Accommodating Changes in Semistructured Databases Using Multidimensional OEM

Yannis Stavrakas[1,2], Manolis Gergatsoulis[2], Christos Doulkeridis[1], and Vassilis Zafeiris[1]

[1] Knowledge & Database Systems Laboratory
National Technical University of Athens (NTUA), 15773 Athens, Greece.
[2] Institute of Informatics & Telecommunications,
N.C.S.R. 'Demokritos', 15310 Aghia Paraskevi Attikis, Greece.
{ystavr,manolis}@iit.demokritos.gr
{cdoulk,bzafiris}@aueb.gr

Abstract. *Multidimensional Semistructured Data* (MSSD) are semistructured data that present different facets under different *contexts* (sets of *worlds*). The notion of context has been incorporated in 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 OEM databases. 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.

1 Introduction and Preliminaries

In this paper we investigate the use of *Multidimensional Object Exchange Model* (*Multidimensional OEM* or *MOEM*) for representing histories of semistructured databases. We start with an introduction to Multidimensional OEM, we explain in detail the way it can be used to model histories of OEM databases, we present an example scenario using our prototype implementation, and we show that Multidimensional OEM can be used to model its own histories as well.

Multidimensional semistructured data (MSSD) [9] are semistructured data [10, 1] 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 used to qualify semistructured data expressions (*ssd-expressions*) [1] and specify sets of *worlds* under which the corresponding ssd-expressions 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 multidimentional entity may differ in value and/or structure, and can in turn be multidimensional entities or conventional information. Each facet is associated with a context that defines the conditions under which the facet becomes a *holding facet* of the multidimensional entity.

A way of encoding MSSD is *Multidimensional XML* (MXML in short) [5, 6], 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 [6].

1.1 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 whose elements are pairs (d, v), where $d \in \mathcal{D}$ and $v \in \mathcal{V}_d$, such that for every dimension name in $\mathcal{D}$ there is exactly one element in w.*

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

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

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`.

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. When two context specifiers represent disjoint sets of worlds they are said to be *mutually exclusive*. The context specifier `[]` is called *universal context* and represents the set of all possible worlds with respect to any set of dimensions $\mathcal{D}$. In [9] we have defined operations on context specifiers, such as *context intersection* and *context union*, and showed how a context specifier can be transformed to the set of worlds it represents w.r.t. a set of dimensions $\mathcal{D}$.

1.2 Multidimensional OEM

Multidimensional Object Exchange Model (*MOEM*) [9] is an extension of *Object Exchange Model* (OEM) [2], suitable for representing multidimensional semistructured data. MOEM 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, playing the role of surrogates for these facets. Multidimensional nodes have a rectangular shape to distinguish them from conventional circular nodes, which are called *context nodes* and represent facets associated with some context.
– *Context edges*: are directed labeled edges that connect multidimensional nodes to their facets. The label of a context edge pointing to a facet p, is a context specifier defining the set of worlds under which p holds. Context edges are drawn as thick lines, while conventional (thin-lined) OEM edges are called *entity edges* and define relationships between objects.

Both multidimensional and context nodes are considered objects and have unique *object identifiers* (oids). 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.

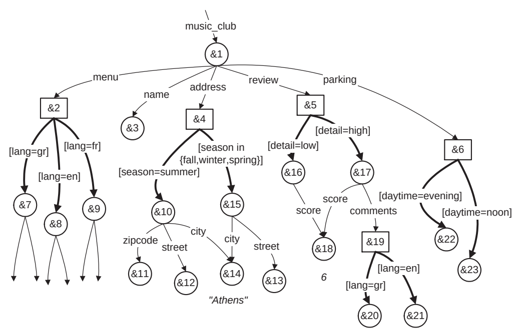


Fig. 1. A multidimensional music-club.

As an example, consider the fragment of an MOEM graph, shown in Figure 1, which represents context-dependent information about a music-club. Notice that 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 from the city center to the vincinity of the sea in the summer). 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 notion of *multidimensional data graph* is formally defined as follows.

