Adding Temporal Constraints to XML Schema

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

Abstract—If past versions of XML documents are retained, what of the various integrity constraints defined in XML Schema on those documents? This paper describes how to interpret such constraints as *sequenced* constraints, applicable at each point in time. We also consider how to add new variants that apply *across* time, so-called *non-sequenced* constraints. Our approach supports temporal documents that vary over both valid and transaction time, whose schema can vary over transaction time. We do this by replacing the schema with a (possibly time-varying) temporal schema and replacing the document with a temporal document, both of which are upward compatible with conventional XML and with conventional tools like XMLLINT, which we have extended to support the temporal constraints introduced here.

Index Terms—cardinality constraint, key constraint, referential integrity, temporal data, XML validation, XML Schema constraint.

1 INTRODUCTION

As with prose documents, spreadsheets, presentations, and data in a database, XML documents also are changed over time. Also, as with these other kinds of documents and as with data in a database, users often would like to retain past versions of XML documents, for several reasons. One, those past versions may contain useful historical information. Secondly, various laws such as the Sarbanes-Oxley Act [1] require that for data that appear in financial reports drawn from prior versions, that those versions be retained for a stated period of time. Third, retaining past versions allows previously-written reports using that data to remain consistent, even if new versions are subsequently added. With XML becoming more prevalent as both a transmission encoding and a document encoding format, it thus becomes important to retain prior versions of an XML document. And indeed, a rich literature on this subject has emerged [2].

Given the existence of such prior versions, one then can ask, what of the various integrity constraints defined on that document? How can such constraints be generalized to apply not just to the current version, but across the entire history of the XML document? And how can new, explicitly temporal constraints be defined? Finally, how can all this be managed effectively over schema changes, which are a fact of life in complex enterprises?

As a motivating example, consider a simple scenario in which a user specifies a conventional schema (Listing 1). The root of this schema is the `<company>` entity. Under that, there are `<emps>`, `<products>` and `<suppliers>`. The `<emp>` element has the sub-elements `<name>` and `<SSN>`, and attributes `ID` and `email`. An `<order>` is a sub-element of `<supplier>`. Note that the schema includes cardinality constraints (e.g., `<minOccurs>`, `<maxOccurs>`), a uniqueness constraint (`<unique>`), and a referential integrity constraint, linking an `<order>` product number to a `<product>` element.

The user creates an initial XML document conforming to the schema (Listing 2) on 2010-01-01. Together, these documents form a conventional system which can be validated with conventional validation tools (e.g., XMLLINT [3]).

So far, the extensive infrastructure around XML applies. The user has defined a schema and a document, and has validated that document against the schema, and all is right in the world.

On 2010-03-17, the user corrects the `email` attribute in the conventional document to produce a new version stored in a new file (Listing 3). Subsequently, on 2010-10-01, a change in email formats leads to another change in the email (Listing 4). The user can validate these documents against the schema. In particular, it is reasonable to assume that the user intends the constraints specified in the schema to apply at each point in time, i.e., `data.xml`, `data.2.xml`, and `data.3.xml` must independently satisfy the stated integrity constraints.

We note a couple of difficulties that now arise. First, the user must manually keep track of the relationships between the versions of the document. Nowhere does it say explicitly that `data.2.xml` is in any way related to document `data.xml`. Second, we have to now rely on the underlying file system to keep track of the dates. If we copy `data.xml` to a new directory, that date will be lost. Third, while we can validate each version separately against `company.xsd`, there is no way in conventional XML Schema to express constraints *across* multiple versions. As one example we will return to later, we cannot state that a product number should never be reused later with a different product. Finally, if the *schema* is also time-varying, that is, if there are multiple versions of `company.xsd`, our job of maintaining the integrity of the document becomes even more challenging.

- *F. A. Currim is with the Department of MIS, University of Arizona, Tucson, AZ, 85721. E-mail:* `currim@email.arizona.edu`
- *S. A. Currim is with UITS, University of Arizona.*
- *C. E. Dyreson is with the Department of Computer Science, Utah State University.*
- *R. T. Snodgrass and R. Zhang are with the Department of Computer Science, University of Arizona.*
- *S. W. Thomas is with the School of Computing, Queen's University, Canada.*

Our design of an upward-compatible extension of XML Schema, τXSchema [4] addresses the first two concerns emphasized in the previous paragraph. τXSchemasupports temporal documents that vary over both valid and transaction time [5], [6], [7], whose schema can vary over transaction time [8], and for which validation is a simple process (to the user) of checking a time-varying document over a schema, which itself is a time-varying document [9], [10]. Related work has formalized language primitives required for managing schema versioning with τXSchema [11].

The challenge addressed by the present paper is how to accommodate both conventional XML integrity constraints, including the identity, referential, cardinality, and datatype constraints illustrated in Listing 1, as well as new *temporal* constraints, across such time-varying schema and data documents. (This schema is very simple, but is sufficient for illustrating both how conventional constraints are applied to time-varying documents and how new temporal constraints can be usefully defined.)

```
<?xml version="1.0"?>
<xs:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://txschema.com"
  xmlns="http://txschema.com" elementFormDefault="qualified">
 <xs:element name="company">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="emps"/>
    <xs:element ref="suppliers"/>
    <xs:element ref="products"/>     ...
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="emps">
  <xs:complexType>
   <xs:element name="emp" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="SSN" type="xs:string" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="ID" type="xs:string"/>
     <xs:attribute name="email" type="xs:string"
       use="optional"/>
    </xs:complexType>
   </xs:element>
  </xs:complexType>
  <xs:key name="employeeIDKey">
   <xs:selector xpath="emp"/>  <xs:field xpath="@ID"/>
  </xs:key>
  <xs:unique name="empEmailUnique">
   <xs:selector xpath="emp"/>  <xs:field xpath="@email"/>
  </xs:unique>       ...
 </xs:element>      ...
 <xs:element name="suppliers">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="supplier" minOccurs="1"
              maxOccurs="unbounded">
     <xs:element name="URL" type="xs:anyURI"
       minOccurs="0" maxOccurs="4"/>
     <xs:element name="order" minOccurs="0"
       maxOccurs="unbounded">   ...
      <xs:keyref name="orderProductRI" refer="productKey">
       <xs:selector xpath="order"/>
       <xs:field xpath="ordProductNo"/>
      </xs:keyref>     ...
     </xs:element>    ...
    </xs:element>    ...
 </xs:element>      ...
</xs:schema>
```

Listing 1. `company.xsd`

After examining related work briefly, we give a quick overview of the goals of τXSchema and outline its approach in Section 3. In short, a single *temporal document* (with timestamps at various locations specified by the user) replaces an entire sequence of versions and a single *temporal schema* replaces a sequence of versions of conventional schemas. Section 4 summarizes the syntax and semantics of those constraints that can be defined within conventional XML Schema, while Section 5 provides the necessary background to understanding their temporal extensions. Section 6 provides the core contribution of this paper: a detailed examination of how each kind of constraint in turn can be supported and extended to apply to time-varying data. We then examine the implications of schema versioning (including changing the constraints themselves!) and the expressiveness of τXSchema. We end with implementation details and an evaluation of our approach (Section 9).

```
<?xml version="1.0" encoding="UTF-8"?>
<company xmlns="http://txschema.com">
 <emps>
  <emp ID="1" email="diana@txschema.com">
   <name>Dana</name>
  </emp>
 </emps>   ...
</company>
```

Listing 2. `data.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<company xmlns="http://txschema.com">
 <emps>
  <emp ID="1" email="dana@txschema.com">
   <name>Dana</name>
  </emp>
 </emps>   ...
</company>
```

Listing 3. `data.2.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<company xmlns="http://txschema.com">
 <emps>
  <emp ID="1" email="ddoe@txschema.com">
   <name>Dana</name>
  </emp>
 </emps>   ...
</company>
```

Listing 4. `data.3.xml`

2 Related Work

Capturing the time-varying nature of web-resident data has been actively researched over the last few years. This area of research has covered a wide range of issues that include architectures to represent changes [12] and collect document versions [13], strategies for storing versions [14], and strategies to retrieve temporal data that is stored as XML [12], [15], [16]. However, enforcing temporal constraints in XML has not been researched previously.

