Representing and Querying Histories of Semistructured Databases Using Multidimensional OEM

Yannis Stavrakas [a,b], Manolis Gergatsoulis [c],

Christos Doulkeridis [a], Vassilis Zafeiris [a]

[a] *Knowledge & Database Systems Laboratory,*

Department of Electrical and Computer Engineering,

National Technical University of Athens (NTUA), 15773 Athens, Greece.

[b] *Institute of Informatics & Telecommunications,*

National Centre for Scientific Research (N.C.S.R.) 'Demokritos',

15310 Aghia Paraskevi Attikis, Greece.

[c] *Department of Archive and Library Sciences, Ionian University,*

Palea Anaktora, Plateia Eleftherias, 49100 Corfu, Greece.

Abstract

Multidimensional semistructured data (MSSD) are semistructured data that present different facets under different contexts. Context represents alternative worlds, and is expressed by assigning values to a set of user-defined variables called dimensions. The notion of context has been incorporated in the Object Exchange Model (OEM), and the extended model is called Multidimensional OEM (MOEM), a graph model for MSSD. In this paper, we explain in detail how MOEM can represent the history of an OEM database. We discuss how MOEM properties are applied in the case of representing OEM histories, and show that temporal OEM snapshots can be obtained from MOEM. We present a system that implements the proposed ideas, and we use an example scenario to demonstrate how an underlying MOEM database accommodates changes in an OEM database. Furthermore, we show that MOEM is capable to model changes occurring not only in OEM databases, but in Multidimensional OEM databases as well. The use of Multidimensional Query Language (MQL), a query language for MSSD, is proposed for querying the history of OEM databases and MOEM databases.

Key words: Context-Dependent Data, Multidimensional Semistructured Data, Multidimensional OEM, Histories of Semistructured Databases, Querying Semistructured Database Histories.

1 Introduction

In this paper we investigate the problem of representing and querying histories of semistructured databases. This problem can be stated as follows: given an OEM graph that comprises the semistructured database, we would like a way to represent dynamically changes in this database as they occur, keeping a history of transitions, so that we are able to subsequently query on those changes.

In order to represent changes in semistructured data we propose the use of *multidimensional semistructured data* (*MSSD*), which are semistructured data that present different facets under different *contexts*. Various aspects of MSSD

Email addresses: `ys@dblab.ntua.gr` (Yannis Stavrakas), `manolis@ionio.gr` (Manolis Gergatsoulis), `cdoulk@aueb.gr` (Christos Doulkeridis), `bzafiris@aueb.gr` (Vassilis Zafeiris).

have been addressed in previous work [Sta03,SG02,GSK01a,GSK$^+$01b,SGDZ02] by the authors. Context in MSSD is a means to represent alternative *worlds*, and is expressed by assigning values to a set of user-defined variables called *dimensions*. The notion of context has been incorporated in OEM, leading to *Multidimensional OEM* (*MOEM*), a graph model for MSSD.

To address the problem of representing histories of semistructured databases, we introduce a set of basic change operations for MOEM graphs, and define a mapping from changes in an OEM database to the MOEM basic change operations. Using this mapping, we produce MOEM graphs that represent the history of OEM databases. Moreover, we show that our approach is powerful enough to represent not only the history of OEM databases, but of MOEM databases as well.

An interesting property of MOEM graphs used to represent histories is that they can give temporal snapshots of the original database for any time instance, by applying a process called *reduction*. Besides, the proposed methodology for representing OEM histories and MOEM histories allows the formulation of complex queries concerning the evolution of objects in the OEM database or the MOEM database, respectively. To express such queries we propose the use of *Multidimensional Query Language* (*MQL*) [Sta03,SPES03], a query language designed especially for posing *context-driven queries* on MOEM databases.

The proposed methodology has been implemented in a prototype system, which is also presented in this paper. This paper extends work that appears in [SGDZ02].

The paper is organized as follows. In Section 2, we present some preliminary material about Multidimensional Semistructured Data. In Section 3, we introduce the basic change operations for MOEM. In Section 4, we define the way MOEM can be used to represent the history of an OEM database. In Section 5, we present an implemented system, and we follow an example scenario that demonstrates how an underlying MOEM database can accommodate changes in an OEM database. In Section 6, we show that MOEM is capable to model changes occurring not only in OEM databases, but in Multidimensional OEM databases as well. In Section 7, we demonstrate how MQL can be used to pose various queries on the history of evolving OEM and MOEM databases. In Section 8, we compare our approach with previous work on representing and querying histories of semistructured databases. Finally, Section 9 concludes the paper.

2 Multidimensional Semistructured Data

In this section we review some preliminary concepts of multidimensional semistructured data that will be used in the sections that follow. *Multidimensional semistructured data* (*MSSD* in short) [SG02] are semistructured data [Suc98,ABS00] which present different facets under different *contexts*. The main difference between conventional and multidimensional semistructured data is the introduction of *context specifiers*. Context specifiers are syntactic constructs that are used to qualify pieces of semistructured data and specify sets of *worlds* under which those pieces hold. In this way, it is possible to have at the same time variants of the same information entity, each holding under a different set of worlds. An information entity that encompasses a number of variants is called *multidimensional entity*, and its variants are called *facets* of the entity.

The facets of a multidimensional entity may differ in value and / or structure, and can in turn be multidimensional entities or conventional information entities. Each facet is associated with a context that defines the conditions under which the facet becomes a *holding facet* of the multidimensional entity.

2.1 Context and Dimensions

The notion of *world* is fundamental in MSSD. A world represents an environment under which data obtain a substance. In the following definition, we specify the notion of world using a set of parameters called *dimensions*.

Definition 1 *Let $\mathcal{D}$ be a nonempty set of dimension names and for each $d \in \mathcal{D}$, let $\mathcal{V}_d$ be the domain of d, with $\mathcal{V}_d \neq \emptyset$. A* world *$w$ with respect to $\mathcal{D}$ is a set of pairs (d, v), where $d \in \mathcal{D}$ and $v \in \mathcal{V}_d$, such that for every $d \in \mathcal{D}$ exactly one (d, v) belongs to w.*

In MSSD, sets of worlds are represented by context specifiers, which can be seen as constraints on dimension values.

Example 2 *The use of dimensions for representing worlds is shown through the following context specifiers:*

```
(a) [time=07:45]
(b) [language=greek, detail in {low,medium}]
(c) [season in {fall,spring}, daytime=noon | season=summer]
```

In Example 2, context specifier (a) represents the worlds for which the dimension `time` has the value `07:45`, while (b) represents the worlds for which `language` is `greek` and `detail` is either `low` or `medium`. Context specifier

(c) is more complex, and represents the worlds where `season` is either `fall` or `spring` and `daytime` is `noon`, together with the worlds where `season` is `summer`. Notice that, according to Definition 1, for a set of $(\mathit{dimension}, \mathit{value})$ pairs to represent a world with respect to a set of dimensions $\mathcal{D}$, it must contain exactly one pair for each dimension in $\mathcal{D}$. Therefore, if $\mathcal{D} = \{\texttt{language}, \texttt{detail}\}$ with $\mathcal{V}_{language} = \{\texttt{english}, \texttt{greek}\}$ and $\mathcal{V}_{detail} = \{\texttt{low}, \texttt{medium}, \texttt{high}\}$, then $\{(\texttt{language}, \texttt{greek}), (\texttt{detail}, \texttt{low})\}$ is one of the six possible worlds with respect to $\mathcal{D}$. This world is represented by context specifier (b) in Example 2, together with the world $\{(\texttt{language}, \texttt{greek}), (\texttt{detail}, \texttt{medium})\}$. It is not necessary for a context specifier to contain values for every dimension in $\mathcal{D}$. Omitting a dimension implies that its value may range over the whole dimension domain.

The context specifier `[]` is a *universal context* and represents the set of all possible worlds with respect to any set of dimensions $\mathcal{D}$, while the context specifier `[-]` is an *empty context* and represents the empty set of worlds with respect to any set of dimensions $\mathcal{D}$. In [SG02] we have defined operations on context specifiers, such as *context intersection* and *context union* that correspond to the conventional set operations of intersection and union on the related sets of worlds. We have also defined how a context specifier can be transformed to the set of worlds it represents with respect to a set of dimensions $\mathcal{D}$. An important case for MSSD is when two context specifiers represent disjoint sets of worlds; in that case the context specifiers are called *mutually exclusive*.

Object Exchange Model (OEM) is a graph data model defined in [AQM$^+$97] as a quadruple $O = (V, E, r, v)$, where V is a set of nodes, E a set of labeled directed edges (p, l, q) with $p, q \in V$ and l a string, r is a special node called the *root*[1] , and v is a function mapping each node to an atomic value of some type (integer, string, etc.), or to the reserved value C which denotes a complex object.

In this section, we introduce *Multidimensional Data Graph* as an extension of OEM, suitable for representing multidimensional semistructured data. Multidimensional Data Graph extends OEM with two new basic elements:

- *Multidimensional nodes* represent multidimensional entities, and are used to group together nodes that constitute facets of the entities. Graphically, multidimensional nodes have a rectangular shape to distinguish them from conventional circular nodes.
- *Context edges* are directed labeled edges that connect multidimensional nodes to their facets. The label of a context edge pointing to a facet, is a context specifier defining the set of worlds under which that facet holds. Context edges are drawn as thick lines, to distinguish them from conventional (thin-lined) OEM edges.

In Multidimensional Data Graph the conventional circular nodes of OEM are called *context nodes* and represent facets associated with some context. Conventional OEM edges (thin-lined) are called *entity edges* in Multidimensional Data Graph and define relationships between objects. All nodes are consid-

[1] Every node in V must be reachable from the root through some path.

