Adding Temporal Constraints to XML Schema

Faiz Currim, Sabah Currim, *Member, IEEE,* Curtis Dyreson, *Member, IEEE,* Richard T. Snodgrass, *Senior Member, IEEE,* and Stephen W. Thomas, *Member, IEEE*

Abstract—[TODO: not decided]
Abstract needs to be added

Index Terms—XML Schema, temporal data, constraints, data management [TODO: complete].

1 Introduction

[TODO: Rick]
Introduction and motivation of our work. Also, background material on temporal data and databases, XML, XML Schema.

1.1 Motivation: Temporal

Limitations of XML Schema for temporal data; why augmenting XML Schema is needed

1.1.1 Motivation: Constraints

Value of constraints; why we need special consideration for temporal augmentations of XML Schema constraints

from TR

need to introduce the company example properly

should include XSD code for company example

We use the `company` example to describe the

The root of this schema is the `company` entity. Under that, there are `products` and `suppliers`. An `order` is considered a sub-element of `suppliers` (with a reference from `order` to `product number` for data integrity).

2 Related Work

[TODO: Sabah]
(TR: Section 4, 15)

- Temporal XML and XML Schema related efforts
- Temporal DB: constraint management (*new*)

from TR: part 1 related work

There are various XML schemas that have been proposed in the literature and in the commercial arena. We chose to extend XML Schema in τXSchema because it is backed by the W3C and supports most major features available in other XML schemas [**?**]. It would be relatively straightforward to apply the concepts in this paper to develop time support for other XML schema languages; less straightforward but possible would be to apply our approach of temporal and physical annotations to other data models, such as UML [**?**] (to produce temporally augmented class diagrams, for example). Previously, we have extended the Unifying Semantic Model, a conceptual model that extends the ER Model, to utilize annotations [**?**], very similar to what we propose here.

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 [**?**], strategies for storing versions [**?**], studies on the frequency of data change [?], temporal query languages [**?**], [**?**] and using events to trigger actions [**?**]. Techniques to capture the semantics of variants (alternatives of an element that can co-exist at a point in time) are orthogonal to our work, but have also been discussed [**?**], [**?**]. The logical representation of deltas between the versions and the aspects of physical storage policy for storing those versions have been proposed so as to maximize the space utilization [**?**]. Grandi and Mandreoli [**?**] sketch an infrastructure for adding valid-time timestamps to a web document, and formatting timeslices from the document using XSLT. They give an XML Schema definition for the timstamps, as we do in τXSchemafor our timestamps. Temporal and physical annotations are not discussed, nor are temporal constraints. Grandi has created a bibliography of previous work in this area [**?**]. More recent papers on version control include [**?**], [**?**], [**?**]. Iwaihara et al [?] discuss a versioned temporal model in the context of access control. The model represents changes between versions with a "delta graph," which logically induces a "version graph" (essentially a timeslice-based representation). The focus of the paper is an access control language for versions, unlike τXSchema, there is no ability to specify which elements are to be versioned, the time domain of the versioning, or the (logical) representation of the versions. Wond and Lam [?] present a version management system for XML data. The system stores a document's history as a combination of some complete versions and deltas. The deltas are edit scripts, and can be used to construct a version from a nearby complete version. They also present part of a query language to retrieve desired versions. The focus of the paper is on efficient storage and retrieval of versions, whereas our focus is on fine-grained control of versioning. Wang and Zaniolo [?] present a comprehensive system for concisely representing a

- *F. Currim is with the Department of Management Sciences, University of Iowa, Iowa City, IA, 52242.*
  E-mail: faiz-currim@uiowa.edu
- *C. Dyreson is with the Department of Computer Science, the Utah State University.*
- *R. Snodgrass and S. Thomas are with the Department of Computer Science, University of Arizona.*

temporally-grouped XML version history. They also give a query language to retrieve past versions. Their extensions, like ours, require no changes to current standards to support versioning. Unlike τXSchema, everything is versioned and there is no support for temporal constraints in the versioning. Temporal and physical schema annotations are not discussed.

In context of time-varying documents, Garcia-Molina and Cho [**?**] provide evidence that some web resources change frequently (though not specifically XML resources). Nguyen et al. [**?**] describe how to detect changes in XML documents that are accessible via the web. In the Xyleme system [**?**], the XML Alerter module periodically (with a periodicity specified by the user) accesses the XML document and compares it with a cached version of the document. The result is a sequence of static documents, each with an associated existence period. Dyreson [**?**] describes how a web server can capture some of the versions of a time-varying document, by caching the document as it is served to a client, and comparing the cached version against subsequent requests to see if anything has changed. Amagasa et al. [**?**] classify the methods used to access XML documents into two general categories: (i) using specialized APIs for XML documents, such as DOM, and (ii) directly editing documents, e.g., with an editor. In the former case, to access and modify temporal XML documents, DOM can be extended to automatically capture temporal information (and indeed, we have implemented such functionality in τ DOM). Franceschet et al. [**?**] have also adopted this approach, but their approach requires the user to specify a valid ER schema and it only supports limited temporal data validation. It is also possible to capture transaction time information in the documents through change analysis, as discussed above and elsewhere [**?**], [**?**], [**?**]. Inconsistencies arise when the documents can be edited directly and methods need to designed to resolve them [**?**], [**?**]. Most previous approaches, irrespective of their methods to access XML documents, assume that timestamps are present on every time-varying element [**?**], [?], [**?**] (whereas our approach enables the schema designer to specify the physical location of the timestamps).

There has been a lot of interest in the representation schemes for time-varying documents. Some version control tools have been developed for data varying XML documents (e.g., [?], [**?**]). Chien, Tsotras and Zaniolo [?] have researched techniques for compactly storing multiple versions of an evolving XML document. Chawathe et al. [**?**] described a model for representing changes in semi-structured data and a language for querying over these changes. A related option, the diff based approach [?], [?] focuses on an efficient way to store time-varying data and can be used to help detect transaction time changes in the document at the physical level. Buneman et al. [**?**], [**?**] provide another means to store a single copy of an element that occurs in many snapshots. Grandi and Mandreoli [**?**] propose a `<valid>` tag to define a validity context that is used to timestamp part of a document. Mandreoli et al. [?] utilize native support, in which an XML document is encoded using inverted lists of tuples with additional position and level numbers. Assuming a data document were stored in this representation, their slicing implementation could be used to implement unsquash efficiently. Finally, Chawathe et al. [?], Dyreson et al. [**?**], Mendelzon et al. [**?**] and Tang et al. [**?**] discuss timestamps on edges (instead of document nodes) in a semi-structured data model.

Recently there has been interest in incremental validation of XML documents [**?**], [**?**], [**?**], [**?**]. These consider validating a snapshot that is the result of updates on the previous snapshot, which has already been validated. In a sense, this is the dual to the problem we consider, which is validating a (compressed) temporal document all at once, rather than once per snapshot (incrementally or otherwise).

None of the approaches above focus on the extensions required in XML Schema to adequately specify the nature of changes permissible in an XML document over time, and the corresponding validation of the extended schema. In fact, some of the previous approaches that attempt to identify or characterize changes in documents do not consider a schema. As our emphasis is on logical and physical data modeling, we assume that a schema is available from the start, and that the desire is for that schema to capture both the static and time-varying aspects of the document. If no schema exists, tools can derive the schema from the base documents [**?**], but the details of that is beyond the scope of this paper. Our research applies at the logical view of the data, while also being able to specify the physical representation. Since our approach is independent of the physical representation of the data, it is possible to incorporate the diff-based approach and other representational approaches [?] in our physical annotations.

from TR: part 2 related work

In this section, we review prior related work in the area of schema versioning. Version and source control for schemas and schema objects is needed, especially in complex, multi-enterprise development environments. The XML Schema working group at W3C has discussed desirable behaviors for use cases that involve schema versioning in XML [**?**]. Various techniques to support evolution of XML schemas, where they allow for extensibility in the original design have also been proposed [**?**]. The emphasis of the paper is to avoid changes to the existing applications by anticipating changes to the schemas and then designing them for evolution. This is typically achieved through a careful use of wildcards, allowing extensions through namespaces, allowing applications to ignore unknown objects, and forcing applications to understand unknown objects when no other option is available. This approach does not address the whole problem, as many schema changes cannot be expressed in their limited notations.

Schema versioning has been previously researched in the context of temporal databases [**?**]. But an XML schema is a grammar specification, unlike a (relational) database schema, so new techniques are required. Although various XML schema languages have been proposed in the literature and in the commercial arena, none model schema changes nor provide 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.

Brahmia et al. propose a six-component taxonomy of schema change operations for use in supporting schema versioning across both valid and transaction time with

XMLSchema [**?**].

Raghavachari and Shmueli consider a problem different from that considered in the present paper: can a nontemporal XML document D that is known to be valid according to nontemporal XML schema S be efficiently validated against a different schema S' [**?**]. However, their problem and proposed solutions are relevant to the validation for a temporal document against a temporal schema as considered in the present paper. As the schema evolves over time, the data is required to also evolve so that he data timestamped with a transaction time at the new time is consistent with the schema timestamped with that transaction time. It is possible for the tool constructing that temporal document, or for the SQUASH tool as it considers a schema change, to efficiently revalidate the data document currently in force against the new schema.

3 Language Design

[TODO: Curtis]

(TR: some material from Sections 5 and 9)

- Background of previous work (sufficient tauXSchema for them to understand subsequent discussions)
- Desiderata for temporal XML Schema and constraints including: allowing sequenced and non-sequenced constraints; valid-time, transaction-time, bitemporal support; also support for data and schema versioning
- Data independence, why a logical annotation document is needed and why temporal constraint annotations belong in annotations

from TR: Part1Design.tex

This section provides the overarching design decisions related to time-varying data within the τXSchema system, and the desiderata and design goals that motivated those decisions.

We start out with some terminology that will be used throughout this document, including conventional and temporal (XML) documents, and conventional and temporal (XML) schemas (which are also XML documents themselves). Also defined is the annotation document and slice (which are also XML documents themselves). We then present some high level design desiderata and goals that motivate the specific decisions listed in Section 3.2. This includes decisions relevant to the temporal schema, annotations, and the temporal document. We conclude by presenting a brief example to illustrates the usage of the τXSchema language.

from TR Part1Design Documents.tex

This section defines terms relevant to τXSchema.

3.1 Terminology

This section defines terms relevant to τXSchema.

- *Conventional Document*: A standard XML document that has no temporal aspects.
- *Temporal Document*: A standard XML document that represents a sequence of conventional documents (i.e., slices). It may be user-created or the result of the SQUASH tool and has the root element `<temporalRoot>`.
- *Conventional Schema*: A standard XML Schema document that describes the structure of the conventional document(s). The root element is `<schema>`.
- *Temporal Schema*: A standard XML document that ties together the conventional schemas and the annotations. In our temporal system, the temporal schema is the logical equivalent to the XML Schema of the conventional world; it describes the rules and format of the temporal documents. The root element is `<temporalSchema>`.
- *Annotation Document*: A standard XML document that specifies a variety of characteristics (e.g., logical, physical, etc.) of a conventional document. For example, *logical* characteristics specify 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; *physical* characteristics specify the timestamp options for the representation, such as where the timestamps are placed and their kind (e.g., valid time or transaction time) and the kind of representation.
- *Slice*: A version of a temporal document at a given point in time. For example, if a temporal document is comprised of two conventional documents d_1 and d_2, which occur at time t_1 and t_2, respectively, then the slice at time t_2 is d_2.

from TR: Part1Design Documents.tex

3.2 Design Decisions

This section outlines the design decisions that resulted from the design goals. We first describe general decisions that apply to all of τXSchema, and then discuss the decisions that apply specifically to temporal schemas, temporal documents, and annotations documents.

3.3 Company Example

This section walks through in detail an example that illustrates the usage of τXSchema. Explanations of a user's actions are given in sequence and the corresponding XML text is provided via a *listing*. In effort to make the example as clear as possible, a few conventions are followed. Note that each convention is used only for clarity and is not a requirement in τXSchema.

- Only transaction time is considered.
- The example does not use default namespaces for τXSchema files (e.g., temporal schemas) in order to emphasize which namespace is being used. However, conventional documents make use of default namespaces for brevity.
- As file contents are changed over time, a version number embedded in the name will also change so that the reader can more easily keep track of the changes. The version number for each file begins at 0 and is constructed as follows.
  - `Company.`S`.xsd` for conventional schemas, where $S = \{$`A, B, C, ...`$\}$ indicates the version of the schema, e.g., `Company.A.xsd`.
  - `data.`S`.`D`.xml` for conventional documents, where S indicates the version of the schema being used and D indicates the version number of the conventional document, e.g., `data.A.0.xml`.

- `temporalDocument.`S`.`D`.xml` for temporal documents, where S indicates the version of the temporal schema being used and D indicates the version number of the latest conventional document, e.g., `temporalDocument.0.3.xml`.
- `temporalSchema.`D`.xml` for temporal schemas, where D indicates the version number of the temporal schema, e.g., `temporalSchema.0.xml`.
- `annotations.`A`.xml` for annotation documents, where A indicates the version of the temporal annotation document, e.g., `annotations.0.xml`.
- `Person.`S`.`E`.xsd` for the Person subschemas, where S indicates the first version of the conventional schema that references this subschema and E indicates the version number of the subschema itself, e.g., `Person.A.0.xml`.
- `Product.`S`.`F`.xsd` for the Product subschemas, where S indicates the first version of the conventional schema that references this subschema and F indicates the version number of the subschema itself, e.g., `Product.A.0.xml`.

In this example, each time the user modifies the conventional schema, a new file is created. He must then modify the conventional document to reference this new, modified schema. In practice, this is awkward and would rarely happen. In a more realistic situation, the user would reuse the same filename by just modifying the file in place, and editors would be responsible for automatically retaining previous versions. Also, in practice the conventional document would change much more frequently than the conventional schema. Figure 1 depicts the overall scenario.

