Dynamic Multi-version Ontology-based Personalization

Fabio Grandi
DISI, University of Bologna
Viale Risorgimento, 2
I-40136, Bologna BO - Italy
+39 051 2093555
fabio.grandi@unibo.it

ABSTRACT

In this paper, we describe a storage scheme that allows the representation and management of the evolving hierarchical structure of a multi-version ontology. The proposed scheme is aimed at supporting ontology-based personalization and temporal access to resources (data, documents, etc.) stored in a dynamic environment.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and networks; H.2.4 [**Database Management**]: Systems - query processing; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing - indexing methods.

General Terms

Algorithms, Management, Design.

Keywords

Ontology, trees, personalization, temporal database, versioning

1. INTRODUCTION

The adoption of reference ontologies and their deployment for the personalization of multi-version resources has been recently proposed by several authors in the medical domain [1,2,3] and other application fields (e.g., e-Government [4]). The considered resources range from descriptive data to textual documents, from Web pages to the specification of processes. References to ontology classes are added to the computer encoding of resources (e.g., for which an XML [5] format can conveniently be used) to introduce a sort of semantic indexing of contents representing their applicability, relevance or eligibility with respect to ontology classes. Hence, starting from a user-supplied list of ontology classes, a suitable query engine can exploit semantic indexing to retrieve the relevant contents only and produce a personalized version of the desired resources.

However, in a dynamic environment, the management of this kind of semantic versioning is interleaved with temporal aspects. For example, we can choose as resources *clinical guidelines* [6], that is "best practices" encoding and standardizing health care procedures, in textual or executable format, and consider their personalization with respect to an ontology of diseases, patients or available hospital facilities they are applicable to [1]. Personalization will produce a guideline version tailored to a specific use case. The fast evolution of medical knowledge and the dynamics involved in clinical practice imply the coexistence of multiple temporal versions of the clinical guidelines stored in a repository, which are continually subject to amendments and modifications. Therefore, it is crucial to reconstruct the *consolidated version* of a guideline as produced by the application of all the modifications it underwent so far, that is the form in which it currently belongs to the state-of-the-art of clinical practice and, thus, must be applied to patients today. However, also past versions are still important, not only for historical reasons: for example, a physician might be called upon to justify his/her actions for a given patient at a past time on the basis of the clinical guideline versions applicable to the pathology of patient and which were valid at that time.

Moreover, in a dynamic environment, the definition of domain ontologies themselves is also subject to modification and, thus, ontologies come out versioned as a consequence of updates periodically effected by domain experts and knowledge engineers or even standardization committees. As we showed in [7] for the legal domain (but it also happens for the medical one), personalization of a resource with respect to a past point in time must be effected by taking into account, in order to consider semantic indexing of the desired temporal version of the resource, the version of the reference ontology which was valid at the same time point. In other words, the selected resource version and the ontology version used for personalization must be mutually *temporally consistent*. Since clinical guidelines have also been recently proposed to be used as evidence of the legal standard of care in medical malpractice litigation [8], enforcement of temporal consistency is crucial to assess the responsibility of physicians having followed the guidelines in the past.

Therefore, in this work we will show how temporal multi-version ontologies can be represented and maintained in a relational setting and how they can be used during the processing of a personalization query. The rest of the paper is organized as follows: in Sec. 2, the ontology-based personalization method proposed in [1,4] is briefly recalled; in Sec. 3, we present our storage scheme and manipulation primitives for temporal ontology versioning; Section 4 is devoted to personalization query processing in the presence of a multi-version ontology. Conclusions can finally be found in Sec. 5.

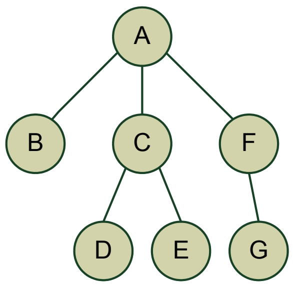
2. A FRAMEWORK FOR ONTOLOGY-BASED PERSONALIZATION

The personalization method proposed in [1,4] is based on the adoption of reference domain ontologies and the introduction of semantic indexing of resource contents with respect to ontology classes. For example, in the medical domain, reference ontologies to be used to this purpose can be derived from the ICD-10[1] international classification of diseases or from the SNOMED-CT[2] comprehensive healthcare terminologies. Semantic indexing can then be used by personalization services to adapt generic resources to specific use cases, for example, to derive and enact individual care plans as proposed in [1,2,3].

The main ontology feature which is relevant for our personalization approach is the hierarchy of classes (taxonomy) induced by the IS-A relationship. Hence, we do not consider properties or other features and also follow the simplified assumption made in [1,4] that the class hierarchy underlying the ontology is tree-shaped, that is each node in the class hierarchy (but the root) has a single parent. Owing to the tree structure, nodes can be assigned a preorder and a postorder code, corresponding to the sequence in which nodes are visited during a preorder or postorder traversal of the tree, respectively. Preorder and postorder codes can be used for efficiently characterizing the descendants of a node [9,10]:

N is a descendant of M iff $M.Pre < N.Pre$ and $N.Post < M.Post$

with obvious meaning of the used dotted notation. For example, we can consider the sample ontology depicted in the left part of Fig.1, where the corresponding preorder, postorder and level code of nodes can be found in the table to the right. Level is the distance from the top, assuming level 1 for the root node. The structure of the class hierarchy is completely defined by the information present in the table (actually, level values are not necessary, but will be used for speeding up query processing as described in Sec. 4), which, thus, can enable storage of the ontology definition in a relational table.



Id	Pre	Post	Lev
A	1	7	1
B	2	1	2
C	3	4	2
D	4	2	3
E	5	3	3
F	6	6	2
G	7	5	3

Figure 1. A sample ontology and its tabular representation.

