Validating Quicksand: Temporal Schema Versioning in τXSchema

Richard T. Snodgrass[1] Curtis Dyreson[*,2] Faiz Currim[3] Sabah Currim[4] Shailesh Joshi[5]

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.

Key words:

* Corresponding author.
Email addresses: `rts@cs.arizona.edu` (Richard T. Snodgrass), `cdyreson@eecs.wsu.edu` (Curtis Dyreson), `faiz-currim@uiowa.edu` (Faiz Currim), `scurrim@ci.fsu.edu` (Sabah Currim), `shailesh@cs.arizona.edu` (Shailesh Joshi).

[1] University of Arizona, Tucson, AZ, USA
[2] Washington State University, Pullman, WA, USA
[3] University of Iowa, Iowa City, IA, USA
[4] Florida State University, Tallahassee, FL, USA
[5] University of Arizona, Tucson, AZ, USA

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 stand. But 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 floundering in quicksand.

An increasing amount of data is being published in XML. The W3C XML Schema recommendation defines the structure and data types for XML documents [15]. 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.

In a dynamic environment, as organizational contexts and user requirements change, it is natural to see consequent changes in application schemas. Schema 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[6]) maintains a repository of XML Schemas[7] 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[8]) "contains genomic, phenotype and clinical information collected from ongoing pharmacogenetic studies." The evolution of its schema has reached version 4.0[9].

[6] `http://www.bgbm.org`
[7] `http://www.bgbm.org/biodivinf/schema/default.asp`
[8] `http://www.pharmgkb.org/`
[9] `http://www.pharmgkb.org/schema/history.html`

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. 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.

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, which we call *schema-constant periods*, first appeared in a paper by one of the authors on temporal aggregation [21]. 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 within 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.

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 seemingly precarious situation of data that changes over time, while its schema and even its representation are also changing over time.

All times mentioned in this paper are from the *transaction time* dimension [20], though τXSchema also supports *valid time* for data versioning. While schema versioning has been considered in the context of valid time [8], doing so is quite complex and in our opinion not worth this complexity. Thus in τXSchema schemas vary and are versioned only over transaction time.

We also note that the emphasis here is on capturing a time-varying schema and validating documents against such a schema. Our approach applies to *unmanaged* environments, where each schema is originally in a separate document paired with one or more data documents at particular points of time. We also support *managed* environments, where a schema editor would be used to maintain the schema(s), which the schema changes captured in a temporal document. *How* the schema changes are made, or what kinds of schema evolution operations are provided, are beyond the scope of this paper.

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>`s instead of a set of `<subject>`s (logically!).

```
<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>
```

Fig. 1. `<ExperimentClass>` element in version 3.1

Now let's examine how this could have been done using τXSchema. (Our emphasis in this paper is on how to validate a time-varying document against a time-varying schema. Hence, we only describe τXSchema to the point where we can explain validation. For more discussion of τXSchema per se, please consult prior papers [9,11].)

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 which are shaded in the figure are specific to an individual

```
<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>
```

Fig. 2. `<ExperimentClass>` element in version 3.2

time-varying document and need to be supplied by a user. We also note that to this point, the schemas (boxes 4, 5, 6, and 7) are static. We'll later relax this assumption.

The designer starts with the base schema (box 4). In the case of PharmGKB the base schema is `root.xsd`. It `xsd:import`s and `xsd:include`s other 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 element itself may appear at certain times (and not at others), and whether its content changes. Elements that are

```
<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>
```

Fig. 3. `<ExperimentClass>` element in version 4.0

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 included in the TXSchema. (Due to space limitations, we won't describe in detail these annotations—more detail can be found elsewhere [9]—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 timestamps is largely independent of which components vary over time. Timestamps can be located either on time-varying components (as specified by the temporal annotations) or somewhere above such components. 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).

τXSchema supplies a default set of physical annotations. (Again, space limitations do not allow us to describe these annotations in detail, for more see [9].) 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 a 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.