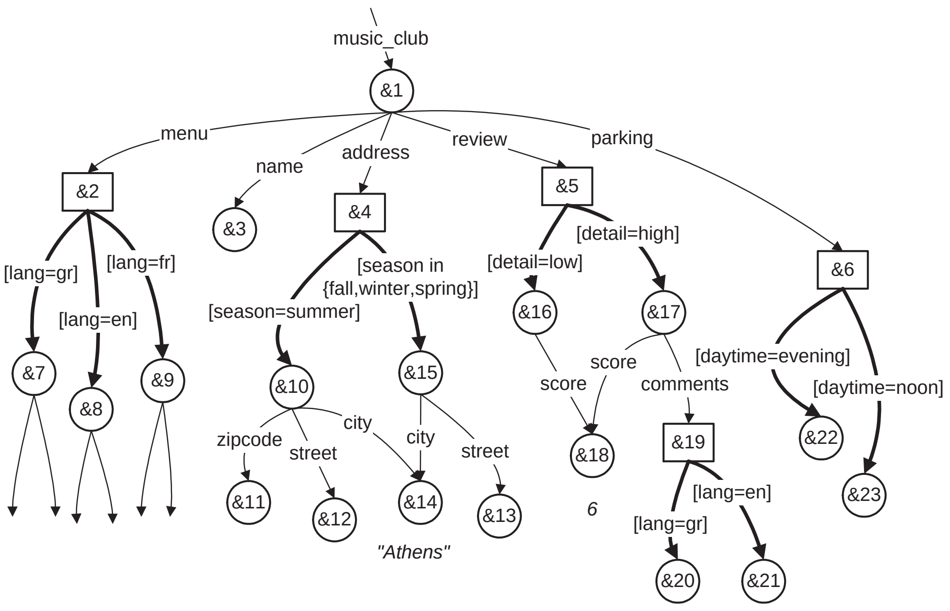


Fig. 1. A multidimensional music-club.

ered objects, and have unique *object identifiers* (*oid*s). Context objects are divided into *complex objects* and *atomic objects*. Atomic objects have a value from one of the basic types, e.g. integer, real, strings, etc. A context edge cannot start from a context node, and an entity edge cannot start from a multidimensional node. Those two are the only constraints on the morphology of Multidimensional Data Graph.

As an example, consider the Multidimensional Data Graph in Figure 1 which represents context-dependent information about a music-club. For simplicity, the graph is not fully developed and some of the atomic objects do not have values attached. The `music_club` with oid `&1` operates on a different address during the summer than the rest of the year (in Athens it is not unusual for clubs to move south close to the sea in the summer period, and north towards the city center during the rest of the year). Except from having a different value, context objects can have a different structure, as is the case of `&10` and `&15` which are facets of the multidimensional object `address` with oid `&4`.

The `menu` of the club is available in three languages, namely English, French and Greek. In addition, the club has a couple of alternative `parking` places, depending on the time of day as expressed by the dimension `daytime`. The music-club review has two facets: node `&16` is the low detail facet containing only the review score with value *6*, while the high detail facet `&17` contains in addition review comments in two languages.

In what follows we formally define Multidimensional Data Graph.

Definition 3 *Let $\mathcal{CS}$ be a set of context specifiers, $\mathcal{L}$ be a set of labels, and $\mathcal{A}$ be a set of atomic values. A* Multidimensional Data Graph *G is a finite directed edge-labeled multigraph $G = (V_{mld}, V_{cxt}, E_{cxt}, E_{ett}, r, v)$, where:*

(1) The set of nodes V consists of multidimensional nodes *and* context nodes*, $V = V_{mld} \cup V_{cxt}$. Context nodes are divided into* complex nodes *and* atomic nodes*, $V_{cxt} = V_c \cup V_a$.*

(2) The set of edges E consists of context edges *and* entity edges*, $E = E_{cxt} \cup E_{ett}$, such that $E_{cxt} \subseteq (V_{mld} \times \mathcal{CS} \times V)$ and $E_{ett} \subseteq (V_c \times \mathcal{L} \times V)$.*

(3) $r \in V$ is the root*, with the property that there exists a path from r to every other node in V.*

(4) v is a function that assigns values to nodes, such that: $v(x) = M$ if $x \in V_{mld}$, $v(x) = C$ if $x \in V_c$, and $v(x) = v'(x)$ if $x \in V_a$, where M and C are reserved values, and v' is a value function $v' : V_a \rightarrow \mathcal{A}$ which assigns values to atomic nodes.

As explained in Step 4, function v uses the reserved values M and C to classify nodes to multidimensional, complex, and atomic. The root of a Multidimensional Data Graph may be a context node or a multidimensional node.

Two fundamental concepts related to Multidimensional Data Graphs are *explicit context* and *inherited context* [SG02]. The explicit context of a context edge is the context specifier assigned to that edge, while the explicit context of an entity edge is considered to be the universal context specifier `[]`. The explicit context can be considered as the "true" context only within the boundaries of a single multidimensional entity. When entities are connected together in a graph, the explicit context of an edge is not the "true" context, in the sense that it does not alone determine the worlds under which the destination node holds. The reason for this is that, when an entity e_2 is part of (pointed by through an edge) another entity e_1, then e_2 can have substance only under the worlds that e_1 has substance. This can be conceived as if the context under which e_1 holds is inherited to e_2. The context propagated in that way is combined with (constrained by) the explicit context of each edge to give the *inherited context* for that edge. The inherited context of a node is the union of the inherited contexts of incoming edges (the inherited context of the root is `[]`). As an example, node `&18` in Figure 1, has inherited context `[detail in {low, high}]`. Worlds where `detail` is `low` are inherited through node `&16`, while worlds where `detail` is `high` are inherited through node `&17`.

In Multidimensional Data Graph leaves are not restricted to atomic nodes, and can be complex or multidimensional nodes as well. This raises the question under which worlds does a path lead to a leaf that is an atomic node. Those worlds are given by *context coverage*, which is symmetric to inherited context, but propagates to the opposite direction: from the leaves up to the root of the graph. The context coverage of a node or an edge represents the worlds under which the node or edge has access to leaves that are atomic nodes. The context coverage of leaves that are atomic nodes is `[]`, while the context coverage of leaves that are complex nodes or multidimensional nodes is `[-]`. For instance, the context coverage of node `&19` in Figure 1 is `[lang in {gr, en}]` (all leaves in Figure 1 are considered atomic nodes).

For every node or edge, the context intersection of its inherited context and its context coverage gives the *inherited coverage* [Sta03] of that node or edge. The inherited coverage represents the worlds under which a node or edge may actually hold, as determined by constraints accumulated from both above and below. A related concept is *path inherited coverage*, which is given by the context intersection of the inherited coverages of all edges in a path, and represents the worlds under which a complete path holds.

A *context-deterministic* Multidimensional Data Graph is a Multidimensional Data Graph in which context nodes are accessible from a multidimensional node under mutually exclusive inherited coverages (hold under disjoint sets of worlds). Intuitively, context-determinism means that, under any specific world, at most one context node is accessible from a multidimensional node. A *Multidimensional OEM*, or *MOEM* in short, is a context-deterministic Multidimensional Data Graph whose every node and edge has a non-empty inherited coverage. In an MOEM all nodes and edges hold under at least one world, and all leaves are atomic nodes. The Multidimensional Data Graph in Figure 1 is an MOEM.

Given a world w, it is possible to *reduce* an MOEM *to a conventional OEM* graph holding under w, by eliminating nodes and edges whose inherited coverage does not contain w. A process that performs such a *reduction to OEM* is presented in [SG02]. In addition, given a set of worlds, it is possible to *partially reduce* an MOEM into a new MOEM, that encompasses only the OEM facets for the given set of worlds. Partial reduction for a context *c* involves the elimination of nodes and edges whose inherited coverage does not contain any common worlds with *c*; eliminated nodes and edges do not hold under any of the worlds represented by *c*.

In [SG02] we extend *ssd-expressions* [ABS00] with context specifiers, and propose *mssd-expressions*, a syntax for representing multidimensional semistructured data. In mssd-expressions curly brackets {} enclose context objects, and angle brackets <> enclose multidimensional objects. As an example, consider the following mssd-expression that describes the (incomplete) graph of Figure 1:

```
&1 {menu: &2 <[lang=gr]: &7 {...},
              [lang=en]: &8 {...},
              [lang=fr]: &9 {...}>,
    name: &3 "...",
    address: &4 <[season=summer]: &10 {zipcode: &11 "...",
                                       street: &12 "...",
                                       city: &14 "Athens"},
                 [season in {fall,winter,spring}]:
                                  &15 {city: &14,
                                       street: &13 "..."}>,
    review: &5 <[detail=low]:  &16 {score &18 6},
                [detail=high]: &17 {score: &18,
                                    comments:
                                        &19 <[lang=gr]: &20 "...",
                                             [lang=en]: &21 "...">},
    parking: &6  <[daytime=evening]: &22 "...",
                  [daytime=noon]: &23 "...">
}
```

Another way of representing multidimensional semistructured data is *Multidimensional XML* (MXML in short) [GSK01a,GSK$^+$01b], an extension of XML that incorporates context specifiers. In MXML, *multidimensional elements* and *multidimensional attributes* may have different facets that depend on a number of dimensions. MXML gives new possibilities for designing Web pages that deal with context-dependent data. We refer to the new method as the *multidimensional paradigm*, and we present it in detail in [GSK$^+$01b].

3 MOEM Basic Change Operations

A conventional OEM graph is defined in [AQM$^+$97] as a quadruple $O = (V, E, r, v)$, where V is a set of nodes, E a set of labeled directed edges (p, l, q) with $p, q \in V$ and l a string, r is a special node called the *root*, and v is a function mapping each node to an atomic value of some type (integer, string, etc.), or to the reserved value C which denotes a complex object. In order to modify an OEM database O, four basic change operations were identified in [CAW99]:

creNode(*nid***,** *val***)**: creates a new node, where *nid* is a new node oid ($nid \notin V$), and *val* is an atomic value or the reserved value C.

updNode(*nid***,** *val***)**: changes the value of an existing object *nid* to a new value *val*. The node *nid* must not have any outgoing edges (in case its old value is C, the edges should have been removed prior to updating the value).

addArc(p**,** l**,** q**)**: adds a new edge labeled l from object p to object q. Both nodes p and q must already exist in V, and (p, l, q) must not exist in E.

remArc(p**,** l**,** q**)**: removes the existing edge (p, l, q). Both nodes p and q must exist in V.

