Using Structured Changes for Elucidating Data Evolution

Yannis Stavrakas[1], George Papastefanatos[2]

Institute for the Management of Information Systems
Mpakou 17, Athens 11524, Greece
[1]yannis@imis.athena-innovation.gr
[2]gpapas@imis.athena-innovation.gr

Abstract— **In this paper we argue that changes should be treated as first class citizens in data management systems. In our approach, changes are not just transformation operations but complex objects retaining structural, semantic, and temporal characteristics. We believe that accommodating structured changes in information modeling and querying can provide users with new insights into data lifecycle. In previous work we proposed a graph model called *evo-graph* for capturing in a coherent way the relationships between evolving data and changes applied on them. We also presented *evo-path*, a path expression language for evo-graphs based on XPath. In the present paper we define an XML representation of evo-graph, and discuss the use of XQuery for expressing a number of interesting query categories. We demonstrate the feasibility and usefulness of our approach through an example inspired by bioscientific databanks.**

I. INTRODUCTION

The wide availability and fast publishing of information enabled by the Web unlocks new potential and new problems for data management. Particularly, an emerging issue concerns collections of Web data (often scientific) that evolve independently, but remain in many ways interconnected. Important interconnections may correlate data at different points in their lifecycle, forming a data space where past and current versions coexist and reference each other.

Consider, for example, biology research communities [2],[11] that produce, consume, and archive rapidly large amounts of data. Scientific communities like that rely increasingly on the Web for the publication and integration of experimental and research results. Moreover, scientists in those communities would often like to review how and why the recorded data have evolved, in order to compare and re-evaluate previous and current conclusions. Such an activity may require a search that moves backwards and forwards in time, spreads across various databanks, and performs complex queries on the semantics of the changes that modified the data. In those cases, issues of evolution and provenance become closely related, since evolution information is needed in order to answer provenance queries; simply revising past document snapshots and differences between versions is not enough.

So far significant work has been done on evolution [5],[8],[14] and provenance [4] of XML and semistructured data. However, previous approaches do not cover all the aspects of the problem, since each of them focuses on the specific questions regarding the framework it addresses.

In this paper we argue that in systems where evolution and provenance issues are paramount, changes should not be treated solely as transformation operations on the data, but rather as first class citizens retaining structural, semantic, and temporal characteristics. Modeling such complex changes explicitly can leverage a number of new interesting queries, and provide additional semantic information for elucidating data evolution and interpreting past data.

In previous work [21] we have proposed a graph model called *evo-graph* for capturing the relationship between evolving data and changes applied on them. We employed this model for representing simple as well as composite evolution operations. Given a time instance, evo-graph can be reduced to the past snapshot of the data holding under that instance. *Evo-path* [21] is a path expression language for evo-graph that extends XPath. Evo-path takes advantage of the complex changes in the evo-graph in order to answer queries about the interpretation of data evolution and the provenance of data.

XML is the standard format for Web data, therefore it is natural to investigate the use of XML technologies for representing and querying evo-graphs. In the present work we introduce *evoXML*, an XML representation of evo-graph that follows a non-replicated approach to transform graph structures. We also demonstrate the use of XQuery for a number of interesting categories of queries. Each query is expressed in both evo-path and XQuery, in order to expose their relative expressiveness and suitability in the frame of our context. We employ a scenario inspired by bioscientific data to motivate the use of evo-graph and evoXML, and to pose example queries with evo-path and XQuery.

The structure of the paper is as follows. In Section II we discuss related work. In Section III we present evo-graph, give a motivating example, and discuss time propagation and snapshot reduction. In Section IV we introduce evoXML, an XML representation of evo-graph. In Section V we give a number of example queries expressed both in evo-path and XQuery, and provide a relevant discussion. Finally, Section VI concludes the paper.

II. RELATED WORK

Modeling and managing evolving Web data has attracted a growing interest in the database research community. Most approaches address the problem of storing efficiently complex and highly evolving data objects, while at the same time providing querying and browsing capabilities over the

versioning schemes, semantics, and relationships that characterize these objects. In one of the early works [5], the authors deal with the representation of changes in semistructured data, and propose DOEM, where evolution operations are expressed as multiple annotations on the nodes and the edges of the graph. In [12] a change-centric method for managing versions in XML data is presented. The authors employ a *diff* algorithm for detecting changes between two versions of a document, and store the last version plus a sequence of deltas. A similar approach is introduced in [6],[7], where instead of deltas calculations, a referenced-based identification of each object is used across different versions. New versions hold only the elements that are different from the previous version whereas a reference is used for pointing to the unchanged elements of past versions. In [10] the authors propose MXML, an extension of XML that uses context information to express time and models multifaceted documents. Recently, there are works that deal with modeling [16] and detection [14] of changes in semantic data, in which the problems of change identification, modeling of temporal properties, and versioning are applied to ontologies and RDF.

