the following, we will refer to this approach as *branching schema versioning*. Lately, schema versioning has also proved useful for other typical object-oriented applications, like multimedia ones [KB96] and GIS [BM98]. Such applications often have temporal requirements, as the evolution of objects has to be tracked and documented. In this context, the temporal aspect of versioning has thus to be considered and *temporal schema versioning* is required in order to represent the history of changes in the object *structure*. While temporal schema versioning has been extensively studied in the context of relational databases [DGS97, Rod96], only a few studies concerning object-oriented databases have taken it into account [GSÖP98, CJK91, GMS98]. Anyway, in all these proposals the time lines involved, valid time and/or transaction time, are represented as a single time line and they do not support alternative sequences of events.

The main attempt of this work is to integrate the branching approach in the temporal schema versioning framework. To this purpose, we present a *generalized* schema versioning framework which provides support for both schema versioning modalities. The availability of both modalities in a single system is aimed at improving its expressive power and application potentialities. For instance, in GIS, the chief versioning modality is presumably temporal versioning, where it is used to represent the history of the structural changes in the modeled reality. Parallel versions can be used to support distinct scenarios as required, for example, in planning activities. In this case, parallel versions can be introduced to foresee (and compare) different land evolutions due to the construction of a bridge or a submarine tunnel across a sea channel. From a technical point of view, this means supporting the bifurcation of the valid-time line necessary to explore hypothetical courses of events (in the past as well as in the future), but impossible with temporal schema versioning only, where a single time line is adopted, unless *branching time* is considered [SR99]. On the other hand, in the engineering and

design field [DL88, Kat90, KL84, Lan86], the chief versioning modality is the use of consolidated versions and the management of their derivation lines. Here the addition of temporal versioning can be used to model the history of all the intermediate schema changes applied to a consolidated version before a new one is released. As already pointed out in [GSÖP98], this adds great flexibility by enabling a full traceability of the design process.

The rest of the paper is organized as follows. Section 2 introduces a new database notion to support the schema development and maintenance process. In Sec. 3 we present our schema versioning approach, with emphasis on the behaviour of the proposed schema change primitives in Sec. 4. An example on how schema changes can be applied can be found in Sec. 5. Finally, Section 6 outlines conclusion remarks and future directions of work.

2 A generalized database model for version management

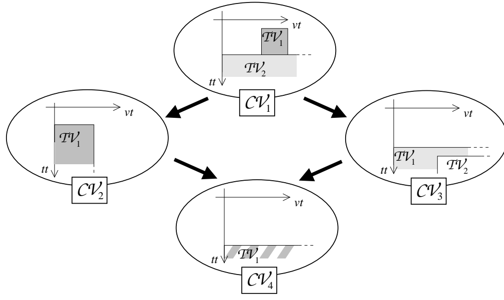

Figure 1. An example of a database

In the proposed model, we revisit the notion of database usually considered in a snapshot system [AHV95] to support the schema development and maintenance process by allowing both temporal and consolidated versions. To this end, we model a scenario where different consolidated versions are stored, with the possibility of deriving a new *consolidated version* from already existing ones. Therefore, a database is represented by a DAG, composed of a set of nodes and edges corresponding to consolidated versions and inheritance relationships, respectively. It is rooted by the first consolidated version supplied by users. For each consolidated version, our model enables users to keep track of all local schema changes by providing a bitemporal schema versioning support similar to our previous proposal [GMS98]. Bitemporal schema versioning [DGS97], based on the valid and transaction time dimensions, enables retro- and pro-active schema changes (keeping track of them in the system) to produce past, present and future schema versions.