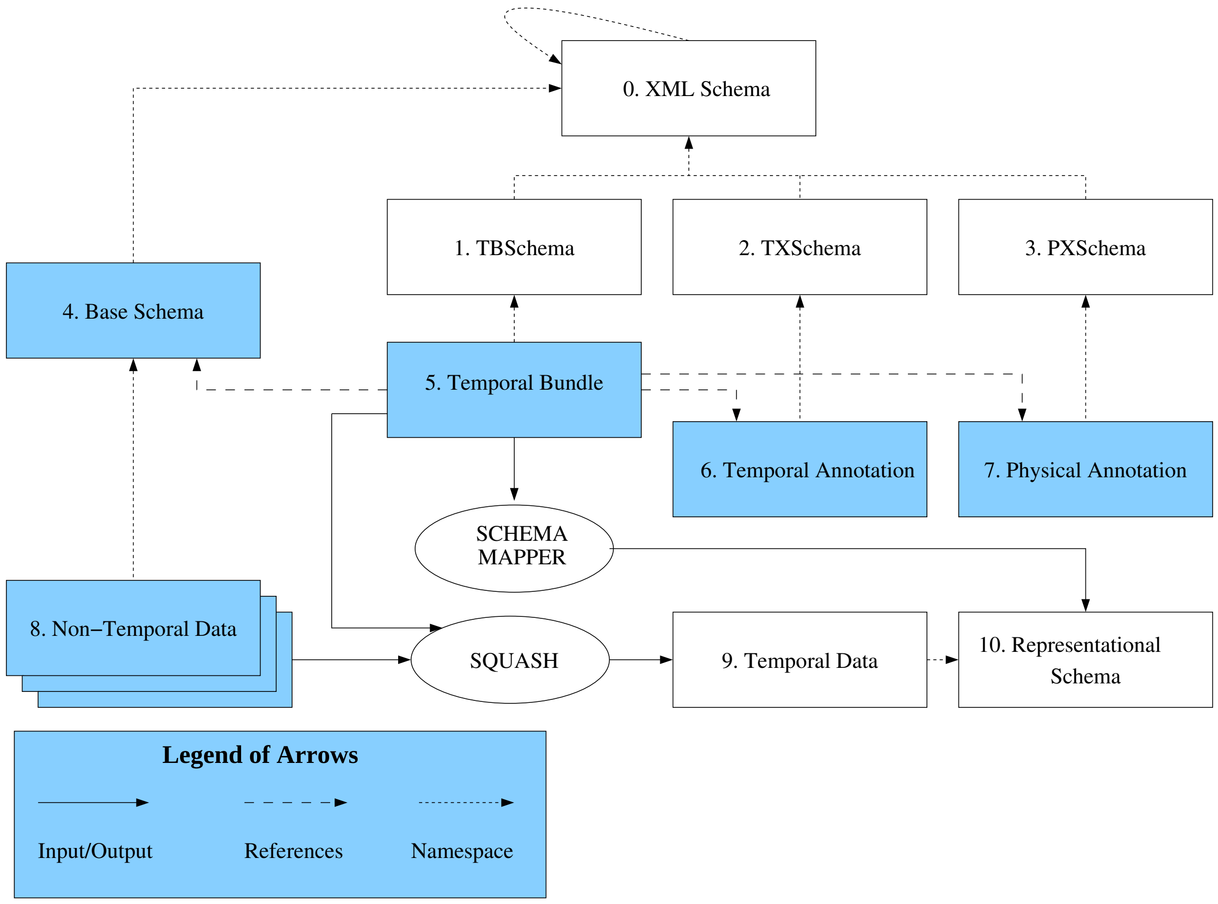

Fig. 4. Architecture

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`, but other formats are possible, e.g., SQL datetimes. 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, a base schema, `root.xsd`, can include a gene sequence schema, `sequence.xsd`, which itself includes a schema for experiment data, `experiment.xsd`.

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`). The temporal XML document (box 9) is essentially a timestamped representation of this sequence of non-temporal XML data files (box 8). 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 (box 9, which is the representational document `rep.xml`), with its XML schema shown as box 10, generated by SchemaMapper from information in the temporal bundle. The schema of the temporal document is its associated representational schema (an XML Schema document). 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, validation is divided into two separate analyses. In one analysis, the conventional validator is applied to the document to check the temporal document against the representational schema. That schema is generated internally by the SchemaMapper, then handed to the conventional validator. However, there are 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 during a second analysis by the time-varying data checker, operating over the temporal document.

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 add two sets of annotations to a base schema, 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.

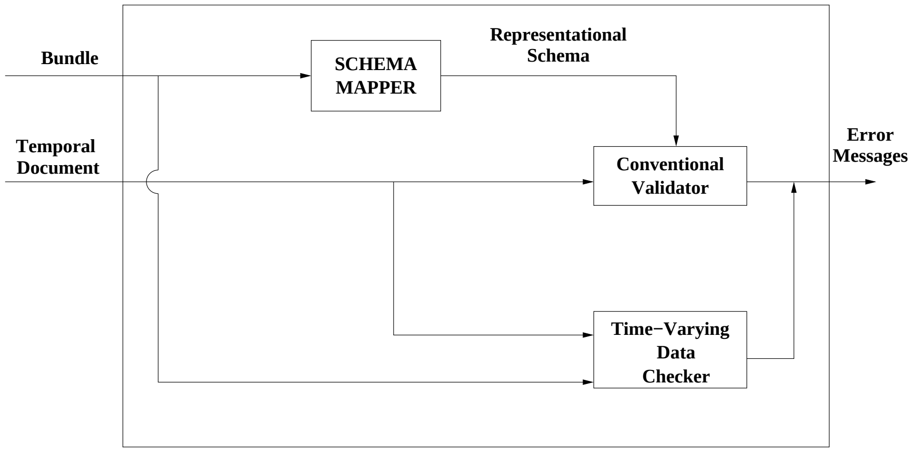

Fig. 5. Validating a Document with Time-Varying Data

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 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`, 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`. 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 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

```
<?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>
```

Fig. 6. A Temporal Bundle for PharmGKB: `bundle.xml`

schema annotation elements. Similarly, we use Squash to generate temporal schemas for `sequence.xsd` and `experiment.xsd`.