Definition 2. *Let $\mathcal{C}$ 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 *is a finite directed edge-labeled multigraph $G = (V, E, r, \mathcal{C}, \mathcal{L}, \mathcal{A}, v)$, where: (1) The set of nodes V*

is partitioned into 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 is partitioned into* context edges *and* entity edges $E = E_{cxt} \cup E_{ett}$*, such that* $E_{cxt} \subseteq V_{mld} \times \mathcal{C} \times V$ *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 assigning 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* $v' : V_a \rightarrow \mathcal{A}$ *assigning values to atomic nodes.*

An MOEM graph is a *context deterministic* multidimensional data graph, that is, the context edges departing from the same multidimensional node have mutually exclusive context specifiers. Two basic concepts related to MOEM graphs are the *explicit context* and the *inherited context*. The explicit context of a context edge is the context specifier assigned to that edge, while the explicit context of an entity edge is 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 an MOEM graph, the explicit context of an edge does not alone determine the worlds under which the destination node holds. The reason 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 (constraint by) the explicit context of each edge to give the *inherited context* for that edge. In contrast to edges, nodes do not have an explicit context; like edges, they do have inherited contexts. The inherited context of a node/edge gives the set of worlds under which the node/edge is taken into account, when reducing the MOEM graph to a conventional OEM graph.

Given a specific world, we can always *reduce* an MOEM graph to a conventional OEM graph holding under that world, using a *reduction* procedure given in [9]. Moreover, it is also possible to *partially reduce* an MOEM into a new MOEM, that encompasses only the OEM facets for the given set of worlds.

2 Representing Histories of OEM Databases

MOEM can be used to represent changes in OEM databases. The problem is the following: given a static OEM graph that comprises the database, we would like a way to represent dynamically changes in the database as they occur, keeping a history of transitions, so that we are able to subsequently query on those changes. In [9] we outlined some preliminary ideas towards a method for modeling OEM histories, and showed that it is feasible to model such histories through MOEM. In this section we further extend those ideas and present the method in detail: we give specific algorithms, and discuss how MOEM properties are applied.

The problem of representing and querying changes in semistructured data has also been studied in [4], where *Delta OEM* (*DOEM* in short), a graph model that extends OEM with *annotations* containing temporal information, has been proposed. 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 arc, containing information that encodes the history of changes for that node or arc. When a basic operation takes place, a new annotation is added to the affected node or arc, 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 the arc removal which results to just annotating the arc. Our approach, although it builds on the key concepts presented in [4], is quite different, as changes are represented by introducing new facets instead of adding annotations.

A special graph for modeling the dynamic aspects of semistructured data, called *semistructured temporal graph* is proposed in [8]. 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 [3], where leaf data nodes can have alternative values, each holding under a time period. However, the model presented in [3] does not allow dimensions other than time, and does not explicitly support facets with varying structure for nodes that are not leaves. Another approach for representing time in XML documents is described in [7], where the use of Multidimensional XML is suggested.

An important advantage of MOEM over those approaches is that a single model can be applied to a variety of problems from different fields; representing valid time is just one of its possible applications. MOEM is suitable for modeling entities presenting different facets, a problem often encountered on the Web. 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 MOEM, like inherited context, reduction, and querying are also used without change in the case of representing semistructured histories. In addition, as shown in section 4, MOEM is a model capable of representing its own histories.

2.1 OEM and MOEM Basic Change Operations

OEM graph is defined in [2] as a quadruple $O = (V, E, r, v)$, where V is a set of nodes, E is a set of labeled directed edges (p, l, q) where $p, q \in V$ and l is 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 (int, string, etc.), or to the reserved value C denoting a complex object. In order to modify an OEM database, four basic change operations were identified in [4]:

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 arcs.

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

remArc(*p*, *l*, *q*): removes the existing arc (p, l, q). Both p and q must exist.

Given an MOEM database $M = (V, E, r, \mathcal{C}, \mathcal{L}, \mathcal{A}, v)$, we introduce the following basic operations for changing 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*, *context*, *id*): creates a new context edge with context *context* from node *mid* to node *id*, where $mid \in V_{mld}$ and $id \in V$.

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

For both OEM and MOEM, object deletion is achieved through arc removal, since 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 to L as defined in [4], 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.

2.2 Using MOEM to Model OEM Histories

The basic MOEM operations defined in section 2.1 can be used to represent changes in an OEM database using MOEM. 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, that uses a dimension d whose domain is time to represent an OEM history H valid [4] for O. We assume that our time domain T is linear and discrete; we also assume: (1) a reserved value *now*, such that $t < now$ for every $t \in T$, (2) a reserved value *start*, representing the start of time, and (3) a syntactic shorthand $v_1..v_n$ for discrete and totally ordered domains, meaning all values v_i such that $v_1 \leq v_i \leq v_n$. 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` $\{t_1\,..\,t_2\}$`]`.