In our model, the versioning granularity is the *temporal version*, that is $\mathcal{TV} = (\mathcal{SV}, \mathcal{SDV})$, where $\mathcal{SV}$ is a schema version (which could be the whole schema of a snapshot database) and $\mathcal{SDV}$ is the corresponding extensional component (stored data version) containing the data instances. Each temporal version has a *temporal pertinence* which is a subset of the Cartesian product of transaction time and valid time domains. Time values vary over a discrete set $\mathcal{TIME} = \{0, 1, \ldots, now, \ldots, \infty\}$ of chronons [JCG+98]. Since the meaning of the domain $\mathcal{TIME}$ is different in the two time dimensions considered [ÖS95], we distinguish $\mathcal{TIME}_t$ from $\mathcal{TIME}_v$, where t means transaction and v means valid.

In order to identify consolidated versions, we adopt a user-defined naming method which allows users to associate symbolic labels to consolidated versions. A combination of this method with the bitemporal timestamping mechanism represents the way in which temporal versions are univocally *referenced* within the database.

The complete definition of a database can be given as follows:

Definition 1 (Database) *Let $\mathcal{TVS}$ be the set of all possible Temporal Versions $\mathcal{TV}$ and $\mathcal{L}$ a set of labels for consolidated versions. A Database graph is a DAG named $\mathcal{DBG} = (\mathcal{L}, \mathcal{E}, \mathcal{DB})$ where $\mathcal{E} \subseteq \mathcal{L} \times \mathcal{L}$ is a set of edges and $\mathcal{DB}$ is a function*

$$\mathcal{DB} : (\mathcal{L} \times \mathcal{TIME}_t \times \mathcal{TIME}_v) \rightarrow \mathcal{TVS} \cup \{\emptyset\}$$

which associates each tuple made up of a label l and a bitemporal chronon (tt, vt) to a temporal version, if it exists, that is:

$$\mathcal{DB}\,(l, tt, vt) = \begin{cases} \mathcal{TV} & \text{if } \exists\mathcal{TV}\text{, in } l\text{, valid in } (tt, vt) \\ \emptyset & \text{otherwise} \end{cases}$$

where $l \in \mathcal{L}$, $tt \in \mathcal{TIME}_t$, $vt \in \mathcal{TIME}_v$.

Adopting the database definition above, facilities to reach a specific consolidated version are also provided. In fact, by applying the $\mathcal{DB}$ function to a label l, all the temporal versions contained in the consolidated version with name l can be obtained. This result represents the full history of the changes applied to the first schema defined with the creation of the consolidated version.

We also define the Current Database function $\mathcal{CDB}(l, vt) = \mathcal{DB}(l, now, vt)$ as a view on current versions only. Such a function will be used to slightly simplify the formalization of the schema changes, which can act on current temporal versions only.

Figure 1 shows a database made up of four consolidated versions ($\mathcal{CV}_i$) represented with bubbles. Each consolidated version history is maintained by means of temporal versions ($\mathcal{TV}_i$) placed along the "private" valid/transaction time lines of the bubbles.

	Schema changes on node		
AddProperty	Add a new property	**ChangeMethCode**	Change a method code
DeleteProperty	Delete an existing property	**AddClass**	Add a new isolated class
ChangePropName	Change a property name	**DeleteClass**	Delete an isolated class
ChangePropType	Change a property type	**ChangeClassName**	Change a class name
AddMethod	Add a new method	**AddSuperclass**	Add a superclass to a class
DeleteMethod	Delete an existing method	**DeleteSuperclass**	Delete a superclass from a class
ChangeMethName	Change a method name		
	Merge-type schema changes		
PickProperty	Pick an existing property	**PickMethod**	Pick an existing operation
PickClass	Pick an existing class	**MergeVersion**	Merge versions
	Schema changes on DAG		
NewNode	Introduce a new isolated node	**NewEdge**	Introduce an edge

Table 1. List of primitive schema changes

3 Schema Version Management

According to the meaning associated with versions, there are different reasons which lead to the introduction of new versions. For instance, in the engineering context, new consolidated versions are introduced to represent stable design stages, whereas temporal versions can be used to represent intermediate design phases. On the other hand, in planning activities, temporal versions are adopted to maintain the history of structural changes, whereas consolidated versions can be used to model different scenarios involving alternative courses of events. Our model provides facilities for introducing and organizing both kinds of versions for all the above purposes by means of a set of *primitive schema changes*, which allows users to:

1. update a consolidated version by means of schema changes, maintaining the history of all its past or future versions;
2. generate a new consolidated version;
3. integrate in a consolidated version characteristics of other consolidated versions;
4. add a semantical derivation relationship between consolidated versions.