3.3.1 Initial Configuration

Consider the following scenario which begins on 2008-01-01. The user has a conventional schema which defines a `<Person>` element, which itself has a `<Name>` element, an `<SSN>` element, and an `ID` attribute.

```
<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.company.org"
  xmlns="http://www.company.org"
  elementFormDefault="qualified">

  <xsd:element name="Company">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Person"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="Person">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string"/>
        <xsd:element name="SSN" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="ID" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

Listing 1. `Company.A.xsd`

He also has a conventional document conforming to the schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<Company xmlns="http://www.company.org">

  <Person ID="1">
    <Name>Steve</Name>
    <SSN>111-22-3333</SSN>
  </Person>

</Company>
```

Listing 2. `data.A.0.xml`

Together, these documents form a conventional system which can be validated with conventional validation tools (e.g., XMLLINT). Of course, τXMLLINT will also validate this conventional system. In the following sections, we will add new versions of the conventional document, add new versions of the conventional schema, break up the conventional schema into multiple subschemas, and specify temporal annotations. Figure 1 shows the relationship between all the documents in the system. Note that Company schema (for details on this schema and example documents, please see Section **??**) will import and include two subschemas: `Person` and `Product`. In this example, both the Person and Product schemas will change over time. Each time there is a new slice created, the Company schema must be updated to reference the new slice. There are other mechanisms available to the user for handling this scenario, as described in the document beginning at section **??**

3.3.2 Adding Temporal Data

On 2008-03-17, the user corrects the `<SSN>` element in the conventional document to produce a new version. The user can now use τXSchema to create temporal documents and use the τXSchema tools to validate these documents.

```
<?xml version="1.0" encoding="UTF-8"?>
<Company xmlns="http://www.company.org">

  <Person ID="1">
    <Name>Steve</Name>
    <SSN>123-45-6789</SSN>
  </Person>

</Company>
```

Listing 3. `data.A.1.xml`

The user creates a temporal document that lists both slices of the conventional document with their associated timestamps.

```
<?xml version="1.0" encoding="UTF-8"?>
<td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD">
  <td:temporalSchemaSet>
    <td:temporalSchema location="./Company.A.xsd"/>
  </td:temporalSchemaSet>

  <td:sliceSequence>
    <td:slice location="data.A.0.xml" begin="2008-01-01" />
    <td:slice location="data.A.1.xml" begin="2008-03-17" />
  </td:sliceSequence>

</td:temporalRoot>
```

Listing 4. `temporalDocument.0.1.xml`

The user uses the conventional schema as the temporal schema. That is, the user does not explicitly create a temporal schema. Note that since no logical or physical annotations have been specified, the defaults will take effect.

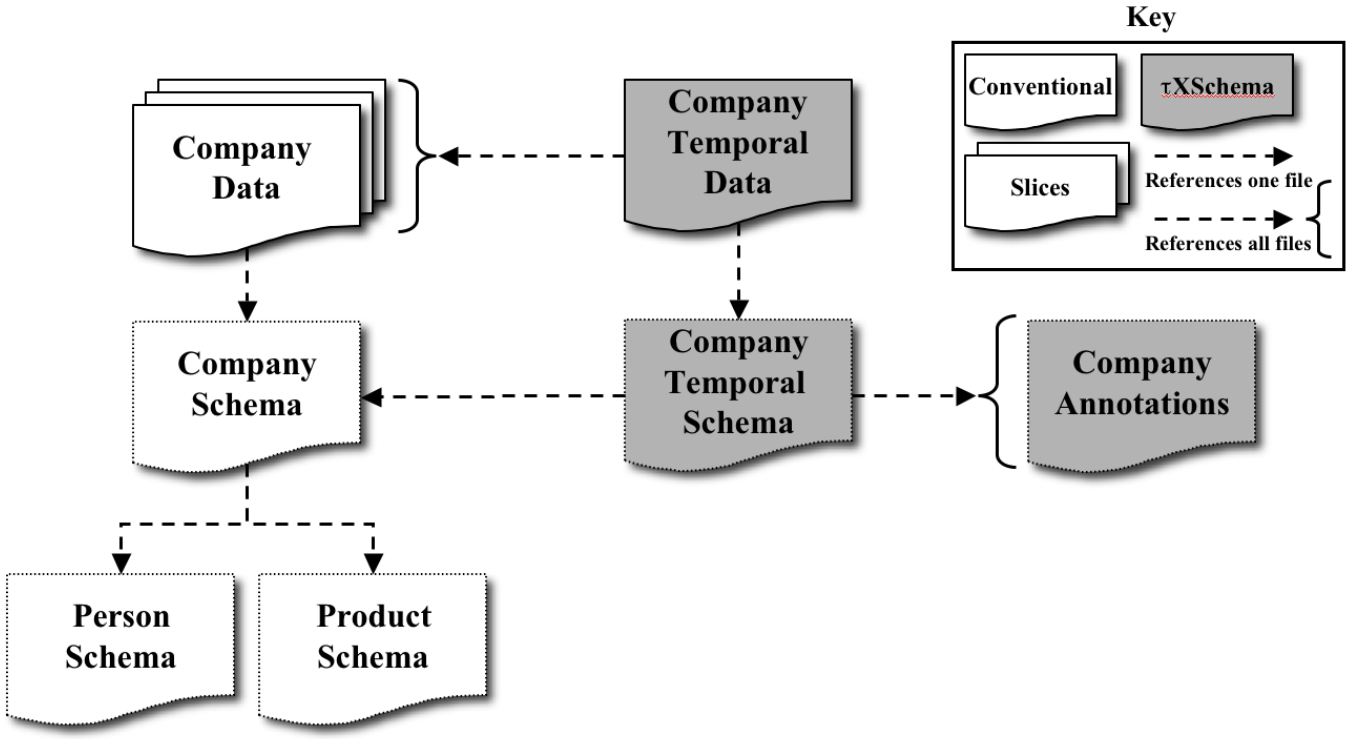

Fig. 1. An overview of the end-state of the Company example.

from Part1Architecture.tex

In this section we describe the overall architecture of τXSchema and illustrate with an example. The design and implementation details of the tools are explained further in Section **??**.

A visual depiction of the architecture of τXSchema is illustrated in Figure 2. This figure is central to our approach, so we describe it in detail and illustrate it with examples. We note that although the architecture has many components, only those components shaded in the figure are specific to an individual time-varying document and need to be supplied by a user. New time-varying schemas can be quickly and easily developed and deployed. We also note that the representational schema, instead of being the only schema in an ad hoc approach, is merely an artifact in our approach, with the conventional schema, logical annotations, and physical annotations being the crucial specifications to be created by the designer.

The designer annotates the conventional schema with logical annotations (box 5). The logical annotations together with the conventional schema form the logical schema. Listing 5 provides an extract of the logical annotations on the WinOlympic schema. The logical 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. For example, `<athlete>` is described as a state element, indicating that the `<athlete>` will be valid over a period (continuous) of time rather than a single instant. Annotations can be nested, enabling the target to be relative to that of its parent, and inheriting as defaults the kind, `contentVarying`, and `existenceVarying` attribute values specified in the parent. The attribute `existenceVarying` indicates whether the element can be absent at some times and present at others. As an example, the presence of `existenceVarying` for an athlete's phone indicates that an athlete may have a phone at some points in time and not at other points in time. The attribute `contentVarying` indicates whether the element's content can change over time. An element's content is a string representation of its *immediate content*, i.e., text, sub-element names, and sub-element order.

As discussed in Section **??**, if *no annotations* are provided whatsoever, the default annotation is that *anything can change*. However, once we begin to annotate the conventional schema, the semantics we adopt are that elements that are not described as time-varying are static. Thus, they must have the same content and existence across every XML document in box 7. For example, we have assumed that the birthplace of an athlete will not change with time, so there is no annotation for `<birthPlace>` among the logical annotations. The schema for the logical annotations document is given by ASchema (box 2).

```
<?xml version="1.0" encoding="UTF-8"?>
<logical
  xmlns="http://www.cs.arizona.edu/tau/tauXSchema/ASchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.cs.arizona.edu/tau/
                        tauXSchema/ASchemaASchema.xsd">
 <default>
    <format plugin="XMLSchema" granularity="gDay"/>
  </default>
```

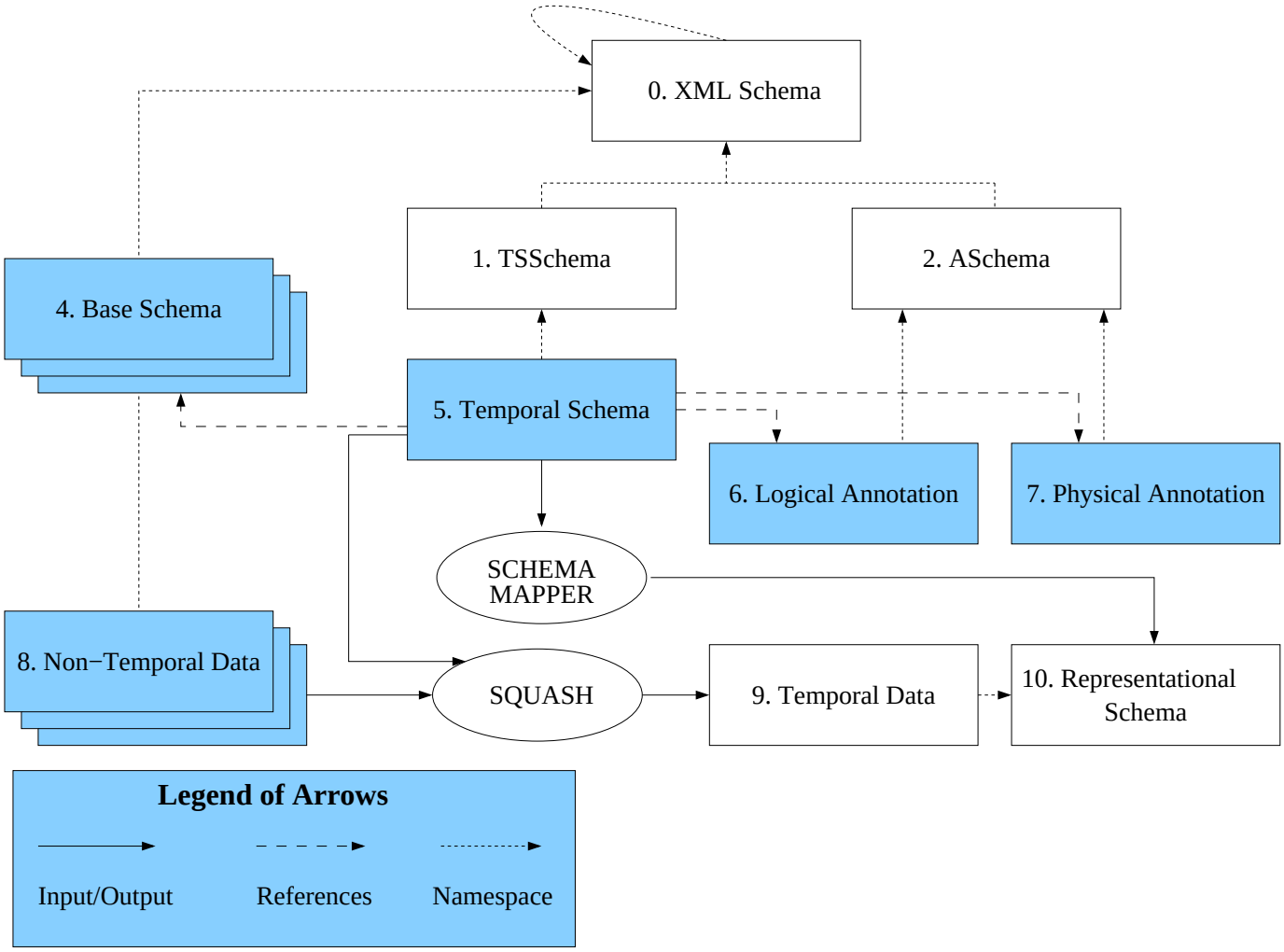

Fig. 2. Overall Architecture of τXSchema

```
...
  <item base="winOlympic/country/athleteTeam">
   <validTime content="constant" existence="varyingWithGaps">
      <maximalExistence begin="1924-01-01" />
   </validTime>
   <itemIdentifier name="teamName"
         timeDimension="transactionTime">
      <field path="./teamName"/>
   </itemIdentifier>
  </item>
...
 <item target="winOlympic/country/athleteTeam/athlete/medal">
   <validTime/>
   <transactionTime/>
   <itemIdentifier name="medalId1"
         timeDimension="bitemporal">
      <field path="./text()"/>
      <field path="../athName"/>
   </itemIdentifier>
  </item>
...
</logical>
```

Listing 5. Sample WinOlympic Temporal Annotation

The next design step is to create the physical annotations (box 6). In general, the physical annotations specify the time-stamp representation options chosen by the user. An excerpt of the physical annotations for the WinOlympic schema is given in Listing 6. Physical annotations may also be nested, inheriting the specified attributes from their parent; these values can be overridden in the child element.

```
<?xml version="1.0" encoding="UTF-8"?>
<physical xmlns="http://www.cs.arizona.edu/tau/tauXSchema/
                                                  ASchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.cs.arizona.edu/tau/
                                      tauXSchema/ASchema
      ASchema.xsd">
  <default>
    <format plugin="XMLSchema" granularity="days"/>
  </default>
...
  <stamp target="winOlympic/country">
    <stampKind timeDimension="transactionTime"
               stampBounds="extent"/>
  </stamp>
...
  <stamp target="winOlympic/country/athleteTeam/athlete">
    <stampKind timeDimension="transactionTime"
               stampBounds="step"/>
  </stamp>
...
</physical>
```

Listing 6. Sample WinOlympic Physical Annotation

Physical annotations play two important roles.

- They help to define where 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 logical annotations). Two documents with the same logical information will look very different if we change the location of the physical timestamp. For example, although the elements `phone` and `athName` are time-varying, the user may choose to place the physical timestamp at the `athlete` level. Whenever any element below athlete changes, the entire `athlete` element is repeated.

- The physical annotations also define the type of timestamp (for both valid time and transaction time). A timestamp can be one of two types: `step` or `extent`. An extent timestamp specifies both the start and end instants in the timestamp's period. In contrast a step-wise constant (step) timestamp represents only the start instant. The end instant is implicitly assumed to be just prior to the start of the next version, or *now* for the current version. However, one cannot use `step` timestamps when there might be "gaps" in time between successive versions. `extent` timestamps do not have this limitation. Changing even one timestamp from `step` to `extent` can make a big difference in the representation.

