Validating Quicksand: Schema Versioning in τXSchema

Curtis Dyreson
Washington State University
cdyreson@eecs.wsu.edu

Richard T. Snodgrass
University of Arizona
rts@cs.arizona.edu

Faiz Currim
University of Iowa
faiz-currim@uiowa.edu

Sabah Currim
University of Arizona
scurrim@eller.arizona.edu

Shailesh Joshi
University of Arizona
shailesh@cs.arizona.edu

Abstract

The W3C XML Schema recommendation defines the structure and data types for XML documents, but lacks explicit support for time-varying XML documents or for a time-varying schema. In previous work we introduced τXSchema which is an infrastructure and suite of tools to support the creation and validation of time-varying documents, without requiring any changes to XML Schema. In this paper we extend τXSchema to support versioning of the schema itself. We introduce the concept of a bundle, which is an XML document that references a base (non-temporal) schema, temporal annotations describing how the document can change, and physical annotations describing where timestamps are placed. When the schema is versioned, the base schema and temporal and physical schemas can themselves be time-varying documents, each with their own (possibly versioned) schemas. We describe how the validator can be extended to validate documents in this seeming precarious situation of data that changes over time, while its schema and even its representation are also changing.

1. Introduction

Much of the power of a database management system stems from the presence of a *schema* that describes the structure of the database. When the data is versioned, a schema helps even more, because it expresses the commonality among the different versions, as well as indicating which parts of the data can change, and how. The schema is the solid ground upon which the data structures can stand.

When the schema itself is versioned, there is no solid ground. How schema versioning is supported makes the difference between a fluid motion between versions and awkward struggling against quicksand.

The W3C XML Schema recommendation defines the structure and data types for XML documents [14]. XML Schema lacks explicit support for time-varying XML documents. We previously proposed a data model and architecture, called τXSchema [9], for creating a temporal schema from a base schema, a temporal annotation, and a physical annotation. The annotations specify which portion(s) of an XML document can vary over time, how the document can change, and where timestamps should be placed. The advantage of using annotations to denote the time-varying aspects is that logical and physical data independence for temporal schemas can be achieved while remaining fully compatible with both existing XML Schema documents and the XML Schema recommendation.

In this paper we extend τXSchema to also support *schema versioning*. In doing so, we leverage both conventional XML Schema and related tools (principally, validator parsers), as well as τVALIDATOR for data versioning.

Schemas designers often edit their schemas, refining and adding element and attribute types. As an example, the Botanic Garden and Botanical Museum in Berlin-Dahlem (BGBM[1]) maintains a repository of XML Schemas[2] related to index terms, keywords, biodiversity data about specimens and observations, meta-level data about collections, organizations, and networks, and various wrapper and configuration files. Most of these XML schemas have had multiple versions over the last two to three years. The BioCASE Collection Profile is up to version 1.24; the Access to Biological Collection Data is up to version 2.06. The *Pharmacogenetics Knowledge Base* (PharmGKB[3]) "contains genomic, phenotype and clinical information collected from ongoing pharmacogenetic studies." Its schema is up to version 4.0; its evolution is documented[4].

[1] http://www.bgbm.org
[2] http://www.bgbm.org/biodivinf/schema/default.asp
[3] http://www.pharmgkb.org/
[4] http://www.pharmgkb.org/schema/history.html

Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW'06)
0-7695-2571-7/06 $20.00 © 2006 IEEE

One challenge is that in this potential quicksand, *anything* can change, and thus must be versioned: the snapshot documents, the base schema, the temporal annotations, the physical annotations, the schema documents included by these documents, even the schemas of these schema components. And, because the physical annotations can change, the concrete representation within a temporal XML document can vary. How can one even define validation in such a fluid environment?

2. Approach

There are several key ideas to our solution. First, a *temporal bundle* (or simply, a bundle) serves the analogous purpose of an XML Schema document for a static document. So we have a single point of reference for the schema of a temporal document. Of course, the bundle may itself contain versions within it. That means that the temporal documents *it* references must also have associated bundles as *their* schemas.

Second, as with quicksand, as you venture outward, eventually you reach solid ground. So eventually you reach a bundle containing no versions, or else you reach a static XML Schema document.

The third key idea first appeared in a paper by one of the authors on temporal aggregation [19], that of what we will call here, *schema-constant periods*. It is possible, even with versioned schemas having themselves versioned schemas, to identify contiguous periods of time when there are no schema changes, *anywhere*. Now, during such schema-constant periods the data may be (and probably is) versioned, but at least you have a fixed base schema and fixed temporal annotations, each of which has a fixed schema. And since the physical annotations are fixed, the representation is also fixed, so it is possible to read and interpret the temporal document during that schema-constant period, and even to validate that portion of the document. (This is just the situation discussed in our previous papers, of a single schema and versions of the data.) So a general temporal document can be viewed as a sequence of data-varying documents, each over a single schema-constant period. Since we can validate each schema-constant period, given the approaches elaborated on earlier, all we have to do is validate *across* schema changes.

The final key idea first appeared in the original presentation of τXSchema [9]: the representational schema (a) is derivable solely from information in the bundle, (b) can be designed to enable some of the temporal integrity constraints to be checked by a conventional validator, and (c) can be computed and cached within τVALIDATOR, completely unbeknown to the user. The bundle is all the user needs for describing the temporal document, just as the conventional XML Schema is all the user needs for describing an XML document.

Of course, there are lots of interesting alleys and excursions during this trip, but these four key ideas capture most of the approach.