Once defined and stored the ontology in this way, node identifiers can be used as a reference to ontology classes for semantic indexing of the resources which are the object of personalization. In [1,4], preorder codes are directly used as node identifiers, whereas we will keep them distinguished and associate preorder, postorder and level codes to time-invariant class identifiers (Id) in order to support ontology versioning. Hence, if the same class belongs to two ontology versions, the class Id is the same in both of them, while the preorder (postorder and level) code is very likely different as long as the two ontology versions have a different structure. In this way, the proposed encoding scheme implies an *indirect reference* from class identifiers used for semantic indexing of resources and preorder and postorder codes used for query processing.

```
…
<foo>
<version number=”1”>
  <pertinence>
    <valid from=”T1” to=”T2”/>
    <applies to=”B”>
  </pertinence>
       Contents of foo–version 1
</version>
<version number=”2”>
  <pertinence>
    <valid from=”T2” to=”UC”/>
    <applies to=”C”>
  </pertinence>
       Contents of foo–version 2
  <bar>
  <version number=”1”>
    <pertinence>
      <applies also=”G”>
    </pertinence>
       Contents of  bar–version 1
  </version>
  </bar>
</version>
</foo>
…
```

Figure 2. A chunk of multi-version XML resource.

In this work, like in [1,4], we consider personalization of resources with an inner hierarchical organization (e.g., a text organized with chapters, sections, subsections and paragraphs), which, thus, can be represented and stored as XML documents. Each element within the resource can be represented by means of multiple versions of its contents, each of which can be assigned a temporal validity (by means of timestamps) and a semantic pertinence (by means of references to ontology classes). Owing to the hierarchical organization of resources, temporal validity and semantic applicability properties of an element are inherited by its subelements, unless locally redefined. Considering applicability properties, because of the IS-A semantics (e.g., an individual which is instance of C is also instance of A in Fig.1), if we are looking for all the resource portions that qualify for an instance of an ontology class, we should retrieve the resource portions which are directly applicable to the ontology class itself and also the resource portions which are applicable to its superclasses. For example, if a query retrieves resources concerning an individual belonging to the ontology class C, then the returned resources should be those applicable to class C but also those applicable to the ancestor classes of C (i.e., class A in Fig. 1). Whichever is the most specific class to which our individual of interest belongs, the query results would include all the resource portions applicable to all its ancestor classes up to the ontology root class, which may come out too generic to be of real interest for a specific use case. For instance, considering an ontology of diseases in the medical domain, this would mean to also retrieve all resources generically applicable to “patients” when looking for resources concerning an individual affected by “microvascular angina”. In such a case, as proposed in [1,4], using a optional *depth* parameter in order to focus on the most interesting resources only, the user can limit the applicability scope of a query to the ancestors located up to *depth* steps above the most specific class our individual of interest

[1] http://www.who.int/classifications/icd/en/

[2] http://www.ihtsdo.org/snomed-ct/

belongs in the class hierarchy (in order to easily find them, the level codes can be used, as will be shown in Sec. 4).

For instance, let us consider the sample chunk in Fig. 2 of a multi-version resource encoded in XML. It is made of an element “foo” with two versions, the former (version 1) valid from T1 to T2 and applicable to class B of the ontology in Fig. 1 and the latter (version 2) valid from T2 on and applicable to class C of the ontology in Fig. 1. The special time value UC (Until Changed [11]) is used to represent the To value of a right-unlimited time interval. The second version of element “foo” contains a subelement “bar”, which inherits the validity of its parent element (from T2 on) and extends the applicability inherited from its parent also to class G of the ontology in Fig. 1 (i.e., the applicability of “bar” is C or G). The only version (version 1) defined for “bar” is necessary in order to redefine the inherited semantic pertinence (notice that the “pertinence” XML element is defined as a subelement of the “version” XML element).

The XML encoding of multi-version resources exemplified in Fig. 2, which has been proposed in [1,4], has been adopted in this work for the reasons which follow:

- is general enough to be applied to any kind of resources (as is independent on the non versioned resource schema) and to allow the seamless adoption of an arbitrary number of temporal and semantic versioning dimensions;
- its simplicity allows a self-contained presentation;
- efficient algorithms implemented in a prototype processor are available for personalization query support [1,4] (required extensions to this approach will be presented in Sec. 4).

However, as far as semantic markup is concerned, other encoding schemes proposed for linking resource contents to ontological information, from the ones proposed as standards like RDFa[3] and microformats[4] to the more exotic ones customary in specific application domains (e.g., medical domain), could also be adopted. In such a case, simple modifications, which are beyond the scope of this work, have to be introduced to the personalization query processing methods presented in Sec. 4.

3. TEMPORAL ONTOLOGY VERSIONING

In this section, we will introduce primitive operations for applying ontology changes to produce a new version, and show how they can be defined in order to maintain a multi-version ontology represented and stored as a valid-time relation in a temporal database [11].

3.1 Ontology Evolution Support

In this Subsection, we will show how the evolution of an ontology with a tree-shaped class hierarchy in tabular representation can be supported. To this purpose, we introduce three primitive change operations, which can be used in sequence and combination to make arbitrary changes to the ontology structure, and present algorithms to implement their action on the tabular representation exemplified in Fig. 1.