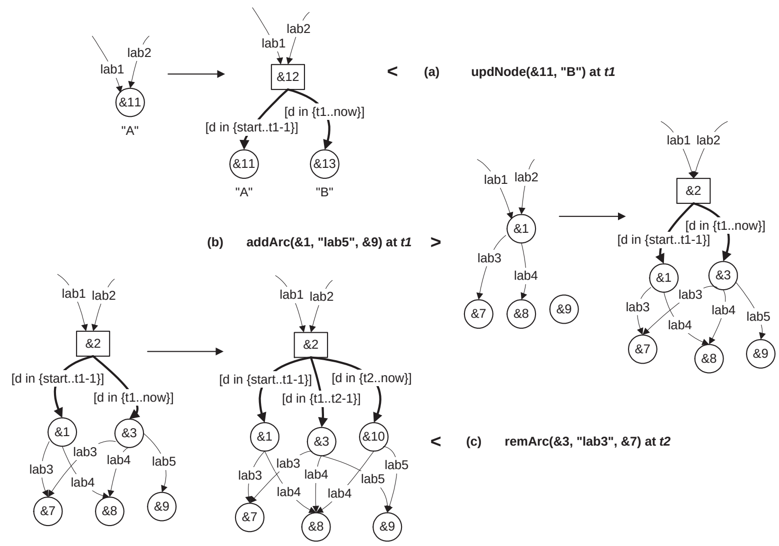


Fig. 2. Modeling OEM basic change operations with MOEM.

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* operation 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) through *addArc* operation(s), which will cause new object facet(s) to be created. Note that, although object identifiers in Figure 2 may change during the OEM history, 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.

id1 ← 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 $(mid, context, id)$ 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. Notice 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 $(mid, context, cid) \in E_{cxt}$. However, this is a property of MOEM graphs constructed for representing OEM histories, and not of MOEM graphs in general.