In the remainder of this paper, we introduce the architecture through a running example, then describe how the validator can be extended to validate documents in this seeming precarious situation of data that changes over time, while its schema and even its representation are also changing over time. We note in passing that all times mentioned in this paper are *transaction times* [2], though τXSchema also supports valid time for data versioning.

3. Example and Architecture

The PHARMGKB XML schema was designed conventionally, without an architecture that supports schema versioning. As new releases of this schema were developed (on May 12, 2004 Version 4.0 was released), all XML documents that were instances of this schema were rendered invalid, with the maintainers responsible for updating their XML documents. The architecture proposed here retains past data and past schemas, while always allowing the current data and schema to be extracted, for tools that are not schema-versioning aware. While our architecture does not limit the kinds of changes a designer can make to a schema, typically as a schema is edited, each new version will add to or refine an existing version rather than entirely replace it.

Prior to Version 3.2, the `<ExperimentClass>` element of PHARMGKB contained nested `<sampleSet>` elements (cf. Figure 1). In Version 3.2, this was replaced with a `<sampleSetXRef>` element (cf. Figure 2), that just mentioned the unique identifier of the sample set, which was moved to the top of the document, with a `pharmgkbId` attribute. (The `AccessionObjectClass` includes an attribute `pharmgkbId` to specify this unique identifier, not shown here.) In Version 4.0 an `<ExperimentClass>` can now cross-reference more than one `<sampleSet>` (cf. Figure 3: note `unbounded` for `maxoccurs`). Additionally, though not shown in the figure, a `<sampleSet>` is now a set of `<sample>` instead of a set of `<subject>` (logically!).

Now let's examine how this could have been done using τXSchema.

Figure 4 illustrates the architecture of τXSchema. This figure is central to our approach, so we describe it in detail and illustrate it with the PHARMGKB schema. We note that although the architecture has many components, only those components shaded gray in the figure are specific to an individual time-varying document and need to be supplied by a user. We also note that to this point, the schemas (boxes 4,

```
<xsc:complexType name="ExperimentClass">
  <xsd:complexContent>
   <xsd:extension base="AccessionObjectClass">
    <xsd:sequence>
     <xsd:element name="name"
          type="NonEmptyTokenType"
          minOccurs="0" maxOccurs="1" />
     ...
     <xsd:element name="sampleSet"
          minOccurs="0" maxOccurs="1" />
      <xsd:complexType>
       <xsd:complexContent>
        <xsd:extension
               base="AccessionObjectClass">
         <xsd:sequence>
          <xsd:element name="name"
               type="NonEmptyTokenType"
               minOccurs="0" maxOccurs="1"/>
          ...
         </xsd:sequence>
        </xsd:extension>
       </xsd:complexContent>
      </xsd:complexType>
     </xsd:element>
     ...
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 1. `<ExperimentClass>` **element in version 3.1.**

```
<xsc:complexType name="ExperimentClass">
  <xsd:complexContent>
   <xsd:extension base="AccessionObjectClass">
    <xsd:sequence>
     <xsd:element name="name"
          type="NonEmptyTokenType"
          minOccurs="0" maxOccurs="1" />
     ...
     <xsd:element name="sampleSetXref"
          type="XrefClass"
          minOccurs="0" maxOccurs="1" />
     ...
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
...
<xsd:element name="sampleSet" />
 <xsd:complexType>
  <xsd:complexContent>
   <xsd:extension base="AccessionObjectClass">
     <xsd:sequence>
      <xsd:element name="name"
           type="NonEmptyTokenType"
           minOccurs="0" maxOccurs="1" />
      ...
     </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
</xsd:element>
```

Figure 2. `<ExperimentClass>` **element in version 3.2.**

5, 6, and 7) are static. We'll later relax this assumption.

The designer starts with the base schema (in the case of PHARMGKB, `root.xsd.xml` and `xsd:imported` or `xsd:included` schemas such as `"http://www.pharmgkb.org/schema/sequence.xsd"`, which itself `xsd:includes experiment.xsd`. The designer annotates the base schema with temporal annotations (box 6). The temporal annotations together with the base schema form the *logical schema*. The temporal annotations specify a variety of characteristics such as whether an element or attribute varies over valid time or transaction time, whether its lifetime is described as a continuous state or a single event, whether the item itself may appear at certain times (and not at others), and whether its content changes. Elements that are not described as time-varying are static and must have the same content and existence across every XML document in box 8.

The schema for the temporal annotations document is given by TXSchema (box 2), which in turn utilizes temporal values defined in a short XML Schema TVSchema (box 1). (Due to space limitations, we won't describe in detail these annotations, but it should be clear what aspects are specified there.)

The next design step is to create the physical annotations (box 7). In general, the physical annotations specify the timestamp representation options chosen by the user. Physical annotations may also be nested, inheriting the specified attributes from their parent; these values can be overridden in the child element. Physical annotations play two important roles. They help to define where in the document tree the physical timestamps will be placed (versioning level). The location of the timestamps is independent of which components vary over time (as specified by the temporal annotations). Two documents with the same logical information will look very different if we change the location of their physical timestamps. The physical annotations also define the kind of timestamp (for both valid time and transaction time). Changing an aspect of even one timestamp can make a big difference in the representation. The schema for the physical annotations document is PXSchema (box 3).

