Schema-Mediated Exchange of Temporal XML Data

Curtis Dyreson[1], Richard T. Snodgrass[2], Faiz Currim[3], and Sabah Currim[4]

[1] Washington State University, Pullman, WA
cdyreson@eecs.wsu.edu
[2] University of Arizona, Tucson, AZ
rts@cs.arizona.edu
[3] University of Iowa, Iowa City, IA
faiz-currim@uiowa.edu
[4] University of Arizona, Tucson, AZ
scurrim@eller.arizona.edu

Abstract. When web servers publish data formatted in XML, only the current state of the data is (generally) published. But data evolves over time as it is updated. Capturing that evolution is vital to recovering past versions, tracking changes, and evaluating temporal queries. This paper presents a system to build a *temporal* data collection, which records the history of each published datum rather than just its current state. The key to exchanging temporal data is providing a *temporal schema* to mediate the interaction between the publisher and the reader. The schema describes how to construct a temporal data collection by "gluing" individual states into an integrated history.

1 Introduction

An XML schema describes the structure of XML data. The schema is used by a publisher to format the data for publication and by a reader to *validate* acquired data and add it to a data collection. Validation ensures that the data conforms to the formatting rules for XML (is well-formed) and to the types, elements, and attributes defined in the schema (is valid). A schema is also used as a guide in interpreting, editing and querying the data. Several schema languages have been proposed for XML; among them XML Schema is the most widely used.

Data formatted in XML is already available from many web servers. One example of a data provider is the National Center for Biotechnology Information (NCBI). Users can search the NCBI databases to locate data on genes and proteins. The data can then be downloaded in several formats, including as XML. In fact NCBI publishes data in three XML schemas. However, NCBI like most XML publishers only provides the current *snapshot* of the data. A snapshot is the data that is available at a single point in time, stripped of its historical context. But a data collection varies over time as new data is inserted and existing data is revised. In general, scientists want to know the *provenance* of their data: who, what, where, and *when* [3]; the evolution of the data is an important part of that provenance. Though NCBI users can download the current snapshot, they are unable to track and download changes to data. Obtaining data in an historical context is useful in many applications. For instance, scientific insights gained by analyzing data often have to be revised when the data changes. To help determine whether a reanalysis is needed, especially in a large data set where manual comparison is infeasible, it is crucial to be able to ascertain whether data has been added, modified, or deleted. One might want to look at coarse changes to an entire XML document or track the evolution over time of specific elements.

Fig. 1 illustrates the process by which a user currently downloads data from a publisher like NCBI. A user requests the current snapshot, D_{now}. The data is then added to the reader's data collection, DB, typically by overwriting a previously acquired version of D in DB. A better strategy, not currently supported by NCBI, is to transmit only the changes to the data, as shown in Fig. 2. A user requests a *change summary* of updates to D from time t, when the user last acquired D, to *now*. The summary, which is represented as "$\Delta\mathsf{D}$," is used to update the local snapshot of D. A Service Data Object (SDO) is one technology that supports change summaries [28]. In contrast, Fig. 3 shows the process of acquiring *temporal data*. A user requests a thick *slice* of

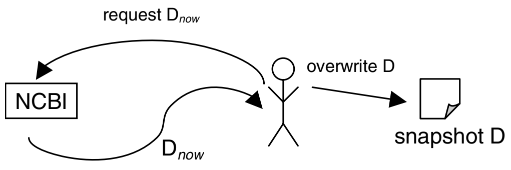

Fig. 1. Download of the current snapshot

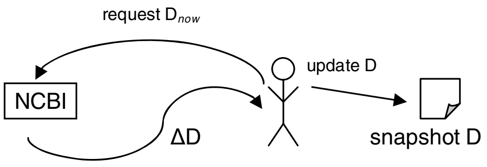

Fig. 2. Download change summary, e.g., in an SDO

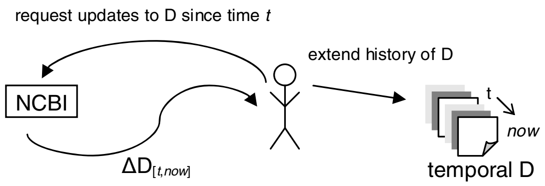

Fig. 3. A download of temporal data

data from time t, when the user last acquired D, to *now*. The slice as returned by the server is represented as "$\Delta\mathsf{D}_{[t,now]}$." The temporal data is then added to DB, extending the history of D. Unlike the snapshot data in Fig. 1 and Fig. 2, temporal data records the entire version history of every data item.

Systems that support the publication of and subscription to temporal data need several novel features.

- A data publisher has to add timestamps and other markup to indicate the lifetime of versions of the data.
- The temporal data produced by a publisher has to be amenable to automatic processing on the reader's side; for instance, the reader has to be able to validate the temporal data and update a temporal data store.
- To conserve bandwidth the slice "$\Delta D_{[t\text{-}now]}$" should be compact. Ideally it will be proportional in size to the changes to D since time *t*.
- It should be possible to validate the changes to a data collection, i.e., $\Delta D_{[t\text{-}now]}$. Unfortunately an SDO's change summary cannot be validated using the data's schema, rather the changes must first be applied to the data, which must then be entirely re-validated. It would be more desirable if it were possible to validate a slice of temporal data in isolation from the rest of the data collection.
- A publisher may have changed its schema since time *t*, so each step in the process must account for changes to the schema as well.