Most approaches dealing with evolution and versioning of semistructured data employ temporal extensions for expressing the lifespan of the different versions. For example, in [1],[5] they enrich data elements with temporal attributes for holding valid and / or transaction time, and extend query syntax with conditions on the time validity of data [8]. In [15], a temporal model for XML is introduced, which attaches transaction time information on the edges of the XML graph. In [9] the authors propose a temporal query language for adding valid time support in XQuery. In [17] the notion of a temporally grouped data model is employed for uniformly representing and querying successive versions of a document. In a more recent work [13], the authors extend this technique for publishing the history of a relational database in XML. Lastly, an interesting technique is presented in [3], where the problem of archiving curated databases is addressed. The authors develop an archiving technique for scientific data that uses timestamps for each version, whereas all versions are merged into one hierarchy.

Compared to the above approaches, our model introduces a change-based perspective for evolving data, in which changes are not derived by data versions but are modelled as first class citizens. Changes are not described through diffs or transformations with edit scripts between document versions, but are complex objects operating on data, and exhibit structural, semantic, and temporal properties: they can be part of other changes, correlate to each other, be transactional, long-termed or instant. Thus, they are mapped to separate elements in the proposed XML representation. Such modeling allows to answer queries about "what" has evolved over time, but also to provide information about "why" and "how" data has evolved.

III. Representing Evolution with Evo-Graph

In this section we discuss *evo-graph*, a graph model for evolving semistructured data where changes are given equal importance as data objects. We give an example of using evo-graph in a bioscientific scenario, and outline some temporal properties of evo-graphs.

A. Modeling Complex Changes

A number of data models have been proposed in the past for semistructured data and XML [5],[19]. In general, those models represent data using trees or directed graphs with values on the leaves. We will use the term *snap-model* for referring to a model that represents data at a single time instance. In the frame of this paper the snap-model is assumed to be a tree, consisting of labeled data nodes (complex and atomic), and edges connecting the nodes. Evo-graph uses similar node and edge components under the following terminology:

- *Data nodes* correspond to the nodes of the snap-model (depicted as circles). They are divided into *complex* and *atomic*: $V_D = V_D^c \cup V_D^a$.
- *Data edges* depart from every complex data node, $E_D \subseteq (V_D^c \times V_D)$.

In addition, we introduce the following new concepts in evo-graph:

- *Change nodes* are nodes that represent change events: basic change operations, and complex changes. Change nodes are depicted as triangles, to distinguish from circular data nodes. They are also divided into *complex* and *atomic*: $V_C = V_C^c \cup V_C^a$.
- *Change edges* connect every complex change node to the (complex or atomic) change nodes it consists of: $E_C \subseteq (V_C^c \times V_C)$. Change edges are depicted as dashed lines.
- *Evolution edges* are edges that connect each change node with two data nodes, the object version before the change and the object version after the change: $E_E \subseteq (V_D \times V_C \times V_D)$. Evolution edges are depicted as thick lines.

A formal definition of evo-graph is given in [21]. Intuitively, the evo-graph consists of two connected graphs: a data graph comprising different versions of data, and a tree of changes. The data graph defines the structure of data, while the change graph defines the structure of changes on data. These two graphs interconnect via evolution edges. Consequently, there are two roots: the data root, r_D, and the change root, r_C. The change root is assumed to be always linked to an evolution edge that originates from the version *T=start* of the data root, and points to the version *T=now* of the data root. Moreover, there are two types of paths: the change paths that follow successive change edges, and the data paths that follow successive data edges.

The main objective of the evo-graph is to represent arbitrarily complex changes. The semantics of a complex change is implied by its structure and constituents, as defined by the users of the databank. An atomic change can only represent one of the basic change operations of the snap-model, however there is no restriction on how atomic changes are combined to form complex changes. Note that, as long as the set of basic change operations is complete (operations can

lead the snap-model to any possible state), *the choice of basic change operations is not restricted* by the evo-graph; alternative sets of basic change operations may be adopted, while the properties of evo-graph remain largely insensitive to which set is selected. We consider a typical set of *basic change operations* for the snap-model, comprised of:

- *create*: creates a new child node, and connects it with the parent node.
- *remove*: removes an edge, deleting a child.
- *update*: updates the value of an atomic node.