```
<xsc:complexType name="ExperimentClass">
  <xsd:complexContent>
   <xsd:extension
          base="AccessionObjectClass">
    <xsd:sequence>
     ...
     <xsd:element name="sampleSetXref"
       type="XrefClass"
       minOccurs="0" maxOccurs="unbounded"/>
     ...
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3. `<ExperimentClass>` **element in version 4.0.**

τXSchema supplies a default set of physical annotations. (Again, space limitations do not allow us to describe these annotations in detail.) We emphasize that our focus is on capturing relevant aspects of physical representations, not on the specific representations themselves, the design of which is itself challenging. Also, since the temporal and physical annotations are orthogonal and serve two separate goals, we choose to maintain them independently. A user can change where the timestamps are located, independent of specifying the temporal characteristics of that particular element. In the future, when software environments for managing changes to XML files over time are available, the user could specify temporal and physical annotations for an element together (by annotating a particular element to be temporal and also specifying that a timestamp should be located at that element), but these would remain two distinct aspects from a conceptual standpoint.

The base schema, temporal annotations, and physical annotations, which are all XML documents, are referenced by a *temporal bundle*. An example bundle for PHARMGKB is shown in Figure 6. The `<format>` element provides information about how timestamps are formatted; here we use XML Schema `dates`. A `<bundleSequence>` contains a sequence of `<schemaAnnotation>` elements, each referencing a snapshot (base) schema, a temporal annotation, and a physical annotation. Note that any of these three documents referenced by a `<schemaAnnotation>` element can include other schemas. For example, `root.xsd.xml` includes `sequence.xsd.xml` which itself includes `experiment.xsd.xml`.

At this point we can contend with time-varying data. Box 8 of the architecture shows a sequence of non-temporal documents, each an instance of the PHARMGKB schema (`root.xsd.xml`). The temporal XML document (box 9) is essentially a timestamped representation of this sequence of non-temporal XML data files (box 8). The schema of the temporal document is its associated representational schema (an XML Schema document). The timestamps are based on the characteristics defined in the temporal and physical annotations (boxes 6 and 7). The sequence of non-temporal documents can be SQUASHed into a temporal document (`rep.xml`), with its XML schema shown as box 10, generated by SCHEMAMAPPER from information in the temporal bundle. The tools are described in greater detail elsewhere [9].

The defining schema of the temporal document (box 9) is the temporal bundle (`bundle.xml`). However, the conventional XML validator does not understand time-varying documents nor their schema, so we have developed τVALIDATOR, a stand-in for the regular validator (see Figure 5). Within τVALIDATOR, SCHEMAMAPPER is invoked to generate the representational schema, which is then handed to the conventional validator. However, this is not sufficient due to the limitations of XML Schema in checking temporal constraints. For example, a regular XML Schema validating parser has no way of checking something as basic as "the time boundaries of a parent element must encompass those of its child." These temporal checks are implemented in the time-varying data checker.

τVALIDATOR, by checking the temporal data, effectively checks the non-temporal constraints specified by the base schema simultaneously on all the instances of the non-temporal data (box 8), as well as the constraints between snapshots, which cannot be expressed in a conventional schema. To reiterate, using the conventional approach, the user would start with the daunting task of manually generating a representational schema (box 10); our proposed approach is to have the user design a base schema and two annotations, with the representational schema automatically generated.

4. Supporting Versioned Schemas

We now generalize the architecture to also support versioned schemas. As noted previously, the PHARMGKB schema has undergone a series of changes. This implies that box 4 is actually a *sequence* of base schemas, three of which are excerpted in Figures 1–3. Not only do these base schemas change over time, but the schemas included by them (e.g., `sequence.xsd`, `experiment.xsd`) can vary over time. Similarly, the temporal annotations (box 6) and those annotations included by them and the physical annotations (box 7) and those annotations included by *them* all can vary over time, resulting in multiple versions.

This versioning is handled by timestamping the `<schemaAnnotation>` element in the bundle. To each such element is added a `<tTime>` element that specifies when that annotation element became applicable. So our