We focus on effectively validating a document while enforcing temporal constraints. Within a document, one may specify a variety of constraints. At the schema level, we want to specify which parts can vary with time and consider how schema changes impact our ability to capture time and validate the document. On the instance level, we want to constrain how the parts vary, which requires new variants of uniqueness, referential integrity, cardinality and datatype constraints.

Most of the topics discussed in this paper have been previously considered in the context of temporal *relational* databases [17], [18], [19]. For example, Chomicki has done extensive work in formalizing temporal constraints using first order logic and applying it to databases [17], [20], [21]. Schema versioning has also been researched in the context of temporal databases [22], [23]. Unlike a relational database schema, an XML schema is a grammar specification so new techniques are required.

Prior work in conceptual modeling for temporal databases has considered extensions to identity [24] and cardinality [25] constraints. Also in the area of conceptual modeling grammars, description logics have been proposed to represent and reason about a variety of temporal constraints [26]. While there are some parallels between conceptual modeling grammars (e.g., ER or UML) and XML Schema, constraint definitions for conceptual grammars naturally focus on constructs such as *entity classes*, *attributes* and *relationships*. Thus, a distinct set of semantics and syntax is required to handle temporal constraints for XML Schema.

Although various XML schema languages have been proposed in the literature and in the commercial arena, none of the approaches provide a systematic approach to encoding time-varying data in XML across schema changes nor to expressing and enforcing integrity constraints over such data. This is where our research makes its contribution.

3 Language Design

We first summarize briefly the design of τXSchema. We start with some relevant terminology.

– *Conventional Document*: An XML document that has no temporal aspects.
– *Temporal Document*: An XML document that represents a sequence of conventional documents (i.e., slices). It has the root element `<temporalRoot>`.
– *Conventional Schema*: An XML Schema document that describes the structure of the conventional document(s). The root element is `<schema>`.
– *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 times t_1 and t_2, respectively, then the slice at time t_2 is d_2.

In augmenting XML Schema to accommodate time-varying data, we had several goals in mind. At a minimum, we desired that our approach exhibit the following benefits.

– Simplify the representation of time for the user.
– Support a three-level architecture to provide data independence, so that changes in the logical and physical level are isolated.
– Retain full upward compatibly with existing standards and not require any changes to these standards.
– Augment existing tools such as validating parsers for XML in such a way that those tools are also upward compatible. Ideally, any off-the-shelf validating parser (for XML Schema) can be used for (partial) validation.
– Support both valid time and transaction time at a logical level; each dimension is treated orthogonally.
– Support instance versioning.
– Support schema versioning. Different versions of a document may conform to different versions of a schema, as both a document and schema are modified over time. Support for schema versioning will ensure that the schema's history can be kept and correctly utilized.

The interaction between the temporal schema and its constituent conventional schemas and related tools is depicted in Fig. 1. 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 now continue the motivating example given at the beginning. We have shown how a conventional document recording information about a company is edited over time, creating a sequence of conventional documents. Each conventional document is intended to conform to a conventional schema.

We start with a conventional schema (Listing 1, box 3 in the figure) and three documents, the original (Listing 2) and two subsequent versions (Listings 3 and 4, identified in the figure as "Conventional XML Data", box 7). These numerous files give us a hint at the complexities that arise as the versions mount and as the schema changes as well (note that there may even be multiple versions of the base schema).

To more easily manipulate these many versions, the user would like to define a "Temporal Schema" (box 4) with the base schema as a component. The two other components are "Logical Annotations" (box 5) and "Physical Annotations" (box 6). 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. Most relevant for our purposes are temporal constraints, which can be inferred from the constraints in the base schema or which are explicitly specified as logical annotations. We'll get into the means of specifying such annotations in Section 6.

Physical annotations specify the timestamp representation options chosen by the user. These annotations 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 the location of the physical timestamp is changed.

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. The physical annotations also provide a user the means to specify temporal granularity, the resolution level at which each timestamp is maintained.

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.

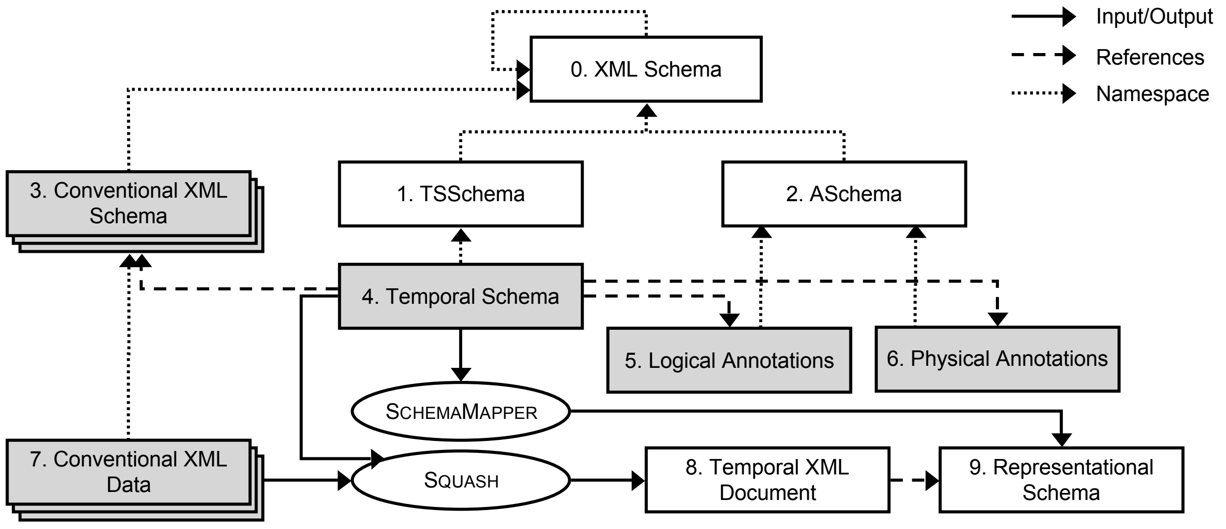


Fig. 1. Overall Architecture of τXSchema

The figure shows a tool called SQUASH that can render a temporal document (box 8) consistent with the logical and physical annotations. Hence, the timestamps are spread out across the document, associated with versions of the elements. This removes a great deal of redundancy found in the non-temporal data, which represents each slice as a separate document. The versions of the temporal document are described with a "Representational Schema" (box 9), generated automatically from the temporal schema by another tool called SCHEMAMAPPER. This 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.