Given an MOEM database $M = (V_{mld}, V_{cxt}, E_{cxt}, E_{ett}, r, v)$, we introduce the following basic operations for changing M. In what follows, $V = V_{mld} \cup V_{cxt}$ is the set of all nodes in M, and $E = E_{cxt} \cup E_{ett}$ is the set of all edges in M.

createCNode(cid**,** val**)**: a new context node is created. The identifier cid is new and must not occur in V_{cxt}. The value val can be an atomic value of some type, or the reserved value C.

updateCNode(cid**,** val**)**: changes the value of $cid \in V_{cxt}$ to val. The node must not have any outgoing arcs.

createMNode(mid**)**: a new multidimensional node is created. The identifier mid is new and must not occur in V_{mld}.

addEEdge(cid**,** l**,** id**)**: creates a new entity edge with label l from node cid to node id, where $cid \in V_{cxt}$ and $id \in V$.

remEEdge(cid**,** l**,** id**)**: removes the entity edge (cid, l, id) from M. The edge (cid, l, id) must exist in E_{ett}.

addCEdge(mid**,** cxt**,** id**)**: creates a new context edge with context specifier cxt from node mid to node id, where $mid \in V_{mld}$ and $id \in V$.

remCEdge(mid**,** cxt**,** id**)**: removes the context edge (mid, cxt, id) from M. The context edge (mid, cxt, id) must exist in E_{cxt}.

For OEM and MOEM, object deletion is achieved through edge removal, since in both OEM and MOEM the persistence of an object is determined by whether or not the object is reachable from the root.

Sometimes the result of a single basic operation u leads to an inconsistent state. For instance, when a new object is created, it is temporarily unreachable from the root. In practice however, it is typical to have a sequence $L = u_1, u_2, \ldots, u_n$ of basic operations u_i, which corresponds to a higher level modification to the database. By associating such higher level modifications with a timestamp, an OEM history H is defined as a sequence of pairs (t, U), where U denotes a set of basic change operations that corresponds [CAW99] to L, and t is the associated timestamp. Note that within a single sequence L, a newly created node may be unreachable from the root and still not be considered deleted. At the end of each sequence, however, unreachable nodes are considered deleted and cannot be referenced by subsequent operations.

4 Representing OEM Histories Using MOEM

In this section we give a detailed explanation of the way MOEM can represent changes in an OEM database. Next, we discuss how Multidimensional Data Graph properties, like inherited coverage and reduction, apply in the case of representing OEM histories. In particular, we use reduction to OEM and partial reduction to obtain temporal snapshots of the OEM database.

Our approach is to map the four OEM basic change operations to MOEM basic operations, in such a way, that new facets of an object are created whenever changes occur in that object. In this manner, the initial OEM database O is transformed into an MOEM graph[2], that uses a dimension d whose domain is time to represent an OEM history H valid [CAW99] for O. We assume that our time domain T is linear and discrete. We also assume:

(1) A reserved value *start*, such that $start < t$ for every $t \in T$, representing the beginning of time.

(2) A reserved value *now*, such that $t < now$ for every $t \in T$, representing the current time.

The time period during which a context node is the holding node of the corresponding multidimensional entity is denoted by qualifying that context node with a context specifier of the form `[d in` $\{t_1\,..\,t_2\}$`]`. In context specifiers the syntactic shorthand $v_1..v_n$ for discrete and totally ordered domains means all values v_i such that $v_1 \leq v_i \leq v_n$.

Figure 2 gives an intuition about the correspondence between OEM and MOEM operations. Consider the sets U_1 and U_2 of basic change operations, with timestamps t_1 and t_2 respectively. Figure 2(a) shows the MOEM representation of an atomic object, whose value "A" is changed to "B" through a call to the basic change operation *updNode* of U_1. Figure 2(b) shows the result of *addArc* operation of U_1, while Figure 2(c) shows the result of *remArc* op-

[2] The initial OEM database is transformed to a Multidimensional Data Graph, which, as we will show in Section 4.2, is always an MOEM.

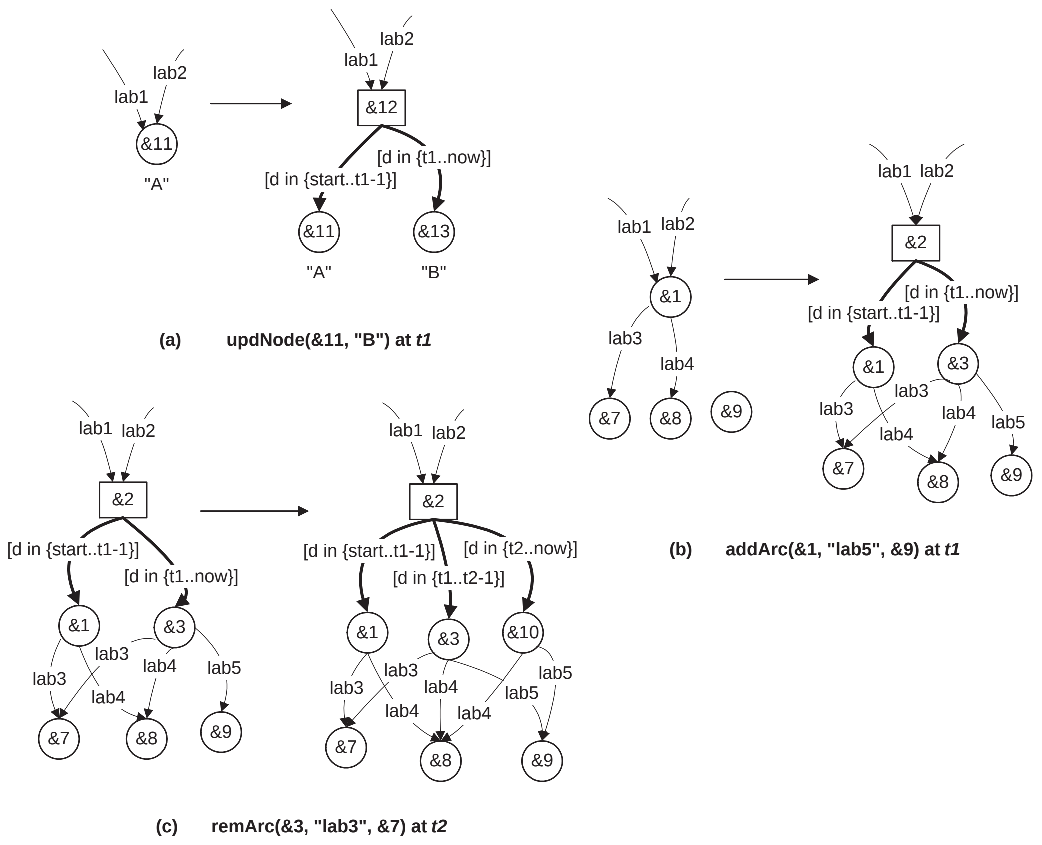


Fig. 2. Modeling OEM basic change operations with MOEM.

eration of U_2, on the same multidimensional entity. It is interesting to notice that three of the four OEM basic change operations are similar, in that they update an object be it atomic (*updNode*) or complex (*addArc*, *remArc*), and all three are mapped to MOEM operations that actually update a new facet of the original object. Creating a new node with *creNode* does not result in any additional MOEM operations; the new node will subsequently be linked with the rest of the graph (within the same set U_i) through *addArc* operation(s), which will cause new object facet(s) to be created.

Note that, as shown in Figure 2, an OEM object may correspond to many MOEM objects with different oids, and this is perceived as if the OEM object changes oids during its history. However, this is more an implementation issue and does not present any real problem. In addition, it is worth noting that the changes induced by the OEM basic change operations affect only localized parts of the MOEM graph, and do not propagate throughout the graph.

Having outlined the approach, we now give a detailed specification. First, the following four utility functions and procedures are defined for MOEM.

(1) **id1 $\leftarrow$ md(id2)**, with $id1, id2 \in V$. Returns the multidimensional node for a context node, if it exists. If $id2 \in V_{cxt}$ and there exists an element (mid, cxt, id) in E_{cxt} such that $id = id2$, then mid is returned. If $id2 \in V_{cxt}$ and no corresponding context edge exists, $id2$ is returned. If $id2 \in V_{mld}$, $id2$ is returned. We assume that there is at most one multidimensional node pointing to any context node, in other words for every $cid \in V_{cxt}$ there is at most one mid such that $(mid, cxt, cid) \in E_{cxt}$. Notice however, that this is a property of MOEM for the specific application, because of the special way the MOEM graph is constructed for representing histories, and does not apply to the general case.

(2) **boolean $\leftarrow$ withinSet(cid)**, with $cid \in V_{cxt}$. Checks whether the context node cid is created within the current set U of basic change operations. This function is used while change operations are in progress, and returns `true` if cid was created within the same set. It returns `false` if cid was created within a previous set of operations.

(3) The following procedure **mEntity(id)**, with $id \in V_{cxt}$, creates a new multidimensional node mid pointing to id, and redirects all incoming edges from id to mid. Note that the procedure alters the graph, but not the information modeled by the graph: the multidimensional entity created by the procedure has node id as its only facet holding under every world.

```
mEntity(id) {
    createMNode(mid)
    addCEdge(mid, [d in start..now], id)
    for every (x, l, id) in $E_{ett}$ {
        addEEdge(x, l, mid)
        remEEdge(x, l, id)
    }
}
```

(4) In the procedure **newCxt(id1, id2, ts)**, with $id1, id2 \in V_{cxt}$ and $ts \in T$, $id1$ is the currently most recent facet of a multidimensional entity, and $id2$ is a new facet that is to become the most recent. The procedure arranges the context specifiers accordingly.

```
newCxt(id1, id2, ts) {
    remCEdge(md(id1), [d in {x..now}], id1)
    addCEdge(md(id1), [d in {x..ts-1}], id1)
    addCEdge(md(id1), [d in {ts..now}], id2)
}
```

The next step is to show how each OEM basic change operation is implemented using the basic MOEM operations. We assume that each of the OEM operations is part of a set U with timestamp ts, and that the node p is the most recent context node of the corresponding multidimensional entity, if such an entity exists. This is because changes always happen to the current snapshot of OEM, which corresponds to the most recent facets of MOEM multidimensional entities. The most recent context node is the one holding in current time, i.e. the node whose context specifier is of the form `[d in {`*somevalue*`..now}]`. The implementation of the OEM basic change operations follows.