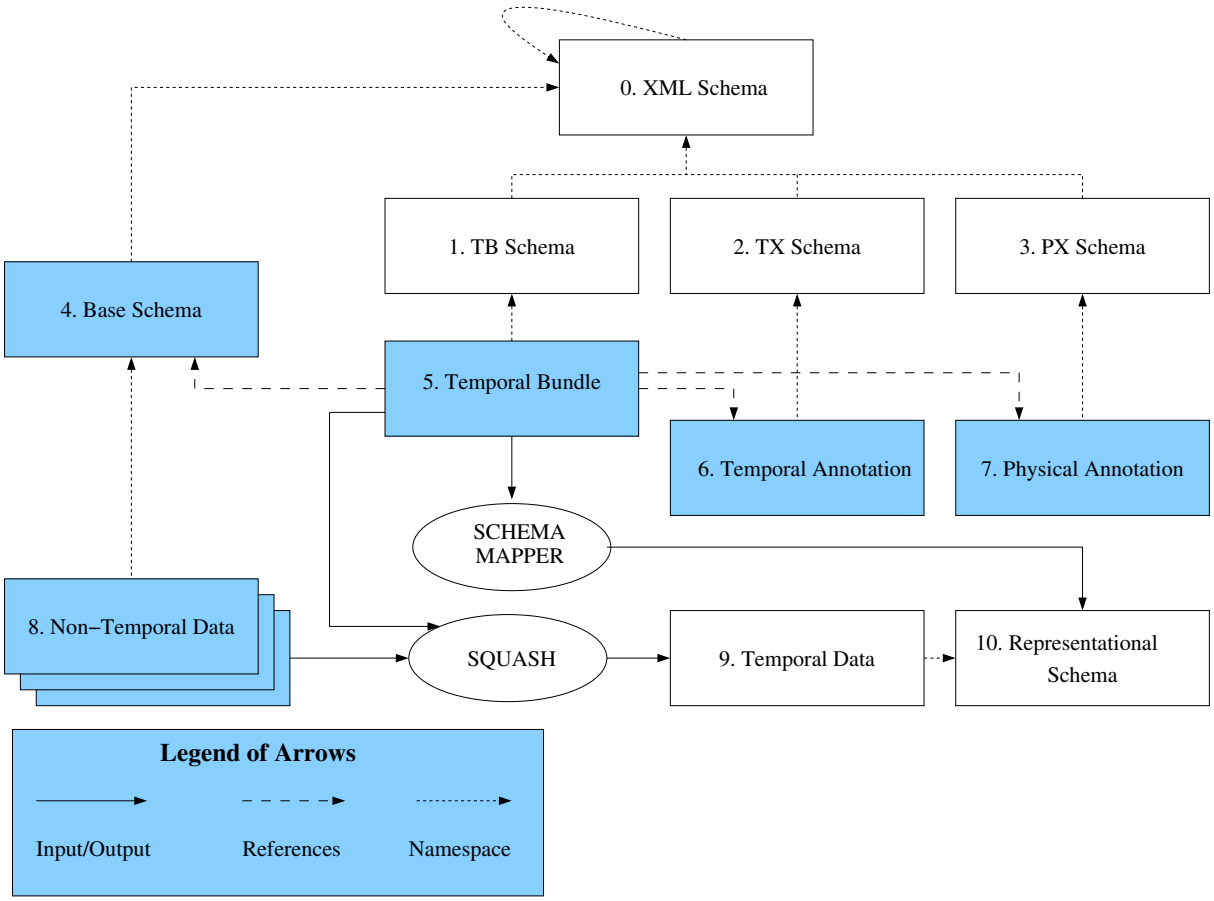

Figure 4. Architecture.

PHARMGKB schema would have many annotation elements, with version 3.1 becoming applicable on April 25, 2003, version 3.2 on May 21, 2003, and version 4.0 on May 12, 2004.

The schema annotation elements reference individual base schemas. One approach is to have a different document (file) for each version, similar to what is shown in box 8. So we might have files named `root.4.25.03.xsd.xml`, etc., or perhaps `root.3.1.xsd`. etc. Each of these files would reference subsidiary schemas, such as `sequence.v3.1.xml.xsd` or `experiment.4.25.03.xsd.xml`. As one can imagine, this becomes rather cumbersome. The problem with this approach is that whenever a subsidiary schema changes, a new version is produced, with its own URI, which requires the referencing schema document to be changed. So a new version of `experiment.xsd` requires a new version of `sequence.xsd`, which requires a new version of `root.xsd`.

While this approach is allowed, τXSchema also permits *temporal schemas*, in place of multiple versions of conventional schemas. Consider the sequence of root

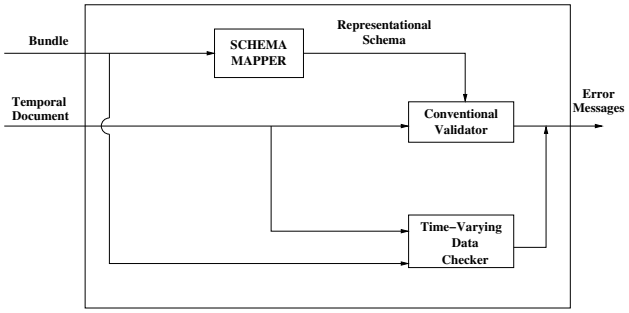

Figure 5. Validating a Document with Time-Varying Data.

schemas: `root.1.0.xsd`, `root.2.0.xsd`, ... We write a simple temporal bundle for these and invoke the SQUASH utility, which produces a single temporal document, `tv_snapshot.xml` which is then referenced by multiple schema annotation elements. Similarly, we use SQUASH to generate temporal schemas for `sequence.xsd.xml` and `experiment.xsd.xml`.

This rather involved state of affairs, with time-varying documents and time-varying schemas, is illustrated with a

```
<?xml version="1.0" encoding="UTF-8"?>
<temporalBundle
    xmlns="http://www.cs.arizona.edu/tau/tauXSchema/TBSchema"
    xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.cs.arizona.edu/tau/tauXSchema/TBSchema">
    <format plugin="XMLSchema" granularity="date"/>
        <bundleSequence defaultTemporalAnnotation="defaultTA.xml"
                        defaultPhysicalAnnotation="defaultPA.xml">
            <schemaAnnotation
                    snapshotSchema="root.xsd"
                    temporalAnnotation="temp_anno.xml"
                    physicalAnnotation="phy_anno.xml">
            </schemaAnnotation>
        </bundleSequence>
</temporalBundle>
```

Figure 6. A Temporal Bundle for PHARMGKB: `bundle.xml`

T Diagram in Figure 8. In this notation, first described almost forty years ago [5], the input of a translator is given on the left arm of the "T" (for example, for SCHEMAMAPPER in the upper right-hand-side of the figure, the input is the logical schema document, `bundle.xml`), the name of the translator is given at the base of the "T" (here, "Schema Mapper"), and the output of the translator is given on the right arm of the "T" (here, a representational schema, `rep.xml`). The name of these diagrams was to the best of our knowledge given by McKeeman, Horning, and Wortman in their classic compiler book [16].