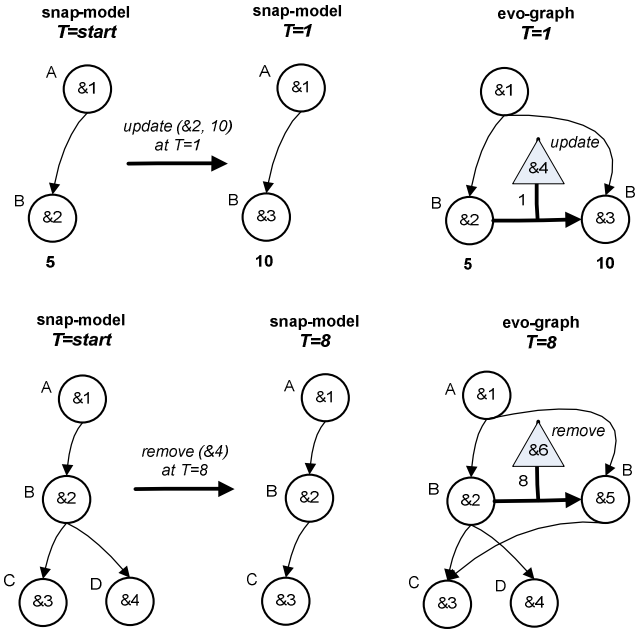

Fig. 1 Modeling of basic change operations with evo-graph

The evo-graph is constructed step by step, as changes occur at the current version of the snap-model. Our approach creates a new object version in the evo-graph whenever a change occurs to the value of that object. The value of an atomic node is a literal attached to the node, while the value of a complex object is the list of references to its children. Each change creates a new change node and a new evolution edge, connecting the previous version with the new version of the object. The new version of the object is then connected to the latest versions of its parents, and the latest versions of its children. Fig. 1 shows how the basic change operations *update* and *remove* are represented in the evo-graph. Nodes contain their respective node ID, while node labels are placed next to each node. In the case of *remove*, when node &4 is removed from the children of node &2, a new version of node &2 with ID &5 is created in the evo-graph to reflect this change. Note that a number of removals and creations of sibling nodes will result to an equal number of parent node versions. In this way evo-graph keeps a detailed record of the data evolution. The number assigned to each change represents the time instance the change occurred. We assume a linear time domain and two special time instances: *start*, representing the beginning of time, and *now*, representing the current moment. As a general rule, changes that affect child nodes create new versions of the parent nodes. The same holds for *update*, since the atomic node &2 can be considered as the parent of an implied "value node".

B. Recording Evolution: an Example

We present a simple scenario which demonstrates how the evo-graph can be used to record changes and relationships between changes in an evolving Web databank that publish bioscientific data. Through this example we attempt to establish the importance of treating complex changes as first class citizens, since they convey indispensable information for interpreting the evolution of data as well as the reasons for their current and previous states.

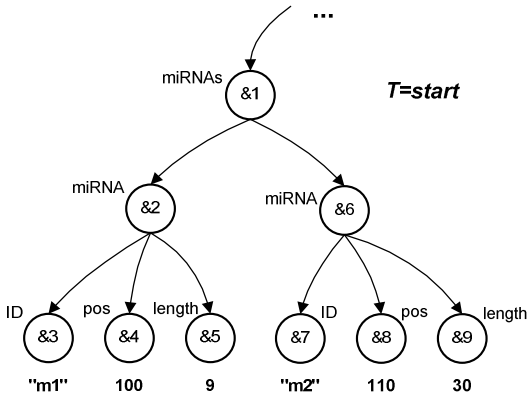

Fig. 2 State of Web databank at T=start

The initial state of the example databank is depicted in Fig. 2 and contains two *miRNA*s. A miRNA is a part of the DNA chain associated with the production of proteins, and under certain circumstances it can attach itself at certain points on the DNA chain, causing important effects. A miRNA is defined by a *start point* in the DNA chain (corresponding to *pos* in Fig. 2) and a *length*. Knowledge on miRNAs advances rapidly, and the databank in Fig. 2 changes often to reflect this: a miRNA may change name and properties, split into two distinct miRNAs, merge with another to form a new miRNA, etc.

Fig. 3 shows the evo-graph at T=5. For simplicity, the data root and the change root are omitted, while the arguments of change operations are implied and do not appear on the figure. At time instance 1 (T=1) the length of the miRNA with ID "m1" is updated from 9 to 19. This basic change operation is expressed by the change node &11 that creates a new version of the length (node &10). After this update, "m1" occupies the positions 100 to 119. This, however, causes a collision with miRNA "m2", which on T=1 starts at position 110. Therefore, the start position of "m2" (node &8) must be updated, as a consequence of the change occurred to "m1". For the sake of the example, we assume that the end position of "m2" at the DNA chain remains fixed. Therefore, an update of the start position of "m2" must be followed by an update of its length, so that its end position remains the same. This is modeled by

the complex change *pos-len-update* that appears in Fig. 3 as node &17. This complex change creates a new version of the specific miRNA, and consists of two atomic changes: an *update* of the start position of "m2" (node &13 introduces node &12), and an *update* on the length of "m2" (node &15 introduces node &14).