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

The 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. The procedure alters the graph, but not the information modeled by the graph: the multidimensional node mid has 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_pln {
           addEEdge(x, l, mid)
           remEEdge(x, l, id) } }
```

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. 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}]`.

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 arc 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_pln
               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.

2.3 Applying MOEM Properties

MOEM graphs that represent OEM histories have special characteristics, not generally encountered in MOEM graphs, which affect the MOEM properties of inherited context, and reduction to conventional OEMs.

Let G be a multidimensional data graph produced by the process specified in section 2.2, let e be a multidimensional entity in G, with multidimensional node

m and facets e_1,e_2,...,e_n, and let c_1,c_2,...,c_n be the context specifiers of the respective context edges. Notice that, as already stated, the process in section 2.2 guarantees that at most one multidimensional node points to any context node. In addition, in the case of representing an OEM history, worlds are time instances. It is easy to observe that G is context deterministic, because for every multidimensional entity e in G, the contexts c_1,c_2,...,c_n always define disjoint sets of worlds, thus for any given time instance at most one of e_1,e_2,...,e_n may hold. Consequently, G is an MOEM graph, and the reduction process (defined in [9]) will always give an OEM graph, for any time instance in T.

In addition, from the procedures *mEntity* and *newCxt* defined in section 2.2, it can be seen that: (a) c_1 has the form `[d in {start..`*`somevalue1`*`}]`, (b) c_n has the form `[d in {`*`somevalueN`*`..now}]`, and (c) the union of the context specifiers c_1,c_2,...,c_n can be represented by `[d in {start..now}]`, for every e in G. Although for every multidimensional entity e in G the corresponding context specifiers c_1,c_2,...,c_n cover the complete `{start..now}` time range, the corresponding inherited contexts denote the true life span of the entity and its facets. To understand why, note that 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 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 e_1,e_2,...,e_n to "survive" under some world. Therefore, for every 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..`*`somevalue1`*`}]`, and c_n =`[d in {`*`somevalueN`*`..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 understand the result of applying MOEM reduction to G. Given an OEM database O and an MOEM database G that represents the history of O, it is possible to specify a time instance t and reduce G to an OEM database O'. Then O' will be the snapshot of O at the time instance t.

3 OEM History

OEM History is an application developed in Java, which implements the method described in Section 2 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 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), that 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, which can be used only on the window with the OEM depicting the current state of the database. These operations are mapped to a number of 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 for $d = \texttt{now}$.

Note that the "tick" button in the left toolbar removes nodes that are not accessible from the root, while the last button 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 and MOEM validity check 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 $d = \texttt{now}$.

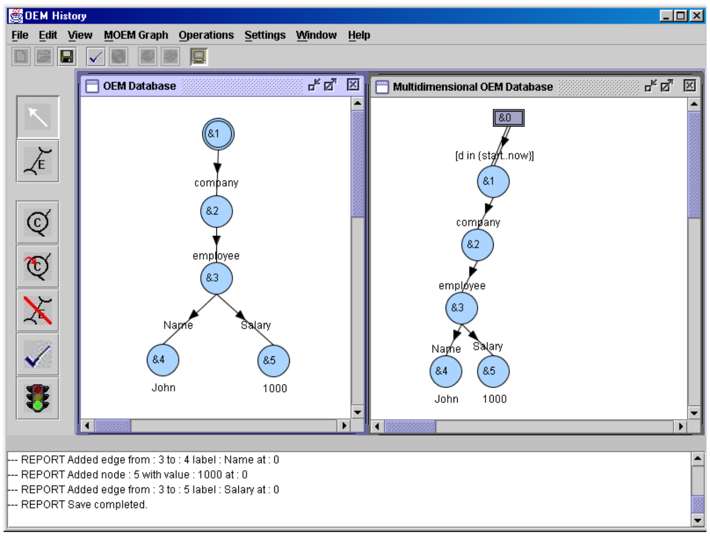
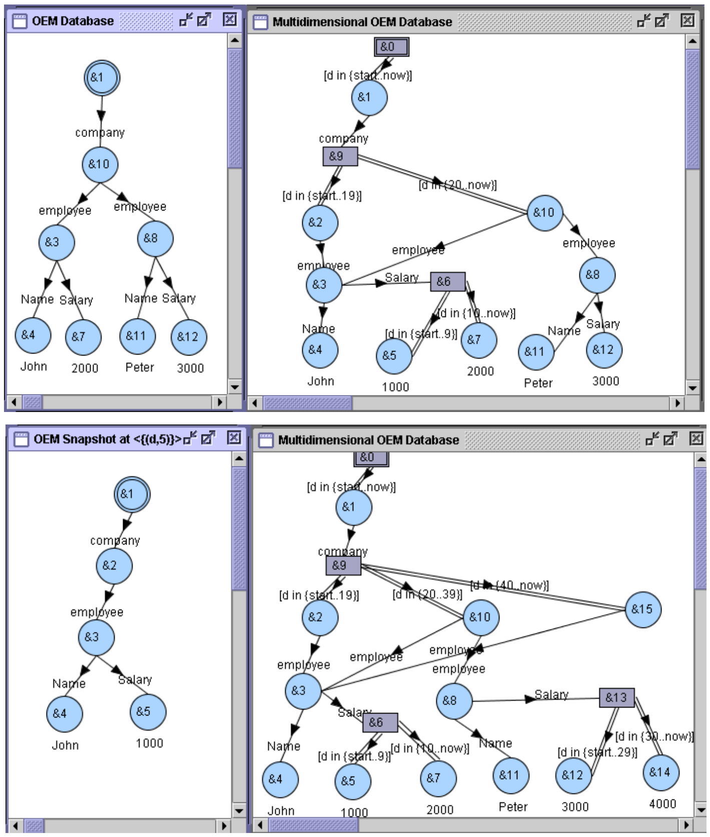


Fig. 3. Initial state of example database in *OEM History* application.

Figure 4 (a) 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 `20` a new employee called `Peter` joined the company with salary `3000`.

In Figure 4 (b) two more change sequences have been applied. The salary of `Peter` increased to `4000` at the time instance `30`, and at the time instance `40` `Peter` left the company. 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 from reducing the MOEM in the right window for $d = \texttt{5}$. That snapshot is identical to the initial state of the database, since the first change occurred at the time instance `10`.