Recall that the base schema (Listing 1) includes cardinality constraints, a uniqueness constraint, and a referential integrity constraint. As noted in Section 1, these constraints apply at each point in time within the temporal document.

Further, the user may wish to specify additional restrictions that guarantee uniqueness of an email *across conventional documents* (for example, that the address `dana@txschema.com` is not re-used by another employee to avoid confusion or problems re-directing emails after the second change). Using XML Schema alone, we cannot specify nor validate such constraints.

Instead, the designer can utilize τXSchema to augment the conventional schema with additional logical annotations, as we will illustrate with examples shortly, thus forming a more expressive temporal schema. As we'll discuss further in Section 7, the schema may be a time-varying document as well, and may even reference other time-varying schemas.

When we had one conventional schema (Listing 1) and one conventional (non-time-varying) document (Listing 2), we could use a tool such as XMLLINT to validate this document against its schema. We now have a similar, though much more flexible situation: a single document and a single schema (being upward compatible, Listing 1 is perfectly adequate). τXMLLINT is a tool we developed as the temporal counterpart to XMLLINT; see Fig. 2. XMLLINT takes as input a conventional document (slice at time *t*) referencing a conventional schema and reports if it is valid. Analogously, τXMLLINT takes as input a single temporal document referencing a temporal schema. τXMLLINT validates the temporal document and reports either success or the errors encountered.

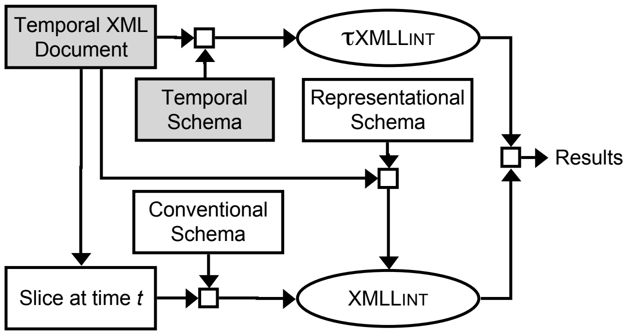


Fig. 2. Using τXMLLINT.

The validation using τXMLLINT is related to that of XMLLINT as follows: if a slice of a temporal document at time t is validated using XMLLINT and results in an error, then the validation of the temporal document using τXMLLINT should also report an error at time t.

With this high-level overview of τXSchema (details are available elsewhere [4], [8], [10]), we can now turn to the challenge at hand: supporting existing conventional and novel temporal constraints concerning a time-varying document. We first examine the constraints that XML Schema provides, and then apply and extend them for temporal documents.

4 XML Schema Constraints

XML Schema provides four types of constraints, namely datatype, cardinality, identity, and referential integrity constraints. These are conventional constraints and restrict a specific XML document. In this paper we extend these constraints in turn with temporal semantics.

Datatype constraints restrict the content of the corresponding element or attribute. A datatype restriction by itself applies fully in the temporal context. For example, the fact that the `name` attribute is a string (XML Schema type `xs:string`) applies equally in the static and temporal context (assuming no schema versioning). The content of the `name` attribute may change, and we consider in Section 6.4 some restrictions on what kinds of changes are permitted.

The *cardinality* of elements in XML documents is restricted by the use of `minOccurs` and `maxOccurs` in the XML Schema document. The default for both `minOccurs` and `maxOccurs` is 1. In the example in Listing 1, while there can be multiple `<emp>` sub-elements within `<emps>`, there can be a maximum of one `<SSN>` per `<emp>`, and there is always at most a single value for each attribute (for example, `ID`). Cardinality for attributes is therefore restricted in `use` to `"optional"` or `"required"`.

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 is that the `key` constraint does not allow a null value in any of the component fields, while missing (null) values do not lead to a violation of the `unique` constraint.

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 `<key>` or `<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., `<emp>` in Listing 1), relative to which the (combination of) `<field>` values is unique (e.g., `@ID`). An identity constraint may be named, and this name can then be used when defining a referential integrity constraint.

Note that the attributes of type `ID` (`IDREF`) are a special case of the `<key>` (`<keyref>`) constraints in XML Schema. In this paper we address the general case. Further discussion on the design choice of only addressing temporal semantics for `<key>` (`<keyref>`) is available in prior work [4].

Referential integrity constraints (defined using `<keyref>`) 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, the `<keyref>` in Listing 1 ensures that only valid product numbers (i.e., those that exist for a `<product>`) are entered for an order.

5 MOVING TOWARDS TIME

Before considering how to adapt the XML Schema constraints we just summarized to be used in time-varying XML documents, we first introduce an orthogonal classification of three flavors of temporal constraints and introduce the concept of a time-varying *item*.

5.1 Three Classes of Semantics

An important concept is the distinction between three orthogonal classes of semantics: *sequenced*, *non-sequenced*, and *current* [27]. All combinations are appropriate and useful. One could contemplate for example a sequenced cardinality constraint or a non-sequenced referential integrity constraint.

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. As discussed earlier, given a conventional XML Schema constraint, the corresponding semantics in τXSchema for a temporal document implies a *sequenced* constraint. For example, a conventional (cardinality) constraint, “There should be between 0 and 4 URLs for each supplier” (Listing 1), has a sequenced equivalent of: “There should be between 0 and 4 URLs for each supplier *at each point in time*.”

For convenience, we also allow the user to add a new sequenced constraint in the logical annotations. Such logical annotations can include an *applicability bound*, $B \subseteq T$, enabling the user to restrict the consideration of that sequenced constraint 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; it would not apply outside of that time period.

A special kind of sequenced constraint is a *current* constraint. A current constraint is applicable (and evaluated) at the current point in time, or *now* [28]. We support current constraints by allowing the user to set the applicability bound of the sequenced constraint to *now*.

A *non-sequenced constraint* is evaluated over some part (or the whole) of the applicability bound rather than at each point in time separately. For such constraints, we include an *evaluation window*, w, which is a time interval (e.g., a day, or a Gregorian month) as well as a *slide size*, ss, and an *applicability bound*, B [29]. The default length for ss is a single granule interval. The default for B is the lifetime of the temporal document. The following relationship must hold among the components of a non-sequenced constraint: $ss \leq w$. When $duration(B)$ is the same size as w, we term it a “fixed-window” constraint (analogously, when both ss and w are a single granule of time, we have a sequenced constraint). Non-sequenced constraints are included in the logical annotations.

For example, suppose the constraint requires “there are between 0 and 4 supplier URLs in the temporal document over a period of any calendar month.” (This is a temporal variant of the cardinality constraint on `<URL>` in Listing 1.) Let’s say this constraint is applicable from 2010-03-01 to 2010-03-31. Here, w and B have the same duration. If instead the applicability were 2010-03-01 to 2010-06-31, then we see a case of a “sliding-window” constraint, as 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*. If 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. In such a case, the size of the slide (ss) is a single day.

5.2 Temporal Data Model

An XML document is usually modeled as a labeled tree. Few additional modeling components are needed in a temporal XML model to capture time. A temporal XML document can be modeled as a timestamped set of XML documents. For simplicity, we discuss a data model with only one time dimension.