This rather involved state of affairs, with time-varying documents and time-varying schemas, is illustrated with a T Diagram in Figure 7. In this notation, first described over forty years ago [4], 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.xsd`). The name of these diagrams was to the best of our knowledge given by McKeeman, Horning, and Wortman in their classic compiler book [17].

We extend these diagrams to allow multiple inputs, which unfortunately complicates them somewhat. As shown in Figure 7, 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 `XMLSchema.xsd`.

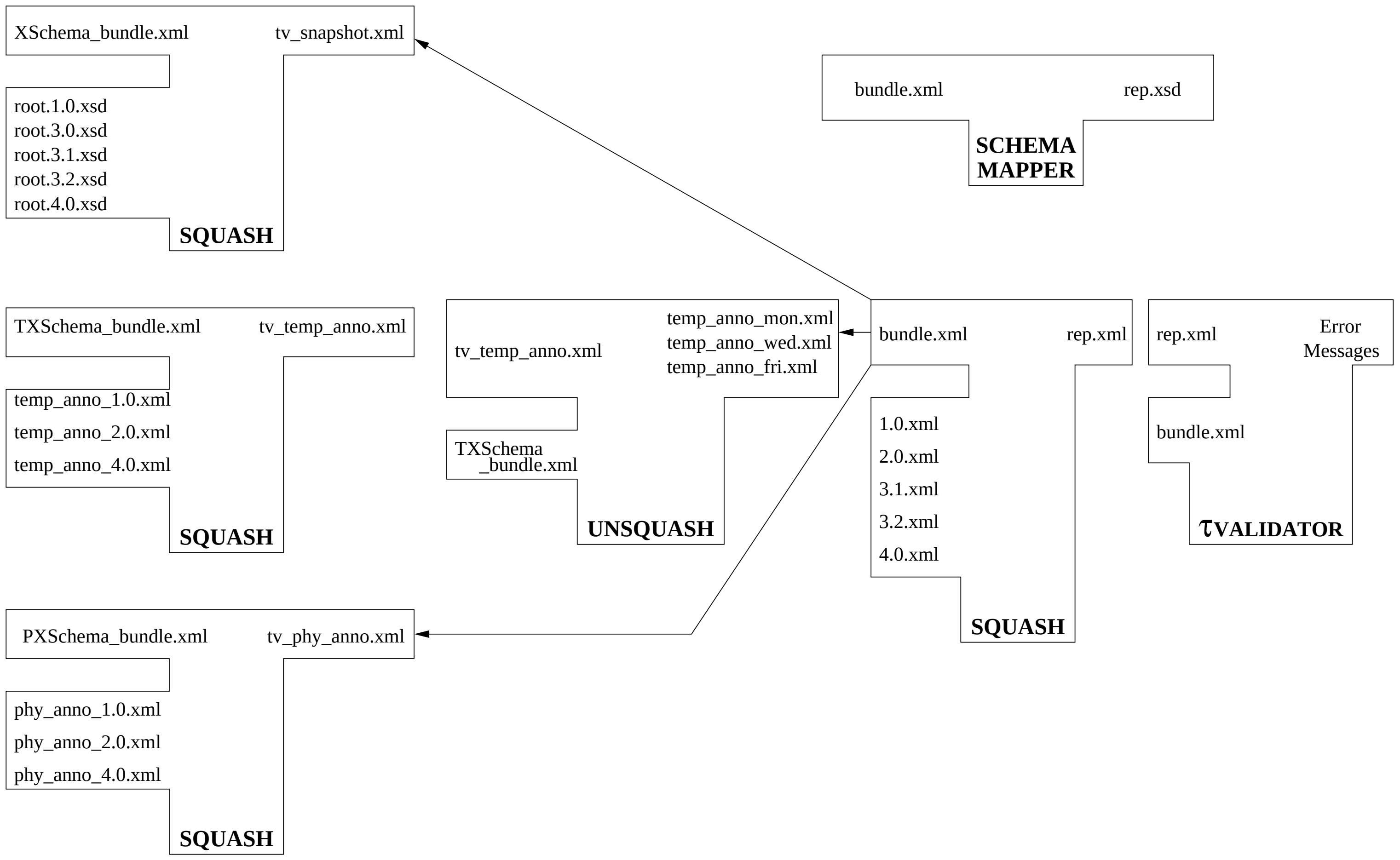

Fig. 7. T Diagram of Validation

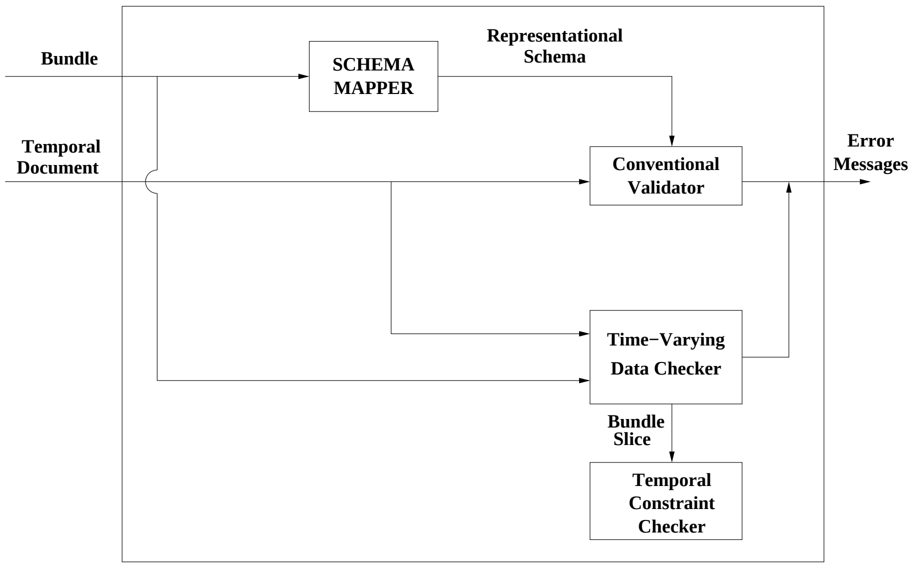

Fig. 8. Validating a Document with a Time-Varying Schema

5 Validating Against a Time-Varying Schema

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 (see Figure 8). 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.

During this process, 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 9. `root.xsd` is a time-varying document, containing several schema versions. In this context, τVALIDATOR will utilize the temporal context 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 passes the same information 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

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

Fig. 9. An excerpt from the time-varying Temporal Bundle for PharmGKB

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 8 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.

This change is reflected in two versions of `experiment.xsd`, 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` subschema document and changing `root.xsd` to also include the new sampleset subschema. We could write a very short `experimentBundle.xml`, then use Squash to create a temporal `experiment.xml` schema, and do the same for the root schema.

What do we do with an actual XML document (such as `3.1.xml`, version 3.1 of PharmGKB), 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. This is the *managed* environment mentioned earlier.)

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 10. 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;

```
<?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>
```

Fig. 10. A portion of a temporal document (`rep.xml`)