All of the above features can be supported by using a *temporal schema* to mediate the exchange of XML data.

This paper utilizes τXSchema (Temporal XML Schema), which is an infrastructure and suite of tools for constructing and validating XML data collections as both the data [8] and schema [10] evolve, though in this paper we consider only the data evolution aspects of τXSchema. τXSchema extends XML Schema with the ability to define *temporal element types*.[1] A temporal element type denotes that an element can vary over time, describes how to associate elements in different snapshots, and provides constraints that broadly characterize how an element evolves. Biologists are reticent to learn a new data model, or even a significant extension of a data model with which they have just gotten comfortable. Similarly, they don't want to have to acquire and learn how to use a new suite of tools that comes with the new data model. Hence, an important goal in the development of τXSchema was to maximally reuse existing XML standards and technology. In τXSchema, any element type can be denoted as a temporal element type by including a single, simple temporal annotation in the type definition. So a τXSchema document is just a conventional XML Schema document with a few temporal annotations. The tools operate in most cases identically to extant tools and in fact utilize those existing tools, such as conventional validating parsers. In most cases, the scientists don't even need to care if their XML data is static or temporal.

This paper is organized as follows. The next section motivates the differences between conventional (static) XML data and temporal XML data. We then discuss how snapshots of a temporal data collection are glued to create *items* and *versions*. The extensions to XML Schema to support temporal data are presented in Section 4. Section 5 sketches the process of constructing a representational schema. The paper concludes with a discussion of related work and a summary.

[1] This use of "temporal element" is a generalization of "XML element," and is not related to the "temporal element" defined by Gadia [11].

2 Example

Assume that data on the gene trypsin 4 (TRY4) is described in an XML data collection called `gene.xml`. The collection has information about gene function, which is described using the Mouse Genome Institute ontology. On 2005-01-01 the function of TRY4 was unknown as shown by the XML in Fig. 4. In subsequent months, new scientific data about TRY4 became available. On 2005-02-14 it was learned that TRY4 is involved in synthesizing the trypsinogen protein. The value of the "function" attribute was updated creating a new version of the data, as shown in Fig. 5. On 2005-03-06, the gene description became more specific, relating TRY4 to β-cell receptors so an additional "desc" element was inserted as shown in Fig. 6.

Researchers that prepared a paper on TRY4 in 2005-01 would like to learn of any updates to the TRY4 data since that time, and in particular how the data has changed. Certain changes will require a new analysis of their experiments. But the data in each figure is the data at a single point in time. Instead of the current snapshot, the researchers need the *version history*, which consists of the information in each version of the data along with a timestamp indicating the version's lifetime. The version history would describe how the knowledge about a particular gene has changed over time. This is of particular interest since new genomic and proteomic data is being constantly generated, and existing data is being revised and corrected. A version history would also aid in time-related analysis such as in tracking how a disease and its symptoms evolve over time (e.g., in an epidemic like the avian flu).

```
<gene name="TRY4">
    <desc>trypsin 4</desc>
    <ontology ref="MGI" function="unknown"/>
</gene>
```

Fig. 4. `gene.xml` on 2005-01-01

```
<gene name="TRY4">
    <desc>trypsin 4</desc>
    <ontology ref="MGI" function="synthesizes trypsinogen"/>
</gene>
```

Fig. 5. TRY4 codes for a protein, as of 2005-02-14

```
<gene name="TRY4">
    <desc>trypsin 4, beta-cell receptor</desc>
    <ontology ref="MGI" function="synthesizes trypsinogen"/>
</gene>
```

Fig. 6. TRY4 is related to β-cell receptors, as of 2005-03-06

Fig. 7 shows the temporal data that captures the history of the TRY4 data. The data is largely a list of gene and ontology *items*. The concept of an item is a central contribution of this paper. An item is an element that *persists* across individual snapshots. Each item has an `itemId` attribute that uniquely numbers the item. There is one gene item in the data, and one ontology item. Each item is referenced by a *temporal element*, which places it in the context in which it appears in a snapshot of the data. For example, in Fig. 7 the element `<ontology`$_{Temporal}$`>` references the ontology item, which indicates that a version of that item appears within the context of a `<gene>` element for each snapshot in the range of the version's timestamp.

```
<data_Temporal>
  <data><gene_Temporal itemRef="1"/></data>
  <gene_Item itemId="1">
    <gene_Version><time start="2005-01-01" end="2005-03-05"/>
      <gene name="TRY4">
        <desc>trypsin 4</desc>
        <ontology_Temporal itemRef="2"/>
      </gene>
    </gene_Version>
    <gene_Version><time start="2005-03-06" end="now"/>
      <gene name="TRY4">
        <desc>trypsin 4, beta-cell receptor</desc>
        <ontology_Temporal itemRef="2"/>
      </gene>
    </gene_Version>
  </gene_Item>
  <ontology_Item itemId="2">
    <ontology_Version><time start="2005-01-01" end="2005-02-13"/>
      <ontology ref="MGI" function="unknown"/>
    </ontology_Version>
    <ontology_Version><time start="2005-02-14" end="now"/>
      <ontology ref="MGI" function="synthesizes trypsinogen"/>
    </ontology_Version>
  </ontology_Item>
</data_Temporal>
```

Fig. 7. Temporal XML data

Whenever the item changes, a new *version* of the item is created. A change is defined, roughly, as a difference in an element's nontemporal content, exclusive of changes to content within the element's temporal subelements. Hence, the gene item has two versions. The second version was created on 2005-03-06 when new text content was added to the nontemporal `<desc>` element. The timestamp for each version indicates the version's lifetime. The end time of the second version is "*now*" indicating that the version is current. The ontology item also has two versions, because an attribute value was changed on 2005-02-14.