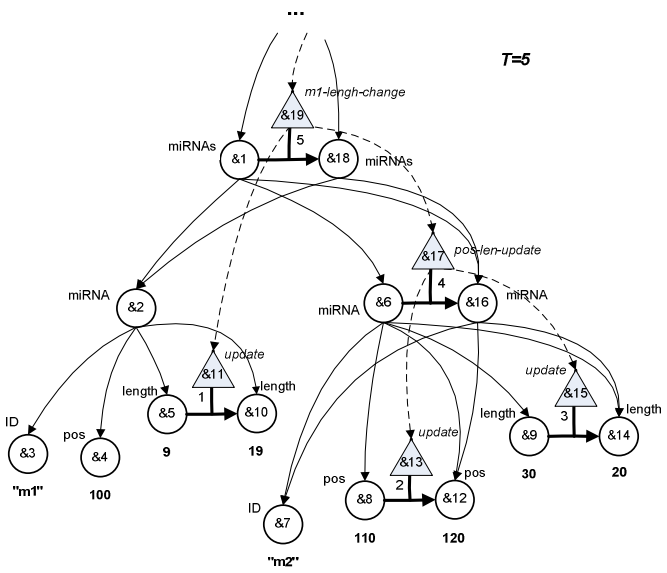

Fig. 3 Evo-graph at T=5

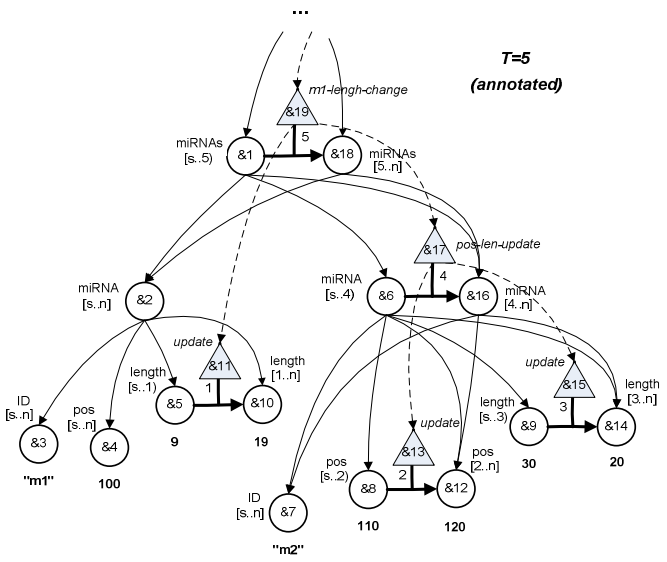

Fig. 4 Annotating the evo-graph of Fig. 3 with validity timespans

Note that newly introduced nodes are connected only to the latest versions of their parents and children, ensuring that the complexity of incoming and outgoing edges is contained within a "local" area. For example, in case node &14 is updated again at T=6, the resulting node &20 (not shown in Fig. 2) would be connected to node &16 but not to node &6.

Change nodes &11 and &17 are further composed into the complex operation *m1-length-change*, represented by node &19. This operation is associated with node &1 and causes the creation of a new version of the *miRNAs* node (node &18). Complex change nodes can represent relationships between changes that take place in disparate places of the databank, *and would otherwise be treated as unrelated*. In this way, it is possible to *model any change operation*, like for instance, *move*, *split*, *merge*, etc.

Summarizing, the evo-graph models evolution using arbitrarily complex changes, with sub-changes applied to objects that can reside far from each other in the data graph. In [21] we also show that the same principles can be used to create and maintain links between disparate databanks, as in the case of copying and pasting information.

C. Time Propagation and Snapshot Reduction

Evo-graph captures in a uniform way multiple data versions along with the evolution operations applied on each version, and represents them in a coherent graph enriched with temporal information. A key difference from existing versioning and temporal approaches on XML and semistructured data, where transaction and valid time are explicitly expressed though timestamps or time intervals assigned to nodes, edges, or paths [20], is that the time dimension is not assigned on the data elements of the graph; instead it is the change nodes that retain and propagate all temporal information on the evo-graph.

A timestamp is assigned to each change node in V_C, denoting the time on which this change occurred (transaction time). Change timestamps determine the *validity timespan* of data nodes and data edges. Every change affects the validity timespan of the two data nodes it is connected with (through the respective evolution edge), in the sense that the previous version of an object stops being valid the moment a new version is created. Timespans propagate to child data nodes, since a child can only exist if it has a valid parent.

In what follows we give a process for obtaining the validity timespans of the data nodes in an evo-graph.

- *Step 1*. Set as the current node v the data root r_D
- *Step 2*. For each outgoing data edge $e_i = (v,v_i) \in E_D$ set $TS(e_i) = TS(v)$, i.e. the timespan of the current node is propagated to all outgoing edges.
- *Step 3*. Set $TS(v_i) = \cup TS(x,v_i)$, where $(x,v_i) \in E_D$, i.e. the timespan of a node is the union of the timespans of all incoming edges (or equivalently to the union of the timespans of all its parents).
- *Step 4*. If an evolution edge $e_{ev} = (v_i,c,v_i') \in E_E$ exists with timestamp t_c, then do steps 4a and 4b, else go to step 5.
- *Step 4a*. The timespan of v_i becomes $TS(v_i) = TS(v_i) \cap [start..t_c)$.

- *Step 4b*. The timespan of v'_i becomes $TS(v'_i) = TS(v'_i) \cup [t_c..now]$.
- *Step 5*. Set v_i as the current node and return to step 2.

The evo-graph of Fig. 3 is shown in Fig. 4 annotated with the validity timespans of data nodes.

The validity timespans of data nodes and edges allow us to perform the operation of *snapshot reduction*, for extracting the specific version that holds under a given time instance. Snapshot reduction takes as input an evo-graph plus a time instance, and produces a snap-model consisting only of those data nodes and data edges for which their validity timespan contains the given time instance.