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.

A temporal constraint is termed as *sequenced* with respect to a similar snapshot constraint in the schema document, if the semantics of the temporal constraint can be expressed as the semantics of the snapshot constraint applied at each point in time [19]. Given a snapshot XML Schema constraint, we can define the corresponding temporal semantics in τXSchema in terms of a sequenced constraint. In the earlier schema, with a `maxOccurs` of 1, the temporal semantics of this integrity constraint is the sequenced constraint, "*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 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 component in τVALIDATOR, 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. In some cases, the representational schema can be designed such that many sequenced constraints can be checked directly by the conventional validator.

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* [2] is ensured.

This arrangement works very well. However, there are four remaining aspects that do not show up with time-varying data, but rather are unique to versioned schemas: (1) an evolving definition of keys, (2) accommodating gaps in lifetimes, (3) the semantics of mixed data and schema changes within a single transaction, and (4) checking non-sequenced constraints across schema changes. We examine each in turn.

6 Accommodating Evolving Keys

When documents vary over time, it is important to identify which elements in successive snapshots are in actuality the same *item*, An item is an element that exists in one or more snapshots. Each change to an item creates a new *version* of that item. We refer to the process of associating elements that persist across various snapshots as *gluing* the elements. So the elements are glued to create an item. SQUASH must do this gluing; the time-varying data checker within τVALIDATOR must also on occasion glue elements.

Determining which elements should be glued into an item 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].

When a schema-change wall is encountered, items across the wall need to be associated. This process is called *cross-wall gluing*, or *bridging*. Figure 11 depicts the concepts of gluing and bridging.