The supported primitive schema changes are listed in Table 1, where they have been partitioned into three categories: “Schema changes on node”, “Merge-type schema changes” and “Schema changes on DAG”. The “Schema changes on node” collection is a complete set of schema updates applicable to an object-oriented schema to support temporal schema versioning. It corresponds to the primitives usually considered in schema change papers [BKKK87] for addition, deletion and modification of all the elements of a generic object-oriented data model (namely properties, methods and classes) [AHV95]. Their exact expression (syntax and required parameters) depends on the particular choice of the underlying object model. For instance, [GMS98] explicits this collection for a temporal schema versioning model based on the ODMG 2.0 model [CBB+97]. They are necessary when an update to a consolidated version has to be applied and no new consolidated version is thus introduced by them. When one of these primitives is applied, a new temporal version is created in the consolidated version. New temporal versions are associated with a valid-time pertinence specified by the user, whereas they are automatically assigned a $[now, \infty]_t$ transaction timestamp by the system. Hence, stored data can be accessed by means of past, present or future temporal schema versions, which can be selected by means of their temporal pertinence. All the other mechanisms for the version management (see enumeration below) are included in the “Merge-type schema changes” and the “Schema changes on DAG” collections. They are necessary for a user-driven control of consolidated versions and their derivation relationships:

- the “Merge-type schema changes” allow users to integrate in a consolidated version an element or a complete temporal version taken from another consolidated version. The main difference between the pick-type schema changes and the corresponding add-type in the “Schema changes on node” collection (e.g. **PickProperty** vs. **AddProperty**) is that the first ones consider populated elements whose values associated

with objects are inherited by the receiver version, while the second ones only add elements with a default value.

- the "Schema changes on DAG" allow users either to introduce a consolidated version (**NewNode**) or to introduce a derivation in the hierarchy (**NewEdge**);

4 Semantics for schema changes

In this Section we introduce the semantics of the schema changes listed in Table 1. Since our model provides a bitemporal schema versioning support, the schema change primitives do not operate in an "update-in-place" fashion but always manipulate and produce temporal versions. Although this is the usual approach adopted in the temporal context, it is a novel issue in the branching one. In fact, the consolidated versions usually managed in the branching context are snapshot and no history of changes is maintained. In our model, the finer granularity temporal versioning actually allows the full history of schema changes to be maintained in the context of each consolidated version.

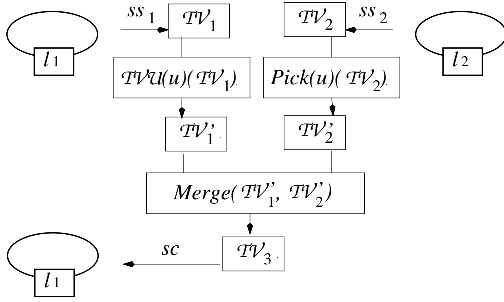

Figure 2. The $\mathcal{DBU}$ behaviour for the schema changes on edge

The way the primitive schema changes are managed in our model is depicted in Fig. 2. Each schema change involves, in general, two consolidated versions (nodes labeled l_1 and l_2). Both can work as sources of the change process, whereas the first one always acts as destination, as it receives the output of the process.

The execution of a schema change can be summarized in the following steps:

1. The set of nodes is updated as follows: $\mathcal{L}' = \mathcal{L} \cup \{l_1\}$. This is necessary when l_1 was not present yet (execution of a **NewNode**), otherwise it has no effect.