- **updNode(*p*, *newval*)** : If p has been created within U, its value is updated directly, and the process terminates. Otherwise, if p is not pointed to by a multidimensional node, a new multidimensional node is created for p, having p as its only context node with context specifier `[d in {start..now}]`. A new facet is then created with value *newval*, and becomes the most recent facet by adjusting the relevant context specifiers. Since a node updated by *updNode* cannot have outgoing edges, no edge copying takes place in contrast to the case of *addArc*.

  ```
  updNode(p, newval) {
       if not withinSet(p) {
             if not exists (x, c, p) in $E_{cxt}$
                   mEntity(p)
             createCNode(n, newval)
             newCxt(p, n, ts)
       } else updateCNode(p, newval)
  }
  ```

- **addArc(*p*, *l*, *q*)** : If p has been created within U, it is used directly: the new edge is added, and the process terminates. Otherwise, if p is not already pointed to by a multidimensional node, a new multidimensional node is created for p, having p as its only context node with context specifier `[d in {start..now}]`. A new "clone" facet n is then created by copying all outgoing edges of p to n. In this case, the context specifiers are adjusted so that ts is taken into account, and n becomes the most recent facet as depicted in Figure 2(b) for $ts = t1$. Finally the new edge specified by the basic change operation is added to the most recent facet. Note that, in the frame of representing changes, an MOEM is constructed in such a way that an entity edge does not point directly to a context node q_c if there exists a context edge (q_m, c, q_c); instead, it always points to the corresponding multidimensional node q_m, if q_m exists. This is achieved by using the function $md(q)$ in combination with $mEntity(p)$.

```
addArc(p, l, q) {
     if not withinSet(p) {
          if not exists (x, c, p) in $E_{cxt}$
               mEntity(p)
          createCNode(n, 'C')
          newCxt(p, n, ts)
          for every (p, k, y) in $E_{ett}$
               addEEdge(n, k, y)
          addEEdge(n, l, md(q))
     } else addEEdge(p, l, md(q))
}
```

- **remArc(*p*, *l*, *q*)** : The process is essentially the same as $addArc(p, l, q)$, with the difference of removing an edge at the end of the process, instead of adding one. Therefore, *remArc* is like *addArc*, except for the last two calls to *addEEdge* which are replaced with calls to *remEEdge* with the same arguments.
- **creNode(*p*, *val*)** : this basic change operation is mapped to $createCNode(p, val)$ with no further steps. New facets will be created when new edges are added to connect node p to the rest of the graph.

In the previous section we explained how the history of an OEM database is incorporated in an MOEM graph, using multidimensional nodes and context edges. However, we have not shown that this graph is actually an MOEM graph. It is obviously a Multidimensional Data Graph, but in order to be an MOEM graph (a) it must be context-deterministic, and (b) every node and edge must have a non-empty inherited coverage.

In this section we examine how the MSSD properties of context-determinism, inherited coverage, and reduction are applied in the particular case of representing OEM histories. We show that a Multidimensional Data Graph constructed as specified in Section 4.1 is always an MOEM, and furthermore, an MOEM that has special characteristics, not generally encountered in MOEMs.

Let G be a Multidimensional Data Graph produced by the process specified in Section 4.1, let e be a multidimensional entity in G, with multidimensional node m and facets $e_1, e_2, \ldots, e_n$, and let $c_1, c_2, \ldots, c_n$ be the context specifiers of the respective context edges. Then, the process in Section 4.1 guarantees the following special properties for G:

(a) Context edges always point to context nodes, thus e_1, e_2, $\ldots$, e_n are exclusively context nodes.

(b) A context node e_i is facet of at most one multidimensional node m, therefore at most one context edge can point to e_i.

(c) Every multidimensional node m (except possibly the root) is pointed by at least one entity edge.

(d) A time instance is a world for G.

(e) All leaf nodes in G are atomic nodes.

Because of the procedure *newCxt*, which constructs the context edges, for every multidimensional entity e in G the explicit contexts c_1, c_2, …, c_n always define disjoint sets of worlds. Consequently, the inherited coverages of the context edges are mutually exclusive, since they are subsets of the corresponding explicit contexts. Because of the special property (a) above, given any world w, if we start from the multidimensional node m we can navigate to at most one context node through context edges that hold under w, hence G is context-deterministic.

In addition, from the procedures *mEntity* and *newCxt* it can be seen that:

(f) c_1 has the form [`d in` $\{start..somevalue_1\}$]

(g) c_n has the form [`d in` $\{somevalue_N..now\}$]

(h) The context union of c_1, c_2, …, c_n is [`d in` $\{start..now\}$], for every e in G.

Although for every multidimensional entity e in G the explicit contexts c_1, c_2, …, c_n cover the complete $\{start..now\}$ time range, this is not the case with the corresponding inherited coverages, which denote the true life span of the entity and its facets. First, let us discuss context coverage. Because of the special properties (c), (e), and (h) above, the context coverage of all context nodes, multidimensional nodes, and entity edges in G, is [`d in` $\{start..now\}$], which is a universal context. Besides, the context coverage of a context edge is always its explicit context. Therefore, in the case of representing OEM histories context coverage does not propagate any constraints, and for every node and edge in G *the inherited coverage coincides with the inherited context.*

Let us examine the meaning of inherited context in G. Each multidimensional entity e in G corresponds to a node that existed at some time in the evolution of the OEM graph. The facets of e correspond to OEM changes that had affected that node. Edges pointing to the multidimensional node m correspond to edges that pointed to that node at some time in the evolution of the OEM graph. In addition, the inherited context of edges pointing to m will be such as to allow to each one of the facets e_1, e_2, …, e_n to "survive" under some world. Therefore, for every facet e_i with $2 \leq i \leq n-1$ the explicit context c_i is also the inherited context of the context node e_i. As we have seen, $c_1 = $ [`d in` $\{start..somevalue_1\}$], and $c_n = $ [`d in` $\{somevalue_N..now\}$]; for facets e_1 and e_n incoming edges restrict the explicit contexts, so that the inherited context of e_1 may have a first value greater than `start`, while the inherited context of e_n may have a second value smaller than `now`.

It is now easy to see that there cannot be any elements in G with empty inherited coverage, and, since G is context-deterministic, G is an MOEM. In addition, the inherited coverage of the root of G is a universal context, meaning that for any given time instance t within $\{start..now\}$ we can reduce G to an OEM holding under t.

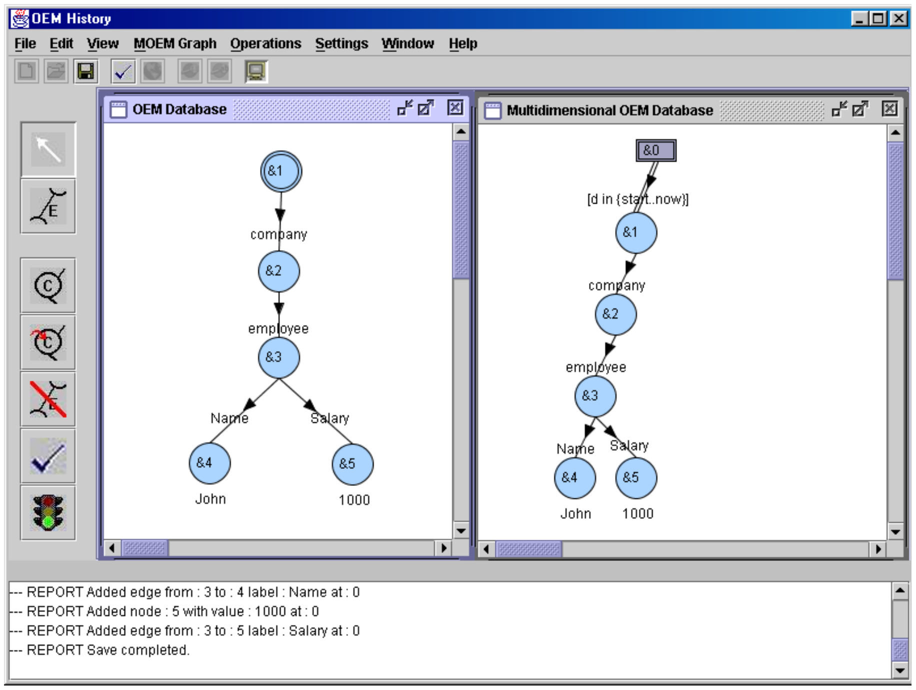
Conceptually, given an OEM database O and an MOEM database M that represents the history of O, it is possible to specify any time instance t from the time domain T and reduce M to an OEM database O' holding under t. Then O' will be the snapshot of O at the time instance t. It is also possible to apply partial reduction to M for a set of time instances [`d in` $\{t_1, t_2, \ldots, t_n\}$]. In this case an MOEM sub-graph of M is returned, encompassing the OEM snapshots at t_1, t_2, …, t_n.

5 Implementation: OEM History

OEM History [OEM] is an application developed in Java, which implements the method described in Section 4.1 for representing OEM histories. As it can be seen in Figure 3, OEM History employs a multi-document interface (MDI) with each internal window displaying a data graph. There are two main windows: one that displays an MOEM graph that corresponds to the internal model of the application, and one that always shows an OEM graph corresponding to the current state of the OEM database. Furthermore, the user can ask for a snapshot of the database for any time instance in T (the time domain), which will be presented as an OEM graph in a separate window. The toolbar on the left side contains buttons that correspond to the four OEM basic change operations, and can be used only on the window with the OEM depicting the current state of the database. In reality, these buttons invoke operations that update the internal MOEM data model of the application, which is the only model actually maintained by OEM History. The current OEM database is the result of an MOEM reduction to OEM under the world where d is `now`.

The "tick" button in the left toolbar removes nodes that are not accessible from the root. The last button in the toolbar marks the end of a sequence of basic change operations, and commits all changes to the database under a common timestamp. Operations like MOEM reduction to OEM can be initiated from the upper toolbar or from the application menu.