Note that the history of each item in a temporal data collection is more than just the current snapshot. It records not only the current state of the data, but all previous versions as well, and has timestamps to indicate when each version was current. Hence, a temporal data collection is unlike an SDO or related technology that records only a single snapshot and/or a summary of changes from the previous version.

A second contribution of this paper is a description of how to construct the temporal data (Fig. 7) by gluing the data in individual snapshots (Fig. 4, Fig. 5, and Fig. 6). The history in Fig. 7 captures the *transaction time* lifetime of each version [14]. Transaction time is the system time when the data was edited.[2]

A third contribution of this paper is explaining how to compactly represent in XML the change across a number of versions. Though the temporal data shown in Fig. 7 appears verbose in this small example, in general, it is actually *compact* in the sense that each edit results in only a localized change to the data (basically, a new version is created within an item). Fig. 8 shows the difference between the first and second versions of the data. The difference is a new version of the ontology element. The ability to represent the difference between two versions in isolation from the rest of the data is useful in both data streaming and refreshing data from a remote source, since the change is usually much smaller in size than the entire collection or even a snapshot. Note that the value of the `itemId` attribute in Fig. 8 is local to the temporal data being exchanged (the value of the attribute could be "23").

```
<data_Temporal>
  <ontology_Item itemId="1">
    <ontology_Version><time start="2005-02-14" end="now"/>
      <ontology ref="MGI" function="synthesizes trypsinogen"/>
    </ontology_Version>
  </ontology_Item>
</data_Temmporal>
```

Fig. 8. The difference between two versions

A fourth contribution is the description of a process to construct a schema to validate and interpret the temporal data. Although publishers can provide temporal data, there must be some means of interpreting such data. Typically, the structure of published data is described in an associated schema document. Assume that the file `gene.xsd` contains the *snapshot schema* for `gene.xml`. The snapshot schema is the schema for an individual version. The snapshot schema is a valuable guide for editing and querying individual snapshots. The snapshot schema is given (in part) in Fig. 9. Note that the schema describes the structure of the fragments shown in Fig. 4, Fig. 5, and Fig. 6. Though the individual snapshots conform to the schema, the temporal data does not. So a snapshot schema such as `gene.xsd` cannot be used (directly) to validate or interpret the temporal data of Fig. 7. Nor can the schema be used to validate version differences, such as the fragment shown in Fig. 8. In our approach a snapshot schema is annotated with additional information to create a *temporal schema*. The temporal schema describes, at a logical level, which elements can vary over time, and how those elements can change. Fig. 10 shows the temporal schema for the running example. The temporal schema includes annotations for both the gene

[2] Temporal data could also record the *valid time* versions (valid time is real world time) but for simplicity we consider only one kind of time in this paper, i.e., the transaction and valid times are the same (other relationships between valid and transaction time [16] can be easily modeled in our framework).

and ontology element type definitions. The annotations are shaded gray in the figure. (Section 4 describes the annotations in detail.) We present the temporal schema here to emphasize that τXSchema is fully-upwards compatible with XML Schema; that is, it extends but does not change XML Schema. A further advantage of our approach is that the temporal schema can also be used to validate the differences between versions, such as the data in Fig. 8.

```
<element name="gene">
  <complexType>
    <attribute name="name" type="text" use="required"/>
    <sequence>
      <element name="desc" type="string"/>
      <element ref="ontology" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>
<element name="ontology">
  <complexType>
    <attribute name="ref" type="text"/>
    <attribute name="function" type="text"/>
  </complexType>
</element>
```

Fig. 9. An extract from the gene data schema

```
<element name="gene">
  <txs:temporal>
    <txs:ItemIdentifier>
      <txs:field path="@name"/>
    </txs:ItemIdentifier>
    <txs:transactionTime kind="state" contentVarying="true"
        existenceVarying="no gaps"/>
  </txs:temporal>
    definition of gene from the snapshot schema omitted for space
</element>
<element name="ontology">
  <txs:temporal>
    <txs:ItemIdentifier>
      <txs:field path="../@name"/><txs:field path="@function"/>
    </txs:ItemIdentifier>
    <txs:transactionTime kind="state" contentVarying="true"
        existenceVarying="gaps allowed"/>
  </txs:temporal>
    definition of ontology from the snapshot schema omitted for space
</element>
```

Fig. 10. An extract from a temporal schema

3 Items and Versions

This section briefly reviews concepts related to temporal data and then discusses how to temporally associate elements in different snapshots to create temporal data.

Let D be an XML document or data collection. D is typically modeled as an ordered tree, $D = (E, V)$, where E is the set of edges and V is the set of nodes. Each edge in E is of the form (v, w, n) where v is the parent, w, is the child, and n is an ordinal representing the position of the child in the lexical ordering of the children. We will refer to XML data acquired from a (non-temporal) document as a *snapshot* indicating that it is the data at a single point in time.