2. The set of edges is updated as follows:

$$\mathcal{E}' = \begin{cases} \mathcal{E} \cup \{(l_2, l_1)\} & \text{if } l_2 \neq null \\ \mathcal{E} & \text{otherwise} \end{cases}$$

   The update is necessary when a new edge has to be added (execution of Merge-type changes or a **NewEdge** primitive).

3. For each source version, one of its current temporal versions is selected. To this end, two valid-time parameters, called *schema selection validities* (ss_1 and ss_2), are specified by users. The selected temporal versions are $\mathcal{TV}_1 = \mathcal{CDB}(l_1, ss_1)$ and $\mathcal{TV}_2 = \mathcal{CDB}(l_2, ss_2)$.

4. A Temporal Version Update function ($\mathcal{TVU}$) is applied to the first selected temporal version in order to execute the required schema change u: $\mathcal{TV}'_1 = \mathcal{TVU}(u)(\mathcal{TV}_1)$.

5. A $Pick$ function is applied to the second selected temporal version in order to isolate part of it according to the required schema change u: $\mathcal{TV}'_2 = Pick(u)(\mathcal{TV}_2)$.

6. A $Merge$ function is used to merge the results of the first two functions in a new temporal version $\mathcal{TV}_3$: $\mathcal{TV}_3 = Merge(\mathcal{TV}'_1, \mathcal{TV}'_2)$.

7. The destination version is updated by placing the new temporal version in it. This is done by updating the $\mathcal{DB}$ function, where the new temporal version is assigned the validity specified by the user through the *schema change validity* sc (which is a valid-time element [JCG+98]) and a transaction-time pertinence defined by the system as $[now, \infty]_t$:

$$\mathcal{DB}'(l, tt, vt) = \begin{cases} \mathcal{TV}_3 & \text{if } l = l_1, \\ & tt \geq now, \\ & vt \in sc \\ \mathcal{DB}\,(l, tt, vt) & \text{otherwise} \end{cases}$$

Notice that, after the creation of the new temporal version $\mathcal{TV}_3$, invariants concerning its consistency should be enforced:

- the correctness of its schema version has to be checked [BKKK87]. To this end, the axiomatic approach proposed in [PÖ97], which is independent on the underlying object model, could be adopted;

- the consistency of its stored data version, with respect to their corresponding type version in the schema version, has also to be guaranteed. To this end, a propagation mechanism on data is included in the $\mathcal{TVU}$, $Pick$ and $Merge$ functions.

The whole schema change procedure described above is formalized in our model via the Database Update function $\mathcal{DBU}$.

Definition 2 (Database Update function) *Let $\mathcal{SC}$ be the set of all possible schema changes, $\mathcal{DBGS}$ the set of all possible database graphs $\mathcal{DBG}$, then:*

$$\mathcal{DBU} : \mathcal{SC} \rightarrow (\mathcal{DBGS} \times \mathcal{L} \times \mathcal{TIME}_v \times \mathcal{L} \times \mathcal{TIME}_v \times 2^{\mathcal{TIME}_v} \rightarrow \mathcal{DBGS})$$

where, if $\mathcal{DBG}$ is a database graph, u a schema change, l_1, l_2 labels, ss_1 and ss_2 schema selection validities, sc the schema change validity ($ss_i \in \mathcal{TIME}_v$, for $i = 1, 2$, $sc \subseteq \mathcal{TIME}_v$), then

$$\begin{aligned} \mathcal{DBG}' &= (\mathcal{L}', \mathcal{E}', \mathcal{DB}') \\ &= \mathcal{DBU}(u)(\mathcal{DBG}, l_1, ss_1, l_2, ss_2, sc) \end{aligned}$$

where the definition of $\mathcal{L}'$, $\mathcal{E}'$ and $\mathcal{DB}'$ has been given above.

The general schema described so far, when applied for the execution of a particular primitive in Tab. 1, has a specialized behaviour which will be detailed in the following Subsections.