[3] http://rdfa.info/

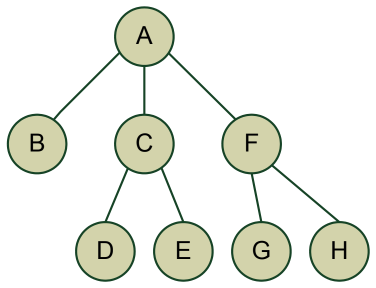
[4] http://microformats.org/

3.1.1 Insertion of a leaf node

The operation **InsertUnder(N)** can be used to create a new leaf node as child of the existing node N. If the node N already has children, the new node is created as the rightmost child (the order of siblings does not matter for personalization query processing, as the ancestor-descendant relationships only are relevant). Owing to the definition of preorder, postorder and level codes, the action of the insertion reflects on their values as explained in the following. All the nodes which were visited after N in postorder and N itself must have their postorder code increased by 1 (as they will be visited after the new node). Notice that, being created as the rightmost child of N, the new node will be visited in preorder right after all the nodes in the subtree rooted on N (which satisfy the descendant relations Pre>N.Pre and Post<N.Post). Hence, the nodes which must have their preorder code increased by 1 are all the nodes which were visited after N both in preorder and in postorder (i.e., nodes visited after N in preorder but not belonging to the subtree rooted on N). The new node must be assigned a preorder code equal to the maximum preorder code found in the subtree rooted on N plus 1, inherits the postorder code from N and has a level equal to the level of N plus 1. A slightly optimized algorithm for updating the tabular representation is the following:

```
InsertUnder(N:NodeRow)
   MaxPreSub:=N.Pre;
   ForEach Node in TreeTable Do
      If Node.Post>=N.Post
      Then Node.Post++
           If Node.Pre>N.Pre
           Then Node.Pre++ EndIf
      ElseIf Node.Pre>N.Pre
             and Node.Pre>MaxPreSub
      Then MaxPreSub:=Node.Pre
      EndIf
    EndFor
    AddRow(NewId(),MaxPreSub+1,N.Post,N.Lev+1)
Return
```

The function NewId() is assumed to create an unused identifier, which acts like a time-invariant key, for the newly added node. For instance, starting from the ontology in Fig. 1, the execution of the operation **InsertUnder(F)** produces the new ontology version shown in Fig. 3 with the new node created as H.



Id	Pre	Post	Lev
A	1	8	1
B	2	1	2
C	3	4	2
D	4	2	3
E	5	3	3
F	6	7	2
G	7	5	3
H	8	6	3

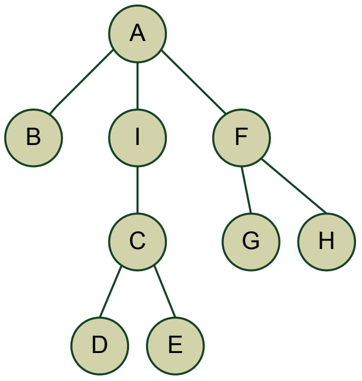
Figure 3. Second version of the ontology in Fig.1.

3.1.2 Insertion of an intermediate node

The operation **InsertOver(N)** can be used to create a new node in the path between the node N and its parent (i.e., the new node becomes the new parent of N and a child of the former parent of N). If N is the tree root node, the created node will become the new root. The action of the insertion reflects on the preoder, postorder and level values as explained in the following. All the nodes which were visited after N in preorder and N itself must have their preorder code increased by 1 (as they will be visited after the new node and no other descendant of N can be visited before). All the nodes which were visited after N in postorder must have their postorder code increased by 1 (as they will be visited after the new node). All the nodes which were in the subtree rooted on N (inclusive) must have their level increased by 1. The new node inherits the preorder code and the level from N and must be assigned a postorder code equal to the postorder code of N plus 1. A slightly optimized algorithm for accordingly updating the tabular representation is the following:

```
InsertOver(N:NodeRow)
   ForEach Node in TreeTable Do
      If Node.Pre>=N.Pre
      Then Node.Pre++
           If Node.Post<=N.Post
           Then Node.Lev++ EndIf
      EndIf
      If Node.Post>N.Post Then Node.Post++ EndIf
    EndFor
    AddRow(NewId(),N.Pre,N.Post+1,N.Lev)
Return
```

For instance, starting from the ontology in Fig. 3, the execution of the operation **InsertOver(C)** produces the new ontology version shown in Fig. 4 with the new node created as I.



Id	Pre	Post	Lev
A	1	9	1
B	2	1	2
C	4	4	3
D	5	2	4
E	6	3	4
F	7	8	2
G	8	6	3
H	9	7	3
I	3	5	2

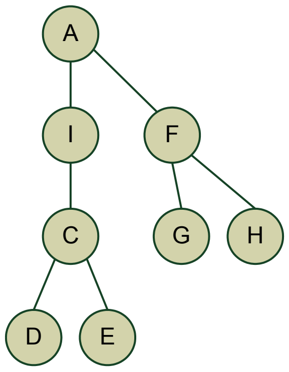
Figure 4. Third version of the ontology in Fig.1.

3.1.3 Deletion of a node

The operation **DeleteNode(N)** can be used to delete node N from the ontology (former children of N become children of the former parent of N). The DeleteNode procedure can be applied to the tree root node only if it has a single child (which becomes the new root). All the nodes which were visited after N in preorder (or postorder) must have their preorder (or postorder) code decreased by 1 (as they will be reached in both visit orders one step earlier). All the nodes which were in the subtree rooted on N must have their level decreased by 1. A slightly optimized algorithm for updating the tabular representation is the following:

```
DeleteNode(N:NodeRow)
   ForEach Node in TreeTable Do
      If Node.Pre>N.Pre Then Node.Pre--
         If Node.Post<N.Post
         Then Node.Lev-- EndIf
      EndIf
      If Node.Post>N.Post Then Node.Post-- EndIf
   EndFor
   DeleteRow(N.Id,N.Pre,N.Post,N.Lev)
Return
```

For instance, starting from the ontology in Fig. 4, the execution of the operation **DeleteNode(B)** produces the new ontology version shown in Fig. 5.