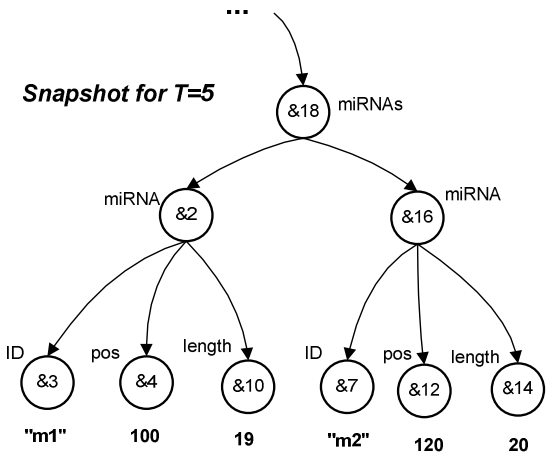

Fig. 5 Snapshot reduction for T=5 of the evo-graph in Fig. 4

The algorithm starts from the data root, and performs a recursive DFS on the evo-graph. It evaluates the time validity of each data node, and if it contains the given time instance the node is added to the output, along with its incoming edge through which it was accessed.

The tree snapshot for the time instance *T=5* of the evo-graph in Fig. 4 is shown in Fig. 5. Since the resulting tree does not contain any change nodes or evolution edges, it can be easily transformed to XML format, following a non replicated top-down traversal [15].

IV. XML Representation of Evo-Graph

There are two main techniques that can be employed for representing evo-graph with XML: the one follows the replicated approach, and the other follows the non-replicated approach [15]. A replicated representation dictates that a node with many parents is replicated as a child element under every different parent. The non-replicated approaches use XML references to connect the parent nodes to the common child element. In the latter approaches, shared children can be contained as elements either in the oldest, or in the latest version of the parent (with all the other versions connected to them through references). Moreover, shared children may use references to point to their parents (bottom-up representation), or alternatively be pointed to by their parents (top-down representation). Evidently, non-replicated techniques are more space efficient, since they avoid the constant replication of potentially big subtrees.

TABLE I
EVOXML FOR THE EVO-GRAPH OF FIG. 3 AT T=1

```
<evo:evoXML xmlns=""
            xmlns:evo="http://web.imis.athena-
                  innovation.gr/projects/c2d#evo">
 <evo:DataRoot evo:id="dataroot">
  <miRNAs evo:id="1">
   <miRNA evo:id="2">
     <ID evo:id="3">m1</ID>
     <pos evo:id="4">100</pos>
     <length evo:id="5">9</length>
     <length evo:id="10" evo:ts="1"
             evo:previous="5">19</length>
   </miRNA>
   <miRNA evo:id="6">
     <ID evo:id="7">m2</ID>
     <pos evo:id="8">110</pos>
     <length evo:id="9">30</length>
   </miRNA>
  </miRNAs>
 </evo:DataRoot>
 <evo:ChangeRoot evo:id="changeroot">
  <update evo:id="11" evo:tt="1"
          evo:before="5" evo:after="10"/>
 </evo:ChangeRoot>
</evo:evoXML >
```

We choose a top-down non-replicated approach for the XML representation of the evo-graph. The resulting representation, called *evoXML*, integrates into a single XML document all the different versions of the data nodes as well as the change nodes, along with some temporal attributes. To distinguish domain elements from special-purpose elements and attributes, we use a namespace called `evo` for qualifying the latter. In evoXML the data root, r_D, and change root, r_C, of the evo-graph are mapped to the elements `evo:DataRoot` and `evo:ChangeRoot` respectively. These top-level elements contain child elements that correspond to the data node hierarchy and the change node hierarchy of the evo-graph. An element is tagged with the label of the respective node, and has an attribute called `evo:id` whose value is the respective node ID in the evo-graph. The values of atomic data nodes become the content of the respective elements, whereas atomic change nodes are represented by empty elements. A change edge between two change nodes is captured as the parent-child relationship of the corresponding elements. A data edge between two data nodes is represented in the same way.

Note that if a child node is pointed to by multiple parent versions, the element corresponding to the child node is contained in the oldest parent element, while subsequent parent versions contain "clone" elements of the child. The "clone" elements are empty elements that point to the "original" child element via the special-purpose attribute `evo:ref`. As an example consider node &6 and node &16 that point to node &7 in Fig. 3. Table 2 shows the evoXML representation for the evo-graph in Fig. 3. In Table 2, node &7 corresponds to the `ID` element with `evo:id="7"` and content

"m2". Node &7 is contained in the miRNA element with evo:id="6". Notice that the miRNA element with evo:id="16" contains an ID element with a reference to the "original" ID element.