Definition [Temporal XML Model] A temporal XML model is a tuple, (X, T, S, A), where

– $X = \{X_1, \ldots, X_n\}$ is a set of *XML data model instances*, where an instance $X_i = (V_i, E_i)$ has a set of nodes V_i (with each node being an element or an attribute) and a set of edges E_i (with each edge being between an element and an attribute or an element and its child element),
– T is a set of *times*,
– $S : X \rightarrow 2^T$ is a *timestamp function* that maps an XML data model instance to a timestamp (a set of times) for which it is current in the time dimension, and
– $A : V \rightarrow V$ is a *temporal association relation* that associates a node in some XML data model instance to a node in some other XML data model instance (as described in Section 5.3). The relation captures a node's identity over time across instances. ■

The **slice** function extracts a slice (an XML data model instance) from a temporal XML document.

Definition [Slice] Let $D = (X, T, S, A)$ be an instance of a temporal XML model. Then for $t \in T$, $\textbf{slice}(t, D) = X_i$, where $X_i \in X$ and $t \in S(X_i)$. ■

Though this model is simple, it is sufficient for the purposes of this paper and its simplicity makes clear that existing XPath, XQuery, and XML Schema constructs can be natively evaluated for any XML data model instance in a temporal XML data model. (Note that we are not proposing to store or represent a temporal XML document using the model, rather we use this model to formalize the semantics of temporal constraints, specifically, in the *Eval* function to be introduced shortly.)

5.3 Items

In order to validate non-sequenced constraints, it is important to identify which elements persist across various transformations of the document. This will allow us, for example in the case of a non-sequenced identity constraint, to verify whether an email address is being repeated for the same employee, or for a different one. (Items are not relevant for sequenced nor current constraints.) This section discusses how to find and associate elements in different slices of a temporal 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 relational 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 [30], [31]. In a similar manner, elements in two slices of a temporal document can be *temporally-associated*. A temporal association between the elements is possible when the element has the same *item identifier* in both slices. We will sometimes refer to the process of associating a pair of elements as *gluing* the elements. When two or more elements are glued, an item is created.

Only elements of types that have temporal annotations 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 defined using 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, X) denote the result of evaluating an XPath expression E from a context node n in an XML data model instance X. Given a list of XPath expressions, $L = [E_1, \ldots, E_k]$, then *Eval*$(n, L, X) =$ $[$*Eval*$(n, E_1, X), \ldots,$ *Eval*$(n, E_k, X)]$. ■

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 of a field, an itemref, or a keyref. A target is an XPath expression that specifies an element's location in the slices (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 slice 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.

A schema designer specifies the item identifiers for the time-varying elements. As an example, a designer might specify that the time-varying element `<emp>` has as its item identifier, the attribute `@ID` employee (syntax example in Listing 5). 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 of an element over time [4].

```
<item target="company/emps/emp">
 <itemIdentifier name="itmIdEmp" timeDimension="bitemporal">
   <field path="@ID"/> ...
</item>
```

Listing 5. Item Identifier for `<emp>`

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} slice of a temporal document D. Let y be an element of type T in the j^{th} slice 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 $\mathit{Eval}(x, L, \mathbf{slice}(i, D)) = \mathit{Eval}(y, L, \mathbf{slice}(j, D))$ and it is not the case that there exists an element z of type T in a slice k between the i^{th} and j^{th} slices such that $\mathit{Eval}(z, L, \mathbf{slice}(k, D)) = \mathit{Eval}(x, L, \mathbf{slice}(i, D))$. ■

A temporal association relates elements that are adjacent in time and that belong to the same item. For instance, the `<emp>` element in Listing 2 is temporally associated with the `<emp>` element in Listing 3 but not the `<emp>` element in Listing 4 (though the `<emp>` element in Listing 3 is temporally related to the one in Listing 4).

5.4 Content and Existence Constraints

Over time, elements in a conventional document can change, e.g., as edits are made. A schema designer may wish to control or constrain what kinds of changes are permitted. In this section we review two constraints, which we proposed in previous research [8], to constrain the ways that an element can vary over time in its existence or content.

Let's first consider the specification of an item's existence. First an item could be "varying with gaps," which means that it may be present in some slices and absent in others. A second, more restrictive form is "varying without gaps." If such an item is present, then it cannot have gaps in its existence, e.g., it must exist through consecutive slices only. The third existence alternative is "constant." Then the item is either always present (in every slice of the document) or never present.

The content of an item may also be constrainted to be constant (no changes are allowed) or varying (the default, changes allowed). A detailed explanation of the restrictions can be found elsewhere [4], [8].

The content and existence constraints are orthogonal. For instance, an item can be constrained to have constant content (i.e., the content does not change) and varying existence (i.e., it's lifetime may have gaps).

6 Temporal Augmentations to XML Schema Constraints

We now show how to augment, with support for time, XML's cardinality, identity, referential integrity, and datatype constraints, in turn. We discussed in Section 5.1 how to interpret any particular XML constraint in a sequenced semantics, as well as how to revise that constraint to be interpreted in the current semantics. In this section, we discuss the specifics of the sequenced semantics for each type of constraint.

We then show how each kind of constraint can be extended in various ways to effect a non-sequenced semantics, that is, evaluated over an item as a whole. Note that the evaluation window and slide size can be specified for such constraints. These non-sequenced constraints are specified in the temporal schema as logical annotations.

6.1 Identity Constraints

Recall from Section 4 that identity constraints restrict uniqueness of elements and attributes in a given document, through `<key>` and `<unique>` constraints. We formally define a sequenced `key` constraint as follows.

Definition [Sequenced `<key>`] For element type E in the conventional schema, let sel be the `selector` (an XPath expression) of an identity constraint and let $F = [f_1, f_2, \ldots, f_m]$ be the field XPath expressions. Then for a temporal document $D = (X, T, S, A)$ the identity constraint is *sequenced* if and only if for all times $t \in T$, if c is a node of type E in $X_t = \mathbf{slice}(t, D)$

$$\forall e_i, e_j \in \mathit{Eval}(c, sel, X_t) : \\ \mathit{Eval}(e_i, F, X_t) = \mathit{Eval}(e_j, F, X_t) \Rightarrow i = j.$$

This proposition asserts that two elements can evaluate to the same key value only if they are in fact the same element. ■

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

A non-sequenced `<unique>` or `<key>` constraint is specified in the logical annotations through one of the following elements: `<nonSeqUnique>`, `<nonSeqKey>` or `<uniqueNullRestricted>` (all constraints, including identity, are sub-elements within an `<item>` annotation). We adopt the usual distinction between key and unique constraints. The sub-elements and attributes of these non-sequenced constraints are provided in Tables 1 (those attributes and subelement common to all temporal constraints and 2 (those components found only in `<nonSeqUnique>`, `<nonSeqKey>` or `<uniqueNullRestricted>`). Within these tables, and subsequent ones, subelements are denoted by enclosing < >; the rest are attributes.

If the `conventionalIdentifier` is included within these constraints the `<selector>` and `<field>` are drawn from the referenced (conventional) constraint; otherwise, those two elements are required. The rest of the attributes and elements are as described, though we elaborate on a few, and provide examples of most of the others, below.

A non-sequenced `<unique>` (or `<key>`) constraint requires that the field value combination of the constrained element (or attribute) is *unique between* items across time (not just at a point in time). For example, if an employee's SSN were unique, i.e., no two employees had the same SSN in a single conventional document as well as the temporal document, we would use a non-sequenced constraint. We envision non-sequenced constraints being used in three ways.