Id	Pre	Post	Lev
A	1	8	1
C	3	3	3
D	4	1	4
E	5	2	4
F	6	7	2
G	7	5	3
H	8	6	3
I	2	4	2

Figure 5. Fourth version of the ontology in Fig.1.

3.2 Use of a Temporal Relation for Storing a Multi-version Ontology

In this Subsection, we show how the whole evolution of a tree-shaped ontology can be represented and maintained as a temporal relation storing all the time-stamped ontology versions. In this work, we assume valid time [11] is used as time dimension, which allows ontology designers to also apply retro- and pro-active modifications. However, the adoption of transaction time [11] (in a transaction-time or bitemporal relation) would require simple modifications to the proposed management. Hence, a multi-version ontology can be stored in a temporal relation with schema:

TreeRelation(Id, Pre, Post, Lev, From, To)

where tuples like the ones considered in Sec. 3.1 are augmented with the timestamping attributes From and To, representing the boundaries of a right-open time interval [From,To). Such relation can be stored in a relational database and manipulated via SQL statements.

Before applying any other modification, an empty TreeRelation temporal table must be initialized via the root creation by means of a call to the following procedure:

```
CreateRoot(T:TimePoint)
   { INSERT INTO TreeRelation
     VALUES (NewId(),1,1,1,T,'UC') }
Return
```

The algorithms presented in Sec. 3.1 for maintenance of ontologies in their tabular representation translate into the procedures which are listed in the rest of this subsection, where embedded SQL statements are also used to manage the ontology stored in the TreeRelation temporal table. For example, we will make use of a kind of snapshot query [11]:

```
   SELECT * INTO TreeCursor
   FROM TreeRelation WHERE To='UC'
```

which extracts the current snapshot from the temporal relation TreeRelation to select all the tuples belonging to the ontology consolidated version, which are then accessed through a cursor TreeCursor in main memory, for further processing by the procedure one tuple at a time (within the ForEach loop).

The three procedures corresponding to the ontology maintenance algorithms in Sec. 3.1 are listed in the following (using pseudo-code with embedded SQL statements). Procedures have a second argument, T, representing the validity start of the modification For the insertion of a new leaf node, the algorithm presented in Sec. 3.1.1 becomes as follows:

```
InsertUnder(N:NodeTuple,T:TimePoint)
  MaxPreSub:=N.Pre;
  { SELECT * INTO TreeCursor
    FROM TreeRelation WHERE To='UC' }
  ForEach Node in TreeCursor Do
     New.Pre:=Node.Pre;
     New.Post:=Node.Post;
     If Node.Post>=N.Post
     Then Node.To:=T; New.Post++
          If Node.Pre>N.Pre Then New.Pre++ EndIf
     ElseIf Node.Pre>N.Pre
            and Node.Pre>MaxPreSub
     Then MaxPreSub:=Node.Pre
     EndIf
     If Node.To=T
     Then { UPDATE TreeRelation
            SET Pre=Node.Pre,Post=Node.Post
                Lev=Node.Lev,To=T
            WHERE Id=Node.Id and From=Node.From;
            INSERT INTO TreeRelation
            VALUES (Node.Id,New.Pre,New.Post,
                    Node.Lev,T,'UC') }
  EndFor
  { INSERT INTO TreeRelation
    VALUES (NewId(),MaxPreSub+1,N.Post,
            N.Lev+1,T,'UC') } EndIf
Return
```

For the insertion of an intermediate node within the class hierarchy, the procedure introduced in Sec. 3.1.2 becomes:

```
InsertOver(N:NodeTuple,T:TimePoint)
  { SELECT * INTO TreeCursor
    FROM TreeRelation WHERE To='UC' }
  ForEach Node in TreeCursor Do
     New.Pre:=Node.Pre;
     New.Post:=Node.Post;
     New.Lev:=Node.Lev;
     If Node.Pre>=N.Pre
     Then Node.To:=T; Node.Pre++
       If Node.Post<=N.Post Then Node.Lev++ EndIf
     EndIf
     If Node.Post>N.Post
     Then Node.To:=T; Node.Post++
     EndIf
     If Node.To=T
     Then { UPDATE TreeRelation

            SET Pre=Node.Pre,Post=Node.Post
                Lev=Node.Lev,To=T
            WHERE Id=Node.Id and From=Node.From;
            INSERT INTO TreeRelation
            VALUES (Node.Id,New.Pre,New.Post,
                    New.Lev,T,'UC') } EndIf
  EndFor
  { INSERT INTO TreeRelation
    VALUES (NewId(),N.Pre,N.Post+1,
            N.Lev,T,'UC') }
Return
```

Finally, the procedure deriving from the algorithm in Sec. 3.1.3 to be used for the deletion of a node is as follows:

```
DeleteNode(N:NodeTuple,T:TimePoint)
  { SELECT * INTO TreeCursor
    FROM TreeRelation WHERE To='UC' }
  ForEach Node in TreeCursor Do
     New.Pre:=Node.Pre;
     New.Post:=Node.Post;
     New.Lev:=Node.Lev;
     If Node.Pre>N.Pre
       Then Node.To:=T;New.Pre--
       If Node.Post<N.Post Then New.Lev-- EndIf
     EndIf
     If Node.Post>N.Post
       Then Node.To=T;New.Post--
     EndIf
     If Node.To=T
     Then { UPDATE TreeRelation
            SET Pre=Node.Pre,Post=Node.Post
                Lev=Node.Lev,To=T
            WHERE Id=Node.Id and From=Node.From;
            INSERT INTO TreeRelation
            VALUES (Node.Id,New.Pre,New.Post,
                    Node.Lev,T,'UC') } EndIf
  EndFor
Return
```

For the sake of simplicity, in writing the code, we assumed so far that only one ontology version be affected by the modification (i.e., the one with To equal to UC, which is part of the consolidated version valid at present time, further assuming that no versions with From>Now are currently stored in the TreeRelation temporal table).