TABLE II
EVOXML FOR THE EVO-GRAPH OF FIG. 3 AT T=5

```
<evo:evoXML xmlns=""
            xmlns:evo="http://web.imis.athena-
                 innovation.gr/projects/c2d#evo">
 <evo:DataRoot evo:id="dataroot">
  <miRNAs evo:id="1">
   <miRNA evo:id="2">
     <ID evo:id="3">m1</ID>
     <pos evo:id="4">100</pos>
     <length evo:id="5">9</length>
     <length evo:id="10" evo:ts="1"
             evo:previous="5">19</length>
   </miRNA>
   <miRNA evo:id="6">
     <ID evo:id="7">m2</ID>
     <pos evo:id="8">110</pos>
     <pos evo:id="12" evo:ts="2"
          evo:previous="8">120</pos>
     <length evo:id="9">30</length>
     <length evo:id="14" evo:ts="3"
             evo:previous="9">20</length>
   </miRNA>
   <miRNA evo:id="16" evo:ts="4" evo:previous="6">
     <ID evo:ref="7"/>
     <pos evo:ref="12"/>
     <length evo:ref="14"/>
   </miRNA>
  </miRNAs>
  <miRNAs evo:id="18" evo:ts="5" evo:previous="1">
   <miRNA evo:ref="2"/>
   <miRNA evo:ref="16"/>
  </miRNAs>
 </evo:DataRoot>
 <evo:ChangeRoot evo:id="changeroot">
  <m1-length-change evo:id="19" evo:tt="5"
                    evo:before="1" evo:after="18">
    <update evo:id="11" evo:tt="1"
            evo:before="5" evo:after="10"/>
    <pos_len_update evo:id="17" evo:tt="4"
                    evo:before="6" evo:after="16">
      <update evo:id="13" evo:tt="2"
              evo:before="8" evo:after="12"/>
      <update evo:id="15" evo:tt="3"
              evo:before="9" evo:after="14"/>
    </pos_len_update>
  </m1-length-change>
 </evo:ChangeRoot>
</evo:evoXML >
```

For representing an evolution edge $e_E(v_1,c,v_2) \subseteq E_E$, we introduce the two special-purpose attributes evo:before and evo:after, which are part of the evoXML element corresponding to the change node c, and reference the elements that represent v_1 and v_2 respectively. In addition, we use the attribute evo:previous in the element representing a new version v_2 of a node v_1 to reference the element representing v_1. This enables us to directly spot the previous version of an element without having to refer to the evo:before attribute of the corresponding change element.

Finally, we use the attribute evo:tt for recording the timestamp (transaction time) of a change node, and the attribute evo:ts for recording the beginning of the validity timespan of a data node. We do not keep the end of the validity timespan, because it can be derived from the evo:ts attribute of the element representing the next version. The attribute evo:ts is used only in elements representing nodes that have evolved (nodes connected to an evolution edge); especially for the first version of such nodes the evo:ts="start" is implied and does not appear explicitly in the element. Moreover, the validity timespan of an element that has not evolved is implied, and is the same as that of its parent.

In Table 1 we give the evoXML for the evo-graph in Fig. 3 until the time instance T=1 (inclusive), while in Table 2 we give the evoXML for the whole evo-graph in Fig. 3 (until the time instance T=5, inclusive). Elements in Table 2 that do not exist in Table 1 appear in bold. Observe that our approach to XML representation is *additive* with respect to evo-graph operations: as the evo-graph changes, only additions of new elements are performed in the corresponding evoXML document.

In [21] we have presented in detail how temporal snapshots can be extracted from evo-graph. The extraction of a specific document version for a given time instance is performed on an evoXML document in a similar way: by traversing the data part of the document, and keeping only those elements that are valid for the specified time instance.

V. QUERYING DATA EVOLUTION THROUGH COMPLEX CHANGES

In this section we give query examples on a number of distinct query categories, and illustrate how they can be answered using both evo-path and XQuery.

A. Evo-path Syntax

In [21] we have proposed *evo-path* as an extension of XPath [18] used to navigate evo-graphs. Similarly to XPath, evo-path uses path expressions to move through and select data nodes. Moreover, evo-path uses constructs that allows the navigation through change nodes, plus predicates that express conditions on evolution edges. Consequently, there are two kinds of path expressions in evo-paths: *data path expressions*, and *change path expressions*.

Data path expressions start from the data root of the evo-graph and return data nodes. Similarly to XPath they are written as a sequence of *location steps* separated by "/" characters. Shortcuts can be used in data path expressions just like in XPath, as shown by the following two equivalent evo-paths:

```
/child::A/descendant-or-self::node()/child::B/
                                  child::*[position()=1]
/A//B/*[1]
```

Change path expressions start from the change root of the evo-graph and return change nodes. They have the same syntax as data path expressions, but are enclosed in square brackets:

```
</location_step1/location_step2/…/location_step_N>
```

A *temporal predicate* is introduced in evo-path in order to express temporal conditions on the evo-graph nodes. The form of the temporal predicate is:

```
[ts() operator {timespan_1, timespan_2, …, timespan_N}]
```

where `operator` is `[not] (in | contains | meets | equals)`. The `ts()` evaluates to the validity timespan of the context node, which is calculated through the process described in Section III-C.

Evolution predicates are used to assert the existence of evolution edges at specific points in the graph. The form of the evolution predicate is:

```
[evo-filter data_path_expr | change_path_expr]
```

The `evo-filter` can be one of: `evo-before()`, `evo-after()`, and `evo-both()`. The filters `evo-before()` and `evo-after()` retain only those data nodes that are on the correct side (left and right respectively) of the change specified by the evolution predicate. On the other hand, `evo-both()` returns *true* for the data nodes on both sides of the evolution edge.