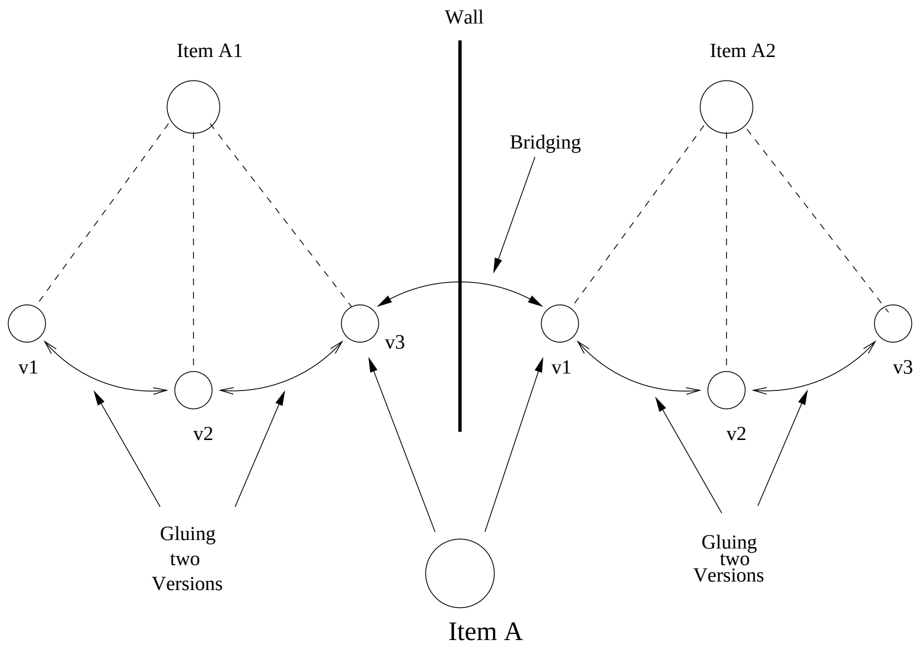

Fig. 11. Gluing and Bridging

In this figure, individual elements in individual versions of an XML document are depicted as small circles in the center of the figure. Here we see six elements, three of which are determined to be versions of the same item (A1) and three of which are determined to be versions of another item (A2). The wall indicates that the schema was changed between the third and fourth version of the document.

Gluing uses the item identifier to associate the first three elements with an item and likewise the second three elements. Bridging determines that the element that is version 3 of item A1 and the element that is version 1 of item A2 are actually versions of the same item, item A. So in fact item A has *six* versions, the three elements before the schema change (indicated in the figure as the wall) and the three elements after the schema change. Gluing and bridging occur in different stages within the validator; both conspire to realize an item across schema changes, which is the first step in checking the temporal constraints associated with that item's definition in the schema.

What is relevant for our purposes here is that each item identifier is specified in the temporal annotations. Usually the item identifier is the same as the (snapshot) key of the corresponding element type [5] as given in the base schema. The identifier is used by τVALIDATOR to extract the items from the temporal document and to 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.

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

This is best described with an example. Say that in March the item identifier is the `assayNumber` attribute of the `<assay>` element. In May, this attribute is renamed `assayID`; we specify a mapping type of `useBoth`. In May, the item identifier is changed to the `name` attribute, with a mapping type of `useNew`. (This attribute has been around since March, but it wasn't used as a key until May.) In June we add a new attribute, `assayKey`, and specify that as the item identifier, with a mapping type of `useOld`. Finally, in July 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 May, the `assayNumber` is used for gluing. When the schema change occurs sometime in May, 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. In May, 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 (`name`) was present but wasn't being used as a key. In a consistent fashion, in June we also glue using the `name` attribute, which was the *old* item identifier.

July 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, the `MappingLocation` attribute in the bundle points to a mapping table that provides a list of pairs, each with an `assayKey` and a `genoID` value.

This list of pairs is termed a *replace mapping list*. As it is instance-based, containing as it does a list of key *values*, the replace mapping list should only be used as a last resort. Its role is to allow bridging for all cases in which the other three mapping types, which have no need for storing instance information in the schema, are inappropriate.

Of course, the mapping location document can also be time-varying; τVALIDATOR extracts the relevant timeslice with UNSQUASH. We will see shortly how this is useful in the presence of gaps.

7 Accommodating Gaps

Bridging is more involved when there are *gaps* in the lifetime of an item. Gaps make the process of finding the correspondence between the items from consecutive schema-constant periods more difficult. If there are gaps in the lifetime of an item, bridging becomes even more complex.

Figure 12 shows three cases that may arise while bridging the items from consecutive schema-constant periods. It shows the data and schema changes along the transaction-time dimension, from left to right. The schema-change walls are shown as bold vertical lines. The horizontal lines depicts the evolution of a particular item (in this case, three separate items). The bridging process is shown by the jumpers over schema change walls. A dotted line indicates when the item did not exist in the database. The first item existed during the entire transaction time period depicted in this figure. There is a single gap in the existence time of the second item: it ceased to exist sometime during P_1 but reappeared in P_2. The third item had a much longer gap, reappearing only in P_3.

We now now examine each item in turn.

1. The item A (the first horizontal line) is present throughout schema-constant periods P_1 and P_2. Thus the last snapshot of P_1 and the first snapshot of