4.1 Schema changes on node

When a schema change on node is applied, the $\mathcal{DBU}$ function operates on a single temporal version belonging to a single existing consolidated version labeled by l_1 ($null \neq l_1 \in \mathcal{L}$, $l_2 = null$ and, thus, $\mathcal{TV}_2 = \emptyset$). In this case, with the positions $Pick(u)(\emptyset) = \emptyset$, $Merge(\mathcal{TV}, \emptyset) = Merge(\emptyset, \mathcal{TV}) = \mathcal{TV}$, we have $\mathcal{TV}_3 = \mathcal{TV}'_1 = \mathcal{TVU}(u)(\mathcal{TV}_1)$.

Hence, the $\mathcal{TVU}$ function generates a new temporal version $\mathcal{TV}'_1$ by applying the schema change u to the current temporal version $\mathcal{TV}_1$ satisfying the schema selection validity condition. The outcome $\mathcal{TV}_3$ with its own pertinence $[now, \infty]_t \times sc$ is placed by the $\mathcal{DBU}$ function in the same consolidated version with label l_1. The database DAG remains unchanged.

An example of detailed $\mathcal{TVU}$ function behaviour for the set of schema changes defined on the basis of the ODMG model can be found in [GMS98].

4.2 Merge-type schema changes

When a merge-type change is applied, two existing consolidated versions are always involved ($null \neq l_1 \in \mathcal{L}$ and $null \neq l_2 \in \mathcal{L}$). As far as the $Pick$ function behaviour is concerned, we distinguish three cases:

PickClass The aim is to integrate a class belonging to the temporal version $\mathcal{TV}_2$ in the temporal version $\mathcal{TV}_1$. When the **PickClass** primitive is applied, $\mathcal{TV}'_2$ has a schema version which only contains the selected class and a stored data version which is made up of all the objects which are instances of that class.

PickProperty or **PickMethod** The aim of these primitives is to integrate part of a class (i.e. a property or a method) belonging to $\mathcal{TV}_2$ in a class with the same name (if it exists) contained in $\mathcal{TV}_1$. In this case, $\mathcal{TV}'_2$ has a schema version which corresponds to the selected class containing the only selected model element with its type. If the **PickProperty** is applied, the corresponding stored data version is not empty but contains all the objects which are instances of the selected class, whose values only contain the value associated with the specified property.

MergeVersion The aim is to integrate the entire temporal version $\mathcal{TV}_2$ in $\mathcal{TV}_1$. When the **MergeVersion** is applied, the $Pick$ function (with the position $Pick$(**MergeVersion**)$(\mathcal{TV}) = \mathcal{TV}$) just makes a copy of the temporal version $\mathcal{TV}_2$.

In all these cases, a new edge is automatically generated (from l_2 to l_1) to indicate the effected semantic derivation, whereas the set of nodes remains unchanged. Moreover, the $\mathcal{TVU}$ function has no effect (formally $\mathcal{TVU}(u)(\mathcal{TV}) = \mathcal{TV}$ for each u in the set Merge-type schema changes) and, thus, $\mathcal{TV}_3 = Merge(\mathcal{TV}_1, \mathcal{TV}'_2)$.

The basic idea behind the $Merge$ function is to integrate two temporal versions. The general schema integration problem considers two or more arbitrary schemas which have been developed completely independent of each other [Bre90]. In contrast, when integrating schema versions of the same DAG, the schemas involved usually model the same real world entities or different portions of the same reality of interest, where the same terminology has probably been adopted. For this reason, the system can infer some semantic relationships between the schemas. In our model, semantic equivalences are reduced to syntactic equivalences. We assume that identical names represent the same semantics; we also introduce a user interaction by allowing users to express further syntactic equivalences. For the sake of brevity and simplicity, we do not formally define the $Merge$ function here but we explain its behaviour. Given two temporal versions $\mathcal{TV}_1 = (\mathcal{SV}_1, \mathcal{SDV}_1)$ and $\mathcal{TV}_2 = (\mathcal{SV}_2, \mathcal{SDV}_2)$ and some syntactic equivalences (in the form $n \leftarrow m$ which means that each term m is substituted with the term n) then the $merge$ function:

1. for each syntactic equivalence $n \leftarrow m$, replaces each occurrence of m with n in $\mathcal{TV}_2 = (\mathcal{SV}_2, \mathcal{SDV}_2)$;
2. merges the schema versions $\mathcal{SV}_1$ and $\mathcal{SV}_2$. If two classes with the same name occur, it just makes the union of their types. Furthermore, if they contain

two properties with the same name, it chooses the most specific type (for more details, with reference to ODMG types, see [GMS98]);

3. recalculates the new hierarchical lattice. This step is necessary because, after the previous step, some classes can be associated with new types. It can be computed by means of a normalization process [TS93] or an intelligent tool [BN94];

4. integrates the stored data version $\mathcal{SDV}_1$ and $\mathcal{SDV}_2$.

In the fourth step, a problem arises when two object versions with the same OID occur in the two stored data versions to be merged, since their outcome has to be a single object version with the same OID (assuming *object equality by identity*). However, they can only descend from a common ancestor of the stored data versions involved. Hence, if they are instances of classes with the same name, they actually represent different versions of the same real world entity and, thus, during step two, a merge of their types has been effected and a merge of the object values is possible. In this case, we obtain a new object version with the common OID and a new value which represents a new version of the same real world entity. Otherwise, if the two object versions with the same OID belong to different classes, the user has to choose one of them. Moreover, in this case, a merge of their values is not possible because they belong to classes with different names for which a merge has not been effected during step two.

Notice that, if $\mathcal{TV}_1 = \emptyset$ (as it happens when $ss_1 = null$), with the position $\mathcal{TVU}(u)(\emptyset) = \emptyset$, the $Merge$ function simply makes $\mathcal{TV}_3 = \mathcal{TV}'_2$. Therefore, a copy of the elements "picked" from l_2 is put by the $\mathcal{DBU}$ function into l_1.

4.3 Schema changes on DAG

The schema changes on DAG can be used for explicit management of DAG elements. Thanks to the proposed semantics of the $\mathcal{TVU}$, $Pick$ and $Merge$ functions (including the positions done), such operations can easily be effected as particular cases of the operations seen before. The **NewNode**, which creates a new empty and isolated consolidated version, can be considered as a special case of *Schema changes on node* application, where the first node did not exist before (i.e. $l_1 \notin \mathcal{L}$ and, thus, $\mathcal{TV}_1 = \emptyset$) and the second one is not specified ($l_2 = null$ and, thus, $\mathcal{TV}_2 = \emptyset$). Therefore, also $\mathcal{TV}_3 = \emptyset$ and the **NewNode** action can simply be expressed as:

$$\mathcal{DBG}' = \mathcal{DBU}(\textbf{NewNode})(\mathcal{DBG}, l_1, null, null, null, null)$$

On the other hand, the **NewEdge**, which creates a derivation relationship between two nodes, can be considered as a special case of *Merge-type schema changes* application, where the two labels l_1 and l_2 reference already existing nodes but nothing to integrate is selected in the second node ($\mathcal{TV}_2 = \emptyset$). Therefore, it can be expressed as:

$$\mathcal{DBG}' = \mathcal{DBU}(\textbf{NewEdge})(\mathcal{DBG}, l_1, null, l_2, null, null)$$

5 An infrastructure planning example

An elective aim of a generalized versioning model is the development of effective decision support systems based on the analysis of complex spatio-temporal data. For instance, GIS applications are often used in planning activities (e.g. concerning transport, infrastructures, facilities, utilities). In this case, the comparison between different solutions, also concerning alternative geoinformation evolutions, is frequently required and simulations effected on alternative scenarios can be very helpful.