We extend these diagrams to allow multiple inputs, which unfortunately complicates them somewhat. As shown in Figure 8, SQUASH takes both a bundle and a sequence of snapshot documents and produces a temporal document, and UNSQUASH does just the opposite (this is illustrated for the temporal annotations, which are SQUASHed into a single `tv_temp_anno.xml` document, then UNSQUASHed back into their constituent time slices.

In this figure we show a bundle (`bundle.xml`, right in the middle of the figure, with the arrows pointing left) referencing two temporal schemas, one of the base schema and one of the physical annotations; the bundle also references several temporal annotation documents. Note that the base schema for the base schema (!) is `XSchema_bundle.xml`, which has as its base schema `XSchema.xml`.

τVALIDATOR treats each URI it encounters as the specification of a temporal *timeslice* operation to select the appropriate version. The timeslice is as of the time of the document or context that contains the URI. For example, consider the excerpt in Figure 7. `root.xsd.xml` is a time-varying document, containing several schema versions. In this context, τVALIDATOR will utilize the temporal context

```
<schemaAnnotation
    snapshotSchema="root.2.0.xsd"
    temporalAnnotation="temp_anno.xml"
    physicalAnnotation="phy_anno.xml">
  <tTime>May 21, 2003</tTime>
</schemaAnnotation>
</schemaAnnotation>
```

Figure 7. An excerpt from the time-varying Temporal Bundle for PHARMGKB.

of "May 21, 2003" to extract a *single* version of the `root` schema. To do so, it calls UNSQUASH, passing it (a) the bundle, (b) the temporal document, and (c) a timestamp. It does so as well, for all the schemas included by that schema, such as `sequence` and `ExperimentClass`. The underlying semantics ensures that at any point in time, there is a single base schema, a single temporal annotation, and a single physical annotation.

Of course, one can carry this further. Because the base schema is versioned, it is associated with a temporal bundle which could itself have multiple schema annotation elements. τVALIDATOR recursively calls UNSQUASH so that at any point in time, there is a single schema in effect.

Let's examine how τVALIDATOR depicted in Figure 5 could handle the versioned schema for PHARMGKB. Recall that prior to Version 3.2, the `<ExperimentClass>` element of PHARMGKB contained nested `<sampleSet>` elements (cf. Figure 1). In Version 3.2, this was replaced with a `<sampleSetXRef>` element (cf. Figure 2), that just mentioned the unique identifier of the sample set, which was moved to the top of the document, with a `pharmgkbId` attribute.

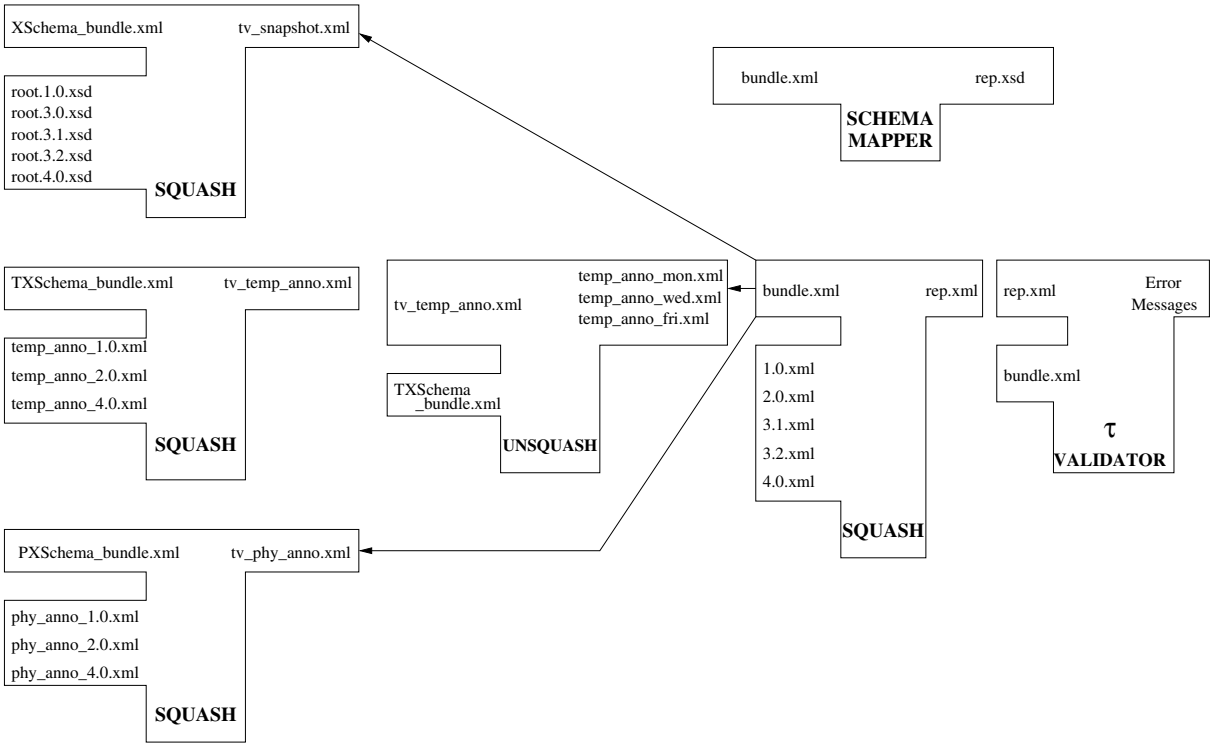

Figure 8. T Diagram of Validation.