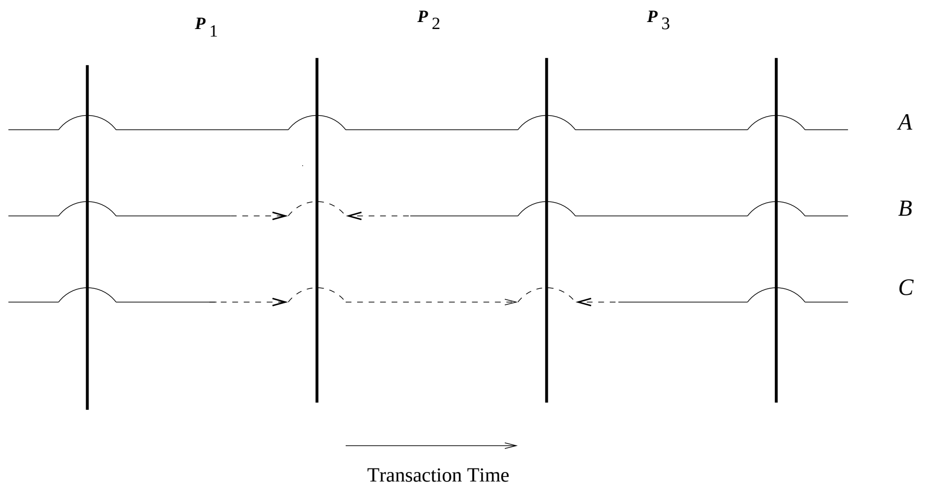

Fig. 12. Cross Wall Gluing

P_2 contains versions of item A. Here, no extra work is needed as the items can be bridged directly using one of the above four methods.

2. The item B (the second horizontal line) disappeared for some time in P_1 and reappeared about halfway through in P_2. Thus the last snapshot of P_1 and first snapshot of P_2 will not contain versions of item B. Bridging these two items in this case involves virtually extending period of item B's last version until the end of P_2 as if it were present during the last snapshot; and virtually extending its first version's period until the start of P_3; and then performing the bridging using one of the above four methods. Each virtual extension is depicted as a dashed line with an arrow indicating the direction the extension was made. In an implementation, this could be done by simply checking item's last version from P_1 and first version from P_2.
3. An item could also disappear for one or more schema-constant periods and then reappear again. Item C (the bottom horizontal line) was present for initial part of P_1. It then disappeared over entire period P_2 and again appeared in the later half of P_3. For such cases, bridging involves virtually extending the period of the item C's last version from P_1 over multiple schema-constant periods followed by bridging using one of above methods. So P_1's version is extended to the wall, then bridged to a virtual element over all of P_2, then bridged to the extended element in P_3.

Figure 13 illustrates the most complex situation of cross-gap gluing over multiple schema-constant periods. Documents in the top right part of the figure show the temporal bundle documents corresponding to schema-change walls in March, May and July respectively. The two time lines correspond to an `<assay>` item. The top time line is that contained in the March document; the bottom one is that contained in the July document. The `replace` and `useNew` methods are used for item correspondences in July and September, respectively. The item identifier during this period is the attribute '`assayKey`' of `<assay>`. In July we replace the `<assay>` element with a `<genotypingAssay>`

element, with a `genoID` attribute as the item identifier and a mapping type of `replace`. In September the item identifier is changed back to the `name` attribute, with a mapping type of `useNew`.

The item is present during the initial part of schema-constant period P_1, but is removed sometime during June, as indicated by a terminated line in the middle of P_1. A schema change takes place in July. Since this item is absent during P_2, no item correspondence is necessary in the replace mapping list.