Temporal data represents the history of a sequence of snapshots. Let D^T be a temporal data collection. The *snapshot operation* extracts a complete *snapshot* of D^T at a particular time. Timestamps are *not* represented in the snapshot. The *snapshot* operation is denoted as $snap(t, D^T) = D$ where D is the snapshot at time t of D^T.

Note that we haven't yet described the structure of temporal data, however it should faithfully capture entire snapshots as stated in the following definition.

Definition [Snapshot reducibility]. Let D^T be a temporal data collection. D^T is said to be *snapshot reducible* to the sequence of snapshots $D_1, \ldots, D_n$ iff for each $1 \leq k \leq n$, $D_k = snap(k, D^T)$.

To create compact temporal data it is important to identify which elements persist through changes to a data collection. We will sometimes refer to the process of associating elements that persist across various snapshots as *gluing* the elements. When a pair of elements is glued, an *item* is created. An item is an element that evolves over time through various *versions*. Only temporal elements (that is, elements of a type that has a temporal annotation as described further in Section 4) are candidates for gluing.

Determining which elements should be glued depends on two factors: the *type* of the element, and the *item identifier* for the element's type. The type of an element is the element's definition in the schema. We will denote the type of an element as $\mathcal{T}$. An element can be glued only to an element or item of the same type. An item identifier is a list of XPath expressions (much like a key in XML Schema) so we first define what it means to evaluate an XPath expression.

Definition [XPath evaluation]. Let $Eval(x, E)$ denote the result of evaluating an XPath expression E from a context node x. Given a list of XPath expressions, $L = [E_1, \ldots, E_k]$, then $Eval(x, L) = [Eval(x, E_1), \ldots, Eval(x, E_k)]$.

Since an XPath expression evaluates to a list of values, $Eval(x, L)$ evaluates to a list of lists. An item identifier is a list of XPath expressions.

Definition [Item identifier]. An *item identifier* for a temporal type, $\mathcal{T}$, is a list of XPath expressions, $L_{\mathcal{T}}$, such that for each element x of type $\mathcal{T}$, $Eval(x, L_{\mathcal{T}})$ names the *item* to which x belongs.

Each item identifier is specified by a schema designer (elsewhere we sketch a method for automatically constructing them [32]). Often an identifier will be the (snapshot) key for the element type given in the schema [4]. But an item identifier may differ from a snapshot key since the identifier should be a temporally-invariant key [22].

Example [Item identifiers]. As an example, a designer might specify the following item identifiers for the temporal elements in Fig. 7.

- `<gene>` $\rightarrow$ `[@name]`
- `<ontology>` $\rightarrow$ `[../@name, @function ]`

The item identifier for a `<gene>` is the name of the gene, while the item identifier for an `<ontology>` is the gene's name (its parent's item identifier) combined with the gene's `function` attribute value.

We will further restrict item identifiers to be *unique* within a snapshot, that is, at most one element in each snapshot can belong to an item. Over time, elements that belong to different snapshots will belong to the same item. Elements that are *temporally adjacent* can be associated within an item as defined below.

Definition [Temporal adjacency]**.** Let x be an element of type $\mathcal{T}$ in $snap(i, D^T)$. Let y be an element of type $\mathcal{T}$ in $snap(j, D^T)$. Finally let $L_\mathcal{T}$ be the item identifier for elements of type $\mathcal{T}$. Then x is *temporally adjacent* to y if and only if $Eval(x, L_\mathcal{T}) = Eval(y, L_\mathcal{T})$ and it is not the case that there exists an element z of type $\mathcal{T}$ in a snapshot between (exclusive) the i^{th} and j^{th} snapshots such that $Eval(z, L_\mathcal{T}) = Eval(x, L_\mathcal{T})$.

When an item is temporally adjacent to an element in a new snapshot, the element either creates a new version of the item or extends the lifetime of the latest version within the item. So an item is a sequence of versions and associated timestamps. The lifetime of each version is a set of maximal, disjoint time periods.

Definition [Item]**.** Let **item**(x) be the item named by Eval(x, $L_\mathcal{T}$) where x is of type $\mathcal{T}$. Then **item**(x) = $[(v_1, t_1), \ldots, (v_n, t_n)]$ where each v_i is a version of x with lifetime t_i ($1 \le i \le n$).

A version represents the content of an item in a snapshot. Basically, the version is a copy of the subtree rooted at the item, and each branch in the copy terminates at a leaf (attribute node, text node, etc.) or at the first element on that branch that is associated with an item. The element is replaced in this version with an *item reference*.

Definition [Version]**.** Let **item**(x) be an item of type $\mathcal{T}$ in snapshot $D=(E, V)$. Let (E_x, V_x) be the subtree rooted at x in D. Then **version**(x, D) = (E_v, V_v) where

$$E_v = \{(a_v, b_v, n) \mid (a_x, b_x, n) \in E_x \wedge (b_x \text{ is an item} \Rightarrow b_v \text{ is an } \textit{item reference}) \wedge (a_x \text{ is an item} \Rightarrow a_v = x) \wedge (a_x \text{ and } b_x \text{ are not items} \Rightarrow a_v = a_x \wedge b_v = b_x)\}$$

and $V_v = \{v \mid (v, \_, \_) \in E_x \vee (\_, v, \_) \in E_x\} \cup \{x\}$.