(a)

(b)

Fig. 4. Example database after **(a)** two sequences of basic changes, and **(b)** four sequences of basic changes upon the initial database state.

OEM History is available at: `http://www.dblab.ntua.gr/~ys/moem/moem.html`

4 Representing Histories of MOEM Databases

Besides representing OEM histories, MOEM is expressive enough to model its own histories. That is, for any MOEM database G evolving over time we can construct an MOEM database G', which represents the history of G. The approach is similar to that of section 2.2; we show that each MOEM basic operation applied to G, can be mapped to a number of MOEM basic operations on G', in such a way that G' represents the history of G. Figure 5 gives the intuition about this mapping, for three basic operations. Context edge labels $c1$,$c2$,...,cN are context specifiers involving any number of dimensions, as in example of Figure 1, while the dimension d is defined in section 2.2. Note that the use of dimension d in G' does not preclude G from using other dimensions ranging over time domains. The MOEM operations depicted in Figure 5 are basic operations occurring on

G, and the corresponding graphs show how those operations transform *G'*. For simplicity, graphs on the left side do not contain context specifiers with the dimension *d*, and all timestamps are *t*1. 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.

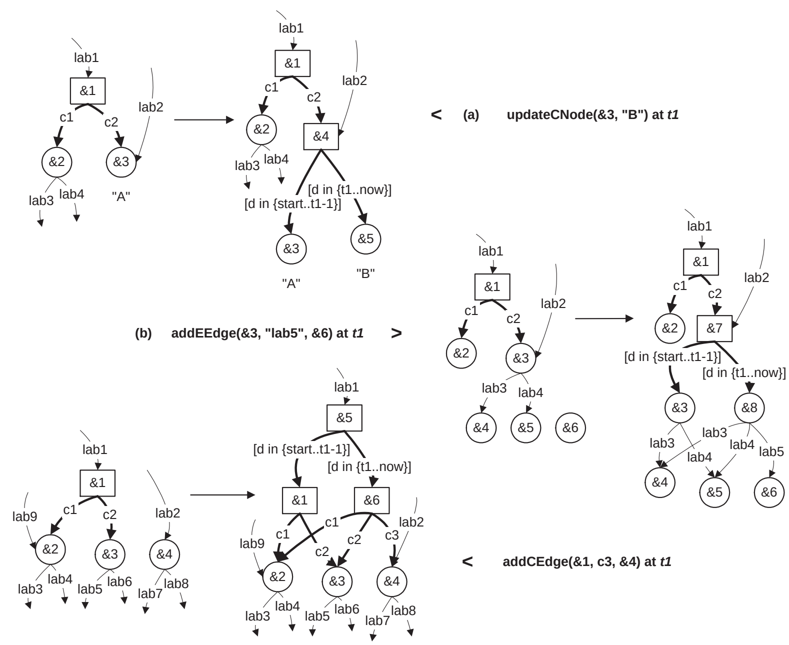


Fig. 5. Modeling Multidimensional OEM basic operations with MOEM.

Figure 5(a) shows a facet with id &3 whose value is changed from "A" to "B" through a call to *updateCNode*. Figure 5(b) shows the result of an *addEEdge* operation. Finally, figure 5(c) depicts the *addCEdge* basic operation. Among MOEM basic operations *not* shown in Figure 5, *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; *G'* will record the change when the new nodes are connected to the rest of the graph *G* through calls to *addEEdge* or *addCEdge*.

An MOEM graph *G'* constructed through the process outlined above represents the history of the MOEM graph *G*. In contrast to the case of OEM histories, where a world is defined by only one dimension *d* representing time, in the case of MOEM histories a world for *G'* in general involves more than one dimensions, including the time dimension *d*. Therefore, by specifying a value *t* for d 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. The process of reducing an MOEM graph under a set of worlds, instead of under a single world, is called *partial reduction* and, as with full reduction, involves intersecting the given set of worlds with those represented by the inherited contexts of edges and nodes in the graph. Therefore, by applying the process of partial reduction to G' for any time instance $t \in T$, G' gives the snapshot of the MOEM database G at that time instance.

5 Future Work

Context-dependent data are of increasing importance in a global environment such as the Web. We have implemented a set of tools for MSSD, which we used to develop the OEM History application. We continue extending this infrastructure that will facilitate the implementation of new MSSD and MOEM applications. Our current work is focused on the implementation of *MQL*, a multidimensional query language.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.
3. T. Amagasa, M. Yoshikawa, and S. Uemura. A Data Model for Temporal XML Documents. In M. Ibrahim, J. Kung, and N. Revell (eds), *Database and Expert Systems Applications (DEXA'00)*, LNCS 1873, pages 334–344. Springer, 2000.
4. S. S. Chawathe, S. Abiteboul, and J. Widom. Managing Historical Semistructured Data. *Theory and Practice of Object Systems*, 24(4):1–20, 1999.
5. 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), Proceedings, Lecture Notes in Computer Science, Vol. 2113, pages 646–656. Springer-Verlag, 2001.
6. 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), LNCS, Vol. 2151, pages 352–365. Springer-Verlag, 2001.
7. 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.
8. 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.
9. Y. Stavrakas and M. Gergatsoulis. Multidimensional Semistructured Data: Representing Context-dependent Information on the Web. In *Proc. of the 14th Int. Conf. on Advanced Information Systems Engineering, Toronto, Canada, May 2002*.
10. D. Suciu. An Overview of Semistructured Data. *SIGACT News*, 29(4):28–38, December 1998.