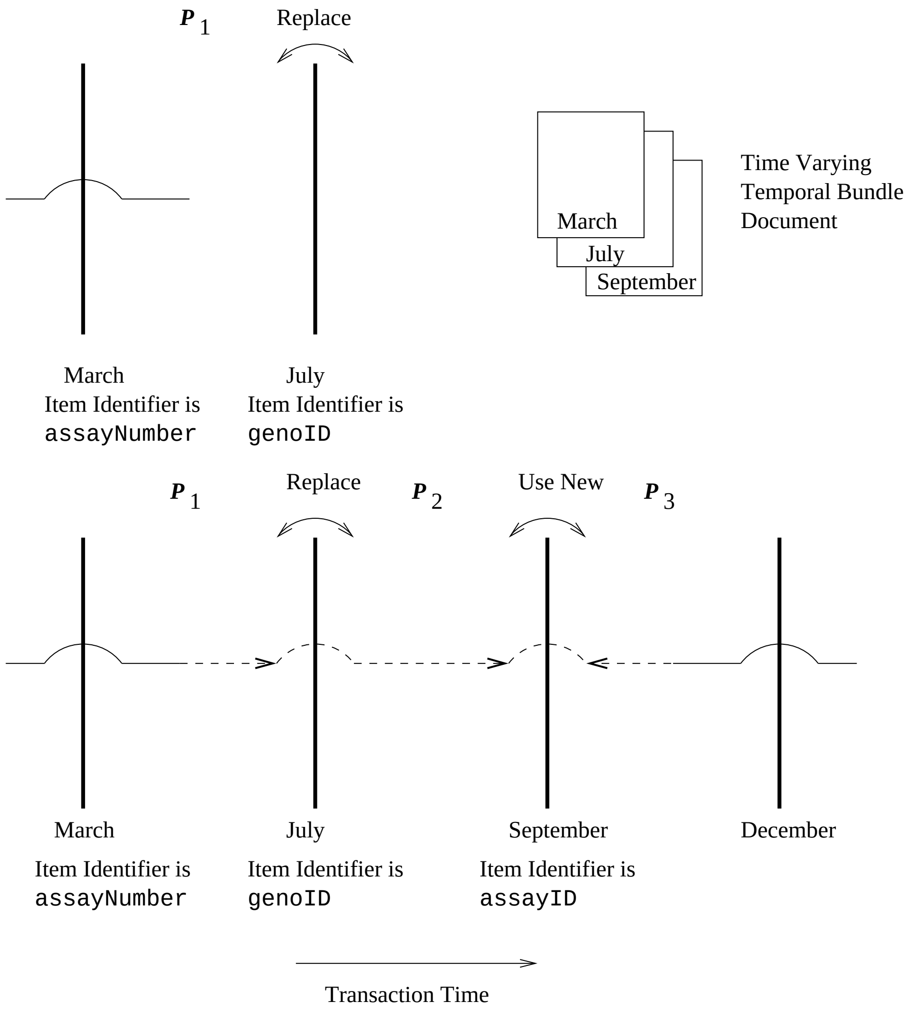

Fig. 13. Cross-Gap Gluing

A second schema-change takes place in the month of September. An `<genotypingAssay>` element that is in fact a version of the old `<assay>` element present in January reappears sometime in November. At this point the user wants to associate this new element with the old one from P_1 since both represent the same assay. In order to perform this association, the user will need to add a pair of identifiers to the old replace mapping list for the month of July to handle this virtually extended element. Multiple versions of the replace mapping list could also be maintained as a temporal document; the temporal validator would then extract the relevant snapshot from it.

8 Transaction Semantics

A data change in XML documents can co-exist with schema changes within a single transaction, and hence can occur at exactly the same (transaction commit) time. With schema changes coming into picture, we also need to consider other factors like name and relative path changes for item identifier fields and other elements that constitute the content of an item, complicating the process of bridging and hence validation.

We considered three ways to handle this situation.

1. Disallow all data change in any transaction containing a schema change. This is the most stringent option and makes the user's job more difficult, forcing her to split the task into multiple transactions. This may not be always feasible from real world point of view. Consider a situation where an element is modified to have a new 'required attribute', data change is mandatory in this case and hence cannot be separated from schema change. It could be argued that this is achievable with the addition of a new 'optional' attribute, followed by required data changes and then making the attribute required. But the burden of work falls on the user.
2. Allow schema changes to coexist with data changes, except for schema changes to item identifier fields. This will eliminate the need to have a replace mapping list since the bridging could always be done using one of the three options '`useNew`', '`useOld`', or '`useBoth`'.
3. Allow data changes to coexist with schema changes within a transaction without any restrictions.

We decided to go with the third approach, as it is the most general. In doing so a refined transaction management semantics was needed. Specifically, we consider any data change in the first snapshot of a schema-constant period to have taken place *after* any schema changes within that same transaction. Thus, even if the the timestamps for the schema change and the data change are identical (as both changes occurred in the same transaction), an assumption is made that the data change took place following the schema change.

9 Non-Sequenced Constraints

A constraint is *non-sequenced* if it is applied to a temporal item as a whole (including the lifetime of the data entity) rather than to individual time slices.

Non-sequenced constraints are defined in the temporal annotation as an extension of snapshot XML Schema constraints. An example of a non-sequenced (cardinality) constraint is: “An item cannot change more than three times in a year.” This type of constraint cannot be validated using the conventional validator and thus needs to be validated using the ‘Temporal Constraint Checker’ module of τVALIDATOR.

As mentioned in Section 2, schemas vary only over transaction time. Hence, non-sequenced constraint validation is easier in valid time, as schema changes cannot occur.

We considered two alternatives for the applicability of a non-sequenced constraint across schema changes.

(1) The constraint is applicable only within the schema-constant period in which it is defined.
(2) The constraint once defined becomes applicable to the entire document.

As per the first approach, any violation of a constraint during previous schema-constant-periods is ignored, while in the second approach, the constraint may be violated even when first defined.

Consider a situation shown in Figure 14. It maintains the same conventions as Figure 12. Changes to an item are shown by X’s. A new non-sequenced constraint is introduced during third schema-constant period P_3 stating that “An item cannot change more than three times in a year.” But the item has already undergone four changes during previous schema-constant periods P_1 and P_2.

According to first alternative listed above, the constraint is not violated as long as the item does not change more than three times in the third schema-constant period. Until there are four changes made after the schema change, the constraint is not considered to be violated.

According to the second alternative semantics, there is immediately a violation of the constraint, due to activity during the previous two schema-constant periods. This situation could be handled in at least two possible ways.

(1) Flag an error and do not perform any further validations.
(2) Flag a warning saying that the item has violated the constraint over previous schema-constant-periods. No further changes would be allowed.