Otherwise, if more than one version can be affected, the SQL SELECT which loads TreeCursor at the beginning of the three procedures must be replaced by the SQL statements which follow:

```
  DELETE FROM TreeRelation WHERE From>=T;
  SELECT * INTO TreeCursor
  FROM TreeRelation WHERE From<=T AND T<To
```

In fact, the creation of a new version valid from T involves all the tuples whose timestamp is totally or partially overlapped by the validity of the modification [T,UC). In order to clarify what happens when a modification is applied to the history of an object in the most general case, we can consider the graphical example shown in Fig. 6.

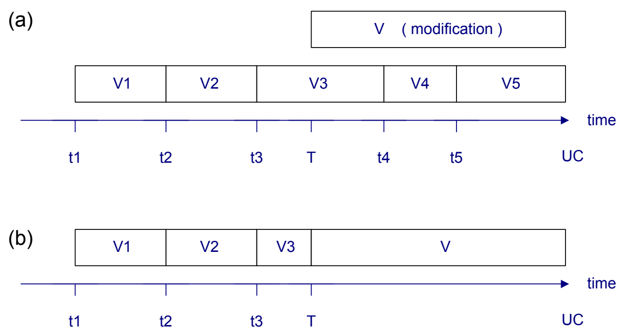


Figure 6. Effects of a modification in the most general case.

In particular, Fig. 6(a) displays the history of an object composed of five versions (i.e., Vi valid from ti to ti+1, i=1..4, and V5 valid from t5) and the placement on the time axis of a modification which must be applied to the history to produce a new version with contents V and validity from T. As shown in Fig. 6(b) presenting the history of the object after the application of the modification, the modification left versions V1 and V2 untouched, completely overlapped versions V4 and V5 (which have been removed) and partially overlapped version V3, whose validity has, thus, been restricted to [t3,T). After the deletion of the completely overlapped versions, the version affected by the modification is the one whose timestamp contains the validity start of the modification (i.e., V3 in Fig. 6(a), as t3≤T<t4).

Hence, the two SQL statements listed above accomplish the tasks, respectively, of deleting all the completely overlapped versions and of putting all the partially overlapped versions into TreeCursor for further processing by the procedures.

Table 1. Temporal relation storing the first ontology version

Id	Pre	Post	Lev	From	To
A	1	7	1	T0	UC
B	2	1	2	T0	UC
C	3	4	2	T0	UC
D	4	2	3	T0	UC
E	5	3	3	T0	UC
F	6	6	2	T0	UC
G	7	5	3	T0	UC

Coming back to the running example introduced in Sec. 3.1, we can easily store the first ontology version depicted in Fig. 1, which has been created with validity starting at time T0, in the temporal relation displayed in Table 1. Then we can consider the application of the following sequence of modifications which correspond to the ontology updates exemplified in Sec. 3.1:

InsertUnder(F,T1);
InsertOver(C,T2);
DeleteNode(B,T3);

For instance, Table 2 shows the contents of the TreeRelation temporal table after the execution of **InsertUnder(F,T1)** on the initial state in Table 1. Notice how nodes A and F are represented through two tuples each, representing their versions belonging to the two ontology versions, respectively (e.g., the first version of A with preorder 1, postorder 7, level 1 and validity [T0,T1) belongs to the first ontology version, whereas the second version of A with postorder changed to 8 and validity [T1,UC) belongs to the second ontology version). Nodes represented through a single tuple (e.g., B) have a single version with validity [T0,UC) shared by both ontology versions.

Table 2. TreeRelation after the creation of leaf node H (it contains the first and second ontology versions)

Id	Pre	Post	Lev	From	To
A	1	7	1	T0	T1
B	2	1	2	T0	UC
C	3	4	2	T0	UC
D	4	2	3	T0	UC
E	5	3	3	T0	UC
F	6	6	2	T0	T1
G	7	5	3	T0	UC
A	1	8	1	T1	UC
F	6	7	2	T1	UC
H	8	6	3	T1	UC

Obviously, the newly created node (H) has a single version with validity [T1,UC) belonging to the second ontology version only.

After the execution of **InsertOver(C,T2)**, the contents of TreeRelation now containing the first three ontology versions are as displayed in Table 3.

Table 3. TreeRelation after the insertion of intermediate node I (it contains the first three ontology versions)

Id	Pre	Post	Lev	From	To
A	1	7	1	T0	T1
B	2	1	2	T0	UC
C	3	4	2	T0	T2
D	4	2	3	T0	T2
E	5	3	3	T0	T2
F	6	6	2	T0	T1
G	7	5	3	T0	T2
A	1	8	1	T1	T2
F	6	7	2	T1	T2
H	8	6	3	T1	T2
A	1	9	1	T2	UC
C	4	4	3	T2	UC
D	5	2	4	T2	UC
E	6	3	4	T2	UC
F	7	8	2	T2	UC
G	8	6	3	T2	UC
H	9	7	3	T2	UC
I	3	5	2	T2	UC

After the execution of **DeleteNode(B,T3),** the final outcome is the temporal relation displayed in Table 4, fully exemplifying the storage of our multi-version ontology in a single temporal relation.