This change is reflected in two versions of the `experiment.xsd.xml` document, one for version 3.1 and one for version 3.2, as well as moving the definition of the `<sampleSet>` element to a new `sampleset.xsd.xml` subschema document and changing `root.xsd.xml` to also include the new sampleset subschema. We could write a very short `experimentBundle.xml` document, then use SQUASH to create a temporal `experiment.xml` schema, and do the same for the root schema.

What do we do with our actual XML document (`3.1.xml` in Figure 8), whose schema is the original root schema (`root.3.1.xsd`)? We take each instance of the `<sampleSet>` element out of its enclosing `<ExperimentClass>` element and move it up to beneath the root of the document (the `<pharmgkb>` element), replacing it with a `<sampleSetXref>` element. Then we take the two documents, the first using the old schema (`3.1.xml`) and the second the updated document (`3.2.xml`) and SQUASH them into a temporal document (`rep.xml`). (Even better, we could use a temporally-aware XML editor to make these changes to the document. Such an editor would output the temporal document.)

What would the representational schema look like for this temporal document? We could see that schema directly by running SCHEMAMAPPER on our bundle. A portion of the temporal document is shown in Figure 9. Note that every change of the base schema (which is what occurred here) or in the physical annotation results in a new `<tv_version_i>` element within the time-varying root (with these names being generated by SCHEMAMAPPER). The conventional validator can thus check to ensure that prior to the schema change on May 25, `<ExperimentClass>` elements contained an `<sampleSet>` element, and afterward, an `<sampleSetXRef>` element. (SQUASH will ensure that the appropriate `<version>` is used in the generated temporal document; τVALIDATOR will also check this.)

Continuing with the example, in Version 4.0 an `<ExperimentClass>` can now cross-reference more than one `<sampleSet>` (cf. Figure 3: note `unbounded` for `maxoccurs`). Additionally, a `<sampleSet>` is now a set of `<sample>` instead of a set of `<subject>`.

The latter change can be checked by the conventional validator because such sub-elements would themselves be enclosed in a new `<tv_version_3>` element. The former change, however, possibly cannot be checked by the conventional validator. In the earlier schema, with a `maxoccurs` of 1, the temporal semantics of this integrity constraint is *sequenced* [18]: *at every point in time*, there can be a maximum of one such element. However, depending on the physical annotations, it may be that the `<sampleSet>` element is itself versioned, which implies that an `<ExperimentClass>` element could have

```
<?xml version="0.1" encoding-"UTF-8"?>
<time-varying-root bundle="bundle.xml" ...>
  <tv_version_1>
    <tTime>May 1, 2004<tTime>
    <pharmGKB>
         ...
      <ExperimentClass>
           ...
        <sampleSet> ... </sampleSet>
           ...
       </ExperimentClass>
     </pharmGKB>
   </tv_version_1>
   <tv_version_2>
     <tTime>May 29, 2004<tTime>
     <pharmGKB>
         ...
       <ExperimentClass>
           ...
          <sampleSetXRef>...</sampleSetXRef>
           ...
       </ExperimentClass>
       <sampleSet>
           ...
       </sampleSet>
      </pharmGKB>
   </tv_version_2>
</time-varying-root>
```

Figure 9. A portion of a temporal document (`rep.xml`).

several `<sampleSet>` elements, each resident at non-overlapping periods, so that at any one time, there wouldn't be more than one. In this case, this integrity constraint would need to be checked separately by the time-varying data checker, which knows the temporal extent of the integrity constraint (from the bundle), and thus could check for a maximum of one only before Version 4.0 went into effect.

τVALIDATOR is a direct replacement for the conventional validator. If it is provided with a conventional schema and a conventional XML document (such as `root.1.0.xsd` and `1.0.xml`), it simply invokes the conventional validator. The UNSQUASH tool is similarly configured. If it is given a temporal document (e.g., `rep.xml`) that references a temporal bundle (versioned or not; here, `bundle.xml`), it will produce a conventional XML document by taking a timeslice at *now* (`4.0.xml`); this conventional document will reference a conventional XML Schema (`root.4.0.xsd`), formed by slicing the bundle at *now*. If UNSQUASH is given a static XML document, it simply returns that document. Hence UNSQUASH can be invoked before any conventional XML tools. In this way, *temporal upward compatibility* [3] is ensured.

This arrangement works very well. However, there are one remaining aspect that does not show up with time-varying data, but rather is unique to versioned schemas: an evolving definition of keys.

5. Accommodating Evolving Keys

When documents vary over time, it is important to identify which elements in successive snapshots are in actuality the same element, varying over time. We refer to the process of associating elements that persist across various snapshots as *gluing* the elements. SQUASH must do this gluing; the time-varying data checker within τVALIDATOR must also on occasion glue elements.

When a pair of elements is glued, an *item* is created. An item is an element that evolves over time through various versions. Determining which elements should be glued depends on two factors: the *type* of the element, and the *item identifier* for that element's type. Elsewhere we describe in detail how item identifiers are specified and how the gluing is accomplished [11]. What is relevant for our purposes here is that item identifiers are specified in the temporal annotations, are usually the (snapshot) key of the element type [6] given in the base schema, and are used by τVALIDATOR to extract the items from the temporal document and then check the temporal constraints on those items.