We decided on the second alternative in both cases: to apply a non-sequenced constraint to all previous schema-constant periods, as it is a more general approach. If such a situation is encountered, the validator would show a warning and would not allow any further changes to that item. Users can find out about

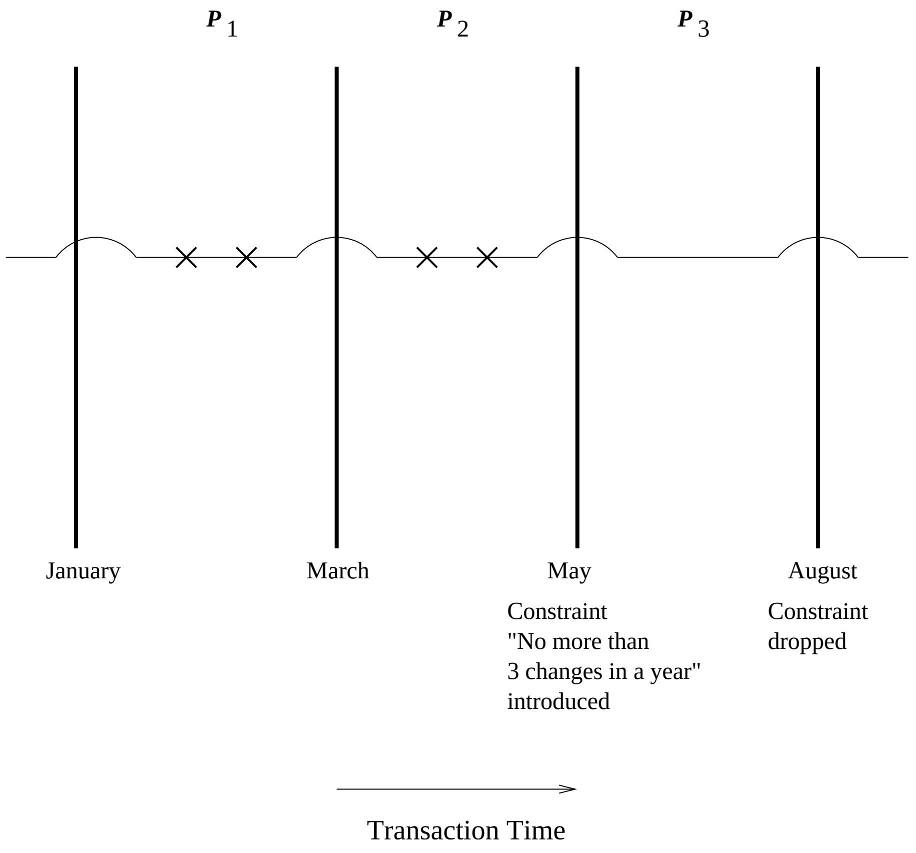

Fig. 14. Non-Sequenced Constraints

the reasons for the violation using some query language (e.g., XQuery) over the previous versions of data. The first approach can be simulated by restricting the *applicability period* of the constraint to a particular schema-constant period.

9.1 Summary

This completes the picture. For each schema-constant 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. 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 [19] does the temporal constraint checker get involved.

The pseudo-code is shown in Figure 15. Here, variables are in italics and helper functions are in boldface. The **timeVaryingDataChecker** returns items encountered in each constant period p. These items are bridged by the **temporalConstraintChecker** to coalesce into items that cross schema change walls.

```
τValidator ( temporalDocument )
    temporalSchema ← extractSchema ( temporalDocument )
    repSchema ← schemaMapper ( temporalSchema )
    Validator ( temporalDocument, repSchema )
    schemaConstantPeriods ← extractSCP ( temporalDocument )
    items ← ∅
    for each p ∈ schemaConstantPeriods do
        items ← items ∪ timeVaryingDataChecker (
                        timeSlice ( temporalDocument , p),
                        timeSlice ( temporalSchema, p) )
    temporalConstraintChecker ( temporalDocument, temporalSchema, schemaConstantPeriods,
```

Fig. 15. Pseudo-code for τValidator

10 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. [10,16]), data modeling of time-varying data (cf. [1]), strategies for storing versions (cf. [6]), studies on the frequency of data change (cf. [7]), and temporal query languages (cf. [13]). Grandi has created a bibliography of previous work in this area [14].

There our only two previous papers on validation of time-varying XML documents: our paper that introduced τXSchema but did not discuss schema versioning [9] and our paper that introduced cross-schema-change validation [12]. The present work extends the latter paper by discussing how to accommodate gaps in the existence time of an item, transaction semantics, and how to accommodate non-sequenced integrity constraints and augments the discussion with additional figures and pseudocode.

Schema versioning has been previously researched in the context of temporal databases [18]. 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. [15]) 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 [3] 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.

11 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.

12 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.

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] 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.

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

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

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

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

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

[8] De Castro, C., F. Grandi, and M. R. Scalas, "Schema Versioning for Multitemporal Relational Databases," *Information Systems*. 22(5): 249-290, 1997.

[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] Dyreson, C., R. T. Snodgrass, F. Currim, S. Currim, and S. Joshi. "Validating Quicksand: Schema Versioning in τXSchema," in *Proceedings of the Third International Workshop on XML Schema and Data Management* (XSDM 2006), Atlanta, Georgia, April, 2006.

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

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

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

[16] 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.

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

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

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

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

[21] 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.