Table 4. TreeRelation storing the multi-version ontology after deletion of B (it contains all the four ontology versions)

Id	Pre	Post	Lev	From	To
A	1	7	1	T0	T1
B	2	1	2	T0	T3
C	3	4	2	T0	T2
D	4	2	3	T0	T2
E	5	3	3	T0	T2
F	6	6	2	T0	T1
G	7	5	3	T0	T2
A	1	8	1	T1	T2
F	6	7	2	T1	T2
H	8	6	3	T1	T2
A	1	9	1	T2	T3
C	4	4	3	T2	T3
D	5	2	4	T2	T3
E	6	3	4	T2	T3
F	7	8	2	T2	T3
G	8	6	3	T2	T3
H	9	7	3	T2	T3
I	3	5	2	T2	T3
A	1	8	1	T3	UC
C	3	3	3	T3	UC
D	4	1	4	T3	UC
E	5	2	4	T3	UC
F	6	7	2	T3	UC
G	7	5	3	T3	UC
H	8	6	3	T3	UC
I	2	4	2	T3	UC

Considering then the execution of a classical snapshot query [11] (retrieving the snapshot valid at time T):

```
SELECT Id,Pre,Post,Lev FROM TreeRelation
WHERE From<=T AND T<To
```

over the temporal relation in Table 4, we can notice that the retrieved snapshot: if T∈ [T0,T1), coincides with the table in Fig. 1; if T∈ [T1,T2), coincides with the table in Fig. 3; if T∈ [T2,T3), coincides with the table in Fig. 4; and if T>T3, the retrieved snapshot coincides with the table in Fig. 5. This clearly confirms how the temporal relation in Table 2 is actually a comprehensive representation and a suitable storage scheme for a multi-version ontology.

As far as indexing of multi-version resources by means of references to ontology classes are concerned, we can underline the fact that the solution —used for the sake of simplicity in [1,4]— based on the bookkeeping of a single ontology version (i.e., the consolidated version) to index all resource versions *is very inefficient from a practical point of view*, besides being simplistic and rather incorrect from a semantic and application requirement point of view [7]. As a matter of fact, a reference ontology might have to be used to index a very large repository of multi-version resources in a realistic environment. Hence, when even small changes (e.g., addition or deletion of a single class) are applied to an ontology in this scenario, where preorder codes are used as class identifiers, a large number of classes in the ontology may have their identifiers changed as a consequence of the update. Thus, such a change in class identifiers has to be propagated to all resources in order to preserve the correct semantic indexing, which would require to access and rewrite a large fraction of the whole resource repository to update class identifiers. With the *indirect reference* solution proposed in this work, ontology changes only affect the corresponding TreeRelation table and do not require any changes to be applied the resource repository.

4. QUERY PROCESSING WITH MULTI-VERSION ONTOLOGIES

The semantic indexing of resources which links their contents to reference ontologies is designed to support personalization queries [1,4]. To this purpose, and in order to show how query processing works in the presence of multi-version ontology, we consider the XQuery-like query template which follows:

```
FOR $x IN resources.xml
WHERE TEXT_CONSTRAINT($x,CC)
  AND VALID($x,T) AND APPLICABLE($x,Cx:depth)
RETURN $x
```

which is a simplified form of the template introduced in [1,4] and for which a query engine has been implemented in a prototype system. The function `TEXT_CONSTRAINT()` applies textual constraints to the contents of the resources to be retrieved. Textual constraints can include both structural and lexical constraints, being expressible as an XPath expression [12] which can be used for matching keywords within the resource structured contents. The function `VALID()` effects a temporal snapshot of the resources by selecting the content versions valid at time "T". The function `APPLICABLE()` effects a semantic slicing of the resources by selecting the content versions which are applicable to instances of ontology class "Cx" and of its ancestors up to "depth" levels (in [1,4], the expression "Cx:depth" is called a *navigational pattern* with respect to the reference ontology). In the presence of multi-version ontologies, the first step in query processing is the determination of the ontology classes denoted by the navigational pattern "Cx:depth" and of the preorder and postorder codes of such classes. This information can be retrieved from the TreeRelation temporal table which stores the encoding of the multi-version ontology. SQL queries similar to the ones which follow can be used to this purpose:

```
SELECT * INTO CX FROM TreeRelation AS Node
WHERE Node.From<=T AND T<=Node.To
  AND Node.Id=Cx;

SELECT * INTO CY FROM TreeRelation AS Node
WHERE Node.From<=T AND T<=Node.To
  AND Node.Pre<CX.Pre AND Node.Post>CX.Post
  AND CX.Lev-Node.Lev=depth
```

The first query retrieves the data of the class CX whose identifier is "Cx" in the ontology version valid at time "T". Then, making use of the level information associated to nodes, the second query retrieves, in the ontology version valid at time "T", the ancestor CY of the class CX, that can be reached in "depth" steps starting from CX. The two queries return two ontology classes which must be used, in the second query processing step, to select the qualifying resource contents through their preorder and postorder codes. In particular, a resource version qualifies if its semantic pertinence implies the query navigational pattern [1,3,4]. Thanks to the properties of the preorder/postorder encoding, this notion of implication translates into verifying whether at least one of the ontology classes which make up the semantic pertinence of the resource is contained in a rectangular region defined in the preorder/postorder plane by the navigational pattern [1,4]. Such rectangular region, in which all and only the nodes in the inheritance path from CY to CX fall, can be determined as the Cartesian product [CY.Pre,CX.Pre] x [CX.Post,CY.Post] (i.e., the lower right corner of the rectangle is CX, whereas the upper left corner is CY). Owing to the fact that preorder, postorder and level codes associated to the same classes are different in different ontology versions, we will have a different containment relationship to be checked for each ontology version.