What if either the snapshot key (specified in the base schema) upon which an item identifier is defined, or if the item identifier itself (specified in the temporal annotation) changes? This is a particularly insidious kind of quicksand. Even worse is when the underlying element type of an item changes. For example, in Version 3.3 of PHARMGKB schema, the `<assay>` element was replaced with `<sequencingAssay>` and `<genotypingAssay>` elements. An item that was a particular `<assay>` element before the schema change could be associated with a particular `<sequencingAssay>` element in the snapshot document associated with the latter schema.

Our solution is to put in the `<schemaAnnotation>` element, which signals a change in some aspect of the schema, an `<itemIdentifierCorrespondence>` element, specifying how old item identifiers are to be mapped to new item identifiers. This element has four attributes: `oldRef`, a string naming an item that appears in the old schema, `newRef`, a string naming an item that appears in the new schema, `mappingType`, an XML Schema enumeration, and optionally a `mappingLocation`, which is a URI. We have defined four mapping types.

- `useNew`: The new identifier must also be present in the old element.

- `useOld`: The old identifier must also be present in the new element.
- `useBoth`: An attribute's name is changed, but its value isn't.
- `replace`: Use an externally-defined mapping.

This is best described with an example. Say that on Monday the item identifier is the `assayNumber` attribute of the `<assay>` element. On Tuesday, this attribute is renamed `assayID`; we specify a mapping type of `useBoth`. On Wednesday, the item identifier is changed to the `name` attribute, with a mapping type of `useNew`. (This attribute has been around since Monday, but it wasn't used as a key until Wednesday.) On Thursday we add a new attribute, `assayKey`, and specify that as the item identifier, with a mapping type of `useOld`. Finally, on Friday we replace the `<assay>` element with a `<genotypingAssay>` element, with a `genoID` attribute as the item identifier and a mapping type of `replace`.

The gluing of elements into items is then done the following way. Before Tuesday, the `assayNumber` is used for gluing. When the schema change occurs sometime on Tuesday, we glue across the schema change by matching the `assayNumber` value of the element before the schema change with the `assayID` value after the change: these (integer) values must match for the two elements to be glued. On Wednesday, we glue across the schema change by matching up old elements and new elements that have the same (string) value for their `name` attribute, the new item identifier. The only difference is that before the schema change, that attribute was present but wasn't being used as a key. In a consistent fashion, on Thursday we also glue using the `name` attribute, which was the *old* item identifier.

Friday is the most complex. We need to glue an `assay` element with an item identifier of `assayKey` with a `genotypingAssay` element with an item identifier of `genoID`. For this, we use the `MappingLocation` attribute in the bundle to access a mapping table that provides a list of pairs, each with an `assayKey` and a `genoID` value. (Of course, the mapping location document can also be time-varying; τVALIDATOR extracts the relevant time-slice with UNSQUASH.)

This completes the picture. To validate a time-varying document associated with a time-varying schema, τVALIDATOR applies the conventional validator to the document, using the representational schema produced by SCHEMAMAPPER. It then determines the times when the schema changes, thus determining the periods when the schema is constant, termed the *schema-constant periods*. These periods will be non-overlapping and continuous; between the periods are schema change *walls*. For each such period, the time-varying data checker is invoked to check the temporal integrity constraints over the time-varying data, with the *single* base schema, temporal annotation, and physical annotation (see Figure 10). Then the temporal constraint checker glues across the schema change walls and performs the temporal checks across these walls. For example, if a temporal annotation states that there can be at most three such values within a year (a rather complex kind of temporal constraint), the temporal constraint checker will ensure that the number of unique values before the wall and the number of unique values after the wall do not together exceed three. For most temporal constraints, it suffices to just check independently before and after the wall. Only for certain kinds of *non-sequenced* constraints [18] does the temporal constraint checker get involved.

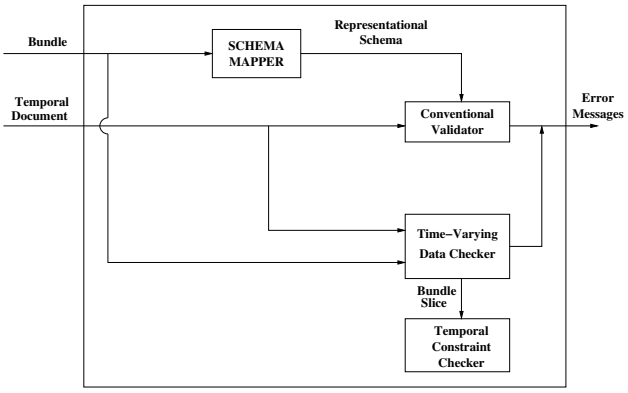

Figure 10. Validating a Document with a Time-Varying Schema.

6. Related Work

Methods to represent temporal data and documents on the web have been actively researched. This research has covered a wide range of issues that include architectures for collecting document versions (cf. [10, 15]), data modeling of time-varying data (cf. [1]), strategies for storing versions (cf. [7]), studies on the frequency of data change (cf. [8]), and temporal query languages (cf. [12]). Grandi has created a bibliography of previous work in this area [13].

There is only one previous paper on validation: our paper that introduced τXSchema but did not discuss schema versioning [9]. Schema versioning has been previously researched in the context of temporal databases [17]. But an XML schema is a grammar specification, unlike a (relational) database schema, so new techniques are required. Though various XML schema languages have been proposed in the literature and in the commercial arena (cf. [14]) for a summary), none model schema changes nor provide for versioning. We chose to base our research on XML Schema because it is backed by the W3C and is the most widely-used schema language.