1) *Between* - Consider the conventional `unique` constraint defined in Listing 1. Suppose a non-sequenced unique constraint is placed on the email address of an employee, with an evaluation window of a year

Term	Definition	Cardinality
`name`	The name of the constraint	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) While applicability bounds conceptually correspond to a temporal element which can be represented by a series of (begin, end) sub-elements, we use a simplified semantics in this paper denoted by a single (begin, end) attribute pair	[0:1] optional
`<selector>`	For the definition of a new constraint. It is similar to the `<selector>` sub-element in the `<uniqueConstraint>` definition. It must be a relative XPath expression.	[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
Common attributes and sub-elements for temporal constraints

Term	Definition	Cardinality
`conventionalIdentifier`	The referenced conventional identifier of the conventional constraint being annotated (if present, the `<selector>` and `<field>` are omitted)	optional
`scope`	Either `within` or `between` (which is the default) semantics can be specified	optional
`nullCountMin`	The minimum number of null values allowable (used only within `<uniqueNullRestricted>`)	optional
`nullCountMax`	The maximum number of null values allowable (used only within `<uniqueNullRestricted>`)	optional

TABLE 2
Attributes for temporal unique constraints `<nonSeqUnique>`, `<nonSeqKey>` and `<uniqueNullRestricted>`

(Listing 6). Then, no two employee items can have the same email address `dana@txschema.com` (for example) in any year, but the same employee (e.g., Dana) can switch from `dana@txschema.com` to `ddoe@txschema.com` and back in a year.

2) *Within* - To specify a uniqueness constraint *within* each item, i.e., if we wished to say that an employee (e.g., Dana Doe) cannot switch from `dana@txschema.com` to `ddoe@txschema.com` and back in a single year, we would need to define a non-sequenced *within* unique constraint on an employee's email address. An example is given in Listing 7, where the `scope="within"` enables within semantics.
3) *Between and within* - To specify that each employee email is unique and also that employees cannot re-use an email, both constraints (Listing 6 and 7) are specified.

```
<item target="company/emps"> ...
 <nonSeqUnique name="employeeEmailNSUnique1"
   conventionalIdentifier="empEmailUnique" scope="between"
   evaluationWindow="year" slideSize="day"/>
</item>
```

Listing 6. Non-seq. constraint "between" employees

```
<item target="company/emps">  ...
 <nonSeqUnique name="employeeEmailNSUnique2"
   scope="within" evaluationWindow="year" slideSize="day">
  <selector xpath="emp"/> <field xpath="@email"/>
 </nonSeqUnique>
</item>
```

Listing 7. Non-seq. constraint "within" each employee

A conventional identity constraint does not imply non-sequenced uniqueness (it only implies that there are no duplicates in a slice). Thus, the same `productNo` (a conventional key) can be *re-used* for another product or changed between slices (for the same product, as long as it remains unique). To place non-sequenced restrictions on elements or attributes, we use *non-sequenced unique* and *non-sequenced key* constraints. These allow us to designate an element or attribute value (e.g., `productNo`) as unique to an item across a temporal document (with slices coalesced across the evaluation window).

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 in any associated XML document. However, this does not preclude the values from changing as long as the new value does not appear elsewhere in the conventional XML document. To designate a time-invariant key, in addition to specifying a conventional key constraint, we restrict the components of the key as time-invariant (`content="constant"`) in the logical annotation of an `<item>`.

We define a `<nonSeqKey>` *between* constraint as follows.

Definition [`<nonSeqKey>`, Between Semantics] Let c be the item containing the `<nonSeqKey>` definition, let F be the list of XPath expressions $[f_1, f_2, \ldots, f_m]$ where f_i is a `field` expression, let sel be the `selector`, and let $D = (X, T, S, A)$ be a temporal document. Then for each window (a time period) $w \subseteq T$, define $U(c, w) = \bigcup_{t \in w}(Eval(c, sel, \mathbf{slice}(t, D)) \times t)$ to be the union of the Cartesian product of the evaluation of the selector for each slice in the window and the time of the slice. The union yields the list of elements, $[(e_1, t_1), \ldots, (e_k, t_k)]$. Finally, let $item(e_i)$ be the item, v, that is the closest ancestor to e_i, i.e., e_i is an element in some slice of v. Then the `<nonSeqKey>` constraint is

$$\begin{array}{l} \forall (e_i, t_i), (e_j, t_j) \in U(c, w) : [\\ \quad Eval(e_i, F, \mathbf{slice}(t_i, D)) = Eval(e_j, F, \mathbf{slice}(t_j, D)) \Rightarrow \\ \qquad\qquad item(e_i) = item(e_j)] \ . \end{array}$$

In other words, if two elements have the same value for their key, then they are elements in the same item, though they may be in different versions of that item. The effect of the slide size is to determine the start point for each successive w. ■

A *within* constraint is similar.

Definition [<nonSeqKey>, Within Semantics] To define a <nonSeqKey> *within* constraint, we replace the constraint given above with the following.

$$\begin{aligned}
&\forall (e_i, t_i), (e_j, t_j) \in U(c, w) : [\\
&\quad (Eval(e_i, F, X_i) = Eval(e_j, F, X_j) \wedge \\
&\qquad item(e_i) = item(e_j)) \;\Rightarrow \\
&\qquad\quad \neg\exists (e_k, t_k) \in U(c, w) : [\; t_i < t_k < t_j \;\wedge \\
&\qquad\qquad\qquad Eval(e_i, F, X_i) \neq Eval(e_k, F, X_k) \;]\;)\;]
\end{aligned}$$

where $X_i = \mathbf{slice}(t_i, D)$, $X_j = \mathbf{slice}(t_j, D)$, and $X_k = \mathbf{slice}(t_k, D)$. The extension adds the constraint that the same field values must be in consecutive slices within any item. ■

We next discuss the <uniqueNullRestricted> constraint. Since the XML Schema definition of unique allows a null value at each point in time, the default semantics for <nonSeqUnique> allows for multiple null values across time (one in each conventional document). A non-sequenced <uniqueNullRestricted> constraint, in addition to specifying uniqueness, also 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. We defer a formal specification of the null counting semantics to Section 6.3 as it is similar to that of a cardinality constraint.

We now present an identity constraint example.

1) *The combination of supplier name and city serves as a* key*. However, at a later point in time we may have a different supplier with a name and city combination that was seen previously. To avoid any problem, we require that reuse should not occur for at least one year after discontinuation. Product numbers on the other hand may not be re-used at any later time. These constraints are applicable between 2005 and 2010.*

```
<item target="company/suppliers">  ...
 <nonSeqKey name="idSupplierNo" dimension="validTime"
   evaluationWindow="year" slideSize="day">
  <applicability begin="2005-01-01" end="2010-12-31"/>
  <selector xpath="supplier"/>
  <field xpath="supName"/> <field xpath="supCity"/>
 </nonSeqKey>
</item>
...
<item target="company/products"> ...
 <nonSeqKey name="idPartNo" dimension="validTime"
   evaluationWindow="lifetime">
  <applicability begin="2005-01-01" end="2010-12-31"/>
  <selector xpath="product"/>
  <field xpath="productNo"/>
 </nonSeqKey>
</item>
```

6.2 Referential Integrity Constraints

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.

Formally, we can define the sequenced <keyref> constraint as follows.