B. Querying Evo-graph and EvoXML

In what follows we continue the scenario presented in Section III-B, and give a few query examples on a number of distinct query categories (appearing in brackets in the headings below). We express each query both as an evo-path and as an XQuery, in order to expose their relative expressiveness and suitability in the frame of the current context. Evo-paths are evaluated against the evo-graph of Fig. 3, while XQueries are evaluated against the corresponding evoXML of Table 2.

1) History of a data element (temporal queries): While browsing the current snapshot of the databank (Fig. 5) a bio-scientist named Brian realizes that the length of the miRNA with ID 'm2' is not what he expected, and engages in finding out what has happened and why. He starts by retrieving the previous versions of the data node &14 (see Fig. 3):

Evo-path

```
//miRNA [ID='m2'] /length [ts() not covers {now}]
```

This is a data path expression that returns the length data nodes of miRNA objects with `ID='m2'`. The temporal predicate `ts()` evaluates to *false* for the current version of `length` (&14) that holds under now, and *true* for every other version. The evo-path returns node &9 in Fig. 3.

XQuery

```
1    for     $m in //miRNA,
2            $i in $m/id,
3            $i2 in $m/id,
4            $l in $m/length,
5            $l2 in $m/length
6    where   ($i = "m2" or
7            ($i/@evo:ref = $i2 and $i2 = "m2"))
8            and
9            $l2/@evo:previous = $l/@evo:id
10   return  $l
```

A key part in this XQuery is the use of the variables `i` and `i2` in order to implement the test `ID='m2'` (lines 6 and 7). Notice that our decision to use a non-replicated approach for the XML representation forced us to use a self join. This self join ensures that the `miRNA` element with `evo:id=16` is not excluded, and that the reference to the `ID` element with `evo:id=7` is correctly evaluated to `'m2'`. Another self join is used in line 9 to ensure that the `length` versions that will be returned will not include the current version. The query returns the `length` element with `evo:id=9` from the evoXML of Table 2.

2) Changes applied on data elements (evolution queries): Brian checks the value of node &9 and wants to learn more about the hows and whys for updating the value 30 of `length` to the current value 20. He wants to get all the complex changes that contain the relevant `update` operation (node &15), and check whether this `update` was part of a larger modification within the *miRNAs* subtree:

Evo-path

```
<//* [evo-both() //miRNAs//*]
     [.//update [evo-after() //length
                        [ts() covers {now}] = 20]]>
```

The first predicate of the above evo-path returns all the change nodes that are applied to a `miRNAs` data node or any of their descendants. On the next two lines, the second predicate dictates that only the changes that have an `update` descendant applied on a `length` object with current value 20 can be returned. The evo-path returns nodes &19 and &17 of Fig. 3.

XQuery

```
1    for     $c in /changeroot//*,
2            $l in /dataroot//length,
3            $l1 in /dataroot//length,
4            $d in //miRNAs//*,
5            $u in $c//update
6    where   ($c/@evo:before = $d/@evo:id
7            or
8            $c/@evo:after = $d/@evo:id)
9            and
10           $u/@evo:after = $l/@evo:id
11           and
12           $l[not
13             (./@evo:id = $l1/@evo:previous)]
14           and
15           $l = 20
16   return  $c
```

Lines 6, 7, and 8 require that the returned changes apply to some element of the `miRNAs` subtree. Line 10 ensures that the changes have some `update` descendant applied on some `length` element. In lines 12 and 13 a self join ensures that only the latest (current) version of `length` elements survives for the next condition `$l=20` in line 15. The query returns the `m1-length-change` element with `evo:id=19` and the `pos-len-update` element with `evo:id=17` from the evoXML of Table 2.

3) Relationships between change elements (causality queries): Realizing that the update of the length of 'm2' has something to do with the complex change &19 `m1-length-change`, Brian decides to check all the prior versions of the

data objects affected by `m1-length-change` and its descendant changes.

Evo-path

```
//* [evo-before() <//m1-length-change//*>]
```

Not taking the predicate into account, the data path expression evaluates to all the data nodes in Fig. 3. The evolution predicate evaluates to *true* only for the data nodes that are connected through an evolution edge with a `m1-length-change` change node (&19) or one of its descendant change nodes (&11, &17, &13, &15). These nodes are &1, &18, &5, &10, &6, &16, &8, &12, &9, and &14. However, due to `evo-before()` in the evolution predicate, only the following nodes are returned as the result: &1, &5, &6, &8, &9. Brian links the dots and realizes that the updates on the miRNA 'm2' are a consequence of the change of the `length` of 'm1'.

XQuery

```
1    for    $c in
2              /changeroot//m1-length-change//*,
3           $d in /dataroot//*
4    where  $c/@evo:before = $d/@evo:id
5    return $d
```

The query returns all the elements that correspond to nodes modified by some change located under the `m1-length-change`. The elements with `evo:id 1,5,6,8` and `9` are returned from the evoXML of Table 2.

C. Discussion

The XQuery expressions in this section are equivalent to, but not a formal translation of the corresponding evo-path expressions. A formal translation would address, part by part, the elements introduced by evo-path that do not exist in XPath, like for instance the temporal predicate (Section V-A) and the `ts()` function. Instead we used XML references to go around the XQuery implementation of the temporal predicate for the specific examples.