In Figure 3, we see the initial state of an OEM database containing information about the employees of a company, and the corresponding MOEM graph. The right window displays the underlying MOEM model, while the left window displays the result of the MOEM reduction for $d =$ `now`.



```
--- REPORT Added edge from : 3 to : 4 label : Name at : 0
--- REPORT Added node : 5 with value : 1000 at : 0
--- REPORT Added edge from : 3 to : 5 label : Salary at : 0
--- REPORT Save completed.
```

Fig. 3. Initial state of example database in *OEM History* application.

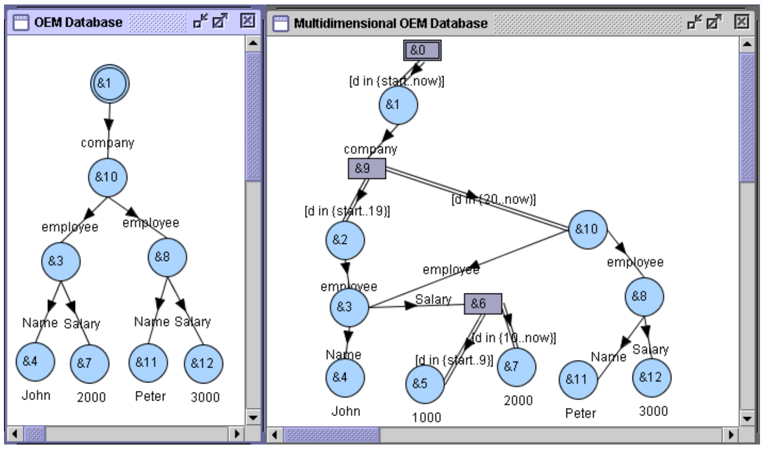


Fig. 4. Example database after two sequences of basic changes upon the initial database state.

Figure 4 shows the current state of the OEM database and the corresponding MOEM graph after a couple of change sequences. First, at the time instance `10` the salary of `John` has been increased from `1000` to `2000`. Then, at the time instance 20 a new employee called `Peter` joined the company with salary 3000.

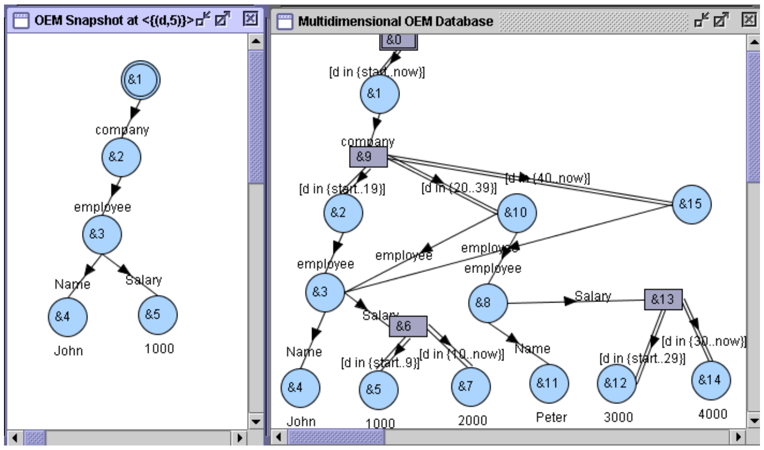


Fig. 5. Example database after another two sequences of basic changes upon the database of Figure 4.

In Figure 5 two more change sequences have been applied. The salary of `Peter` increased to 4000 at the time instance 30, but quite ungratefully `Peter` left the company at the time instance 40. Note that, as shown on the caption, the left window does not display the current OEM. Instead it depicts a snapshot of the OEM database for the time instance 5, which is obtained by reducing to OEM the MOEM in the right window for $d = 5$. That snapshot is identical to the initial state of the database, since the first change occurred at the time instance 10.

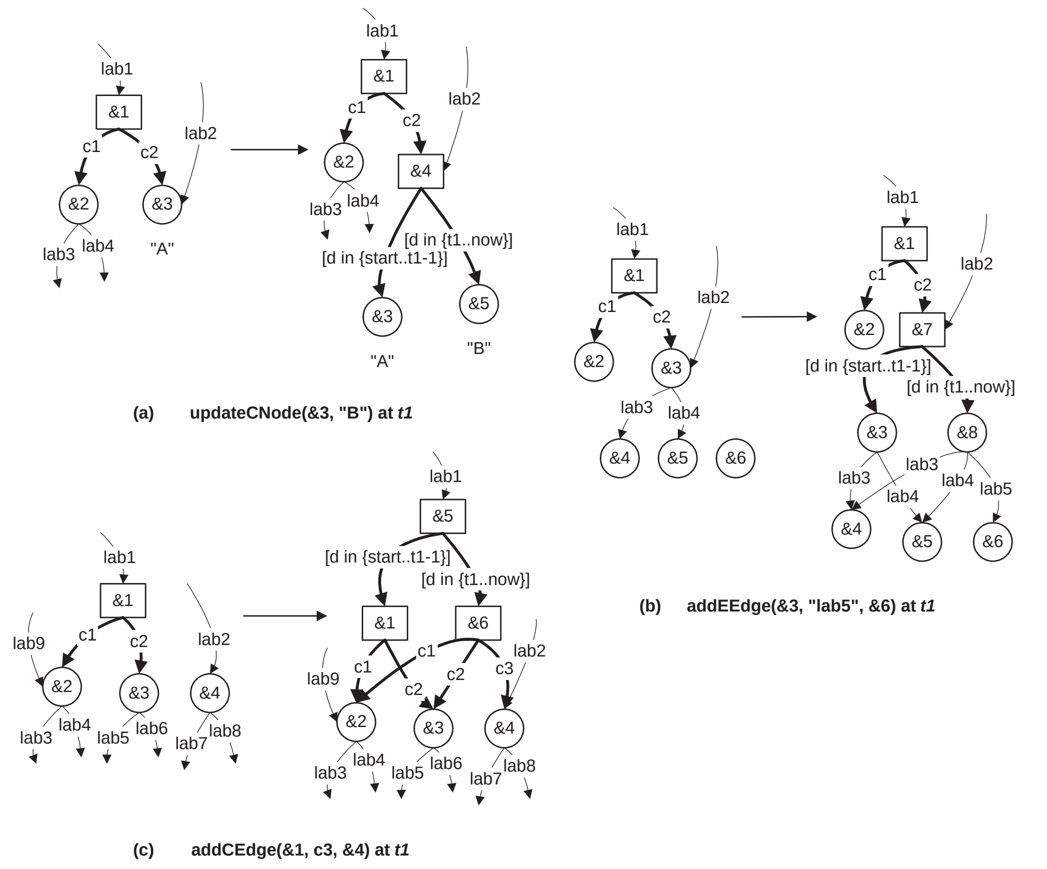
6 Representing Changes in MOEM Databases

Besides representing the history of OEM databases, MOEM has another interesting property. In this section we show that MOEM is expressive enough to *model its own histories*. In other words, for any MOEM database M evolving over time it is possible to maintain an MOEM database M', such that M'

represents the history of M.

The approach is similar to that of Section 4.1; we show that each of the MOEM basic operations (defined in Section 3) that is applied to M, can be mapped to a number of MOEM basic operations on M', in such a way that M' represents the history of M. Figure 6 gives the intuition about this mapping, for three basic operations. Context edge labels $c1$, $c2$, and $c3$ denote context specifiers involving any number of dimensions, while the dimension d is defined in Section 4.1. Note that the use of dimension d in M' does not preclude M from using other dimensions that range over time domains. The MOEM operations depicted in Figure 6 are basic operations occurring on M, and the corresponding graphs show how those operations transform M'. For simplicity, graphs on the left side do not contain context specifiers with the dimension d, and all timestamps are $t1$. It is however easy to envisage the case where d is also on the left side and timestamps progressively increase in value, if we look at Figure 2 (b) and (c) which follow a similar pattern.

Figure 6(a) shows a facet with id `&3` whose value is changed from “A” to “B” through a call to *updateCNode*. Figure 6(b) shows the result of an *addEEdge* operation, and Figure 6(c) depicts the *addCEdge* basic operation. Among MOEM basic operations *not* shown in Figure 6, *remEEdge* is very similar to *addEEdge*; the difference is that an entity edge is removed from facet `&8` instead of being added. In addition, *remCEdge* is similar to *addCEdge*: instead of adding one context edge to `&6`, one is removed. Finally, the MOEM basic operations *createCNode* and *createMNode* are mapped to themselves; M' will record the change when the new nodes are connected to the rest of the graph M through calls to *addEEdge* or *addCEdge*.



(a) updateCNode(&3, "B") at *t1*
(b) addEEdge(&3, "lab5", &6) at *t1*
(c) addCEdge(&1, c3, &4) at *t1*
[d in {start..t1-1}]
[d in {t1..now}]

Fig. 6. Modeling Multidimensional OEM basic operations with MOEM.

An MOEM graph M' constructed through the process outlined above represents the history of the MOEM graph M. In contrast to the case of OEM histories where a world is defined by giving a value to a single dimension d representing time, in the case of MOEM histories a world for M' involves in general more than one dimensions, including the time dimension d. Therefore, by specifying a time instance t we actually define the set of worlds for which $d = t$. In that set, dimensions other than d may have any combination of values from their respective domains.

In order to get a temporal snapshot of M from M', *partial reduction* is used. By applying the process of partial reduction to M' for any time instance $t \in T$, we get the snapshot of the MOEM database M at t. It is also possible to apply partial reduction to M' for a set of time instances [d in $\{t_1, t_2, \ldots, t_n\}$]. In this case an MOEM sub-graph of M is returned, encompassing all MOEM snapshots at $t_1, t_2, \ldots, t_n$.