Recently there has been interest in the incremental validation of XML documents [4] using static schemas, which has application in the area of data streaming. To the best of our knowledge, the effect of changes to the schema during incremental validation is an open area of research. We do not address incremental validation in this paper.

7. Conclusion

This paper shows how schema versioning can be integrated with support for time-varying documents in a fashion consistent and upwardly-compatible with XML, XML Schema, and conventional XML validators. Schema versioning in its full generality is supported, including (time-varying) schemas that include or reference other (time-varying) schemas. Bundles are used uniformly to denote the schema of a temporal document; SCHEMAMAPPER is used to generate a representational schema when needed.

By identifying when schema changes occur, the schema-constant periods can be identified. Such periods have the very useful property that there is an unchanging schema (comprised of a single base schema, a single temporal annotation document, and a single physical annotation). The dance between the conventional validator, the time-varying data checker, and the temporal constraint checker ensures that most of the checking is done by the conventional validator, with most of the remaining checking done by the time-varying data checker.

In the future, we plan to integrate τXSchema with a schema-aware XML-based editor like XMLSpy. Schema-aware editors generate easy-to-use templates for updating each type of element defined in a schema. But they do not track changes to either the schema or the data. Enabling versioning for both will support unlimited undo/redo, improve change tracking, and aid in cooperative editing. Another direction of future work is to add versioning to XUpdate. XUpdate is a language for specifying changes to an XML document. By specifying how the evaluation of an XUpdate statement on an XML Schema document modifies a bundle, we should be able to support schema versioning in XUpdate.

8. Acknowledgments

We thank Lingeshwaran Palaniappan and Eric Roeder for their help in the development of SCHEMAMAPPER, SQUASH, UNSQUASH, and τVALIDATOR. NSF grants IIS-0100436, IIS-0415101, and EIA-0080123 and grants from the Boeing Corporation, Microsoft, and the Arizona Technology and Research Initiative Fund through the University of Arizona Internet Technology, Commerce and Design Institute provided partial support for this work. The reviewers provided helpful comments.

References

[1] Amagasa, T., M. Yoshikawa and S. Uemura, "A Data Model for Temporal XML Documents," In *Database and Expert Systems Applications, 11th International Conference, DEXA 2000*, pages 334–344, London, UK, September 2000.

[2] Snodgrass, R. T. and I. Ahn, "Temporal Databases," *IEEE Computer* 19(9):35–42, September, 1986.

[3] Bair, J., M. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Notions of Upward Compatibility of Temporal Query Languages," *Business Informatics (Wirtschafts Informatik)* 39(1):25–34, February, 1997.

[4] Barbosa, D., A. Mendelzon, L. Libkin, L. Mignet,and M. Arenas, "Efficient Incremental Validation of XML Documents." in *ICDE*, pp. 671–682, 2004.

[5] Bratman, H., "An Alternate Form of the "UNCOL Diagram"," *CACM* 4(3):142, 1961.

[6] Buneman, P., S. Davidson, W. Fan, C. Hara, and W. Tan, "Keys for XML," *Computer Networks* 39(5): 473–487, 2002.

[7] Chien, S., V. Tsotras, and C. Zaniolo, "Efficient schemes for managing multiversionXML documents," *VLDB Journal*, 11(4): 332–353.

[8] Cho, J. and H. Garcia-Molina, "Estimating Frequency of Change," *ACM Transactions on Internet Technology*, 3(3): 256–290, 2003.

[9] Currim, F., S. Currim, C. Dyreson and R. T. Snodgrass, "Effecting Data Independence for Temporal XML Schemas," in *Proceedings of the International Conference on Extending Data Base Technology*, Crete, pp. 348–365, 2004.

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

[11] Dyreson, C., R. T. Snodgrass, F. Currim, and S. Currim, "Schema-mediated Exchange of Temporal XML Data," Technical Report, November, 2005.

[12] Gao, D. and R. T. Snodgrass, "Temporal Slicing in the Evaluation of XML Queries," in *VLDB*, pp. 632–643, 2003.

[13] Grandi, F., "A Bibliography on Temporal and Evolution Aspects in the World Wide Web," *TimeCenter* TR-75, 2003.

[14] Lee, D. and W. Chu, "Comparative Analysis of Six XML Schema Languages," *SIGMOD Record* 29(3):76–87, September 2000.

[15] Marian, A., S. Abiteboul, G. Cobena and L. Mignet, "Change-Centric Management of Versions in an XML Warehouse," in *VLDB*, Roma, Italy, pp. 581–590, 2001.

[16] McKeeman, W. M., J. J. Horning, and D. B. Wortman, **A Compiler Generator**, Prentice-Hall, Englewood Cliffs, NJ., 1970.

[17] Roddick, J. F., "Schema Evolution in Database Systems—An Annotated Bibliography," *SIGMOD Record*, 21(4), pp. 35–40, 1992.

[18] Snodgrass, R. T., **Developing Time-Oriented Database Applications in SQL**, Morgan Kaufmann Publishers, Inc., San Francisco, CA, July, 1999, 504+xxiv pages.

[19] Snodgrass, R. T., S. Gomez and E. McKenzie, "Aggregates in the Temporal Query Language TQuel," *IEEE Transactions on Knowledge and Data Engineering* 5(5):826–842, October, 1993.