The schema for the physical annotations is also contained within ASchema (box 2). τXSchema supplies a default set of physical annotations, which is to timestamp the root element with valid and transaction time using step timestamps, so the physical annotations are optional. However, adding them can lead to more compact representations.

We emphasize that our focus is on capturing relevant aspects of physical representations, not on the specific representations themselves (the design of which is challenging in itself). Also, since the logical 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, independently 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 logical 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 temporal schema (box 4) ties the schema, logical annotations and physical annotations together. This document contains sub-elements that associate a series of conventional schema with logical and physical annotations, along with the time span during which the association was in effect. The schema for the temporal schema document is TSSchema (box 1).

At this point, the designer is finished. She has written one conventional XML schema (box 3), specified two sets of annotations (boxes 5 and 6), and provided the linking information via the temporal schema document (box 4). We provide boxes 1 and 2; XML Schema (box 0) is of course provided by W3C. Thus new time-varying schemas can be quickly and easily developed and deployed.

Let's now turn our attention to the tools that operate on these various specifications. The temporal schema document (box 4)

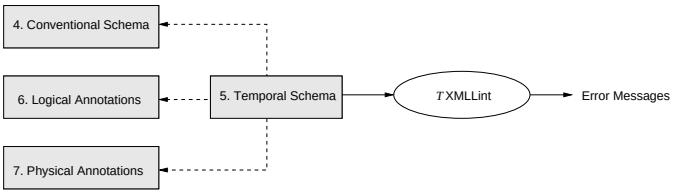

Fig. 3. τValidator: Checking the schemas

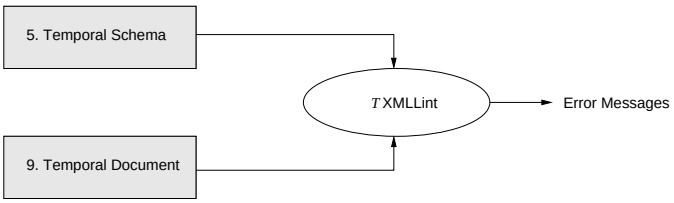

Fig. 4. τValidator: Checking the instance

is passed through the τValidator (see Figure 3) which checks to ensure that the temporal and physical annotations are consistent with the conventional schema. The τValidator utilizes the conventional validator (e.g., XMLLINT) for many of its checks. For instance, it validates the logical annotations against the ASchema. But it also checks that the logical annotations are not inconsistent. Similarly, the physical annotations document is passed through the τValidator to ensure consistency of the physical annotations. The temporal constraint checker then evaluates the temporal constraints expressed in the schema (see Section **??** for more details). Finally, the temporal validator reports whether the temporal document was valid or invalid.

Once the annotations are found to be consistent, the *Schema Mapper* (software oval, Figure 2) generates the *representational schema* (box 9) from the original conventional schema and the logical and physical annotations. The representational schema is needed to serve as the schema for a time-varying document/data (box 8). The time-varying data can be created in four ways:

1) automatically from the non-temporal data (box 7) using τXSchema's squash tool (described in Section **??**),
2) automatically from the data stored in a database, i.e., as the result of a "temporal" query or view,
3) automatically from a third-party tool, or
4) manually.

The time-varying data is validated against the representational schema in two stages. First, a conventional XML Schema validating parser is used to parse and validate the time-varying data since the representational schema is an XML Schema document that satisfies the snapshot validation subsumption property. But as emphasized in Section **??**, using a conventional XML Schema validating parser 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 valid time boundaries of a parent element must encompass those of its child". These types of checks are implemented in the τValidator. So the second step is to pass the temporal data to τValidator as shown in Figure 4. A temporal XML

data file (box 8) is essentially a timestamped representation of a sequence of non-temporal XML data files (box 7). The namespace is set to its associated XML Schema document (i.e., representational schema). The timestamps are based on the characteristics defined in the logical and physical annotations (boxes 5 and 6). The τValidator, by checking the temporal data, effectively checks the non-temporal constraints specified by the conventional schema simultaneously on all the instances of the non-temporal data (box 7), as well as the constraints between snapshots, which cannot be expressed in a conventional schema.

To reiterate, the conventional approach to storing timestamped data would require the user start with a representational schema (box 9) and use it to validate the temporal data (box 8). Both these documents become very complex if time varying data and schema are to be handled, and are non-intuitive to work with directly. Our proposed approach is to have the user design a conventional schema, add logical and physical annotations (boxes 5 and 6), leading to the representational schema (and temporal data) being automatically generated. In the second part of this technical report (Section **??** onwards), we discuss the user specification of the temporal schema (box 4), which is only needed if the conventional schema (box 3) and annotation documents (boxes 5 and 6) themselves can vary.

4 Theoretical Framework

[TODO: Curtis]

(TR: Section 6)

- Sufficient details to allow the reader to have a background for constraints and understand the *eval* function

from TR

This section sketches the process of constructing a schema for a time-varying document from a conventional schema. The goal of the construction process is to create a schema that satisfies the snapshot validation subsumption property, which is described in detail below. In the relational data model, a schema defines the structure of each relation in a database. Each relation has a very simple structure: a relation is a list of attributes, with each attribute having a specified data type. The schema also includes integrity constraints, such as the specification of primary and foreign keys. In a similar manner, an XML Schema document defines the valid structure for an XML document. But an XML document has a far more complex structure than a relation. A document is a nested collection of elements, with each element potentially having (text) content and attributes.

4.1 Snapshot Validation Subsumption

Let D^T be an XML document that contains timestamped elements. A timestamped element is an element that has an associated timestamp. (A timestamped attribute can be modeled as a special case of a timestamped element.) Logically, the timestamp is a collection of times (usually periods) chosen from one or more temporal dimensions (e.g., valid time, transaction time). Without loss of generality, we will restrict the discussion in this section to lifetimes that consist of a single period in one temporal dimension. The timestamp records (part of) the lifetime of an element. We will use the notation x^T to signify that element x has been timestamped. Let the lifetime of x^T be denoted as $\mathit{lifetime}(x^T)$. One constraint on the lifetime is that the lifetime of an element must be contained in the lifetime of each element that encloses it.

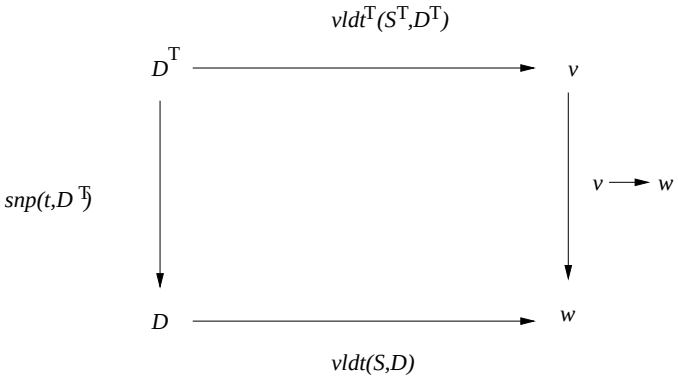

Fig. 5. Snapshot Validation Subsumption

The snapshot operation extracts a complete snapshot of a time-varying document at a particular instant. Timestamps are not represented in the snapshot. A snapshot at time t replaces each timestamped element x^T with its non-timestamped copy x if t is in $\mathit{lifetime}(x^T)$ or with the empty string, otherwise. The snapshot operation is denoted as

$$\mathit{snp}(t, D^T) = D$$

where D is the snapshot at time t of the time-varying document, D^T.

Let S^T be a representational schema for a time-varying document D^T. The snapshot validation subsumption property captures the idea that, at the very least, the representational schema must ensure that every snapshot of the document is valid with respect to the conventional schema. Let $\mathit{vldt}(S, D)$ represents the validation status of document D with respect to schema S. The status is *true* if the document is valid but *false* otherwise. Validation also applies to time-varying documents, e.g., $\mathit{vldt}^T(S^T, D^T)$ is the validation status of D^T with respect to a representational schema, S^T, using a temporal validator.

Property [Snapshot Validation Subsumption] Let S be an XML Schema document, D^T be a time-varying XML document, and S^T be a representational schema, also an XML Schema document. S^T is said to have snapshot validation subsumption with respect to S if

$$\mathit{vldt}^T(S^T, D^T) \Leftrightarrow \forall t[\exists \mathit{lifetime}(D^T) \Rightarrow \mathit{vldt}(S, \mathit{snp}(t, D^T)]$$

Intuitively, the property asserts that a good representational schema will validate only those time-varying documents for which every snapshot conforms to the conventional schema. The subsumption property is depicted in Figure 5.

4.2 Content and Existence Variance

The data stored in XML documents may change over time. It is useful to be able to validate the way data can change.

The XML Schema standard provides a way to validate XML documents, but does not define how an XML document is allowed to change with time. To meet this need, τXSchema was created as an extension of the XML standard that validates time-varying XML documents.

The two ways that a node in an XML document can vary with time are (1) in its content or (2) in its existence. The content of an item includes the entire sub-tree rooted at a node. Each branch in the sub-tree terminates at the first item on the branch, or at a leaf (text value, attribute, empty element). Some nodes, especially those containing loose text, will change their content. Some nodes will exist in one version of an XML instance document but will not be present in another version. Other nodes will have both their content and existence change over time.

An item definition specifies how a data node may vary in its content and its existence. Let's first consider how an item specifies existence. There are three possible alternatives. The first is "varying with gaps", which means that each of its corresponding data nodes may be present in some versions of the XML instance document and absent in others. A second, more restrictive form is "varying without gaps." The data node is not required to always be present. When it is present there may not be any gaps in its existence. The third value is "constant". Then the corresponding data node is either always present or never present. Again the existence-constant can have many different semantics. We have identified three of them and provide support for the first two in our implementation.

- Existence is constant over all time (exists in every instant in lifetime of universe).
- Existence is constant over document lifetime (document lifetime may have gaps).
- Existence is constant over the lifetime of the immediate ancestor's item.

The other aspect an item may specify is content. The content of a data node depends on its node type. The content may change in the data node at any time if the corresponding item specifies content as varying. There are restrictions on how a data node's content may change over time when the corresponding item specifies content as constant. The restrictions are different for each of the type of content (e.g., elements, attributes and loose text). The detailed explanation of the restrictions can be found in Section **??**.

Content-varying and existence-varying are orthogonal concepts. The only restriction is that, when an item is content-constant, the item's immediate descendants should be existence-content, but switching of parents is allowed. When an item specifies content or existence as varying, the corresponding data node may vary with time, but is not required to.

4.3 Items

In order to create a temporal document it is important to identify which elements persist across various transformations of the document. This section discusses how to find and associate elements in different snapshots of a temporal XML document. When elements are temporally-associated, an *item* is created. An item is a collection of XML elements that represent the same real-world entity. An item is a logical entity that evolves over time through various versions.

In a temporal database, a pair of value-equivalent tuples can be coalesced, or replaced by a single tuple that has a lifespan equivalent to the union of the pair's lifespans. *Coalescing* is an important process in reducing the size of a data collection (since the two tuples can be replaced by a single tuple) and in computing the maximal temporal extent of value-equivalent tuples. In a similar manner, elements in two snapshots of a temporal XML document can be *temporally-associated*. A temporal association between the elements is possible when the element has the same *item identifier* in both snapshots. We will sometimes refer to the process of associating a pair of elements as *gluing* the elements. When two or more elements is glued, an item is created.

Only temporal elements (that is, elements of types that have a temporal annotation) are candidates for gluing. Determining which pairs should be glued depends on two factors: the type of the element, and the item identifier for the element's type. The type of an element is the element's definition in the schema. Only elements of the same type can be glued. An item identifier serves to semantically identify elements of a particular type. The identifier is a list of XPath expressions (much like a key in XML Schema) so we first define what it means to evaluate an XPath expression.

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

Since an XPath expression evaluates to a list of nodes, $Eval(n, L)$ evaluates to a list of lists.

Definition [Item identifier] An item identifier for a type, T, is a list of XPath expressions, L, such that the evaluation of L partitions the set of type T elements in a (temporal) document. Each partition is an item.

An item identifier has a target and at least one field, an itemref or a keyref. A target is an XPath expression that specifies an element's location in the snapshots (relative to the item under which it is defined). A field, itemref and a keyref can each specify part of an item identifier. A field contains an XPath expression that specifies an element or attribute that is part of the item identifier. A keyref references a snapshot key and an itemref references an item identifier. This way an item may be specified in terms of an existing item or schema key. An itemref and keyref use the name of an item/key and are not XPath expressions. The item identifier may consist of any combination of field(s), itemref(s) and keyref(s). Each field expression specifies either an attribute or an element. If an attribute is indicated, then the item identifier uses the attribute's value. If an element is indicated, then the item identifier uses the element's loose text. The current implementation supports only fields.

A schema designer specifies the item identifiers for the temporal elements. As an example, a designer might specify the following item identifiers for the temporal elements `<athlete>` and `<medal>`.

- `<athlete>` ⇒ `[athName/*]`

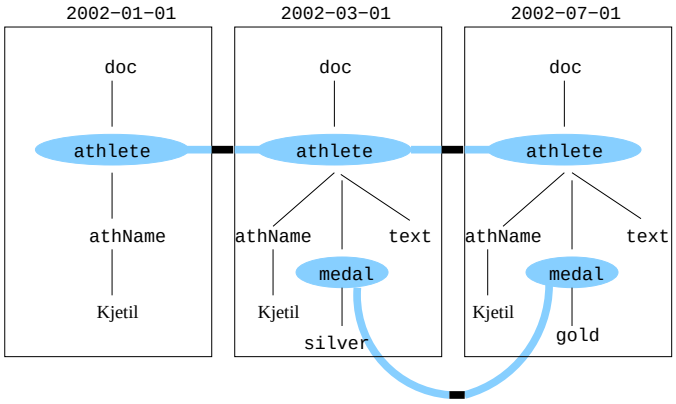

Fig. 6. Items and Versions

- `<medal>` ⇒ `[../athName/*, ./* ]`

The item identifier for an `<athlete>` is the name of the athlete, while the item identifier for `<medal>` is the athlete's name (the parent's item identifier) combined with the description of the event (the text within the medal element). An item identifier is similar to a (temporal) key in that it is used for identification. Unlike a key however, an item identifier is not a constraint; rather it is a helpful tool in the complex process of computing versions.

Over time, many elements in a temporal document may belong to the same item as the item evolves. The association of these elements in an item is defined below.