For example, let us consider the multi-version ontology stored in our sample TreeRelation displayed in Table 4 and the query navigational pattern "D:2". Depending on the time "T" of interest, the CX and CY values retrieved by the above two queries navigating the ontology will be as summarized in the Table which follows:

Table 5. Evaluation of the navigational pattern "D:2" in different versions of the ontology of Tab. 2

	CX			CY		
Time	Id	Pre	Post	Id	Pre	Post
[T0,T1)	D	4	2	A	1	7
[T1,T2)	D	4	2	A	1	8
[T2,T3)	D	5	2	I	3	5
[T3,UC)	D	4	1	I	2	4

Let us further consider the resource chunk in Fig. 2. The element foo(v1) is applicable to class B in [T1,T2); the element foo(v2) is applicable to class C in [T2,UC); the element foo/bar(v1) is applicable to class C or class G in [T2,UC). The relative positioning of such resource pertinences with respect to the regions individuated by the navigational pattern "D:2" in the preorder/postorder plane for different time values is displayed in Fig. 7.

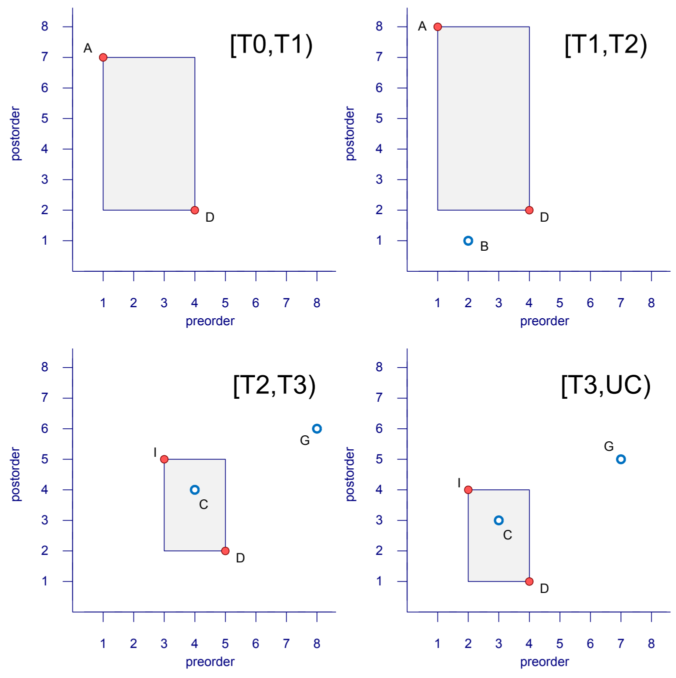


Figure 7. Query processing in the preorder/postorder plane. Placement of candidate resources is shown with blue circles

Hence, at any time T∈[T0,T1) there are no contents in the considered resource chunk which can qualify (the navigational pattern "D:2" translates into the [1,4] x [2,7] region). At any time T∈[T1,T2) the only valid element is foo(v1), which does not qualify since its semantic pertinence is class B (which has coordinates (2,1) in the preorder/postorder plane) which lays outside of the region individuated by the navigation pattern "D:2" (which is [1,4] x [2,8] in the plane). At any time T∈[T2,T3), valid elements are foo(v2), which qualifies since its semantic pertinence is class C (which has coordinates (4,4)) and is contained in the region individuated by "D:2" (which is [3,5] x [2,5]), and its subelement foo/bar(v1), which also qualifies as it inherits the applicability class C from its parent (whereas its other applicability class G with coordinates (8,6) lays outside of the region). At any time T∈[T3,UC), valid elements are foo(v2), which qualifies since its semantic pertinence is class C (which has coordinates (3,3)) and is contained in the region individuated by "D:2" (which is [2,4] x [1,4]), and its subelement foo/bar(v1), which also qualifies as it inherits the applicability class C from its parent (whereas its other applicability class G with coordinates (7,5) lays outside of the region also in this case). Therefore, considering the chunk in Fig. 2 as the only available resource, a query with navigational pattern "D:2" returns an empty result if the temporal selection condition involves a time T<T2 and retrieves the second version of the element foo (inclusive of the only version of the subelement foo/bar) if the temporal selection involves a time T≥T2.

As outlined in [1; Sec. 3.6], in order to obtain a fully fledged efficient and scalable personalization engine, the selection of resource contents based on the semantic indexing described above can be combined with the holistic technology described in [1] and relying on the holistic temporal slicing techniques presented in [13]. In a few words, the holistic technology relies on a four-level architecture on which stack-based algorithms can be executed for efficient path and twig matching in querying an XML file [14]. For details on such an approach for personalization and for better characterization of usefulness of the personalization approach in the medical domain, readers are referred to [1].

Finally, we can observe that the query template considered in this section can easily be extended to support other temporal selection operators (e.g., to test overlap or containment of intervals) and to retrieve data valid over temporal intervals (i.e., also belonging to more than one temporal version of the resource) like in the more general formulation presented in [1]. Furthermore, also the applicability constraint can be extended to the general form presented in [1], where combinations with "AND" and "OR" logical operators of several navigational patterns in positive or negated form can be specified (in order to qualify for a negated navigational pattern, a resource must have its representative point outside the region defined by the navigational pattern in the plane). Also applicability constraints involving multiple reference ontologies in the same query can be specified and processed as shown in [1].

5. CONCLUSIONS

In this work, we introduced a storage scheme based on a temporal relation which can be used to represent and manage a multi-version ontology (embodying a tree-shaped class hierarchy) in a relational database. The definition of primitive operations, which can be used for the maintenance of a multi-version ontology in such a framework, has also been provided. Finally, it has been shown how the query processing method described in [1] has to be augmented in order to deal with multi-version ontologies in the presence of the storage scheme presented in this work.

In future work, we will also consider performance aspects of the proposed solutions. In particular, we will test the efficiency of the approach in the presence of very large ontologies, with thousands of classes and hundreds of versions each. In such a case, the adoption of traditional indices like B^{+}trees or of some sort of temporal index structure might reveal itself necessary in order to cope with the size growth of the TreeRelation temporal table, and avoid excessive execution times for ontology modifications and for the first step of resource personalization queries.