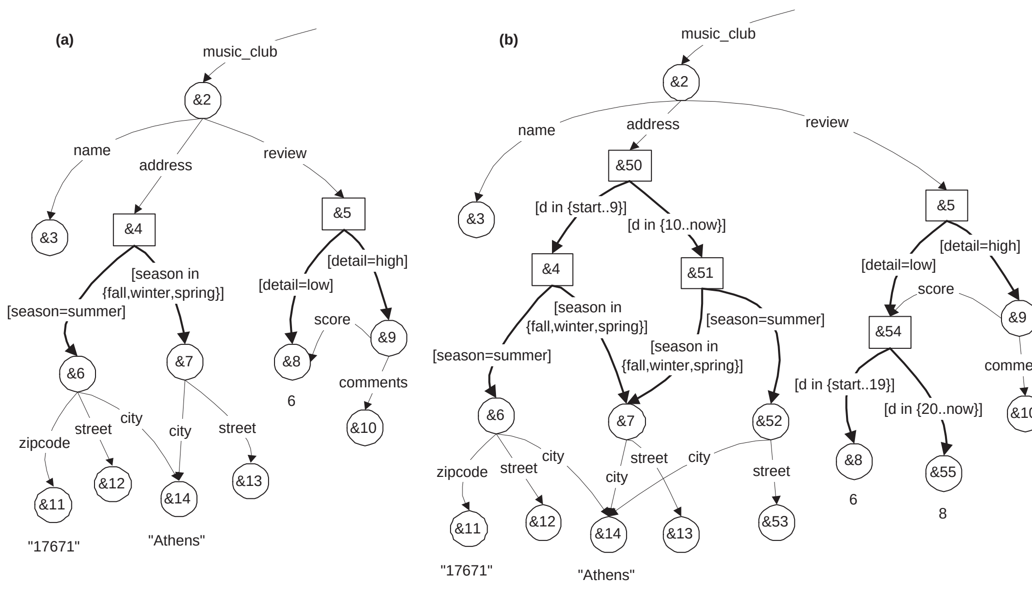


Fig. 7. Representing the history of the multidimensional music club in (a) with the MOEM in (b).

As an example, consider the MOEM graph of Figure 7(b) that represents the history of the multidimensional music club in Figure 7(a). The initial state of the database as depicted in Figure 7(a) does not contain the nodes &50, &51, &52, &53, &54, and &55 as well as their outgoing edges, which where introduced in Figure 7(b) because of change operations.

Two sequences of changes have occurred: (1) at the time instance 10 the summer address of the club changed to another address with a different street and without a zipcode, and (2) at the time instance 20 the review score changed from 6 to 8. The first sequence of changes involves the redirection of a context edge, so that it points to the newly added node &52 instead of node &6. The redirection is achieved through a call to *remCEdge* and a subsequent call to *addCEdge*, which are addressed in Figure 6(c). The second sequence of changes is actually a call to *updateCNode* for the node `&8`, which is addressed in Figure 6(a).

7 Querying Changes with MQL

The approach we presented above keeps different "versions" of each object as they change, encompassing their histories within the corresponding multidimensional entities as a collection of object snapshots. It is interesting to notice that a simpler representation could also have allowed for reduction of the database to temporal snapshots: context edges that depart from a multidimensional root lead to the graphs of all database states that have occurred, and they are labeled with the valid period of the corresponding database state. In effect, this method keeps a copy of the whole database after each sequence of basic change operations, and is clearly a waste of space, but it allows reduction in a straightforward way. An important benefit of representing changes at the level of multidimensional entities rather than as a collection of database snapshots stems from the queries we can formulate to retrieve information about the evolution of those entities. In this section we give examples of such queries using MQL, a query language for multidimensional semistructured data. We start with a brief introduction to MQL, continue with querying histories of OEM databases, and close the section with querying histories of MOEM databases.

Multidimensional Query Language [Sta03], or *MQL* in short, is a query language especially designed for MOEM databases. MQL is based on a "core language" for semistructured data described in [ABS00], and is effectively an extension of Lorel [AQM+97] combined with elements from UnQL [BFS00].

An important feature of MQL is *context path expressions*, which are path expressions qualified with context specifiers, *context variables*, and *context patterns*. Context path expressions take advantage of the fact that every Multidimensional Data Graph can be transformed to a *canonical form* [Sta03], with the property that every possible path is formed by a repeated succession of one context edge and one entity edge. Therefore, context path expressions consist of a number of *entity parts* and *facet parts* succeeding one another. Entity parts are matched against entity edges, while facet parts are matched against context edges:

```
[detail=high]music_club::[-].review::[-] X
```

In this context path expression, `music_club` and `review` are entity parts, while the two empty context specifiers `[-]` are facets parts. A facet part matches a corresponding context edge, if it is subset of the explicit context of the edge, in other words, if every world it defines is covered by the explicit context of the edge. Consequently, the empty context `[-]` matches any context edge. The context specifier `[detail=high]` is an *inherited coverage qualifier* and is matched against the *path inherited coverage* of a path. For a path to match an inherited coverage qualifier, it must hold under every world specified by the qualifier. An inherited coverage qualifier may precede any entity part or facet part in a context path expression. Facet parts can often be omitted, implying the empty context `[-]`. Therefore, the above context path expression can also be written as:

```
[detail=high]music_club.review X
```

Evaluated on the graph of Figure 1, this context path expression causes the *context object variable* `X` to bind to node `&17`.

The general form of an MQL query is the following:

`select` <*results template*>
`context` <*definition of new context variables*>
`from` <*list of context path expressions*>
`where` <*predicate*>
`within` <*context predicate*>

The `select` clause must exist in an MQL query, whereas the rest of the clauses are optional. Generally speaking, the `from` clause contains context path expressions that result in variable bindings and should be considered first; the `where` and `within` clauses filter the bound values and are considered next; the `context` clause defines new context variables that are used in the `select` clause; finally, the `select` clause is considered, which constructs the results. The result of an MQL query is always a Multidimensional Data Graph in the form of an mssd-expression. MQL has been implemented [SPES03] on top of LORE [MAG$^+$97].

For our query examples, we will use the MOEM database depicted in Figure 5. Context path expressions in MQL queries are matched against the canonical form of an MOEM database, and they often use path inherited coverage. To facilitate the comprehension of queries, Figure 8 depicts the database of Figure 5 transformed to canonical form and supplemented with the inherited coverages of edges, which appear in boldface characters. As shown in Figure 8, the transformation to canonical form introduced the multidimensional nodes `&16`, `&17`, `&18`, and `&19`.

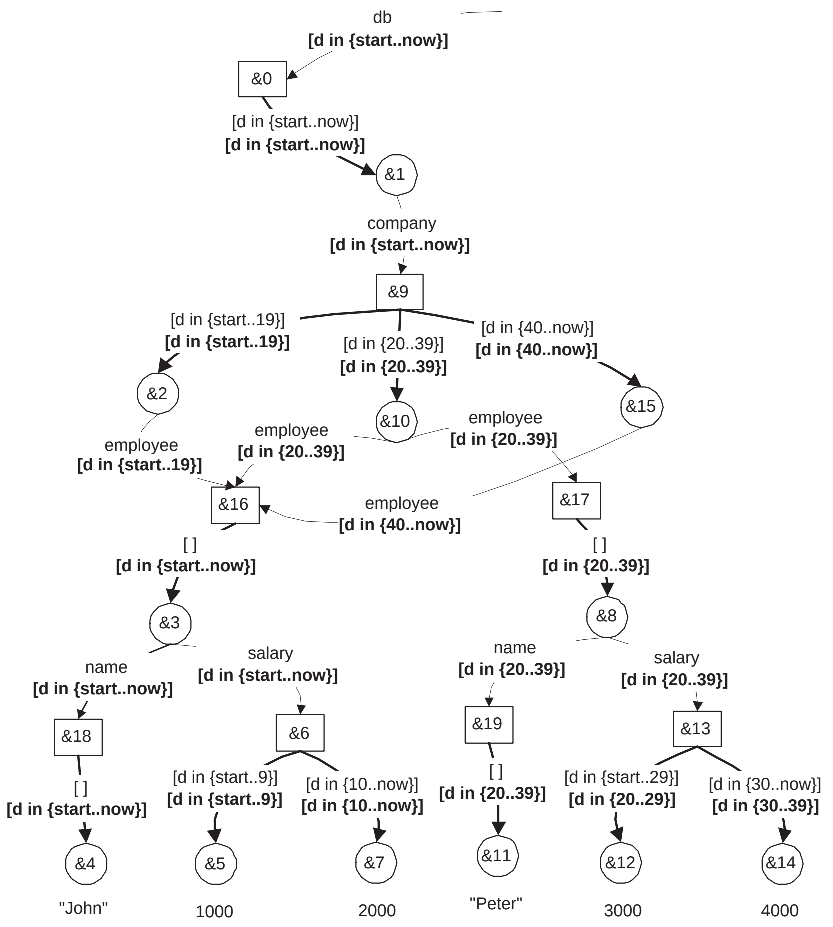


Fig. 8. The MOEM database of Figure 5 in canonical form, supplemented with the inherited coverages of edges.

It is important to note that the transformation of the underlying MOEM database to canonical form can take place at any stage during the evolution of an OEM database. The process specified in Section 4.1 for modeling OEM histories can be equally applied whether or not the underlying MOEM is in canonical form, with a single extension: to maintain the MOEM in canonical form, whenever a new context node is created as a result of $creNode(p, val)$, a multidimensional node and a context edge pointing to that context node are created as well.

Multidimensional entities in the database of Figure 8 model OEM objects that may have changed, and MQL queries on those entities can be interpreted as queries on the history of the corresponding OEM objects.

Getting the value of an OEM object at a specific time.

The query that follows returns the salary of `Peter` at time instance 32:

```
select salary: S
from [d=32]db.company.employee{X}.salary S
where X.name = "Peter"
```

The result of the query is the mssd-expression {`salary: &14 4000`}. Similarly to Lorel, it is possible in MQL to introduce object variables in the middle of context path expressions, by enclosing them in curly brackets. As an example, consider the object variable `X` above which binds to node `&3` and node `&8`. The context path expression `[d=32]db.company.employee.salary` is equivalent to the conventional path expression `db.company.employee.salary` evaluated on the conventional OEM that holds under time instance `32`. Therefore, the object variable `S` binds to the salary facet that exists in that OEM.

A similar query returns the name(s) of the employee(s) whose salary at time instance 25 was greater than or equal to 2500:

```
select name: Y
from [d=25]db.company.employee{X}.salary S,
      X.name Y
where S >= 2500
```

The result is {`name: &11 "Peter"`}.

Retrieving information concerning the history of an OEM object.