The XQuery examples in this section are quite complex. This complexity comes from two main reasons. The first reason is our choice of XML representation. The non-replicated approach we followed uses references to represent common children in the data part of the evo-graph. Conditions on such children require self joins in XQuery in order to take references into account. The second reason has to do with the temporal properties of nodes and edges in the evo-graph. Temporal predicates are not straightforward to express in XQuery. Notice that the third XQuery example above is easy to follow because it does not involve time constraints, and does not pose conditions on the data part of the evo-graph. The change part of the evo-graph is a tree, and it is simple to access with XQuery.

Summarizing, the modeling of complex changes in evo-graph enables a wide range of useful queries to be expressed in a uniform way. The above discussion leads us to believe that it would be meaningful for a query language in our context to provide direct support for complex changes and evolution edges.

VI. Conclusions and Future Work

In this paper we argued that changes should be treated as first class citizens in data management systems. Evo-graph is a graph model that represents, in addition to data, arbitrarily complex changes. We discussed how evo-graph is constructed through an example on bioscientific data, and showed how temporal snapshots of the data can be produced. Moreover, we introduced evoXML, an XML syntax for representing evo-graphs. We presented a number of interesting query categories that can be answered by evo-graph and evoXML, showing the potential of using *change objects* just like data objects in models and queries. Evo-path is an extension of XPath for navigating and querying evo-graphs. We expressed our example queries in both XQuery and evo-path, for giving an intuition of the difference between the two. Concluding, we argue that treating changes as first class citizens enables a uniform solution to a number of evolution and provenance issues in Web data.

Although XML technologies seem a natural choice for representing evolving Web data, the principles behind evo-graph (the treatment of changes as first class citizens) can be applied to other data management frameworks like RDF. In addition to investigating this direction, the next steps of our ongoing work include: (a) to finalize a query language for evo-graphs and evoXML, (b) to specify a language for defining types (templates) for complex changes, and investigate their instantiation on data, and (c) to implement and experiment with tools for recording and querying complex changes.

References

[1] T. Amagasa, M. Yoshikawa, S. Uemura. A Data Model for Temporal XML Documents. In DEXA 2000.

[2] A. Bairoch et al. The Universal Protein Resource (UniProt). In Nucleic Acids Research, 2005, Vol. 33, Database issue D154-D159, http://www.uniprot.org/.

[3] P. Buneman, S. Khanna, K. Tajima, W.C. Tan. Archiving Scientific Data. In ACM Transactions on Database Systems, Vol. 20, pp 1-39, 2004.

[4] P. Buneman, A.P. Chapman, J. Cheney. Provenance Management in Curated Databases. In SIGMOD 2006.

[5] S. Chawathe, S. Abiteboul, J. Widom. Managing Historical Semistructured Data. In Journal of Theory and Practice of Object Systems, Vol. 24(4), pp.1-20, 1999.

[6] S-Y. Chien, V. J. Tsotras, C. Zaniolo, D. Zhang. Storing and Querying Multiversion XML Documents using Durable Node Numbers. In WISE 2001.

[7] S-Y. Chien, V. J. Tsotras, C. Zaniolo. Efficient Management of Multiversion Documents by Object Referencing. In VLDB 2001: 291-300.

[8] C. Dyreson. Observing Transaction-Time Semantics with TTXPath. In WISE 2001.

[9] D. Gao, R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In VLDB 2003.

[10] M. Gergatsoulis, Y. Stavrakas. Representing Changes in XML Documents using Dimensions. In 1st International XML Database Symposium, (XSym 2003).

[11] M. A. Harris et al. The Gene Ontology (GO) database and informatics. In Nucleic Acids Research, 2004(1), Vol. 32, Database issue D258-61, http://www.geneontology.org/.

[12] A. Marian, S. Abiteboul, G. Cobena, L. Mignet. Change-Centric Management of Versions in an XML Warehouse. In VLDB 2001.
[13] H.J. Moon, C. Curino, A. Deutsch, C.Y. Hou, C. Zaniolo. Managing and querying transaction-time databases under schema evolution. In VLDB' 08, pp. 882-895, 2008.
[14] V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, V. Christophides. On Detecting High-Level Changes in RDF/S KBs. In ISWC 2009.
[15] F. Rizzolo, A. A. Vaisman. Temporal XML: modeling, indexing, and query processing. VLDB J. 17(5): 1179-1212 (2008).
[16] F. Rizzolo, Y. Velegrakis, J. Mylopoulos, S. Bykau. Modeling Concept Evolution: a Historical Perspective. In ER 2009.
[17] F. Wang, C. Zaniolo. Temporal Queries in XML Document Archives and Web Warehouses. In TIME 2003: 47-55.
[18] W3C. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, January 2007.
[19] W3C. The XML data model. http://www.w3.org/XML/Datamodel.html, August 2005.
[20] F. Grandi. Introducing an Annotated Bibliography on Temporal and Evolution Aspects in the World Wide Web. SIGMOD Record 33(2): 84-86 (2004).
[21] Y. Stavrakas, G. Papastefanatos. Supporting Complex Changes in Evolving Interrelated Web Databanks. Conference on Cooperative Information Systems (CoopIS 2010), October 2010.