Let us consider an infrastructure planning example which concerns sea straits (e.g. the Straits of Messina, between Sicily and peninsular Italy) with the related road and train traffic problems. Up to now, the straits can only be crossed by ferry-boats. Many plans have been studied so far, involving the construction of a bridge or a submarine tunnel. We may start with a GIS application describing the current Straits of Messina data, modelled through a single schema which does not provide support for bridges or submarine tunnels. If our system allows generalized schema versioning, the geographic and travel impact of the realization of both plans can be evaluated and compared by analysing two scenarios based on alternative consolidated versions, the former including bridges and the latter submarine tunnels. Afterwards, each new schema version may evolve independently from the other, as a consequence of the application of changes acting on the schema version features, so that independent simulations can be effected on the two solutions (e.g. to analyse traffic evolutions). In the end, a plan will be realized and all previous solutions will converge into a single consolidated version, representing the final outcome of the planning process.

In this case, the course of the structural changes undergoes a bifurcation in order to support the comparison between different solutions. A "root" scenario (called $straits$) maintains all the temporal versions, with their corresponding temporal pertinences, describing the Straits of Messina data available up to now. The bifurcation is implemented by the **NewNode** primitive which creates new scenarios. For example,

$$\mathcal{DBU}(\textbf{NewNode})$$
$$(\mathcal{DBG}, bridge, null, null, null, null)$$

creates a new scenario called $bridge$, which can be initially

populated with a temporal version copied from $straits$ (selected as the current present temporal version) as follows:

$$\mathcal{DBU}(\textbf{MergeVersion})$$
$$(\mathcal{DBG}, bridge, null, straits, now, [now, \infty])$$

Its validity is set to $[now, \infty]$. In the same way, another scenario, called $tunnel$, can also be introduced. Afterwards, each new scenario may evolve independently. For example,

$$\mathcal{DBU}(\textbf{AddClass}_{Bridge})$$
$$(\mathcal{DBG}, bridge, now, null, null, [2010, \infty])$$

adds a new temporal version with validity $[2010, \infty]$ in the $bridge$ scenario by adding the class $Bridge$ to the temporal version which is current and valid now. In the end, the final scenario can be created and then populated by making a copy of one temporal version of the selected $bridge$ alternative:

$$\mathcal{DBU}(\textbf{MergeVersion})$$
$$(\mathcal{DBG}, final, now, bridge, now, [2010, \infty])$$

Some characteristics of temporal versions of the other alternative scenario can be inherited by means of the pick-type schema changes. For instance, we may want to integrate into the final scenario an optic-fiber cable telecommunication connection intially included in the tunnel scenario only. This can be done via the following operation:

$$\mathcal{DBU}(\textbf{PickClass}_{Cable})$$
$$(\mathcal{DBG}, final, now, tunnel, 2000, [2010, \infty])$$

which integrates, into the current temporal version valid now and belonging to the $final$ scenario, the populated class $Cable$ belonging to the current temporal version in the $tunnel$ scenario valid in 2000. The outcome is a new temporal version included in the $final$ scenario valid in the interval $[2010, \infty]$. Moreover, a derivation edge from node $tunnel$ to node $final$ is automatically added to the database DAG.

6 Conclusions and Future work

We have proposed a generalized schema versioning model which integrates the temporal schema versioning with the branching versioning approach. A complete set of schema changes for the manipulation of versions is also defined. In this way, our model provides effective facilities for introducing and organizing versions for all the application requirements indicated in the literature and for the development of new advanced applications, by improving the versioning support of an object-oriented database.

In our future work, we will further investigate the properties of our model, also taking into account semantic and implementation issues. We also plan to extend it by considering other versioning dimensions (e.g. spatial coordinates [RGMS99]).

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, MA, 1995.

[BKKK87] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM-SIGMOD Annual Conference*, pages 311–322, San Francisco, CA, May 1987.