The query below returns `Peter`'s salaries, together with the periods they were valid:

```
select [Y]: S
from db.company.employee{X}.salary::[Y][-] S
where X.name = "Peter"
```

The context variable `[Y]` is an inherited coverage qualifier that binds to the path inherited coverage of context edges leading to salary facets, and is used to turn this into explicit context in the results:

```
<[d in {20..29}]: &12 3000,
 [d in {30..39}]: &14 4000>
```

The result graph consists of a multidimensional root from which a couple of context edges depart, leading to the context nodes `&12` and `&14`. Observe that the inherited coverage of the context edge leading to node `&12` in the database is `[d in {20..29}]`, and reflects the fact that `Peter` joined the company at the time instance 20. In a similar way, the inherited coverage of the context edge leading to node &14 in the database reflects the fact that Peter left the company at the time instance 40. Both inherited coverages appear as explicit contexts in the results.

Getting the values of an OEM object that were valid anytime within a period.

The following query looks for Peter's salaries in the period between 25 and 50.

```
select salary: S
from [Y]db.company.employee{X}.salary S
where X.name = "Peter"
within [Y] * [d in 25..50] != [-]
```

For every value of S, the context variable [Y] binds to a path inherited coverage, which represents the time instances under which OEMs hold, containing a path leading to this value of S. Then, [Y] is used in the within clause to filter out salary facets that do not hold under any of the time instances between 25 and 50. The symbol * stands for context intersection. The query returns:

```
{salary: &12 3000,
 salary: &14 4000}
```

Had we used a time instance greater than 29 instead of the time instance 25, only node &14 would have been returned. For the period between 40 and 50 no salary exists, because at time instance 40, Peter left the company.

Posing general questions on the history of the OEM database.

The query below returns the employees whose salary has not changed since time instance 32:

```
select employee_data: {name: Z, salary: S, valid_period: [Y]}
from db.#.employee{X}.salary::[Y][-] S,
      X.name Z
within [Y] >= [d in {32..now}]
```

The result is:

```
{employee_data: {name: &4 "John",
                 salary: &7 2000,
                 valid_period: "[d in {10..now}]"}
}
```

The query uses the wildcard # that matches zero or more entity edge / context edge pairs. The within clause checks whether the period under which a salary facet holds covers the whole period from 32 until the present time. Notice that node &14 is not included in the results, because the inherited coverage correctly indicates the fact that at the time instance 40 the salary ceased to exist.

Combining values and valid times of different OEM objects.

We can combine values and valid times of different OEM objects to formulate queries like "find the value(s) of an object while another object was having the specified value(s)".

The following query returns the salary(-ies) of John, while Peter's salary was 3000:

```
select john_salary: S1
```

```
from db.company X,
      X.employee{Y1}.salary::[Z1][-] S1,
      X.employee{Y2}.salary::[Z2][-] S2
where Y1.name = "John" and Y2.name = "Peter" and S2 = 3000
within [Z1] * [Z2] != [-]
```

The result of the query is {`john_salary: &7 2000`}.

Since the result of an MQL query is a Multidimensional Data Graph, we can apply the process of *reduction to OEM* to query results to get an OEM holding under a specific time instance. Moreover, we can apply the process of *partial reduction* for a set of time instances to get MOEM subgraphs that encompass all OEMs that hold under any of those time instances. Partial reduction can also be achieved using the keyword `holding` in the `select` clause of an MQL query.

7.3 Querying MOEM Histories

In addition to querying OEM histories, MQL can be used to query histories of MOEM databases. In this section we present some example queries on MOEM histories, focusing on the differences from querying OEM histories.

For our example queries we will use the MOEM depicted in Figure 9, which in fact is the MOEM of Figure 7(b) transformed to canonical form. Observe that the transformation to canonical form has introduced three new multidimensional nodes, namely `&60`, `&61` and `&62`, and has removed the multidimensional nodes `&4` and `&51`.

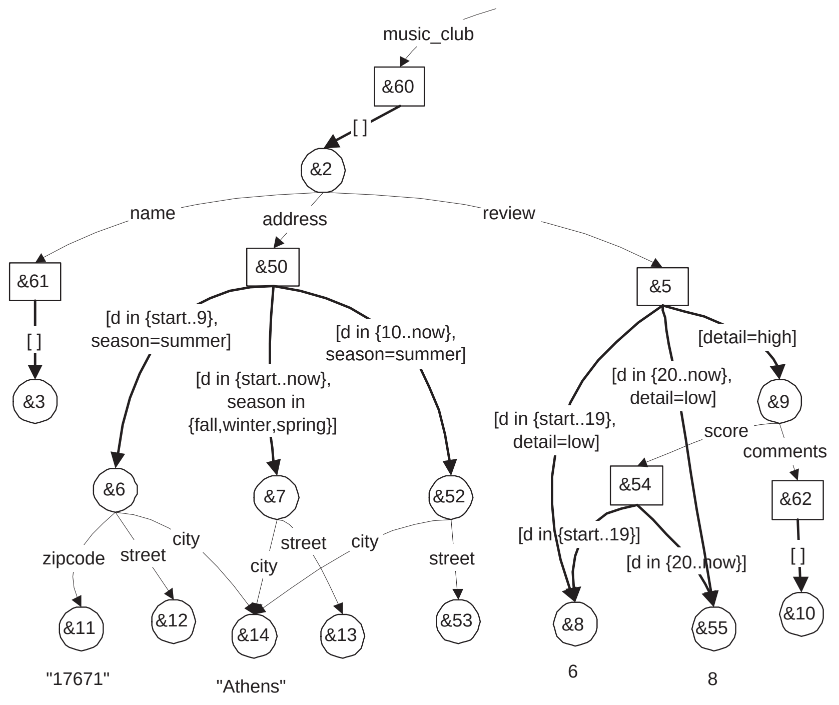


Fig. 9. The MOEM of Figure 7(b) in canonical form.

The following "plain vanilla" query returns the review of the music club at time instance `15` in low detail:

```
select review: X
from [d=15,detail=low]music_club.review X
```

The result is the mssd-expression {`review: &8 6`}.

This simple query specifies values for all dimensions related to `review`, and gets a `review` facet as a result. However, we may be interested in specifying constraints for some dimensions only. As we can see in Figure 9, when representing MOEM histories the dimension `d` is normally used together with other dimensions in context specifiers. We often want to "isolate" this dimension in queries and pose constraints to that dimension alone, irrespective of values of other dimensions. An elegant way to achieve that, is to use *context patterns* [Sta03], which allow to express contexts by specifying only some of the dimensions, while the rest can take any value.

The query below retrieves the music club address(es) at time instance `15`, irrespective of the values of other dimensions:

```
select address: X
from [% d=15]music_club.address X
```

The context pattern `[% d=15]` matches path inherited coverages that contain (at least) a world where `d` is `15`, and the query returns the objects `&7` and `&52`.

The next step elaborates on the previous query, and returns addresses together with the contexts under which those addresses hold. However, we are not interested in getting back dimension `d`, which we have already specified in the `from` clause:

```
select address: <[Z]: X>
context [Z] as [Y] * [% season in ALL]
from [% d=15]music_club.address::[Y][-] X
```

The result is:

```
{address: <[season in {fall,winter,spring}]: &7 {...},
           [season=summer]: &52 {...}> }
```

The contexts in the query results contain only the dimension `season`. This happens because the `context` clause uses the context pattern `[% season in ALL]` to screen out dimensions other than `season` in contexts bound to `[Y]`. The above query demonstrates the value and versatility of context patterns.

As another example, the following query retrieves the current address and the name of clubs, which at time instance 6 were located in Athens with zipcode 17671:

```
select club: {name: X.name, address: <[Q]: Z>}
from music_club X,
      X.[% d=6]address Y,
      X.[% d=now]address::[Q][-] Z
where Y.city = "Athens" and Y.zipcode = 17671
```

The query results comprise two current addresses for a club that at time instance 6 was located in Athens with zipcode 17671, one address for the summer and one for the rest of the year.

8 Related Work

The ability to model the temporal dimension of the real world is essential to many computer applications. Over the past two decades there has been a substantial amount of research on extending databases to support time [SA85,JCG$^+$92,ZCF$^+$97,OS95,TG95,BCW93,Org96]. In addition, some recent research investigates ways for incorporating time in data warehouses [MV00,EKK02,EK02], XML [GS98,AYU00,GM00,MGSI01,Dyr01,Nor02,Gra02,SD02], and semistructured data [CAW99,OQT01,DOQT01,GS98].

The problem of representing and querying changes in semistructured data has been studied in [CAW99], where *Delta OEM* (DOEM in short) was proposed. DOEM is a graph model that extends OEM with annotations containing temporal information. Four basic change operations, namely *creNode*, *updNode*, *addArc*, and *remArc* are considered by the authors in order to modify an

OEM graph. Those operations are mapped to four types of annotations. Annotations are tags attached to a node or an edge, containing information that encodes the history of changes for that node or edge. When an OEM basic change operation takes place, a new annotation is added to the affected node or edge, stating the type of the operation, the timestamp, and in the case of $updNode$ the old value of the object. The modifications suggested by the basic change operations actually take place, except from edge removal, which results to just annotating the edge. To query DOEM databases, the query language *Chorel* is proposed. Chorel extends *Lorel* [AQM+97] with constructs called *annotation expressions*, which are placed in path expressions and are matched against annotations in the DOEM graph. Annotation expressions may contain *time variables* that allow stating conditions on timestamps. Chorel deals with single time instances and time intervals by introducing special keywords in annotation expressions, like `at`, `during`, and `snap`.

Our approach is based on some of the key concepts presented in [CAW99]. It is however quite different, as changes are represented by introducing new facets instead of adding annotations. Similar to DOEM, MOEM can give the snapshot of an OEM database at any time instance, through the process of *reduction to OEM.* The principal difference between the query languages stems, as one would expect, from the different representations of OEM histories. Chorel, with its annotation expressions, maintains at a central role the notion of OEM basic change operations, which apply to a specific element and occur at a specific time instance. Then, starting from this basis, Chorel builds higher-level concepts, like for example an element holding under a time interval, or a complete path holding under a time instance. On the other hand, the notion of OEM basic change operations does not exist in MQL. MQL queries are addressed to the results of those operations, kept as facets of multidimensional entities. The central notion in MQL is context, which means that MQL directly handles the valid time intervals of database objects and database paths.