Definition [Sequenced <keyref>] For each possible referring element sel_r, let $Eval(sel_r, F_r, \mathbf{slice}(t, D))$ denote the result of evaluating the list F_r of <keyref> XPath field expressions relative to the selector element sel_r in a slice of temporal document D at time t. Similarly, let $Eval(sel_k, F_k, \mathbf{slice}(t, D))$ denote the result of evaluating the referenced key (or unique) constraint at time t. Finally, let B be the applicability bound The <keyref> constraint is satisfied when

$$\begin{aligned}
&\forall t \in B \;\; (\exists e_k \in Eval(sel_k, F_k, \mathbf{slice}(t, D))(\\
&\qquad \exists e_r \in Eval(sel_r, F_r, \mathbf{slice}(t, D)) : e_r = e_k)).
\end{aligned}$$ ■

A non-sequenced referential integrity constraint is useful to specify a reference to some past state of the XML document. Suppose we added a <largestOrder> sub-element within suppliers to represent the "largest order" (in dollar terms) placed with that supplier (with a <keyref> to orderNo). We represent a non-sequenced referential integrity constraint using a <nonSeqKeyref> element in the logical annotations in the example below. Table 3 provides the different attributes and sub-elements for the <nonSeqKeyref>, along with the components listed in Table 1.

1) *For each transaction-time slice, for each supplier, the actual order referenced (through* orderNoKey*) by the* largestOrderNo *attribute of the supplier must exist at some valid time, perhaps different from the valid time of that* largestOrderNo *attribute. The referential integrity constraint is applicable from 2008–2012, and no corresponding conventional constraint exists.*

```
<item target="company/suppliers"> ...
 <nonSeqKeyref name="supLargestOrd"
    refer="orderNoKey" dimension="validTime">
  <applicability begin="2008-01-01" end="2012-12-31"/>
  <selector xpath="supplier"/>
  <field xpath="largestOrderNo"/>
 </nonSeqKeyref>
</item>
```

2) *There exists a conventional referential integrity constraint* orderProductKeyref *(cf. Listing 1), which references a valid product number. This is interpreted as a sequenced constraint, in both valid and transaction time, over the temporal document. A related non-sequenced constraint: for each transaction-time slice, for each order, the product referenced (specified by the* orderProductRI *constraint) must exist at some valid time, perhaps different from the valid time of that order. The constraint applicability bounds span all valid time (i.e., the default).*

```
<nonSeqKeyref name="orderProductNSKeyref"
   conventionalConstraint="orderProductRI"
   dimension="validTime">
</nonSeqKeyref>
```

6.3 Cardinality Constraints

The cardinality of elements in conventional documents is restricted by minOccurs and maxOccurs, and that of attributes by setting use to "optional" or "required". These induce sequenced constraints in the temporal document.

Term	Definition	Cardinality
`conventionalConstraint`	The conventional referential integrity constraint being extended via this annotation (if present, the `<selector>` and `<field>` are omitted	optional
`refer`	The referenced identifier; the corresponding `dimension` attribute must be compatible.	optional

TABLE 3
Attributes and sub-elements for `nonSeqKeyref`

Term	Definition	Cardinality
`newOnly`	specifies whether only changes to "new" values should be considered (default `"false"`); only for `<nonSeqCardinality>`	optional
`minOccurs`	default: `"0"`	optional
`maxOccurs`	default: `"unbounded"`	optional
`<group>`	The level at which grouping is performed relative to the `<selector>` (default: `self`)	optional

TABLE 4
Attributes and sub-elements for `<seqCardinality>` and `<nonSeqCardinality>`

Augmented sequenced cardinality constraints use a new element, `<seqCardinality>`, whose syntax is summarized in Table 4 (along with the syntax in Table 1), except for `newOnly`, which doesn't apply to sequenced cardinality constraints. The `minOccurs` and `maxOccurs` attributes are analogous to those in XML Schema.

1) *At every point in time there should be a maximum of 250 orders for the company. The constraint is to be enforced during 2010-11.*

```
<item target="company">  ...
 <seqCardinality name="suppliersSeq"
   maxOccurs="250" dimension="validTime">
  <selector xpath="."/>
  <field xpath="suppliers/supplier/order"/>
  <applicability begin="2010-01-01" end="2011-12-31"/>
 </seqCardinality>
</item>
```

It could be the case that a specific `<order>` may be placed with several `<supplier>`s, in which case the repetitious `<order>` elements are considered as a single `<order>`. To count the shared `<order>`s distinctly, we allow the user to refine the count by *group*ing `<supplier>`s. The conventional cardinality constraints are not designed to handle this. This is our motivation behind introducing the `group` option for a cardinality constraint.

2) *At every point in time there should be a maximum of 250 orders for the company across suppliers (constraint applicability is 2010-11.*

```
<item target="company"> ...
 <seqCardinality name="suppliersSeq2"
   maxOccurs="250" dimension="validTime">
  <selector xpath="."/>
  <group xpath="suppliers/supplier">
  <field xpath="order"/>
  <applicability begin="2010-01-01" end="2011-12-31"/>
 </seqCardinality>
</item>
```

Non-sequenced cardinality constraints can be used to restrict the cardinality over time. Consider the example of an `<order>` element in Listing 8. We see that the `<deliveredOn>` element may not be present in a specific document slice. Let us further say, that while it may be empty at the time the order was placed, we require it to appear at some point (say within a month of the order being placed). So, even though a sequenced `minOccurs="0"` is satisfactory for a conventional document, we may desire the analogous non-sequenced `minOccurs="1"` for a temporal document. For attributes, a similar requirement may be specified (i.e., a conventional `"optional"` attribute, may be `"required"` over some evaluation window). The syntax for `<nonSeqCardinality>` constraints is given in Table 4.

```
<xs:element name="order">
 ... <xs:element name="deliveredOn" minOccurs="0"
       maxOccurs="1" type="xs:date"/> ...
</xs:element>
```

Listing 8. Orders with an optional `<deliveredOn>`

3) *There should be a* `deliveredOn` *element at some time for each order.*

```
<item target="company">  ...
 <nonSeqCardinality name="deliveredOnNonSeq"
   minOccurs="1" dimension="validTime"
   evaluationWindow="month">
  <selector xpath="."/>
  <field xpath="deliveredOn"/>
 </nonSeqCardinality>
</item>
```

Another refinement that may be desired for a cardinality constraint is to constrain the cardinality of a descendant that is not a child, which is not possible in XML Schema. Consider the schema in Listing 1. This says that at any point in time, each company has at least one supplier, for which there may or may not be an order. A non-sequenced cardinality constraint can be used to place a limit of less than or equal to 1500 `<order>`s for the company in any calendar month. A third refinement that may be desired is to distinguish "new" values, which are values that have not previously been seen in the evaluation window. For example, suppose an order `status` attribute can have one of the five following values: `"placed"`, `"underReview"`, `"beingProcessed"`, `"shipped"`, and `"returned"`. It is possible that changes to the order can have it swap back and forth between `"underReview"` and `"beingProcessed"`. Over a period of a month, it might have, say, seven total changes to the value of which only four are distinct. To count each change, the user would set `changes="newOnly"`, otherwise all changes are counted.

We represent a non-sequenced cardinality constraint in logical annotations using a `<nonSeqCardinality>` element; a `<seqCardinality>` element is used for sequenced cardinality constraints. The syntax for both elements is summarized in Table 4. In the following examples, each constraint is specified within the *scope* of some item. Relative to that scope, the `<selector>` locates items that are to be constrained. (Hence, the scope, the `target` of the enclosing item, is just a prefix for the selector.) Combinations of `<field>`s are counted for each `<group>`, and the counts are summed over a group to determine the cardinality of each item located by the `<selector>`. The computed cardinality must fall between the `min` and `max` to satisfy the constraint.