We will also consider the extension of the present approach to non tree-shaped ontologies, that is ontologies where a class is allowed to be the child of more than one parent in the class hierarchy (e.g., where intersection classes can be defined and multiple inheritance is allowed). To this aim, we plan to try to extend our temporal relation approach to the GRIPP numbering scheme used in [15], which provides for the introduction of *non-tree edges* in order to apply the preorder/postorder numbering scheme of trees also to general directed acyclic graphs.

6. REFERENCES

[1] Grandi, F., Mandreoli, F., and Martoglia, R. 2012. Efficient management of multi-version clinical guidelines. *Journal of Biomedical Informatics* 45, 6 (Dec. 2012), 1120–1136. DOI= http://dx.doi.org/10.1016/j.jbi.2012.07.005.

[2] Riaño, D., Real, F., López-Vallverdú, J.A., Campana, F., Ercolani, S., Mecocci, P., Annicchiarico, R., and Caltagirone, C. 2012. An ontology-based personalization of health-care knowledge to support clinical decisions for chronically ill

patients. *Journal of Biomedical Informatics* 45, 3 (June 2012), 429–446 . DOI=http://dx.doi.org/10.1016/j.jbi.2011.12.008.

[3] Tu, S.W., Peleg, M., Carini, S., Bobak, M., Ross, J., Rubin, D., and Sim, I. 2011. A practical method for transforming free-text eligibility criteria into computable criteria. *Journal of Biomedical Informatics* 44, 2 (April 2011), 239–250. DOI=http://dx.doi.org/10.1016/j.jbi.2010.09.007.

[4] Grandi, F., Mandreoli, F., Martoglia, R., Ronchetti, E., Scalas, M.R., and Tiberio, P. 2009. Ontology-Based Personalization of E-Government Services. In *Intelligent User Interfaces: Adaptation and Personalization Systems and Technologies*, P. Germanakos and C. Mourlas, Eds. Information Science Reference Series. IGI Global, Hershey, PA, 167–187. DOI=http://dx.doi.org/10.4018/978-1-60566-308-1.ch010.

[5] W3C Consortium, Extensible Markup Language (XML) Home Page, Retrieved December 1, 2012 from http://www.w3.org/XML/.

[6] Field, M.J., and Lohr, K. N. (Eds.). 1990. *Clinical Practice Guidelines: Directions for a New Program*. National Academy Press, Washington, DC.

[7] Grandi, F., and Scalas, M.R., 2009. The Valid Ontology: a Simple OWL Temporal Versioning Framework. In *Proceedings of the 3rd International Conference on Advances in Semantic Processing* (Sliema, Malta, October 11 - 16, 2009). SEMAPRO'09. IEEE, Los Alamitos, CA, 98–102. DOI=http://dx.doi.org/10.1109/SEMAPRO.2009.12.

[8] Mackey, T.K., and Liang, B.A. 2011. The Role of Practice Guidelines in Medical Malpractice Litigation. *Virtual Mentor* 13, 1 (January 2011), 36–41. Retrieved December 1, 2012 from http://virtualmentor.ama-assn.org/2011/01/hlaw1-1101.html.

[9] Dietz, P.F. and Sleator D.D. 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (New York, NY, May 25 - 27, 1987). STOC'87. ACM, New York, NY, 365–372. DOI=http://doi.acm.org/10.1145/28395.28434.

[10] Grust, T., van Keulen, M. and Teubner, J. 2004. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.* 29, 1 (March 2004) 91–131. DOI= http://doi.acm.org/10.1145/974750.974754.

[11] Jensen, C.S., Dyreson, C.E., Böhlen, M.H., Clifford, J., Elmasri, R., Gadia, S.K., Grandi, F., Hayes, P.J., Jajodia, S., Käfer, W., Kline, N., Lorentzos, N.A., Mitsopoulos, Y.G., Montanari, A., Nonen, D.A., Peressi, E., Pernici, B., Roddick, J.F., Sarda, N.L., Scalas, M.R., Segev, A., Snodgrass, R.T., Soo, M.D., Tansel, A.U., Tiberio, P., and Wiederhold, G. 1998. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In *Temporal Databases, Research and Practice*, O. Etzion, S. Jajodia and S. Sripada, Eds. LNCS, Vol. 1399. Springer Verlag, Heidelberg, Germany, 367–405. DOI=http://www.doi.org/10.1007/BFb0053710.

[12] Clark, J. and DeRose, S. 1999. XML Path Language (XPath) Version 1.0. (November 1999). W3C Recommendation. Retrieved December 1, 2012 from http://www.w3.org/TR/xpath/.

[13] Mandreoli, F., Martoglia, R. and Ronchetti, E. 2006. Supporting Temporal Slicing in XML Databases. In *Proceedings of the 10th International Conference on Extending Database Technology* (Munich, Germany, March 26 - 31). EDBT'06. LNCS, Vol. 1399. Springer Verlag, Heidelberg, Germany, 295–312. DOI=http://dx.doi.org/10.1007/11687238_20.

[14] Bruno, N., Koudas, N. and Srivastava, D. 2002. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the SIGMOD International Conference on Management of Data* (Madison, WI, June 3 - 6, 2002). SIGMOD'02. ACM, New York, NY, 310–321. DOI=http://doi.acm.org/10.1145/564691.564727.

[15] Trißl, S. and Leser, U. 2007. Fast and practical indexing and querying of very large graphs. In *Proceedings of the SIGMOD International Conference on Management of Data* (Beijing, China, June 12 - 14, 2007). SIGMOD'07. ACM, New York, NY, 845–856. DOI=http://doi.acm.org/10.1145/1247480.1247573.