Definition [Temporal association] Let x be an element of type T in the i^{th} snapshot of a temporal document. Let y be an element of type T in the j^{th} snapshot of the document. Finally let L be the item identifier for elements of type T. Then x is *temporally-associated* to y if and only if $Eval(x, L) = Eval(y, L)$ and it is not the case that there exists an element z of type T in a snapshot between the i^{th} and j^{th} snapshots such that $Eval(z, L) = Eval(x, L)$.

A temporal association relates elements that are adjacent in time and that belong to the same item. For instance, the `<athlete>` element in Listing **??** on page **??** is temporally associated with the `<athlete>` element in Listing **??** but not the `<athlete>` element in Listing **??** (though the `<athlete>` element in Listing **??** is temporally related to the one in Listing **??**).

4.4 Versions

When an element in a new snapshot is temporally-associated with an item, the association either creates a new version of the item or extends the lifetime of the latest version within the item. A version is extended when "no difference" is detected in the associated element. Differences are observed within the context of the Document Object Model (DOM).

Definition [DOM equivalence] A pair of elements is DOM equivalent if the pair meets the following conditions.

- Their parents are the same item or their parents are non-temporal elements.
- They have the same number of children.
- For each child that is a temporal element, the child is the same item as the corresponding child of the other (in a lexical ordering of the children).
- For each child that is something other than a temporal element the child's children are each DOM-equivalent to the corresponding children of the other child (in a lexical ordering of the children and grandchildren), and the child's *value*, *type* (e.g., element or text), and *name* (e.g., tag name) are also the same.
- They have the same set of attributes (an attribute is a name, value pair).

The third bullet in the above definition applies to non-temporal children of a node. The idea is that the "value" of a non-temporal child is the entire subtree rooted at the child. The subtree terminates at either (non-temporal) leaves or (temporal) items.

As an aside, we observe that DOM equivalence in a temporal XML context is akin to value equivalence in a temporal relational database context [**?**]. DOM equivalence is used to determine versions of an item, as follows.

Definition [Version] Let x be an item of type T in a temporal document, with a lifetime that ends at time t. Let y be an element of type T in a snapshot at time $t + k$ that is temporally associated to the latest version of x, v_t. If v_t is DOM equivalent to y then the lifetime of v_t is extended to include $t + k$. Otherwise, version v_{t+1}, consisting of y, is added to item x.

A version's lifetime is extended when the element from the next snapshot (or a future snapshot) is DOM equivalent (the lifetime can have gaps or holes, although having a gap may violate a schema constraint as described in section 4.2). A new version is created when a temporal association is not DOM equivalent.

Figure 6 depicts the items and versions in the example. An abstract representation of the DOM for each snapshot of the document is shown. The items in the sequence of snapshots are connected within each shaded region. There is one athlete item and one medal item. The athlete item has two versions; the transition between versions is shown as a black stripe between the regions.

5 XML Schema Constraints

[TODO: Sabah]

(TR: Section 7.1)

Overview of XML Schema constraints (conventional)

From TR

In this section we discuss XML Schema constraints and their temporal extensions. XML Schema provides four types of constraints.

- Identity constraints
- Referential Integrity constraints
- Cardinality constraints (in the form of `minOccurs` and `maxOccurs` for sub-elements and `required` and `optional` for attributes)
- Datatype restrictions (which constrain the content of the corresponding element or attribute)

XML Schema constraints are conventional constraints since they restrict a specific conventional document. We briefly explain each of these XML Schema constraints in turn, and then proceed to their temporal extensions using the company example.

The root of this schema is the company entity. Under that, there are products and suppliers. An order is considered a sub-element of suppliers (with a reference from order to product number for data integrity).

5.1 XML Schema Constraints

5.1.1 Identity Constraints

Identity constraints restrict uniqueness of elements and attributes in a given document. As with the relational model, XML Schema allows users to define both key and unique constraints. The distinction between these two constraint types is that the evaluation of the key constraint should not yield any NULL values in any of the component fields, while the fields in a unique constraint are allowed to evaluate to NULL.

Identity constraints are defined in the schema document using a combination of a selector and one or more field elements. These are sub-elements within a <xs:key> or <xs:unique> container element. Both selector and field contain an XPath expression (the evaluation of which in an XML document yields the value of the constrained element or attribute). The selector is used to define a contextual node in the XML document (e.g., product), relative to which the (combination of) field values is unique (e.g., @productNo).

An identity constraint may be named, and this name can then be used when defining a referential integrity constraint (similar to foreign keys in the relational model). A sample XML Schema identity constraint is in Listing 7.

```
...
<xs:key name="productKey">
   <xs:selector xpath="product"/>
   <xs:field xpath="@productNo"/>
</xs:key>
...
```

Listing 7. Sample Identity Constraint Definition

Formally, we can define a key constraint as follows. Let sel be a context node (defined by selector in an identity constraint), with the list of corresponding field expressions $F = (f_1, \ldots, f_m)$. In an instance document, there may be n such selector nodes across the scope of the document. Using the example in 7, we may have n product nodes. Let $Eval(sel, F)$ denote the resultset of evaluating the XPath expression-list F from the context node sel. Evaluated on a specific instance document, the resultset will contain n elements, corresponding to each product node (for example). For convenience, let us denote the i^{th} element of $Eval(sel, F)$ by e_i. Then, $\forall i, j : e_i \neq e_j$ unless $i = j$; $i, j \leq n$. In the case of a unique constraint, the only differences are: $e_i \in \{Eval(sel, F) \cup \omega\}$ (where ω represents the *null* value), and $\forall i, j; e_i$ is not ω then $e_i \neq e_j$, unless $i = j$.

There are some similarities between the functionality of key and unique constraints and the XML 1.0 ID definitions (and the equivalent ID simple type in XML Schema). However, the XML Schema key and unique constraints have a number of advantages. We take advantage of this distinction when we discuss temporal extensions to the constraints (in Section **??**).

Advantages of XML Schema key over the XML ID are as follows. XML 1.0 provides a mechanism for ensuring uniqueness using the ID attribute (and referential integrity using the associated IDREF and IDREFS attributes). An equivalent mechanism is provided in XML Schema through the ID, IDREF, and IDREFS simple types, which can be used for declaring XML 1.0-style attributes. XML Schema also introduces two other mechanisms to ensure uniqueness using the key and keyref constraints that are more flexible and powerful in the following ways.

- XML Schema keys can be applied to both elements and attributes. Since ID is an attribute (in DTDs; in XML Schema an element's type can be defined as xs:ID), it cannot be applied to other attributes.
- Using key and keyref allows the specification of the scope within which uniqueness applies (done by the selector element; i.e., it is "contextual uniqueness") while the scope of an XML ID is the whole document. Thus using a key constraint one can enforce: "within each order, the part numbers should be unique", to ensure that each order line has a different part number. This cannot be done using XML IDs.
- Finally, XML Schema enables the creation of a key or a keyref constraint from combinations of element and attribute content and does not restrict the possible datatypes for valid keys. XML IDs consist of single attribute content, and must be of the ID datatype.

5.1.2 Referential Integrity Constraints

Referential integrity constraints (defined using the keyref element in an XML Schema document) are similar to the corresponding constraints in the relational model. Each referential integrity constraint refers to a valid key or unique constraint and ensures that the corresponding key value exists in the document. For example, a keyref can be defined to ensure that only valid product numbers (i.e., those that exist for a <product> element) are entered for an order.

A sample definition of a referential integrity constraint in XML Schema to specify that an order should always be for a valid product follows.

```
...
<xs:keyref name="ordersProductRef" refer="productKey">
   <xs:selector xpath="order"/>
   <xs:field xpath="oProductNo"/>
<xs:keyref>
...
```

Listing 8. Sample Referential Integrity constraint

Formally, we can define the keyref constraint as follows. Let $Eval(sel_r, F_r)$ denote the result of evaluating the list F_r of keyref XPath field expressions relative to the selector element sel_r. Let e_r be an element from the list defined by $Eval(sel_r, F_r)$ (e.g., one of the product nodes in an XML document). Similarly, let $Eval(sel_k, F_k)$ denote

the result of evaluating the referenced `key` constraint, and $e_k \in Eval(sel_k, F_k)$. The `keyref` constraint is satisfied when $\exists e_k$ (in the document) such that $e_r = e_k$.

5.1.3 Cardinality Constraints

The cardinality of elements in XML documents is restricted by the use of `minOccurs` and `maxOccurs` in the XML Schema document. For example, to limit there being from zero to four website URLs for suppliers, the `minOccurs` of `<sURL>` is set to `0`, and the `maxOccurs` to `4`.

While there can be multiple sub-elements with the same name, there can be a maximum of one attribute (for example, `supplierNo`) with a given name. The cardinality for attributes is therefore restricted using either `optional` or `required`. An example of cardinality definitions in XML Schema follows.

```
...
<xs:element name="supplier" minOccurs="0"
            maxOccurs="unbounded">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="sURL" type="xs:anyURI"
                  minOccurs="0" maxOccurs="4"/>
      ...
    </xs:sequence>
    <xs:attribute name="supplierNo"   type="xs:integer"
                  use="required"/>
    <xs:attribute name="supplierName" type="xs:integer"
                  use="required"/>
  </xs:complexType>
</xs:element>
...
```

Listing 9. Cardinality definitions using XML Schema

Let (n, c) be the list of child elements c within node n. We use $|(n, c)|$ to represent the cardinality of the list (n, c). Then, $\texttt{minOccurs}(c) \leq |(n, c)| \leq \texttt{maxOccurs}(c)$.

5.1.4 Datatype Restrictions

Datatype definitions in XML Schema can restrict the structure and content of elements, and the content of attributes. We currently consider datatypes defined using the XML Schema `simpleType` element. A simple type is used to specify a value range. In the simplest case, a built-in XML Schema datatype (e.g., `integer`) imposes a value range. For more complicated requirements, a simple type can be derived from one of the built-in datatypes.

An example of an XML Schema datatype definition follows.

```
...
<xs:simpleType name="supplierRating">
  <xs:restriction base="xs:string">
    <xs:enumeration value="A"/>
    <xs:enumeration value="B"/>
    <xs:enumeration value="C"/>
  </xs:restriction>
</xs:simpleType>
...
```

Listing 10. XML Schema data type definition

Formally, let $type(n)$ be the set of values that the datatype assigned to node (element or attribute) n allows. Then, in any given document instance, the XML expression $n/text() \in type(n)$.

6 Temporal Augmentations to XML Schema Constraints

[TODO: Faiz]

(TR: Section 7.2)

- Basics of temporal constraint terminology and approach (conventional)
- Implicit (e.g., existence time of a child within bounds of its parent) vs. Explicit temporal document constraints (XML schema specified)
- How schema versioning affects constraints

from TR

Thus far we have considered conventional XML Schema constraints. We now proceed to discuss temporal augmentations to these constraints.

The time frame over which a constraint is evaluated classifies it into one of two types, either *sequenced* or *non-sequenced*. A temporal constraint is sequenced with respect to a similar conventional constraint in the schema document, if the semantics of the temporal constraint can be expressed as the semantics of the conventional constraint applied at each point in time. A constraint is non-sequenced if it is evaluated over a temporal element as a whole (including the lifetime of the data entity) rather than at a point in time.

Given a conventional XML Schema constraint, the corresponding semantics in τXSchema for a temporal document, is a *sequenced* constraint. For example, a conventional (cardinality) constraint, "There should be between zero and four website URLs for each supplier," has a sequenced equivalent of: "There should be between zero and four website URLs for each supplier *at each point in time*."

For the non-sequenced extension to constraints, we consider a window of evaluation, w, which can be a temporal element. The user specifies the window of evaluation (e.g., a day, or a Gregorian month). The user can also specify a slide size, ss, and applicability bounds, B [**?**]. The default for ss is the granularity, $gran$, of the underlying element (or attribute). The default for B is the lifetime of the temporal document. We've established the following relationship among the components of a non-sequenced constraint: $gran \leq ss \leq w \leq B$

The applicability bounds, $B \subseteq T$, allow the user to restrict their consideration from the lifetime of the document, to some desired subset they are interested in. For example, a constraint may only be valid between 1999-2005, at which time it's replaced by a new constraint. Strictly speaking, applicability bounds can be introduced for sequenced constraints as well. While the effect of applicability bounds (for a sequenced constraint) can be simulated by "removing" the constraint from the schema document (during some time slice), this restricts it to cases where the transaction time and valid time are identical.

Non-sequenced constraints are evaluated over a temporal element rather than at a point in time. The window of evaluation must be within the applicability bounds. So, for non-sequenced constraints, we replace the evaluation point t, where $t \in T$, with $w \in \mathcal{P}(B)$. When $size(w)$ is the same as $size(B)$, we term it a "fixed-window" constraint. For example, suppose the constraint requires there to be between 0 and 4 supplier URLs in the temporal document over a period of any

calendar month. Let's say this constraint is applicable from 2009-03-01 to 2009-03-31. Here, w and B have the same size. If, instead the applicability were (2009-03-01 to 2009-06-31), then we see a case of a "sliding-window" constraint (since the evaluation would take place during *each* month from March through June. Here, we see the the size of the slide is implicitly a *calendar month* as well. Let's say instead, the constraint evaluation window were a period of 30 days. Then the user may wish to restrict how this evaluation window would slide. For example, one may choose to evaluate it from March 1–30, then from March 2–31, and so on. Here, the size of the slide (ss) is a single day.

Non-sequenced constraints are listed in the logical annotations document. In a few cases (when we extend a particular XML Schema constraint for additional functionality), sequenced constraints are also listed in the logical annotations document. We now proceed to discuss temporal enhancements to each of the XML Schema constraints described in section 5.1. The general approach is to add non-sequenced extensions to each constraint (though for sequenced cardinality constraints, we add new semantics as well).

7 Identity Constraints

[TODO: Faiz]

(TR: Section 7.2.1)

- Definition and how it relates to the XML Schema constraint
- Semantics (formal)
- Specification (how the annotation works + example)

from TR

Conventional identity constraints restrict uniqueness in a given XML document and induce sequenced identity constraints in the temporal document. Non-sequenced extensions may further be defined for these constraints.