4) *No supplier should be given more than 100 orders in a calendar month. Furthermore, across all of the suppliers at most 500 products could be ordered in total (a product that is in two different orders is counted as two different products, hence the* `<group>`*).*

```
<item target="company/suppliers"> ...
 <nonSeqCardinality name="supOrders"
   max="100" dimension="validTime"
   evaluationWindow="month" slideSize="month">
  <selector xpath="supplier"/>
  <field xpath="order/ordProductNo"/>
 </nonSeqCardinality>
 ...
 <nonSeqCardinality name="supProducts"
   max="500" dimension="validTime"
   evaluationWindow="month" slideSize="month">
  <selector xpath="supplier"/>
  <group xpath="order"/>
  <field xpath="productNo"/>
 </nonSeqCardinality>
</item>
```

5) *A product could change names (hence,* `newOnly`*) up to three times a month, but can have at most four distinct names in a year. This is in force from 2008–2011.*

```
<item target="/company/products"> ...
 <nonSeqCardinality name="prodNameMonth"
   dimension="validTime" evaluationWindow="month"
   newOnly="true" slideSize="day"
   minOccurs="0" maxOccurs="3">
  <selector xpath="product"/>
  <field xpath="@productName"/>
  <applicability begin="2008-01-01" end="2011-12-31"/>
 </nonSeqCardinality>
 ...
 <nonSeqCardinality name="prodNameYear"
   dimension="validTime" evaluationWindow="year"
   slideSize="day" minOccurs="1" maxOccurs="4">
  <selector xpath="product"/>
  <field xpath="@productName"/>
  <applicability begin="2008-01-01" end="2011-12-31"/>
 </nonSeqCardinality>
</item>
```

Definition [Cardinality Constraint] Formally, we define a cardinality constraint as follows. Let

– D be a temporal document,
– c be the context item for the constraint (item being annotated),
– $item(e)$ be the item, v, that is the closest ancestor to e, i.e., e is an element in some version of v,
– S be the `<selector>` XPath expression relative to c,
– G be the `<group>` expression relative to S (by default it is "`.`"), and
– F be a list of `<field>` expressions.

Both S and G must locate items, that is, they must locate elements that correspond to the `target` expression for some item in the logical annotations. Then for each window (a time period), w, in the constraint define

$$A(c,w) = \{(t, item(x), item(y), Eval(y, F, \mathbf{slice}(t, D))) \mid t \in w \wedge x \in Eval(c, S, \mathbf{slice}(t, D)) \wedge y \in Eval(x, G)\} \ .$$

$A(c,w)$ is a set of tuples, $\{(t_1, s_1, g_1, v_1)$, $\{(t_2, s_2, g_2, v_2)$, $\ldots$, $(t_k, s_k, g_k, v_k)\}$. From this set we can extract tuples that represent a change as follows.

$$Changes(A(c,w)) = \{(t,s,g,v) \mid (t,s,g,v) \in A(c,w) \wedge \neg\exists k[k = t-1 \ \wedge \ (k,s,g,v) \in A(c,w)]\}$$

While $Changes(A(c,w))$ extracts all changes, we are sometimes interested in only changes to "new" values, hence we modify the above definition to capture changes that represent only changes that have not previously occurred in the window.

$$NewChanges(A(c,w)) = \{(t,s,g,v) \mid (t,s,g,v) \in A(c,w) \wedge \neg\exists k[k < t \ \wedge \ (k,s,g,v) \in A(c,w)]\}$$

We are now in a position to count the changes. Let

$$Count(A(c,w)) = \{(s, \mathbf{card}(\{(t,s,g,u)\})) \mid (t,s,g,u) \in Changes(A(c,w))\}$$

count the number of changes for each item, s, located by the `<selector>`. To count the number of "new" changes, we would use $NewChanges$ in place of $Changes$ in the above definition.

A cardinality constraint for a context c and a window w tests the following predicate.

$$\neg\exists s[\exists x : (s,x) \in Count(A(c,w)) \ \wedge \ (x < \mathit{min} \vee \mathit{max} < x)]$$

A non-sequenced cardinality constraint differs from a sequenced cardinality constraint only in the size of the window; for the former the window can be any size, but for a sequenced constraint the window is a single instant. ■

The formal definition of a `<uniqueNullRestricted>` constraint, which restricts the number of null values, is similar to that of a non-sequenced cardinality constraint, but changes resulting in null values are counted rather than all changes.

6.4 Datatype Restrictions (Constraints)

The XML Schema `<simpleType>` element is used to specify a value range and induces a sequenced constraint that ensures conventional document values conform to this range.

We now consider non-sequenced augmentations of such simple types. A non-sequenced equivalent of this type of constraint can be considered either at the schema level (i.e., evolution of the *datatype* within schema evolution) or at the instance level (i.e., evolution of the *value* within instances, that is, 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 non-sequenced constraint could restrict discrete and continuous changes. *Discrete changes* are handled by defining a set of value transitions for the data. For example, it could be specified that while supplier ratings can change over

Term	Definition	Cardinality
`<valuePair>`	Sub-element listing possible pairs for discrete changes (only if `<valueEvolution>` not present)	[0:U]
`<old>`, `<new>`	Sub-elements of `valuePair`	[1:1]
`<valueEvolution>` (`direction`)	Sub-element specifying direction of continuous changes (only if `<valuePair>` not present) can be one of: `"LT"` (less than), `"GT"` (greater than), `"GE"` (greater or equal), `"LE"` (less or equal), `"EQ"` (equal) and `"NE"` (not equal)	[0:1] required

TABLE 5
Attributes and sub-elements of `<transitionConstraint>`

time, the changes can only occur in single-step increments (e.g., a rating changing from value `"B"` to either `"A"` or `"C"`). In this scheme, to allow for successive values being the same, the `<old>` and `<new>` entries will have the same content. *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. The details of these logical annotations are given in Table 5, along with the components listed in Table 1.

1) *Supplier ratings can move up or down a single step at a time in valid time; no restrictions are placed in transaction time, since a data entry error might be made. This is applicable between 2008 and 2011.*

```
<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="2011-12-31"/>
</transitionConstraint>
```

2) *Employee salaries should not go down, but may increase (i.e., each salary value is $>=$ the previous one) between 2008 and 2010. However, a salary freeze is in place between January and June 2010 due to economic factors.*

```
<transitionConstraint name="employeeSalary1"
     dimension="validTime">
  <selector xpath="emp"/>
  <field xpath="salary"/>
  <valueEvolution direction="GE"/>
  <applicability begin="2008-01-01" end="2010-12-31"/>
</transitionConstraint>
...
<transitionConstraint name="employeeSalary2"
    dimension="validTime">
  <selector xpath="emp"/>
  <field xpath="salary"/>
  <valueEvolution direction="EQ"/>
  <applicability begin="2010-01-01" end="2010-06-30"/>
</transitionConstraint>
```

7 Implications of Schema Versioning

Schemas designers often edit their schemas, refining and adding element and attribute types. One challenge with schema versioning is that, in this potential quicksand, anything can change, and thus must be versioned: the conventional documents, the base schema, the annotations, the schema documents included by these documents, even the schemas of these schema components. And, because the physical annotations can change, the concrete representation within a temporal XML document can vary. Thus, it becomes even more difficult to even define validation in such a fluid environment.