A special graph for modelling the dynamic aspects of semistructured data, called *semistructured temporal graph* is proposed in [OQT01]. In this graph, every node and edge has a label that includes a part stating the valid interval for the node or edge. Modifications in the graph cause changes in the temporal part of labels of affected nodes and edges.

An approach for representing temporal XML documents is proposed in [AYU00], where leaf data nodes can have alternative values, each holding under a time period. However, the model presented in [AYU00], does not explicitly support facets with varying structure for nodes that are not leaves. Other approaches to represent valid time in XML include [GM00,MGSI01,Gra02]. In [Dyr01] the XPath data model is extended to support transaction time. The query language of XPath is extended as well with transaction time axis to enable to access past and future states. Constructs that extract and compare times are also proposed. Finally, in [SD02] the XPath data model and query language is extended to include valid time, and XPath is extended with an axis to access valid time of nodes.

An important advantage of MSSD over other approaches for managing histories of semistructured data is that MSSD can be applied to a variety of problems from different fields; representing time is just one of the possible applications of MSSD. MOEM is suitable for modelling entities that present different facets, a problem often encountered on the Web, and the representation of semistructured database histories can be seen as a special case of this problem.

Properties and processes defined for the general case of MSSD (like path inherited coverage, reduction to OEM, partial reduction) are also used without change in the case of representing semistructured histories. Similarly, MQL is not made especially for querying histories of semistructured databases, but is targeted at the general case of context-dependent semistructured data. Therefore, the concepts we proposed as part of MOEM and MQL have a wider applicability, and are not confined in the frame of a specific problem. Moreover, as we showed in Sections 6 and 7.3, MOEM and MQL are capable of representing and querying MOEM's own histories, without having to introduce any additional concepts.

9 Conclusions

In this paper we explained in depth how Multidimensional OEM, a graph model for context-dependent semistructured data, can be used to represent the history of an OEM database. We introduced the basic change operations of MOEM, and specified in pseudocode the process that encodes in an MOEM the history of an OEM database. We discussed how Multidimensional Data Graph properties apply in this particular case, and showed that temporal OEM snapshots can be obtained from MOEM. We presented OEM History, an implemented system, and demonstrated through an example the process of using an underlying MOEM database to model changes in OEM. In addition, we showed that MOEM is capable to model changes occurring not only in conventional OEM databases, but in MOEM databases as well.

An important benefit of our approach stems from the queries that can be posed on OEM histories and MOEM histories. Our approach maintains snapshots of the successive states of an object as facets of a multidimensional entity. This allows the formulation of queries that relate different states of the same or different object(s) at various times. We presented a number of MQL query examples, and through them we confirmed the expressiveness of MQL and the value of some of its features, like for example context patterns.

In our view, this example application of MOEM and MQL demonstrates the potential of MSSD. The applicability of MOEM and MQL is not exhausted in representing histories of semistructured data; context-dependent data are of increasing importance in a global environment such as the Web. We believe that MSSD can be used in a variety of fields, among which: in information integration, for modelling objects whose value or structure vary according to sources; in digital libraries, for representing metadata that conform to similar formats; in representing and querying geographical information, where possible dimensions are *scale* and *theme*.

References

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.

[AQM$^+$97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[AYU00] T. Amagasa, M. Yoshikawa, and S. Uemura. A Data Model for

Temporal XML Documents. In M. T. Ibrahim, J. Kung, and N. Revell, editors, *Database and Expert Systems Applications, 11th International Conference (DEXA 2000), London, UK, Sept. 4-8, Proceedings*, Lecture Notes in Computer Science (LNCS) 1873, pages 334–344. Springer-Verlag, Sept. 2000.

[BCW93] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases. In L. Fariñas del Cerro and M. Penttonen, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 294–320. The Benjamin/Cummings Publishing Company, Inc, 1993.

[BFS00] P. Buneman, M. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[CAW99] S. S. Chawathe, S. Abiteboul, and J. Widom. Managing Historical Semistructured Data. *Theory and Practice of Object Systems*, 24(4):1–20, 1999.

[DOQT01] E. Damiani, B. Oliboni, E. Quintarelli, and L. Tanca. Modeling Users' Navigation History. In *Proceedings of the Workshop on Intelligent Techniques for Web Personalization, Seatle, Washington, USA, August 4-6, 2001*, 2001.

[Dyr01] Curtis E. Dyreson. Observing Transaction-time Semantics with TTXPath. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering (WISE 2001), Kyoto, Japan, Dec. 2001*, pages 193–202. IEEE Computer Press, 2001.

[EK02] J. Eder and C. Koncilia. Representing Knowledge about Changes in Data Warehouse Structures. In *Proceedings of the 3rd International Workshop on Knowledge Management in Electronic Government (KMGov'02), Copenhagen, Denmark*, May 2002.

[EKK02] J. Eder, C. Koncilia, and H. Kogler. Temporal Data Warehousing: Business Cases and Solutions. In *Proceedings of the International Conference on Enterprise Information Systems (ICEIS'02), Cindad, Real, Spain*, pages 81–88, June 2002.

[GM00] F. Grandi and F. Mandreoli. The Valid Web: an XML/XSL Infrastructure for Temporal Management of Web Documents. In T. M. Yakhno, editor, *Advances in Information Systems. First International Conference (ADVIS'02), Izmir, Turkey, 25-27 October*, pages 294–303, 2000.

[Gra02] Fabio Grandi. XML Representation and Management of Temporal Information for the Web-Based Cultural Heritage Applications. *Data Science Journal*, 1(1):68–83, 2002.

[GS98] F. Grandi and M. R. Scalas. Extending Temporal Database Concepts to the World Wide Web. In *Proceedings of SEBD'98, National Conference on Advanced Database Systems, Italy, June 1998*, pages 53–70, 1998.

[GSK01a] M. Gergatsoulis, Y. Stavrakas, and D. Karteris. Incorporating Dimensions to XML and DTD. In H. C. Mayr, J. Lanzanski, G. Quirchmayr, and P. Vogel, editors, Database and Expert Systems Applications (DEXA' 01), Munich, Germany, September 2001, Proceedings, Lecture Notes in Computer Science (LNCS), Vol. 2113, pages 646–656. Springer-Verlag, 2001.

[GSK$^+$01b] M. Gergatsoulis, Y. Stavrakas, D. Karteris, A. Mouzaki, and D. Sterpis. A Web-based System for Handling Multidimensional Information through MXML. In A. Kaplinskas and J. Eder, editors, Advances in Databases and Information Systems (ADBIS' 01), Proceedings, Lecture Notes in Computer Science (LNCS), Vol. 2151, pages 352–365. Springer-Verlag, 2001.

[JCG$^+$92] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. A Glossary of Temporal Database Concepts. *SIGMOD RECORD*, 21(3):35–43, September 1992.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, September 1997.

[MGSI01] T. Mitakos, M. Gergatsoulis, Y. Stavrakas, and E. V. Ioannidis. Representing Time-Dependent Information in Multidimensional XML. *Journal of Computing and Information Technology*, 9(3):233–238, 2001.

[MV00] A. O. Mendelzon and A. A Vaisman. Temporal Queries in OLAP. In A. L. Abbadi, M. L. Brodie, S. Chacravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)*, pages 242–253. Morgan Kaufmann, 2000.

[Nor02] Kjetil Norvag. Algorithms for Temporal Query Operators in XML Databases. In *Proceedings of EDBT Workshops*, Lecture Notes in Computer Science (LNCS) Vol 2490, pages 169–183. Springer-Verlag, 2002.

[OEM] OEM History. The application can be downloaded from `http://www.dblab.ntua.gr/∼ys/moem/moem.html`.

[OQT01] B. Oliboni, E. Quintarelli, and L. Tanca. Temporal Aspects of Semistructured Data. In *Proc. of the 8th International Symposium on Temporal Representation and Reasoning (TIME-01)*, pages 119–127, 2001.

[Org96] M. A. Orgun. On Temporal Deductive Databases. *Computational Intelligence*, 12(2):235–259, 1996.

[OS95] G. Ozsoyoglu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

[SA85] R. Snodgrass and Ilsoo Ahn. A Taxonomy of Time in Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 236–246, 1985.

[SD02] S. Shang and C. E. Dyreson. Adding Valid Time to XPath. In *Database and Network Information Systems, Proceedings of DNIS 2002*, Lecture Notes in Computer Science (LNCS), Vol. 2544, pages 29–42. Springer-Verlag, 2002.

[SG02] Y. Stavrakas and M. Gergatsoulis. Multidimensional Semistructured Data: Representing Context-Dependent Information on the Web. In A. B. Pidduck, J. Mylopoulos, C. Woo, and T. Oszu, editors, *Advanced Information Systems Engineering, 14th International Conference (CAiSE'02), Toronto, Ontario, Canada, May 2002. Proceedings.*, Lecture Notes in Computer Science (LNCS), Vol. 2348, pages 183–199. Springer-Verlag, 2002.

[SGDZ02] Y. Stavrakas, M. Gergatsoulis, C. Doulkeridis, and V. Zafeiris. Accomodating Changes in Semistructured Databases Using Multidimensional OEM. In Y. Manolopoulos and P. Navat, editors, Advances in Databases and Information Systems (ADBIS' 02), Proceedings, Lecture Notes in Computer Science (LNCS), Vol. 2435, pages 360–373. Springer-Verlag, 2002.

[SPES03] Y. Stavrakas, K. Pristouris, A. Efantis, and T. Sellis. Implementing a Query Language for Context-dependent Semistructured Data. Submitted for publication, 2003.

[Sta03] Y. Stavrakas. *Multidimensional Semistructured Data: Representing and Querying Context-Dependent Multifacet Information on the Web*. PhD Thesis, Department of Electrical and Computer Engineering, National Technical University of Athens, Greece, June 2003.

[Suc98] D. Suciu. An Overview of Semistructured Data. *SIGACT News*, 29(4):28–38, December 1998.

[TG95] V. J. Tsotras and B. Gopinath. Efficient Management of Time-Evolving Databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):591–607, August 1995.

[ZCF$^+$97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.