We have shown in section 5.1.1 the advantages of XML Schema identity constraints over defining an element or attribute to have a type of `ID`. This motivates the following design decision: we extend the semantics of XML Schema identity constraints to support non-sequenced semantics, but do not consider non-sequenced extensions to `ID` types. If an element or attribute in an XML Schema document is said to have a type of `ID`, then that only translates to a sequenced constraint.

Formally, we define a sequenced `key` constraint as follows. Let T be the set of time points associated with a temporal XML document over its lifetime. At any given time t (where $t \in T$), we can extract a snapshot of the document. As with the conventional case, let sel be a context node (defined by the underlying `selector`), with the list of corresponding field expressions $F = (f_1, \ldots, f_m)$. Let $Eval^t(sel, F)$ denote the resultset of evaluating the list of XPath expressions F from the context node sel at point t. We denote the i^{th} element of $Eval^t(sel, F)$ by e_i^t. Then for $\forall t \in T, \forall i, j : e_i^t \neq e_j^t$ unless $i = j$; $i, j \leq n$.

The definition of a sequenced `unique` constraint is similar (but allows null values).

The non-sequenced extensions to the identity constraint we consider are: *time-invariant unique*, *time-invariant key*, *non-sequenced unique*, and *non-sequenced key*. We discuss each of these in turn.

A time-invariant restriction specifies that the value of the given conventional `unique` or `key` constraint should not change over time. Without this restriction, conventional `unique` and `key` constraints simply say that the values must not have duplicates. However, this does not preclude the values from changing as long as the new value does not appear elsewhere in the conventional XML document. For example, given the key definition for product in Listing 7, the following snippets (Listings 11 and 12) reflect a perfectly legal change from one state to another for the `productNo` attribute (from `500` to `599`) within the first `<product>` element.

```
...
<product productNo="500">
  <name>17" LCD, Model 350</name>
  <qtyOnHand>25</qtyOnHand>
</product>
<product productNo="501">
  <name>19" LCD, Model 370</name>
  <qtyOnHand>10</qtyOnHand>
</product>
...
```

Listing 11. Initial State for `productNo` attribute

```
...
<product productNo="599">
  <name>17" LCD, Model 350</name>
  <qtyOnHand>25</qtyOnHand>
</product>
<product productNo="501">
  <name>19" LCD, Model 370</name>
  <qtyOnHand>10</qtyOnHand>
</product>
...
```

Listing 12. Changed State for `productNo` attribute

As seen in the previous example, conventional identity constraints do not necessarily imply a *non-sequenced identity constraint* (i.e., a value that can uniquely identify the particular element or attribute across time). Thus, the same `productNo` (a conventional key) can be changed between snapshots (as long as it remains unique) and re-used for another product. Additionally, we may end up with two or more products with the same product number over time. This is why we do not use the term "temporal identity constraints," and instead select *non-sequenced unique* or *non-sequenced key* constraints. This also helps avoid confusion or overlap with the concept of *itemIdentifiers* introduced in Part **??** (for schema versioning).

With respect to time-invariance, if `productNo` is declared as time invariant, then no change can be made to its value. A new `productNo` value indicates an instance of a new product. The simplest way to ensure time-invariance, is by defining the item as non-temporal in the logical annotations. In combination with a unique constraint on the element or attribute, this is sufficient to provide for time-invariance.

A non-sequenced unique constraint indicates that the unique value should not be re-used at a later time within an evaluation window. When writing the non-sequenced restrictions in the logical annotations, we use a newly introduced element `<uniqueConstraint>`. We adopt the usual distinction in

semantics between key and unique (i.e., the permissibility of *null* values).

With the refinements introduced in Section **??**, we define a non-sequenced key constraint as follows. Let $Eval^w(sel, F)$ denote the resultset from the evaluation of the key constraint over window w. An element of this set is e_i^w, where $e_i^w \in Eval^w(sel, F)$. Then, $\forall w \in \mathcal{P}(B), \forall i, j : e_i^t \neq e_j^t$ unless $i = j$; $i, j \leq n$. The effect of the slide size is to determine the start point for successive $i, i+1, \ldots, n$. Or in other words, to constrain the power-set $\mathcal{P}(B)$ to have specific temporal elements.

The next kind of constraint we discuss is uniqueNullRestricted. Since the XML Schema definition of unique allows a NULL value at each point in time, the default semantics for unique allows for multiple NULL values across time (one in each conventional document). A non-sequenced uniqueNullRestricted constraint restricts the appearance of the number of NULL values by allowing the user to specify a finite number (one or more) across time; the default number being one. Setting the number of nulls allowed across time to 0 is equivalent to specifying a non-sequenced key constraint. A non-sequenced key constraint, as might be expected, disallows NULL values in any of the key fields at any time.

We use the term $item$, to refer to a constrained attribute or element, and val to a specific value (including null) that we are interested in. Let $temp(D^B, w, item, val)$ evaluate to the set of maximally coalesced temporal elements associated with an $item$ within the document D, during the evaluation window w (applicability bound of B), where the value of $item$ = val. Then setting val to *null*, returns the set te where the $item$ is *null*. The cardinality, $|te|$, of the set is the number of times *null* appears (counting each contiguous appearance as a single block); nullCountMin $\leq |te| \leq$ nullCountMax.

A more powerful version of the unique (or key) constraint allows the user to specify exactly how many times any key (or unique) value can appear across time other than NULL. The default is 1—in which case it is identical to a non-sequenced unique or a non-sequenced key constraint. We term this constraint as a *value cardinality constraint*, but do not explore it for now since it has no XML Schema equivalent.

We now proceed to discuss the different attributes and sub-elements for the uniqueConstraint (summarized in Table 1; the sub-elements are indented).

- name: This allows the user to name the constraint and is useful in case the constraint is referenced elsewhere (e.g., in a referential integrity constraint).
- conventionalIdentifier: Specifies the name of the identifier in the conventional schema document. If this is not specified, then it implies a new constraint is being defined and the selector and field sub-elements should not be empty.
- type: The type is one of key, unique, or uniqueNullRestricted, the semantics of which have been discussed previously. By default, the type is the same as that of the conventional constraint (i.e., key or unique). In the event a new constraint is defined, the usage of this attribute (type) is required. Further, an existing conventional unique constraint can be restricted to be a non-sequenced key (i.e., no null values allowed), "for a given period of applicability". The converse is not meaningful since coercing a conventional key constraint to unique in a non-sequenced context would violate the key constraint during some conventional document state. Conventional unique constraints can also have a type of uniqueNullRestricted.
- nullCountMin: Used only in conjunction with the uniqueNullRestricted constraint to specify how many nulls are allowed over the non-sequenced time extent (minimum).
- nullCountMax: Used in conjunction with the uniqueNullRestricted constraint to specify how many nulls are allowed over the non-sequenced time extent (maximum).
- dimension: Specifies the dimension in which the unique constraint applies and is one of validTime, transactionTime, or bitemporal. The default is assumed to be validTime since that is closely related to capturing real world restrictions, rather than restrictions on data entry.
- evaluationWindow:[1] Specifies the time window over which the constraint should be checked. This allows uniqueness to be specified for an interval, e.g., year. This is useful when, for example, a particular key value should not be re-used for a period of a year. The value then must be "unique over any period of a year". By default the evaluation window is the lifetime of the time varying XML document. Assuming we use XML Schema to specify the datatypes for time intervals, we can extend that with a union of the string: lifetime. This will allow us to set the time interval for evaluationWindow (and other attributes) to a value of lifetime (indicating a temporal element equivalent to the lifetime of the XML document). In the constraint examples that follow, we assume this datatype extension is done and use the keyword lifetime when needed.
- slideSize: Specifies the size of the slide; must be used in conjunction with an evaluation window. By default, it takes the temporal granularity of the underlying item being constrained.
- applicability:[2] The applicability of a constraint specifies when it was valid. Thus a key constraint may be enforced between 2005 and 2010. Strictly, the applicability need not be a single range, and may be a temporal element, which is why we specify the applicability as (begin, end) attribute pairs within a wrapping sub-element called applicability. If nothing is specified, the default is assumed to be the lifetime of the document. The applicability and evaluation window of the constraint are related. Defining an evaluation window that exceeds the applicability of the constraint is not really meaningful, as it cannot be checked beyond the constraint applicabil-

1. Strictly speaking, the evaluation should be defined as a series of <begin> ... <end> periods. For simplicity we keep it as an attribute.

2. Applicability applies to the valid time dimension. Transaction-time applicability would be when a constraint exists in the schema document.

ity. In such a case, a warning should be returned and the evaluation window should be shortened to the maximum allowable within the constraint applicability limits.

- `selector` and `field`: The `selector` specifies the context within which the combination of `field` XPath expressions should evaluate to unique. A `selector` can have one of two attributes specified. If the `xpath` attribute it specified, it is evaluated relative to the point of definition (of the constraint) within the document. The other option is to use a `itemref` attribute. This provides schema versioning support by allowing the `selector` reference to have flexibility across versions. The only other requirement for schema versioning, is that the elements and attributes picked by `field`, do not change across schemas (or if they do, the constraint is redefined). Multiple `field` sub-elements may be listed. The combination of these are taken for the constraint specification. The `field` sub-elements have a usage identical to their conventional XML Schema counterparts, and have a single `xpath` attribute. `selector` and `field` are needed to specify a new constraint (i.e., those that were not defined as identifiers in the conventional schema). If a new constraint is defined, the `conventionalIdentifier` attribute should not be used. A new constraint can be either defined as either a `key` or a `unique` constraint.

We now describe some non-sequenced `unique` constraint examples.

1) *Supplier numbers are keys but may be re-used over time—however the reuse should not occur for at least one year after discontinuation. Supplier numbers are also allowed to change as long as no other supplier has that number. Part numbers on the other hand may not be re-used later, but may change. The constraints are applicable between 2005 and 2009.*

```
...
<uniqueConstraint type="key" name="idSupplierNo"
    conventionalIdentifier="supplier_key"
    dimension="validTime"
    evaluationWindow="year" slideSize="day">
   <applicability begin="2005-01-01" end="2009-12-31"/>
</uniqueConstraint>
...
<uniqueConstraint type="key" name="idPartNo"
    conventionalIdentifier="part_key"
    dimension="validTime"
    evaluationWindow="lifetime">
   <applicability begin="2005-01-01" end="2009-12-31"/>
</uniqueConstraint>
...
```

2) *A product's key (in both valid and transaction time) is its RFID number. The constraint is applicable from 2007 onwards. A null value may be allowed in the beginning, but once an RFID tag is attached to the product, it should not revert to null at a later point.*

```
...
<uniqueConstraint type="key" name="product_RFID"
     dimension="bitemporal"
     evaluationWindow="lifetime" nullCountMax="1">
   <applicability begin="2007-01-01" />
   <selector xpath="product"/>
   <field xpath="@RFID"/>
</uniqueConstraint>
...
```

3) *Employee email addresses are optional. If they do exist, they should be unique and should not be re-used for a two-year period.*

```
...
<uniqueConstraint type="unique" name="employee_email"
     evaluationWindow="two-years" slideSize="day"/>
...
```

8 Referential Integrity Constraints

[TODO: Faiz]

(TR: Section 7.2.2)

- Similar structure of 3 sections as with Identity constraints; so also for next two sections.

from TR

Each referential integrity (`keyref`) constraint for a conventional document leads to a sequenced counterpart in a temporal document. Thus, each conventional keyref obeys referential integrity.

A non-sequenced referential integrity constraint is useful to specify a reference to some past state of the XML document. Suppose, for example, the "largest order" (in dollar terms) placed by a customer is stored with the customer data (with a `keyref` to `orderNo`). Also, to maintain a compact XML document, orders that are over a year old are deleted from the document. Therefore, a non-sequenced referential the integrity constraint could state, "The largest order the customer has placed should be for an order that existed in the document at some time."

Formally, we can define the temporal `keyref` constraint as follows. Let $Eval^t(sel_r, F_r)$ denote the result of evaluating the list F_r of `keyref` XPath `field` expressions relative to the `selector` element sel_r (at time t, $t \in B$, during the applicability bounds B). Let e_r be an element from the list defined by $Eval^t(sel_r, F_r)$. Similarly, let $Eval^t(sel_k, F_k)$ denote the result of evaluating the referenced `key` constraint, and $e_k \in Eval^t(sel_k, F_k)$. The `keyref` constraint is satisfied when $\exists e_k$ (in the document) such that $e_r = e_k$.

One might think that there should be a limitation preventing referential integrity constraints within state data referring to event data. However, for XML, there does not need to be such a limitation. Consider the following example: Scientists take readings about the temperature and humidity levels at an observation post. Each observation can be considered an event. Information on the scientists on the other hand is state data. Depending on the structure of the document, `<scientist>` can be the enclosing element with `keyrefs` to the appropriate `<observation>` or `<observation>` can be the enclosing element with a reference to the scientist(s) who were responsible for it. Both options can be defined using XML Schema and should be allowed.

Intuitively, a non-sequenced `keyref` constraint should refer to the definition of an identifier that does not permit re-use. Without the restriction of not permitting re-use, the semantics of the referential integrity may not be well defined. For example, if the order number could be re-used, then a customer's largest order may end up referencing an order that

Term	Definition	Cardinality
name	The name of the constraint	optional
conventionalIdentifier	The referenced conventional identifier	optional
type	key, unique, or uniqueNullRestricted (default: the conventional constraint type)	optional
nullCountMin	The number of null values allowable (used only within uniqueNullRestricted)	optional
nullCountMax	The number of null values allowable (used only within uniqueNullRestricted)	optional
dimension	validTime, transactionTime, or bitemporal (default: validTime	optional
evaluationWindow	Time window over which the constraint should be checked (default: lifetime of document)	optional
slideSize	Size of the slide for successive evaluation windows (default: granularity of constrained data type); Only used in conjunction with evaluationWindow	optional
applicability (begin, end)	When the constraint is applicable (default: lifetime of document) Temporal element to specify applicability, with a series of intervals	[0:1] [0:U]
selector	For the definition of a new constraint. It is similar to the selector sub-element in the uniqueConstraint definition	[0:1]
field	For the definition of a new constraint. It is similar to the field sub-element in the uniqueConstraint definition	[0:U]

TABLE 1
Attributes and sub-elements for uniqueConstraint

was not originally placed by that customer. Not permitting reuse of order numbers however is a strict constraint that can be omitted if the largestOrderNo has a valid-time timestamp. Then, the non-sequential reference can be understood to be for a specific order number that was valid at the begin time of the largestOrderNo.