[BM98] P. A. Burrough and R. A. McDonnell. *Principles of Geographical Information Systems*. Oxford University Press, New York, NY, 1998.

[BN94] S. Bergamaschi and B. Nebel. Automatic Building and Validation of Multiple Inheritance Complex Object Database Schemata. *International Journal of Applied Intelligence*, 4(2):185–204, 1994.

[Bre90] Y. Breitbart. Multidatabase Interoperability. *ACM SIGMOD Record*, 19(3):53–60, 1990.

[CBB+97] R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Ware, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Francisco, CA, 1997.

[CJK91] W. Cellary, G. Jomier, and T. Koszlajda. Formal Model of an Object-Oriented Database with Versioned Objects and Schema. In *Proc. of the 2nd Int'l Conf. on Database and Expert Systems Applications (DEXA)*, pages 239–244, Berlin, Germany, 1991.

[DGS97] C. De Castro, F. Grandi, and M. R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.

[DL88] K. R. Dittrich and R. A. Lorie. Version Support for Engineering Database Systems. *IEEE Transactions on Software Engineering*, 14(4):429–436, 1988.

[GMS98] F. Grandi, F. Mandreoli, and M. R. Scalas. A Formal Model for Temporal Schema Versioning in Object-Oriented Databases. Technical Report CSITE-014-98, CSITE - CNR, November 1998. Available on ftp://csite60.deis.unibo.it/pub/report.

[GSÖP98] I. A. Goralwalla, D. Szafron, M. T. Özsu, and R. J. Peters. A Temporal Approach to Managing Schema Evolution in Object Database Systems. *Data & Knowledge Engineering*, 28(1):73–105, 1998.

[JCG+98] C. S. Jensen, J. Clifford, S. K. Gadia, P. Hayes, and S. Jajodia et al. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases - Research and Practice*, pages 367–405. Springer-Verlag, 1998. LNCS No. 1399.

[Kat90] R. H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, 1990.

[KB96] S. Khoshafian and A. B. Baker. *MultiMedia and Imaging Databases*. Morgan Kaufmann, San Francisco, CA, 1996.

[KC88] W. Kim and H.-T. Chou. Versions of Schema for Object-Oriented Databases. In *Proc. of the 14th Int'l Conf. on Very Large Databases (VLDB)*, pages 148–159, Los Angeles, CA, August 1988.

[KL84] R. H. Katz and T. J. Lehman. Database Support for Versions and Alternative of Large Design Files. *IEEE Transactions on Software Engineering*, 10(2):191–200, 1984.

[Lan86] G. S. Landis. Design Evolution and History in an Object-Oriented CAD/CAM Database. In *Proc. of 31st COMPCON Conference*, San Francisco, CA, March 1986.

[Lau97] S.-E. Lautemann. Schema Versioning in Object-Oriented Database Systems. In *Proc. of the 5th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 323–332, Melbourne, Australia, April 1997.

[Odb96] E. Odberg. Database Schema Evolution. World Wide Web Page, URL: `http://home.sol.no/~eodberg/smm.html`, 1996.

[ÖS95] G. Özsoyoğlu and R. T. Snodgrass. Temporal and Real Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[PÖ97] R. J. Peters and M. T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transaction on Database Systems*, 22(1):75–114, 1997.

[RGMS99] J. F. Roddick, F. Grandi, F. Mandreoli, and M. R. Scalas. Towards a Model for Spatio-Temporal Schema Selection. In *Proc. of the DEXA'99 STDML Workshop*, Florence, Italy, August 1999.

[Rod96] J. F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1996.

[SR99] N. L. Sarda and P. V. Siva Prasada Reddy. Handling of Alternatives and Events in Temporal Databases. *Knowledge and Information Systems*, 1(3):193–227, 1999.

[TS93] C. Thieme and A. Siebes. Schema Integration in Object-Oriented Databases. In *Proc. of the 8th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 54–70, Paris, France, June 1993.