Example [Items]**.** Versions appear throughout the example of temporal data shown in Fig. 7. The first version of the `<gene>` item is a copy of the `<gene>` element in Fig. 4, which is the first snapshot of the data. Note that the `<ontology>` element is an item, so it has been replaced in Fig. 7 by an item reference whereas the `<desc>` element is unchanged since it is not an item.

A lifetime of a version is computed separately. The lifetime is extended when "no difference" is detected in the associated element. Differences are observed within the context of the Document Object Model (DOM).

Definition [DOM equivalence]**.** A pair of item versions is *DOM equivalent* if the pair meets all of the following conditions: they have the same number of children, same element tag, same set of attributes (an attribute is a name, value pair), and same text content, and for each child, the child is DOM equivalent to the corresponding child of the other (in a lexical ordering of the children).

As an aside, we observe that DOM equivalence in a temporal XML context is akin to *value equivalence* in a temporal relational database context.

DOM equivalence is used to determine versions of an item, as follows.

Definition [Versioning]**.** Let **item**(x) = [(v_1, t_1), …, (v_n,t_n)]. Let **item**(y) in snapshot D be temporally adjacent to **item**(x). Assume D is current during the period [t, $t+k$] where t is later than any time in t_n. If v_n is DOM equivalent to **version**(y, D) then the lifetime of v_n is extended to be $t_n \cup [t, t+k]$. Otherwise, version v_{n+1}, consisting of **version**(y, D), is added to **item**(x). The lifetime of v_{n+1} is [t, $t+k$].

A version's lifetime is extended when the version from the next snapshot (or a future snapshot) is DOM equivalent (the lifetime can have gaps or holes, although having a gap may violate a schema constraint as described in Section 4). A new version is created when temporally adjacent elements in the same item are *not* DOM equivalent.

Example [Versions]**.** Fig. 11 depicts the items and versions in the example in Section 2. An abstract representation of the DOM for each snapshot of the data is shown. The items in the sequence of snapshots are connected within each gray shaded region. There is one gene item and one ontology item. Each item has two versions. The transition between versions is shown as a black rectangle on the gray connection arcs. The gene item has a new version when the content of the `<desc>` element changes and the ontology item has a new version when its content is modified on 2005-02-14.

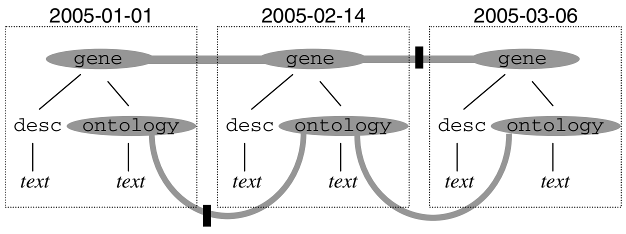

Fig. 11. Items and versions in the example

4 XML Schema Extensions

τXSchema extends XML Schema with a single annotation to denote temporal element types, but otherwise leaves XML Schema unchanged. The annotation is a `<txs:temporal>` element that can appear in the content of any element type definition. The annotation denotes that elements of that type can be time-varying. The `txs` namespace indicates that the annotation is part of τXSchema. Within a `<txs:temporal>` element there must appear an item identifier. Such an identifier has the following general form.

```
<txs:itemIdentifier
   <txs:field path="XPath expression"/>
   ...
   <txs:field path="XPath expression"/>
</txs:itemIdentifier>
```

An item identifier is list of fields, each of which is a (relative) XPath path expression.

Temporal constraints are optional. The constraints are evaluated after an item is glued. The constraints are separately specified for each kind of time, though in this paper we focus only on transaction time. The constraint specification for a `<txs:transactionTime>` element has the following general form.

```
<txs:transactionTime
    txs:kind="state (default) | event"
    txs:contentVarying="false (default) | true"
    txs:existenceVarying="false | gaps allowed (default) | no gaps" />
```

The `kind` attribute specifies whether the lifetime of an item has duration; a *state* kind of annotation implies continuity, while an *event* signifies that the lifetime is a single instant. The terminology is borrowed from temporal databases [14] where events occur at a single instant in time (e.g., a wedding on July 14, 2005), whereas a state occurs over a period of time (e.g., married from July 14, 2005 until now). The `contentVarying` attribute is used to specify whether an item's content must be constant over time, or can vary. The `existenceVarying` attribute governs whether the item can come and go in various snapshots. If the value of the attribute is *false*, then the item must be in every snapshot (or never appear). If the existence is *no gaps*, then once the item has been deleted from a snapshot, it cannot reappear in a later snapshot. Otherwise, the item's existence is unrestricted. Each attribute is optional, as is the transaction time element. If the attribute is not specified, the indicated default value applies.

Example [τXSchema]**.** The biologists in our running example are interested primarily in tracking two kinds of changes to the NCBI data: revisions of the gene itself and revisions of the ontology elements. Since NCBI does not publish a temporal schema, biologists must download individual snapshots and maintain a temporal data collection locally. Towards this end they create the temporal schema given in Fig. 10. The gene and ontology element type definitions given in the snapshot NCBI schema are annotated to indicate that they are temporal element types, and so a version history will be kept for each element of those types. While genes can be both content and existence varying, a gene's existence is slightly constrained to disallow gaps since a gene. The constraint specifies that in order for the data to be valid a gene cannot be deleted and then (later) reinserted.