We represent a non-sequenced referential integrity constraint using a nonSeqKeyref element in the logical annotations. Next, we proceed to discuss the different attributes and sub-elements for the nonSeqKeyref (summarized in Table 2; the sub-elements are indented).

- name: This allows the user to name the constraint and is useful in case the constraint is referred to elsewhere. In a managed environment, this would also aid in allowing constraints to be disabled or dropped.

refer: Denotes a referenced constraint (either conventional or temporal), i.e., the name of the constraint, that the non-sequenced referential integrity constraint refers to. If the constraint being referred to is a conventional keyref, then it is in effect just extending the semantics of the conventional constraint (e.g., with an applicability bounds). In this case, it inherits the referred key constraint information. If this is a new constraint, then we need to refer to both an existing key (either conventional or temporal), and define the selector and field properties.

- applicability: A non-sequenced keyref can be associated with a particular applicability that specifies when it was in effect. If the applicability is not specified, the default is assumed to be the lifetime of the document. As with uniqueness constraints, the applicability can be a temporal element.
- selector and field: These two sub-elements have a usage identical to their uniqueConstraint counterparts, but are needed to specify a new constraint (i.e., those that were not defined as referential integrity constraints in the conventional schema).

New non-sequenced referential integrity constraints may be defined (i.e., those that were not defined as a keyref in the conventional schema).

resume

1) *A non-sequenced referential integrity constraint is to be defined for the product number in orders. We assume an existing referential integrity constraint exists in the conventional schema. The name of the corresponding keyref is* ordersProductRef *which references a valid part number. The constraint is applicable from 2005–2009.*

```
...
<nonSeqKeyref name="ordersProductRef_NS"
              refer="ordersProductRef">
   <applicability begin="2005-01-01" end="2009-12-31"/>
</nonSeqKeyref>
...
```

2) *A non-sequenced referential integrity constraint is to be defined for the customer email in orders. It should reference a valid email address. The corresponding unique constraint within customers is defined as* custEmailsUnique. *The referential integrity constraint is applicable from 2008–2012, and no corresponding conventional constraint exists.*

```
...
<nonSeqKeyref name="ordersCustEmailRef"
              refer="custEmailsUnique">
   <applicability begin="2008-01-01" end="2012-12-31"/>
   <selector xpath="order"/>
   <field xpath="oCustEmail"/>
</nonSeqKeyref>
...
```

9 Cardinality Constraints

[TODO: Faiz]

(TR: Section 7.2.3)

from TR

As discussed in Section 5.1.3, the cardinality of elements in conventional documents is restricted by using minOccurs and maxOccurs, and that of attributes by using optional and required. These automatically induce sequenced constraints in the temporal document.

Non-sequenced constraints can be used to restrict the cardinality over time. Consider the example of an order element

Term	Definition	Cardinality
name	The name of the constraint	optional
refer	The referenced identifier or referential integrity constraint	optional
applicability	When the constraint is applicable (default: the lifetime of the document)	[0:U]
selector	Used in the definition of a new constraint	[0:1]
field	Used in the definition of a new constraint	[0:U]

TABLE 2
Attributes and sub-elements for nonSeqKeyref

in Listing 13. We see that the deliveredOn element may not always be present in a specific document snapshot. Let us further say, that while it may be empty at the time the order was placed, we require it to be appear at some point (say within three months of the order being placed). So, even though minOccurs="0" is satisfactory for a conventional document, we may desire the equivalent minOccurs="1" for a temporal document.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
   <xsd:element name="order">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="orderNo" type="xsd:string" />
          <xsd:element name="orderDate" type="xsd:date" />
          <xsd:element name="deliveredOn"
                       minOccurs="0" maxOccurs="1"
                       type="xsd:date" />
          ...
          <xsd:element ref="product" minOccurs="1"
                                     maxOccurs="unbounded" />
          ...
        </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
...
```

Listing 13. Orders with an optional deliveredOn

For attributes, a similar requirement may be placed (i.e., a snapshot optional attribute, may be required over some evaluation window).

Another refinement that may be desired for a non-sequenced cardinality constraint is the *aggregation level* at which the count is being performed. Let's consider the schema in Listing 14. A non-sequenced cardinality constraint can be used to place a limit of one hundred orders from a supplier in any given year. In this case, order is the direct child of suppliers, and the conventional maxOccurs constraint (on order) would be used to restrict the number of child order elements a supplier can have. Suppose, we wished to further constrain the number of orders for the company across the all suppliers to 1500 per month. In other words, the number of order elements that were descendants of company) should be $\leq$ 1500 in any calendar month. The conventional cardinality constraints are not designed to handle this. This is our motivation behind introducing the aggLevel option for a cardinality constraint.

```
...
<xsd:element name="company">
...
  <xsd:element name="supplier" minOccurs="1"
                               maxOccurs="unbounded">
  ...
     <xsd:element ref="order" minOccurs="0"
                              maxOccurs="unbounded" />
  ...
  </xsd:element>
..
</xsd:element>
...
```

Listing 14. Considering Aggregation Levels for an order

We represent temporal cardinality constraints using a cardConstraint element in the logical annotations document. Formally, we define the cardConstraint (of type childList)as follows. Let:

- sel be a context node (defined by selector in a cardinality constraint), with the list of corresponding field expressions $F = (f_1, \ldots, f_m)$.
- $ancestorOf(n, a)$ return true if a is an ancestor of node n, and false otherwise.
- For two lists L_1 and L_2, let $L_1 \uplus L_2$ return the result of appending the members of L_2 to L_1.
- $eval^w(sel, F)$ be the result returned by the evaluation of F relative to sel (over the evaluation window w). There may be many nodes corresponding to a given sel (e.g., many supplier nodes), and each such (supplier) node can have many children. Therefore $eval^w(sel, F)$ returns a list of lists.
- $childList_i \in eval^w(sel, F)$ be the i^{th}, member (also a list) of the result.
- $childListAggr = aggr(childlist_i, aggLevel)$ be the aggregation of the various $childList_i$ at the ancestor level of $aggLevel$; i.e., $childListAggr = (childList_1 \uplus \ldots childList_i \uplus \ldots childList_n)$, where $\forall i, ancestorOf(sel, aggLevel)$ is true.

Then the cardConstraint restricts: min $\leq |childListAggr| \leq$ max, if an aggLevel is defined. If the aggLevel is empty, then: min $\leq |childList| \leq$ max. Here $|list|$ is the count of members in the $list$.

The definition for childList can be modified for set semantics, childSet, by only considering distinct elements (i.e., duplicates are not considered). This brings up the issue of how to determine if two nodes are duplicates. One option is to go with the understanding of DOM-equivalence defined in Section 4.4. Another option is to consider two nodes equivalent if the values of their corresponding unique constraints match. When defining the uniqueref attribute in cardConstraint, the selector XPath for the unique constraint definition should match the node being counted in the cardinality constraint (i.e., the field XPath expression in cardinality).

Formally, we let:

- $childSet_i \in eval^w(sel, F)$ be the list of child nodes returned by the evaluation of the XPath expressions F

relative to sel (over the evaluation window w).

- $childSetAggr = aggr(childSet_i, aggLevel)$ be the aggregation of the $childSet_i$ resultsets at the ancestor level of $aggLevel$; i.e., $childSetAggr = (childSet_1 \cup \ldots \cup childSet_i \cup \ldots \cup childSet_n)$, where $\forall i, ancestorOf(sel, aggLevel)$ is `true`.

Then the `cardConstraint` restricts: `min` $\leq |childSetAggr| \leq$ `max`, if an `aggLevel` is defined. If the `aggLevel` is empty, then: `min` $\leq |childSet| \leq$ `max`. Here we use the notation $|childSet|$ is the cardinality, that is, the count of members in the $childSet$. As can be seen, the definitions for the `restriction` type of `childSet` are very similar to the definition for the type of `childList`.

resume

1) *There should be no more than fifty active suppliers (i.e., in the "database") in any year. This constraint is true between 2007 and 2009. [*`childList` *constraint]*

```
...
<cardConstraint name="supplierCardYear"
    restriction="childList" dimension="validTime"
    evaluationWindow="year" slideSize="day"
    max="50">
   <selector xpath="company" />
   <field xpath="supplier" />
   <applicability begin="2007-01-01" end="2009-12-31"/>
</cardConstraint>
...
```

2) *No supplier should be given more than one hundred orders in a calendar month month. These orders should not be for more than five hundred different products.* Note: we do not do SUM type constraints here, since they are not an extension of minOccurs or maxOccurs (Different kinds of aggregation).

```
...
<cardConstraint name="supOrders"
    restriction="childList" dimension="validTime"
    evaluationWindow="month" slideSize="month"
    max="100">
   <selector xpath="company/supplier" />
   <field xpath="order" />
</cardConstraint>
<cardConstraint name="supParts"
    restriction="childList" dimension="validTime"
    evaluationWindow="month" slideSize="month"
    aggLevel="company/supplier" max="500">
   <selector xpath="company/supplier/order" />
   <field xpath="product" />
</cardConstraint>
...
```

3) *There should be a maximum of 250 potential suppliers for the company across all products. We assume there exists a unique constraint on the potential supplier's supplierNo attribute. This constraint is to be enforced during 2009. [Sequenced constraint; use of* `childSet`*]* This is a sequenced constraint. However it cannot be enforced by a combination of a `minOccurs` and `maxOccurs`.

```
...
<cardConstraint name="potential_suppliers_seq"
    restriction="childSet"
    uniqueRef="potential_supplierNo"
    dimension="validTime" sequenced="true"
    aggLevel="company" max="250">
   <selector xpath="company/product" />
   <field xpath="potential_supplier" />
   <applicability begin="2009-01-01" end="2009-12-31"/>
</cardConstraint>
...
```

Another kind of constraint we consider is restricting the cardinality of the `valueList`, i.e., the min/max number of "values" that an element or attribute can have over a specific evaluation window. This constraint does not have an XML Schema equivalent. So it does not fit into a strict extension of XML Schema semantics.

A `valueList` restriction is related to the datatype of the item (which specifies the possible values an item can take). For example, suppose an order `status` attribute can have one of the five following values: `placed`, `underReview`, `being_processed`, `shipped`, and `returned`. It is possible that changes to the order can have it swap back and forth between `underReview` and `being_processed`. Therefore over a period of a month, it can potentially have seven values. However the number of *distinct* values that status can have is five or fewer. In this sense, the `valueList` and `valueSet` restriction kinds are analogous to the SQL notion of `COUNT(`$attribute$`)` and `COUNT(distinct` $attribute$`)`.

For both of the two `valueList` restrictions, child elements (or attributes) are not being counted. Instead it is the value of the element (or attribute) itself. So, the semantics of the `cardConstraint/selector` element is different from that for `childList` or `childSet`. In the latter, the `selector` is used to set up the context node, relative to which the child items described by the `field` nodes are counted. With `valueList` constraints, the `selector` is used to decide the item for which the values will be counted. Typically for the `valueList` cardinality constraints, the `field` expression will contain a terminal `/text()` function.

The formal definition for `valueList` and `valueSet` constraints are similar to those for `childList` and `childSet`. The main difference being in the $eval^w(sel, F)$ function, which instead of returning a list (or set) of nodes (element or attribute), returns the *value* or content of those nodes. An example of a `valueList` and `valueSet` cardinality constraints follows.

resume

1) *A product should have only one name in any month, but can have up to three distinct names in a year. This is in force during 2008–2010. [*`valueList` *and* `valueSet` *constraints; different evaluation window sizes used]*

```
...
<cardConstraint name="prodNameMonth"
    restriction="valueList" dimension="validTime"
    evaluationWindow="month" slideSize="day"
    min="1" max="1">
   <selector xpath="product" />
   <field xpath="@productName/text()" />
   <applicability begin="2008-01-01" end="2010-12-31"/>
</cardConstraint>
<cardConstraint name="prodNameYear"
    restriction="valueSet" dimension="validTime"
    evaluationWindow="year" slideSize="day"
    min="1" max="3">
   <selector xpath="product" />
   <field xpath="@productName/text()" />
   <applicability begin="2008-01-01" end="2010-12-31"/>
</cardConstraint>
...
```

We now proceed to discuss the different attributes and sub-elements for the `cardConstraint` (summarized in Table 3; the sub-elements are indented).

- `name`: This allows the user to name the constraint and is useful in case the constraint is referenced later.
- `restriction`: Cardinality constraints can restrict the `childList`, `childSet`, `valueList`, and `valueSet` counts of elements and attributes. `childList` refers to the actual number of sub-elements that can appear over time, and is analogous to the conventional minOccurs and maxOccurs for sequenced constraints. The difference between `childList` and `childSet` is similar, in that duplicate sub-elements are not counted for `childSet`. Duplication is determined using by referencing an applicable uniqueness constraint (which in terms specifies the fields to be evaluated).
- `uniqueRef`: Used along with `childSet` to eliminate duplicates.
- `dimension`: Specifies the dimension in which the cardinality constraint applies and is one of `validTime`, `transactionTime`, or `bitemporal`.
- `evaluationWindow`: Associated with a non-sequenced cardinality constraint is the time window over which the constraint should be checked. This allows cardinality minimum and maximum ranges to be specified for an interval, e.g., year. This is useful, for example, when a restriction needs to be put on how many orders suppliers can handle in any given period. By default the time window is the lifetime of the XML document.
- `slideSize`: Associated with the time window of evaluation. By default it is the granularity of the underlying data type.
- `sequenced`: Denotes if the constraint is sequenced or not (using either `true` or `false`). This is allowed in the constraint specification since XML Schema only allows `minOccurs` and `maxOccurs` to be aggregated at the target parent level. Allowing a different aggregation level is useful, for example, if instead of restricting the number of potential suppliers for a product (assuming `<potential_suppliers>` is a child element of `<product>`), we wish to restrict the total number of potential suppliers the company maintains relationships with at any time. If a constraint is specified as sequenced, the `evaluationWindow` attribute must not be used.
- tt aggLevel: Specifies the level at which the aggregation is performed for cardinality constraints; by default it is at the level of the target's parent. This is also the reason why we allow sequenced cardinality specifications. For a sequenced constraint to be useful, the aggregation level should not be the target's parent.
- `min` and `max`: Specify the minimum and maximum cardinality respectively.
- `selector` and `field`: These two sub-elements have a usage identical to XML Schema usage for conventional constraints.
- `applicability`: The constraint applicability specifies when it was in effect. If the applicability is not specified, the default is assumed to be the lifetime of the document. The applicability can be a temporal element.

10 Datatype Restrictions

[TODO: Faiz]
TR: Section 7.2.4
from TR

10.1 Datatype Restrictions (Constraints)

As mentioned in section 6.1.4, we currently consider non-sequenced augmentations to the XML Schema `simpleType` element. A simple type is used to specify a value range and induces a sequenced constraint that ensures conventional document values conform to this range.

A non-sequenced equivalent of this type of constraint can be considered either at the schema level (i.e., datatype evolution—within schema evolution) or at the instance level (transition constraints). Schema-level constraints restrict the kinds of changes possible to the datatype of an item. However, we do not see much need for this type of a constraint.

At the instance level (i.e., conforming to a particular type specification), a temporal constraint could restrict discrete and continuous changes. Discrete changes can be handled by defining a set of value transitions for the data. For example, it could be specified that while supplier ratings can change over time, the changes can only occur in single-step increments (i.e., B to either A or C). Continuous changes are handled by defining a restriction on the direction of the change. For a transition constraint to be applicable, a corresponding datatype should be defined at the conventional schema level.

We now proceed to discuss the different attributes and sub-elements for the `transitionConstraint` (summarized in Table 4; the sub-elements are indented).

- `name`: This allows the user to name the constraint and is useful in case the constraint is referenced elsewhere.
- `dimension`: Specifies the dimension in which the unique constraint applies and is one of `validTime`, `transactionTime`, or `bitemporal`. The default is `validTime` since a cardinality constraint on transaction time is akin to specifying how many "data entry changes" can be made to an element or attribute.
- `selector` and `field`: These two sub-elements have a usage identical to their conventional XML Schema counterparts.
- `valuePair`: This is used to list possible pairs for discrete changes. The pairs themselves are specified as `<old>` and `<new>` sub-elements. valuePair cannot be used simultaneously with `valueEvolution`. The values listed here should be within the range of values defined for the conventional `simpleType` datatype.
- `valueEvolution`: This sub-element lists the direction for continuous changes. Only one of `valuePair` and `valueEvolution` should be used. The values listed here should be within the range of values defined for the conventional `simpleType` datatype. Continuous

Term	Definition	Cardinality
name	The name of the constraint	optional
restriction	One of childList, childSet, valueList, valueSet	required
uniqueRef	Reference to a uniqueness constraint—only used with childSet	optional
dimension	Either validTime or transactionTime (default: validTime)	optional
evaluationWindow	Time window over which the constraint should be checked (default: lifetime of document)	optional
slideSize	Size of the slide for successive evaluation windows (default: granularity of constrained data type); Only used in conjunction with evaluationWindow	optional
sequenced	If it is a sequenced constraint (default: false)	optional
aggLevel	The level at which the aggregation is performed (default: parent level)	optional
min	minOccurs equivalent (default: 0)	optional
max	maxOccurs equivalent (default: unbounded)	optional
selector	Role and definition is similar to the selector sub-element in the conventional XML Schema constraint definitions (e.g., for keyref constraints)	[1:1]
field	Similar to the field sub-element in the conventional XML Schema constraint definitions. Allowing for multiple field elements lets us constraint combinations of entities.	[1:U]
applicability	When the constraint is applicable (default: lifetime of the document)	[0:U]

TABLE 3
Attributes and sub-elements for cardConstraint

changes of the following direction are currently supported:

- strictlyIncreasing: the value should be strictly increasing
- strictlyDecreasing: the value should be strictly decreasing
- nonIncreasing: the value should be non-increasing
- nonDecreasing: the value should be non-decreasing
- equal: the value should be equal, i.e., no change allowed.

The last type, equal, should only be used in conjunction with the applicability begin and applicability end to restrict when the value of a particular element or attribute (e.g., salary) should not change. This allows us flexibility over annotating salary to be non-temporal since the user may wish to place this restriction only between "March 2009 and June 2009".

- applicability: The constraint applicability specifies when it was in effect. If the applicability is not specified, the default is assumed to be the lifetime of the document. The applicability can be a temporal element.

resume

1) *Supplier Ratings can move up or down a single step at a time (for example, from A to B, or B to A; but not from A to C) in valid time but no restrictions are placed in transaction time (since a data entry error might be made). This is applicable between 2008 and 2010.*

```
...
<transitionConstraint name="supplierRating"
     dimension="validTime">
   <selector xpath="supplier"/>
   <field xpath="supplierRatingType"/>
   <valuePair> <old>A</old> <new>B</new> </valuePair>
   <valuePair> <old>B</old> <new>A</new> </valuePair>
   <valuePair> <old>B</old> <new>C</new> </valuePair>
   <valuePair> <old>C</old> <new>B</new> </valuePair>
   <applicability begin="2008-01-01" end="2010-12-31"/>
</transitionConstraint>
...
```

2) *Employee Salaries should not go down, but may increase between 2008 and 2009. However, a salary freeze is in place between January and June 2009 due to economic factors.*

```
...
<transitionConstraint name="employeeSalary1"
     dimension="validTime">
   <selector xpath="employee"/>
   <field xpath="salary"/>
   <valueEvolution direction=">=" />
   <applicability begin="2008-01-01" end="2009-12-31"/>
</transitionConstraint>

<transitionConstraint name="employeeSalary2"
     dimension="validTime">
   <selector xpath="employee"/>
   <field xpath="salary"/>
   <valueEvolution direction="=" />
   <applicability begin="2009-01-01" end="2009-06-30"/>
</transitionConstraint>
...
```

11 Implementation Considerations

[TODO: Steve]

(TR: Section 10.6)

- An architectural overview of how to implement constraint validation (without actually discussing specific functions or algorithmic details)
- An explanation where schema constraint functionality should be placed

From TR To this point we have focused on how the user describes his temporal documents and their schemas. We now turn to examine where schema constraint functionality is placed and the issues that arise when validating temporal constraints. In this section we focus on the latter.

Before facing these issues, it is convenient to discuss the approach that the τXSchema tools take to validate temporal constraints. Figure 2 shows the overall architecture of the tools as they manage XML documents and their schemas. A sequence of non-temporal documents is input into SQUASH to create a temporal representation; this document can then be validated using τXMLLINT and SCHEMAMAPPER. UNSQUASH can be used to reconstruct the original non-temporal documents

Term	Definition	Cardinality
name	The name of the constraint	optional
dimension	Either validTime or transactionTime (default: validTime)	optional
selector	Role and definition is similar to the selector sub-element in the conventional XML Schema constraint definitions (e.g., for keyref constraints)	[1:1]
field	Similar to the field sub-element in the conventional XML Schema constraint definitions. Allowing for multiple field elements lets us constraint combinations of entities.	[1:U]
valuePair	Sub-element listing possible pairs for discrete changes	
old, new	Sub-elements of valuePair	
valueEvolution	Sub-element specifying direction of continuous changes	
applicability	When the constraint is applicable (default: lifetime of document)	optional

TABLE 4
Attributes and sub-elements for transitionConstraint

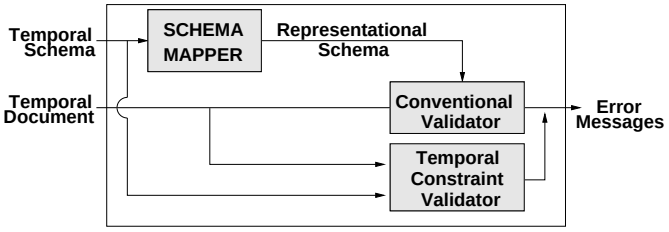

Fig. 7. Validating a document with Time-Varying Data: τXMLLINT.

from the temporal representation, while RESQUASH can be used to create a new representation (e.g., different timestamp locations) from a given representation.

Figure 7 provides the validation procedure used by τXMLLINT. The first step is to pass the temporal schema into τXMLLINT, which ensures that the logical and physical annotations are consistent with the conventional schema and with each other. Once the annotations are found to be consistent, SCHEMAMAPPER is invoked to generate a representational schema from the original conventional schema and the logical and physical annotations. The representational schema is then used as the schema for the temporal document and input into a conventional validator (in this case, XMLLINT). The next step is to pass the temporal document and the temporal schema to *Temporal Constraint Validator Module*. This step is to enforce temporal constraints that are not possible to be enforced by the representational schema alone.

A key design decision during the validation of temporal constraints is the placement of functionality: should a constraint be implemented in the representational schema or within the temporal constraint validator? Implementing (or *expressing*, or *enforcing*) constraints in the representational schema may provide faster validation since a conventional validator can be invoked directly, but may result in increased size and complexity of the representational schema. Conversely, implementing constraints in the temporal validator may yield small and compact schemas, but requires more time to perform the validation since the tools must perform checks across all slices sequentially and individually in the worst case.

In the following sections we explore two issues related to the placement of functionality. First, we determine which temporal constraints are possible to be expressed in a representational schema using the item-based representation class[3] and which can only be implemented in the temporal constraint validator. Second, for those constraints that can be implemented in both the schema and the temporal constraint validator, we provide a brief analysis of the tradeoffs between the two placements.

11.1 Constraints

In this section we discuss both *sequenced* (enforced at each point in time) and *non-sequenced* (enforced on the temporal document as a whole) constraints and determine for each whether it is possible to express that constraint in the representational schema. For both sequenced and non-sequenced constraints, we focus on the following classes of constraints.

- Identity constraints: These constraints restrict uniqueness of elements and attributes in a given document. Identity constraints are defined in the schema document using a combination of <selector> and <field> sub-elements within an <key> or <unique> element.
- Referential Integrity constraints: These constraints, defined using the <keyref> element, are similar to the corresponding constraints in the relational model. Each referential integrity constraint refers to a valid key or unique constraint and ensures that the corresponding key value exists in the document. For example, a <keyref> can be defined to ensure that only valid product numbers (i.e., those that exist for a <product> element) are entered for an order.
- Cardinality constraints: The cardinality of elements in XML documents is restricted by the use of minOccurs and maxOccurs in the XML Schema document. The cardinality of attributes is restricted using optional, required, or prohibited.
- Datatype : Datatype definitions can restrict the structure and content of elements, and the content of attributes. For example, a datatype definition can restrict the content of an element <age> to be between 0 and 100.

Table 5 provides a sneak-peak summary of which of the eight classes of constraints we claim can be implemented in the representational schema in the general case. We now provide an argument for each cell of this table in turn.

3. Section **??** introduces and describes the four kinds of representation classes and Section 11.4 outlines how the other three classes affect this analysis.

	Identity	Referential	Cardinality	Datatype
Sequenced	×	×	✓	✓
Non-sequenced	×	×	×	×

TABLE 5
The classes of constraints that can be implemented in a representational schema in the general case.

11.2 Sequenced Constraints

In this section we examine whether each class of sequenced constraints can be enforced by a representational schema. Given a conventional XML Schema constraint, we define the corresponding logical semantics in XML Schema in terms of a *sequenced constraint*. For example, a conventional (cardinality) constraint, "There should be between 0 and four 4 URLs for each supplier," has the following sequenced constraint: "There should be between 0 and 4 website URLs for each supplier *at every point in time*."

For each sequenced constraint below we use the following approach. If we claim that the constraint can be enforced by a representational schema, we outline a method that can be used by the τXSchema tools to transform the sequenced constraint syntax into standard XML Schema syntax. If, on the other hand, we claim that the constraint cannot be enforced by a representational schema, we provide a counter example that illustrates the specific shortcoming of XML Schema that forbids the constraint to be enforced.

11.2.0.1 Identity Constraints: We claim that identity constraints of elements and attributes *cannot* be enforced in a representational schema in the general case. To see this, consider the following example. In this example, we require `<zip>` elements to have unique `code` attributes via an identity constraint named `zipUnique`.

```
...
   <xs:unique name="zipUnique">
     <xs:selector xpath="zip"/>
     <xs:field xpath="@code"/>
   </xs:unique>
...
```

Listing 15. XML Schema `<unique>`.

```
...
  <zip code="85721"> Tucson,  AZ </zip>
  <zip code="85001"> Phoenix, AZ </zip>
...
```

Listing 16. Unique `code`s (slice 1).

Now suppose the user were to change the `code` for Tucson to be the same as Phoenix (violating the conventional schema's identity constraint), and then back again.

```
...
  <zip code="85001"> Tucson,  AZ </zip>
  <zip code="85001"> Phoenix, AZ </zip>
...
```

Listing 17. Slice 2 (invalid).

```
...
  <zip code="85721"> Tucson,  AZ </zip>
  <zip code="85001"> Phoenix, AZ </zip>
...
```

Listing 18. Slice 3 (valid).

Assuming that the physical annotations place the timestamps at the `<zip>` element level, the above actions would create an item-based representation similar to the one shown in Listing 19.

```
...
  <zip_RepItem>
    <zip_Version begin="1" end="1">
      <zip code="85721"> Tucson, AZ </zip>
    </zip_Version>
    <zip_Version begin="2" end="2">
      <zip code="85001"> Tucson, AZ </zip>
    </zip_Version>
    <zip_Version begin="3">
      <zip code="85721"> Tucson, AZ </zip>
    </zip_Version>
  </zip_RepItem>

  <zip_RepItem>
    <zip_Version begin="1">
      <zip code="85001"> Phoenix, AZ </zip>
    </zip_Version>
  </zip_RepItem>
 ...
```

Listing 19. Squashed version of the three slices.

With this representation, there is no way to create an identity constraint in XML Schema that can detect that both `code` values at time 2 are the same. If the constraint were constructed to restrict all `./zip_RepItem/zip_Version/zip/@code` values[4] to be unique, this would fail since at times 1 and 3, Tucson has a `code` value of 85701, and this is legal in our temporal constraint. If the constraint were constructed to require all `./zip_Version/zip/@code` within each `./zip_RepItem` to be identical, this would also fail since the user is allowed to change the zip code from slice to slice.

One could imagine extending the constraint shown in Listing 15 to include key specification fields `begin` and `end` for the valid times associated with each version of the `zip` element, as shown in Listing 20 on lines 29 and 30, with the corresponding attributes in the element specification.