Currently, τXSchema has a restricted set of temporal constraints. Richer classes of temporal constraints have been proposed [7], but for simplicity and brevity we limit the variety of constraints in the current system.

5 The Representational Schema

The representational schema is a conventional XML Schema document that is automatically generated from a τXSchema document. It is used to validate temporal data using a conventional validating parser. This section describes how to weave the temporal annotations into a snapshot schema to create the representational schema. The representational schema is transitory; it is needed only for validation, and in fact need never be seen by the user.

An XML Schema specification can be viewed as a grammar. The grammar consists of productions of the following form for each element type.

$S \rightarrow$ `<s>` α `</s>`

In the above production, α describes the content of elements of type *S*.

A temporal schema denotes that some of the element types are time-varying. To construct a representational schema, several new productions are added to the schema for each temporal element type; no productions are removed from the non-temporal schema though some are modified. Since only elements can be temporal, this section focuses on the element-related components of a schema. The construction process consists of several steps. We'll illustrate the process by describing what is done for a single, representative temporal element type, *S*.

The first step is to add a production to indicate that the element type *S* is temporal. The temporal production has following form:

$S_{Temporal} \rightarrow$ `<s`$_{Temporal}$ `itemRef="`m`"/>`

where `<s`$_{Temporal}$`>` denotes a temporal element of type *S* and `itemRef` is a reference to an item of type *S*. Next a production is added to define the *S* item type.

$S_{Item} \rightarrow$ `<s`$_{Item}$ `itemId="`n`">` $S_{Version}+$ `</s`$_{Item}$`>`

An item has a unique `itemId` value, and consists of a list of *versions*. The third step is to add a production to specify each version of type *S*. The production for a version of an element of type *S* has the following form:

$S_{Version} \rightarrow$ `<s`$_{Version}$`>` $\tau\, S$ `</s`$_{Version}$`>`

where τ is the schema of the timestamp and *S* is the non-temporal definition of the element's type. The timestamp in a version records the lifetime of the version. We do not impose a particular schema for a timestamp, rather we assume that the schema is given separately and imported into the temporal document's schema. Without loss of generality we will assume that each timestamp has the following form.

$\tau \rightarrow$ `<time start="…" end="…"/>`

The next step is to modify the *context* in which a temporal element appears. For each temporal element type, *S*, that appears in the left-hand-side of a production, replace *S* with $S_{Temporal}$. For example, assume that the schema has a production of the following form:

$X \rightarrow$ `<x>` $\beta\, S\, \gamma$ `</x>`

where β and γ describe arbitrary content before and after *S*, respectively. The production is replaced by the following production.

$X \rightarrow$ `<x>` $\beta\, S_{Temporal}\, \gamma$ `</x>`

Only the element type is replaced, any other constraints on the element are kept (e.g., minoccurs and maxoccurs are unaffected).

This process is repeated for every temporal element type. The final step is to augment the root element type with an additional production that appends a list of items. Let the root be an element of type *R*. Then the new root becomes the following.

$R_{Temporal} \rightarrow$ `<data`$_{Temporal}$`>` $R?\ X_{Item}{}^{*}$ `</data`$_{Temporal}$`>`

where X_{Item} is a list of item types. The production for X_{Item} is given below, where each S^{i}_{Item} is one of k item types.

$$X_{Item} \rightarrow S^{1}_{Item} \mid \ldots \mid S^{k}_{Item}$$

An additional step is needed to recast constraints that appear in the original schema. One such constraint is the uniqueness constraint imposed by a DTD identifier or XML Schema key definition. Since the same identifiers and key values can appear in multiple versions of an element, such values are no longer unique in a temporal document, even though they are unique within each snapshot. In temporal relational databases, the concept of a *temporal key*, which combines a snapshot key with a timestamp, has been introduced. Temporal keys can be enforced by a temporal validating parser, but not by a conventional parser. So constraints that impose uniqueness within a snapshot must be relaxed or redefined as follows. The value of each `id` type attribute in a time-varying element is rewritten to be a unique value; `idRefs` are similarly rewritten. Finally, schema keys are rewritten to include `itemIds` and version start and end times, creating a temporal key.

It is important to note that the production for the root of the temporal data specifies that it is just a list of items. This enables temporal data to be incrementally validated, which is critical in data streaming applications.

Example [Representational schema construction]**.** Let's go through the construction process with an example. Assume that the productions in the schema for the example in Fig. 6 are given below.

$$R \rightarrow \texttt{<data>}\ G{+}\ \texttt{</data>}$$
$$G \rightarrow \texttt{<gene>}\ D[\ N \mid \textit{text}\ ]^{*}\ \texttt{</gene>}$$
$$D \rightarrow \texttt{<desc>}\ \textit{text}\ \texttt{</desc>}$$
$$N \rightarrow \texttt{<ontology ref="}\textit{text}\texttt{">}\ \textit{text}\ \texttt{</ontology>}$$

Next, assume that the `<ontology>` element type is temporally annotated, as in Fig. 10. The schema would be transformed as follows. First, productions are added for the temporal elements.

$$N_{Temporal} \rightarrow \texttt{<ontology}_{Temporal}\ \texttt{itemRef="}m\texttt{"/>}$$

Next, productions are added for the items of temporal elements.

$$N_{Item} \rightarrow \texttt{<ontology}_{Item}\ \texttt{itemId="}n\texttt{">}\ N_{Version}{+}\ \texttt{</ontology}_{Item}\texttt{>}$$

Productions are then added for each version type, and for the timestamp(s) in each version.

$$N_{Version} \rightarrow \texttt{<ontology}_{Version}\texttt{>}\ \tau\ N\ \texttt{</ontology}_{Version}\texttt{>}$$
$$\tau \rightarrow \texttt{<time start="…" end="…"/>}$$

Next, the root is modified to include the new productions.

$$R_{Temporal} \rightarrow \texttt{<data}_{Temporal}\texttt{>}R?\ [G_{Item} \mid N_{Item}]^{*}\ \texttt{</data}_{Temporal}\texttt{>}$$

6 Related Work

Temporal databases has been an area of intense study for the past 25 years [[29]], with Oracle now perhaps having the most mature temporal support: transaction-time, valid-time, and bitemporal tables, current modifications, and automatic support for temporal referential integrity [[25]]. Concerning the representation of temporal data and documents on the web, Grandi has created a bibliography of previous work in this area [13]. Marian et al. [20] discuss versioning to track the history of downloaded documents. Chien, Tsotras and Zaniolo [5] have researched techniques for compactly storing multiple versions of an evolving XML document. Buneman et al. [4] provide another means to store a single copy of an element that occurs in many snapshots. This paper differs from all of the above papers since our focus is on temporal schemas and validation.

It is possible to capture transaction time information for documents through change analysis, as discussed below. Cho and Garcia-Molina [6] provide evidence that some web resources change frequently (though not specifically XML resources). Nguyen et al. [23] describe how to detect changes in XML documents that are accessible via the web [30]. Dyreson et al. [9] describe how a web server can capture some of the versions of a time-varying document. Yu and Popa provide an algorithm to convert either a list of changes or just the original and altered schema to a (more semantic) evolution mapping [31].

There are various XML schemas that have been proposed in the literature and in the commercial arena. We chose to extend XML Schema because it is backed by the W3C and supports most major features available in other XML schemas [19]. It would be relatively straightforward to apply the concepts in this paper to develop time support for other XML schema languages; less straightforward but possible would be to apply our approach to other data models, such as UML [24]. As an example, we have extended the Unifying Semantic Model, a conceptual model similar to the ER Model, to utilize annotations [17] very similar to what we propose here.

Recently there has been interest in incremental validation of XML [2][26]. τXSchema takes a orthogonal approach to incremental validation in so far as the changes to documents can be validated in isolation.

Only one paper has previously addressed the issue of validating temporal data [8]. In previous work we developed the τXSchema data model and architecture. In this paper we extend the architecture with items and versions, and a different construction process for the representational schema. Also, this paper directly extends XML Schema, unlike our previous paper.

τXSchema focuses on *instance versioning* (representing a time-varying XML instance document) rather than *schema versioning* [12][27]. The schema describes which aspects of an instance document change over time. But we assume that the schema itself is fixed, with no element types, data types, or attributes being added to or removed from the schema over time. In other work we consider schema versioning [10].

One final area of related work is *intensional XML data* (also termed dynamic XML documents [1]), that is, parts of XML documents that consist of programs that generate data [21]. Incorporating intensional XML data is beyond the scope of this paper.

7 Conclusion

This paper presents τXSchema, which extends XML Schema to support temporal data. τXSchema helps schema designers easily convert existing snapshot schemas to temporal schemas for the construction, management, and validation of temporal data and documents. A temporal schema is created by adding annotations to denote that some element types are temporal. Each annotation includes an item identifier, which is used to glue elements, yielding an item. Each change in an item over time creates a new version of the item. To validate a temporal document, a temporal schema is first converted to a *representational* schema, which is a conventional XML Schema document that describes how the temporal information is represented. The representational schema is carefully constructed to ensure every snapshot of the temporal document conforms to the snapshot schema (which is the temporal schema without the temporal annotations). A conventional validating parser is then used to validate the temporal document against the representational schema. The temporal document is also checked by a temporal constraint checker.

The architectural design of the infrastructure and even of the schema language itself is driven by the critical requirement from biologists, and indeed, from data users generally, of *upward compatibility*, of data, of schemas, and even of tools and infrastructure, in the support of time-varying data. This paper has demonstrated how a schema for time-varying data can be extended very simply from a snapshot schema, and then how the data manipulation, principally gluing and validation of such data and schema, can be done, utilizing conventional, well-understood tools.

In future we plan to integrate τXSchema with an XML-based editor. By incorporating τXSchema, an editor should be able to provide improved revision control and a *change tracking* feature. We have done this for an editor for the afore-mentioned temporal USM conceptual model [18]; it turns out that the upward-compatibility of the language design extends even to design support environment. Another broad area of work is optimization and efficiency. Currently there is no separation of elements or attributes based on the relative frequency of update. In the situation that some elements (for example) vary at a significantly different rate than other elements, it may prove more efficient to split the schema into pieces such that elements with similar "rates of change" are together [15].