```
...
   <xs:unique name="zipUniqueAttempt">
     <xs:selector xpath="zip"/>
     <xs:field xpath="@code"/>
     <xs:field xpath="@begin"/>
     <xs:field xpath="@end"/>
   </xs:unique>
...
```

Listing 20. XML Schema `<unique>` with additional fields.

As long as the `begin` and `end` attributes are maintained in proper order (which can be checked by τXMLLINT), the keys will uniquely identify each key within each snapshot. Listing 21 below shows an example where the addition of

4. This is XPath code [**?**].

such attributes will achieve the desired functionality. Here, the conventional validator would detect that the `zip` elements on lines 95 and 104 are in violation of the unique constraint, which is indeed correct.

```
...
  <zip_RepItem>
    <zip_Version begin="1" end="1">
      <zip code="85721" begin="1" end="1"> Tucson, AZ </zip>
    </zip_Version>
    <zip_Version begin="2" end="2">
      <zip code="85001" begin="2" end="2"> Tucson, AZ </zip>
    </zip_Version>
  </zip_RepItem>

  <zip_RepItem>
    <zip_Version begin="1" end="1">
      <zip code="85001" begin="1" end="1"> Phoenix, AZ </zip>
    </zip_Version>
    <zip_Version begin="2" end="2">
      <zip code="85001" begin="2" end="2"> Phoenix, AZ </zip>
    </zip_Version>
  </zip_RepItem>
 ...
```

Listing 21. Squashed document with multiple changes

However, this approach will not succeed in the general case because it only enforces uniqueness at the interval end points and not anywhere within the interval. For example, consider the excerpt from a squashed document shown in Listing 22. We see that the elements on lines 95 and 101 conflict our desired constraint, but since the `begin` attributes are distinct, XML does not detect an error.

```
...
  <zip_RepItem>
    <zip_Version begin="1" end="1">
      <zip code="85721" begin="1" end="1"> Tucson, AZ </zip>
    </zip_Version>
    <zip_Version begin="2" end="2">
      <zip code="85001" begin="2" end="2"> Tucson, AZ </zip>
    </zip_Version>
  </zip_RepItem>

  <zip_RepItem>
    <zip_Version begin="1" end="2">
      <zip code="85001" begin="1" end="2"> Phoenix, AZ </zip>
    </zip_Version>
  </zip_RepItem>
 ...
```

Listing 22. Squashed document with multiple changes

We are thus forced to conclude that XML Schema lacks sufficient capability to discriminate time boundaries in a way that would allow sequenced identity constraints to be enforced.

11.2.0.2 Referential Integrity Constraints: We claim that referential integrity constraints *cannot* be implemented in a representational schema. The argument is similar to that for identity constraints: there is no way to create a constraint in XML Schema that can both satisfy referential integrity and time issues. Consider the example shown below.

```
...
<!-- Defines a key named "pNumKey" -->
<key name="pNumKey">
  <selector xpath="states/state"/>
  <field xpath="@id"/>
</key>

<!-- Says that the "state" attribute -->
<!-- in <zip><city></zip> elements   -->
<!-- must match a pNumKey.           -->
<keyref name="stateMatcher"
        refer="r:pNumKey">
  <selector xpath="regions/zip/city"/>
  <field xpath="@state"/>
</keyref>
...
```

Listing 23. A referential constraint.

```
...
<regions_RepItem>
  <regions_Version begin="1" end="1">
    <regions>
      <zip code="85701">
        <city state="1"/>
      </zip>
    </regions>
  </regions_Version>
  <regions_Version begin="2" end="3">
    <regions>
      <zip code="85701">
        <city state="6"/>
      </zip>
    </regions>
  </regions_Version>
</regions_RepItem>

<states_RepItem>
  <states_Version begin="1" end="2">
    <states>
      <state id="1">Arizona</state>
      <state id="2">California</state>
  </states>
 </states_Version>
 <states_Version begin="3" end="3">
   <states>
     <state id="6">Arizona</state>
     <state id="2">California</state>
   </states>
 </states_Version>
</states_RepItem>
  ...
```

Listing 24. Squashed document.

Here, we have a constraint that says *"A city element's state attribute must match an existing state element's id attribute at every point in time."* The squashed document shows that this constraint is satisfied at times 1 and 3, but violated at time 2 since Arizona will point to a non-existent state `id`. To construct an XML Schema that could describe this situation, one would need to be able to somehow discriminate between different `<regions_Version>` elements according to their `begin` and `end` attributes, but there is no such way to accomplish this without the help from a procedural language like XQuery [**?**]. We are again forced to conclude the XML schema lacks sufficient mechanisms to enforce a referential constraint.

11.2.0.3 Cardinality Constraints: We claim that the cardinality of both elements and attributes *can* be enforced in the representational schema. Consider an element e which has created a logical item i. If the lowest timestamp is located at a ancestor or descendent of e, then no change to the definition of e from the original schema is necessary, only a direct copy into the representational schema. If a timestamp is located at i, then the cardinality constraint information must be moved from e up to i in the representational schema. Since there must be one item for each original element, ensuring that we have a particular number of items is the same as ensuring that we have a particular number of original elements.

Listings 25 and 26 below show an example constraint: *"The element* `<supplier>` *can occur exactly 1 or 2 times."* We first assume that the physical timestamps—specified in the temporal schema—are placed at a predecessor or successor element

of the <supplier> element. In this case the specification of the <supplier> element requires no modification in the representational schema.

```
...
  <xs:element name="supplier"
          minOccurs="1" maxOccurs="2">
    ...
  </xs:element>
...
```

Listing 25. Conventional schema 1.

```
...
  <xs:element name="supplier"
          minOccurs="1" maxOccurs="2">
    ...
  </xs:element>
...
```

Listing 26. Representational schema 1.

Listings 27 and 28 show the same example as above, except now the physical timestamps are located at the level of the <supplier> element. In this case, the transformation pushes the constraints up to the <supplier_RepItem> element.

```
...
  <xs:element name="supplier"
          minOccurs="1" maxOccurs="2">
    ...
  </xs:element>
...
```

Listing 27. Conventional schema 2.

```
...
  <xs:element name="supplier_RepItem"
          minOccurs="1" maxOccurs="2">
    ...
    <xs:element name="supplier_Version">
      ...
      <xs:element name="supplier">
      ...
      </xs:element>
    </xs:element>
  </xs:element>
...
```

Listing 28. Representational schema 2.

11.2.0.4 Datatype Constraints: We claim that datatype definitions of both elements and attributes *can* be enforced in the representational schema. This can be achieved by copying the datatype definition for each element in the original schema into the representational schema. Since datatype restrictions are not affected by the location of timestamps, the transformation is trivial in all cases. See Listings 29 and 30 for an example of the datatype constraint: *"The element* <age> *must have a value between 0 and 100, inclusive, at all times."* No changes to the constraint must be made in the transformation.

```
...
  <xs:element name="age">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0"/>
        <xs:maxInclusive value="100"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
...
```

Listing 29. Datatype conventional schema.

```
...
  <xs:element name="age">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0"/>
        <xs:maxInclusive value="100"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
...
```

Listing 30. Datatype rep. schema.

11.3 Non-sequenced Constraints

Non-sequenced constraints are constraints applied to a temporal element as a whole (including the lifetime of the data entity) rather than individual time slices. Non-sequenced constraints are not defined on conventional XML Schema equivalents. An example of a non-sequenced (cardinality) constraint is: "There should be no more than 10 URLs for each supplier *in any year*."

We claim that in general it is *not* possible to enforce non-sequenced constraints within a representational schema. Since non-sequenced constraints can reference arbitrary sections of time that don't necessarily correspond to slice lifetimes or schema change (*schema wall*) boundaries, it is impossible to use XML Schema to isolate and thus validate these sections. For example, consider the simple non-sequenced cardinality constraint: *"There should be two or three unique suppliers in any given year."* If the document were changed at intervals that were less than one year in duration, we could have a representation that looked similar to Listing 31.

```
...
<suppliers_RepItem>
  <suppliers_Version begin="1" end="1">
    <supplier id="1">IBM</supplier>
    <supplier id="2">HP</supplier>
  </suppliers_Version>
  <suppliers_Version begin="2" end="100">
    <supplier id="1">IBM</supplier>
    <supplier id="3">Sun</supplier>
  </suppliers_Version>
    <suppliers_Version begin="100" end="600">
    <supplier id="3">Sun</supplier>
    <supplier id="4">Apple</supplier>
  </suppliers_Version>
</suppliers_RepItem>
  ...
```

Listing 31. Squashed version. One day equals one unit of time.

It is easy to see that there are in fact four suppliers between the times 1 and 365, violating our example contraint. However, there is no way to construct an XML Schema to successfully validate this, since we would need some way to accumulate the number of unique <supplier>s across <supplier_Version> and then check this number against the constraint; but there is no such way to perform this accumulation in XML Schema.

However, we do note that there exist specific circumstances in which non-sequenced constraints *may* be validated. Again consider the non-sequenced cardinality constraint: *"There should be 2 or 3 unique suppliers in any given year."* Also suppose that the timestamps were placed at some element above the <supplier> element and that slices were created

exactly once per year. The result will be a representation that closely mimics the individual slices. We see that it is possible to create a representational schema to enforce this constraint.

```
...
  <company_RepItem>
    <company_Version begin="1" end="2">
      <company>
        <suppliers>
          <supplier id="123"/>
          <supplier id="456"/>
        </suppliers>
      </company>
    </company_Version>
    ...
...
```

Listing 32. Item-based temporal representation #1.

```
...
  <xs:element name="company_RepItem"
    ...
    <xs:element name="company_Version">
      ...
      <xs:element name="supplier">
          minOccurs="2" maxOccurs="3">
      ...
      </xs:element>
    </xs:element>
  </xs:element>
...
```

Listing 33. Non-sequenced representational schema #1.

In this case, we are guaranteed to have one `<suppliers>` element per year. Thus, validating each element in each company version will validate the constraint.

As another example, consider the non-sequenced cardinality constraint: *"There should be between 2 and 4 players on the team in any given year."* If the slices happen to have a one-to-one correspondence with the boundaries for a year, and the timestamp happens to be at or above the `<team>` element, then we could have the following representational schema.

```
...
<team_RepItem>
  <team_Version begin="1" end="1">
    <team>
      <player>Steve</player>
      <player>Bob</player>
      <player>Mark</player>
      <player>Paul</player>
    </team>
  </team_Version>
  <team_Version begin="2" end="2">
    <team>
      <player>Steve</player>
    </team>
  </team_Version>
</team_RepItem>
  ...
```

Listing 34. Item-based temporal representation #2.

```
...
  <xs:element name="player"
          minOccurs="2" maxOccurs="4">
    ...
  </xs:element>
...
```

Listing 35. Non-sequenced representational schema #2.

In general, we see that such special cases can be constructed when both of the following conditions are met.

- Placing the physical timestamp at or above the highest element that is involved in the constraint.
- Versioning the conventional document so that the lifetime of each slice matches the time unit specified by the constraint (e.g., if the constraint involves one year, then there would be exactly one slice per year).

Clearly, these situations are of limited practical use since they are constricting and unlikely to occur naturally. Nevertheless, one might argue that the tools could simply adopt the following strategy. "If a special case occurs, place the functionality in the representational schema; otherwise, place the functionality in the tools." We argue that this process would add complexity that is not justified by the marginal performance gains, especially when there are multiple constraints defined and only some would meet the special-case criteria.

11.4 Functionality of Other Representation Classes

In the above sections we considered whether constraints could be expressed in an XML schema using the item-based representation class. We now provide a brief commentary on the ability of each of the remaining three representation classes to express constraints. Briefly, the remaining three representation classes provide the same or worse level of capability as the item-based class.

The slice-based class allows the same set of constraints to be expressed as the item-based class. This is because the slice-based class is a special case of the item-based class; it possesses no unique characteristics and thus the same limitations apply. The reference-based class also allows the same set to be expressed. This can be seen by viewing the reference-based class as an optimized, but similar version of the item-based class. The reference-based class has the same structure as the item-based class (e.g., items, versions, physical timestamps); the only difference is that the reference-based class avoids data duplication by providing multiple references to subtrees that occur more than once. This process does not gain the reference-based class any benefits that can be used to enforce constraints. The edit-based class is not able to express any temporal constraints within the representational schema since it reduces changes to the XML tree to simple text content that cannot reliably be parsed and examined.

11.5 Placement of Functionality

For those constraints that can be implemented within either the representational schema or the tools (i.e., sequenced cardinality and datatype constraints), the question remains: where should the functionality be placed? To address this question, we provide a discussion below.

Consider the model of validation used by τXMLLINT shown in Figure 7. First, the temporal document is validated against the representational schema using a conventional validator (i.e., XMLLINT). Then the *Temporal Constraint Validator Module* is invoked to explicitly and exhaustively check all temporal constraints. This module uses DOM to parse and traverse each slice and manually checks each constraint present in the logical annotation set. From the description of these steps we draw two simple observations. First, the conventional validator is always invoked on the temporal document, no matter which constraints are being implemented

in the representational schema. Second, temporal constraints which are "hard" to implement are done so using DOM. Thus, since the conventional validator is empirically much faster than DOM, and is being invoked anyway, we argue that all constraints, when possible, should be implemented within the representational schema. This will provide much better performance in terms of time required, and as we have shown in the previous sections, will not greatly increase the complexity of the representational schema. Furthermore, SCHEMAMAPPER will not require extensive modifications in order to create a schema that can enforce these constraints, since the transformation is trivial in most cases and relatively simple in the rest.

For these reasons, we conclude that the functionality of sequenced cardinality and datatype constraints be placed within the representational schema and not within the temporal constraint validator.

12 CONCLUSION AND FUTURE WORK

[TODO: Rick]

did not copy paste from TR; overlap seemed minimal

ACKNOWLEDGEMENTS

We thank Lingeshwaran Palaniappan for the development of the initial version of the logical to representational mapper and the temporal data validator. Eric Roeder took over from Lingsesh; he was an integral part of the design of the time-varying document support and implemented a refined version of the mapper and validator, as well as first implementations of squash and unsquash. Shailesh Joshi wrote a superb MS Thesis [**?**] describing further refined tools to support a refined language design. NSF grants IIS-0100436, IIS-0415101, IIS-0515101, IIS-0639106, IIS-0803229, and EIA-0080123 and grants from the Boeing Corporation and Microsoft provided partial support for this work.