References

[1] S. Abiteboul et al., "Dynamic XML Documents with Distribution and Replication," in *SIGMOD*, 2003. San Diego, CA. pp. 527-538.

[2] D. Barbosa et al., "Efficient Incremental Validation of XML Documents," in *ICDE*, 2004. Boston, MA, pp. 671-682.

[3] P. Buneman, S. Khanna, and W. C. Tan, "Why and Where: A Characterization of Data Provenance," in *ICDT*, 2001. pp. 316-330.

[4] P. Buneman et al., *Keys for XML*. Computer Networks, 2002. **39**(5): 473-487.

[5] S. Chien, V. Tsotras, and C. Zaniolo, *Efficient schemes for managing multiversion XML documents*. VLDB Journal, 2002. **11**(4): pp. 332-353.

[6] J. Cho and H. Garcia-Molina, *Estimating frequency of change*. ACM Trans. on Internet Technology, 2003. **3**(3): pp. 256-290.

[7] J. Chomicki, *Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding*. ACM Transactions on Database Systems, 1995. **20**(2): pp. 149-186.

[8] F. Currim et al., "A Tale of Two Schemas: Creating a Temporal XML Schema from a Snapshot Schema with τXSchema," in *EDBT*. 2004, pp. 348-365.

[9] C. Dyreson, H.-L. Lin, and Y. Wang. "Managing Versions of Web Documents in a Transaction-time Web Server," in *WWW*, 2004. New York, NY. pp. 422-432.

[10] C. Dyreson et al., "Validating Quicksand: Schema Versioning in τXSchema," in *XSDM*, 2006. Atlanta, GA, to appear.

[11] S. K. Gadia and J. H. Vaishnav. "A Query Language for a Homogeneous Temporal Database," in *PODS*. 1985, pp. 51-56.

[12] F. Grandi, "SVMgr: A Tool for the Management of Schema Versioning," in *ER*, 2004, pp. 860-861.

[13] F. Grandi, *An Annotated Bibliography on Temporal and Evolution Aspects in the World-WideWeb*. 2003, TimeCenter Technical Report.

[14] C. S. Jensen and C. Dyreson (eds.), "The Consensus Glossary of Temporal Database Concepts – Feb. 1998 Ver.," in *Temporal Databases*, 1998. pp. 367-405.

[15] C. S. Jensen and R. T. Snodgrass, *Semantics of Time-Varying Information.* Information Systems, 1996. **21**(4): pp. 311-352.

[16] C. S. Jensen and R. T. Snodgrass, *Temporal Specialization and Generalization.* IEEE Trans. on Knowledge and Data Engineering, 1994. **6**(6): pp. 954-974.

[17] V. Khatri, S. Ram, and R. T. Snodgrass, *Augmenting a Conceptual Model with Geospatiotemporal Annotations.* IEEE Transactions on Knowledge and Data Engineering, 2004. **16**(11): pp. 1324-1338.

[18] V. Khatri, S. Ram, and R. T. Snodgrass, *On Augmenting Database Design-Support Environments to Capture the Geo-Spatio-Temporal Data Semantics*. Information Systems, 2005: pp. 1-37.

[19] D. Lee and W. Chu, *Comparative Analysis of Six XML Schema Languages.* SIGMOD Record, 2000. **29**(3): pp. 76-87.

[20] A. Marian et al., "Change-Centric Management of Versions in an XML Warehouse," in *VLDB*, 2001. Roma, Italy, pp. 581-590.

[21] T. Milo et al., "Exchanging Intensional XML Data, " in *SIGMOD*, 2003. San Diego, CA, pp. 289-300.

[22] S. B. Navathe and R. Ahmed, *Temporal Relational Model and a Query Language*. Information Sciences, 1989. **49**(1): pp. 147-175.

[23] B. Nguyen et al., "Monitoring XML Data on the Web," in *SIGMOD*. 2001. Santa Barbara, CA, pp. 437-448.

[24] OMG, Unified Modeling Language (UML), v1.5. 2003.

[25] Oracle Corporation, *Application Developer's Guide – Workspace Manager*, 10g Release 1, December 2003.

[26] Y. Papakonstantinou and V. Vianu, "Incremental Validation of XML Documents," in *ICDT*, 2003. Siena, Italy, pp. 47-63.

[27] J. Roddick, *A Survey of Schema Versioning Issues for Database Systems*. Information and Software Technology, 1995. **37**(7): pp. 383-393.

[28] *Service Data Objects for Java Specification.* http://www-128.ibm. com/dev-eloperworks/ webservices/library/specification/ws-sdo, current as of March 2006.

[29] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummins Publishing Company, 1993

[30] L. Xyleme, *A dynamic warehouse for XML Data of the Web.* IEEE Data Engineering Bulletin, 2001. **24**(2): p. 40-47.

[31] C. Yu and L. Popa, "Semantic Adaptation of Schema Mappings when Schemas Evolve," in *VLDB*, 2005. Trondheim, Norway, pp. 1006-1017.

[32] S. Zhang, C Dyreson, and R. T. Snodgrass. "Schema-Less, Semantics-Based Change Detection for XML Documents," in *WISE*, 2004. Brisbane, pp. 279-290.