Elsewhere [10] we delve into the specifics of how to accommodate schema versioning within τXSchema. Our approach exploits the concept of *schema-constant periods* [32]. It is possible, even with versioned schemas having themselves versioned schemas, to identify contiguous periods of time when there are no schema changes, *anywhere*. Now, during such schema-constant periods the data may be (and probably is) versioned, but at least one has a fixed base schema and fixed logical annotations, each of which has a fixed schema. And since the physical annotations are fixed, the representation is also fixed, it is possible to read and interpret the temporal document during that schema-constant period, and even to validate that portion of the document. So a general temporal document can be viewed as a sequence of data-varying documents, each over a single schema-constant period. Since one can validate within each schema-constant period, given the approaches elaborated on earlier, all that is necessary is to validate *across* schema changes.

As a concrete example, Listing 1 includes the key constraint for the `ID` attribute of `<emp>`. In the temporal document, this is interpreted as a sequenced constraint. Suppose that employees at some point are divided into permanent and contract, identified by the elements `<permanent>` and `<contract>`, respectively. Each employee may end up in either of the two new elements; we wish to retain the unique constraint semantics.

One approach would use object-oriented methodology to "specialize" the class "emp" into "permanent" and "contract" subclasses. Then the constraint specification on "emp" would be inherited by both subclasses. XML Schema does not however support such modeling. While XML Schema supports inheritance for type definitions (through extension and restriction), type definitions do not have constraints (only element definitions do). So in XML Schema, constraint inheritance is not supported. To specify that `ID` is unique across permanent and contract elements a new constraint should be defined with a selector that selects both kinds of elements, e.g., `<selector path="permanent | employee"/>`.

With that background let us consider how this change would be modeled in τXSchema. When the schema evolves so that every (or only some) employee becomes a permanent or contract employee, the designer would then also specify key constraints within the two new elements to require that permanent and contract employees have unique IDs. The data in the document from that point forward would have to correspond to this new schema.

The one remaining issue that concerns temporal constraints is how to check *non-sequenced* constraints across schema changes. Note that schemas vary only over transaction time. Hence, non-sequenced constraint validation is easier in valid time, as schema changes cannot occur. And sequenced constraints over transaction time are effectively checked at each point in transaction time.

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

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

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

We decided on a modified version of the first alternative: to apply a non-sequenced constraint only within the schema-constant period in which it is defined, if there were a schema change to any of the items involved in the constraint. The non-sequenced constraints are "restarted" on any such schema change. In effect the schema change deletes all the old constraints and then adds them back as new constraints. For example, consider the first example in Section 6.3: *there should be no more than fifty active suppliers in any calendar year.* If the schema changed on July 1 concerning `<supplier>`, this non-sequenced constraint would be checked twice, for the first half of the year and for the second half.

8 Expressive Power

As mentioned in Section 2, there has been very little done in the area of temporal constraints for XML. But for the work that has been done, we can evaluate the expressiveness of our approach to these other approaches.

Rizzolo and Vaisman's temporal extension to XML [33] specifies (in Definition 3 on page 1184) six conditions for a valid temporal document in their model. While the first four of these conditions are specific to their encoding (recall that our approach supports multiple encodings, including that proposed by Rizzolo and Vaisman), the last two of their conditions are relevant to temporal constraints.

The fifth condition states, "For any containment edge $e_c(n_i, n_j, T_{e_c})$, if n_j is an attribute of type REF, such that there exists a reference edge $e_r(n_j, n_k, T_{e_r})$, then $T_{e_c} = T_{e_r}$ holds." As discussed in Section 5.1, in our model a non-temporal referential integrity constraint is mapped in a temporal document to one that applies in each slice. Here we differ with Rizzolo and Vaisman, as what they *require* in their model is what in our design is a *non-sequenced referential integrity constraint* (also discussed in Section 5.1). Our design is more uniform in that we utilize a per-slice semantics for *all* non-temporal constraints when applied to a temporal document, permitting the user to specify additional non-sequenced variants of such non-temporal constraints. As we argue there, a per-slice (that is, sequenced) semantics is very natural to the user.

The last of their conditions states, "Let $e_r(n_i, n_j, T_{e_r})$ be a reference edge. Then, $T_{e_r} \subseteq \textit{lifespan}(n_j)$ holds." This states that a reference edge applies in a subset of the slices in which the destination node exists, which is a quite specific kind of non-sequenced referential-integrity constraint. Again, we prefer a per-slice semantics for referential integrity, as with all other explicit non-temporal integrity constraints, while allowing the user to specify additionally non-sequenced variants.

Finally, we note that our approach provides all the benefits listed in Section 3. We provide a more in-depth look in previous work [34].

9 Checking Temporal Constraints

τXSchema provides tools to construct and validate temporal documents, including an extension of XMLLint [3]. As discussed in Section 3, to validate a temporal document, τXMLLint first converts a temporal schema into a *representational schema*, which is a conventional XML Schema document that describes how the temporal information is represented in the temporal document. XMLLint is then used to validate the temporal document against the representational schema. Finally, the temporal document is validated against the temporal schema by the temporal constraint validator. Fig. 2 depicts this process.

9.1 Where to Enforce Constraints?

Given the architecture in Fig. 2, there are two places where temporal constraint functionality could be enforced.

1) Express the constraint within the representational schema, and hence when the conventional validator validates the temporal document against the representational schema it also validates the temporal constraint.
2) Enforce the temporal constraint directly within the temporal constraint validator code.

The representational schema serves at least two important functions. First it ensures that every slice of the temporal document is syntactically valid with respect to the corresponding conventional schema. Second, the representational schema is important in constructing, evaluating, and optimizing temporal queries. Can the representational schema also help in the validation of non-sequenced constraints?

At first glance, this seems attractive: we could use an existing conventional validator to validate our temporal documents in a simple and straightforward manner. However, expressing constraints in this way could result in a large and complex representational schema, making the conventional validation process inefficient. Further, some temporal constraints cannot be expressed in the representational schema at all [34].

Consider, for example, the (sequenced) `employeeIDKey` constraint in Listing 1: an `<emp>` element has an `ID` attribute and we require the attribute to be globally unique. As shown in the example in Fig. 3, this constraint is initially valid at time t_1, but is violated by the change at t_3.

The temporal document shown in Fig. 4 encodes the change history of the documents, but no representational schema can be constructed to match the intention of the sequenced identity constraint. If, for instance, the representational schema were to place an identity constraint globally on the `ID` attribute, the conventional validator would detect a conflict at the `Tandy` elements at times t_1 and t_8, but this is incorrect behavior (Tandy should be allowed to have the same `ID` at different times). However, not having an identity constraint at all

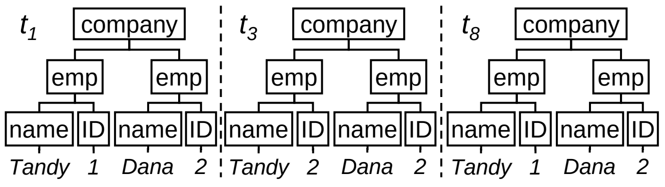


Fig. 3. Three <company> slices.

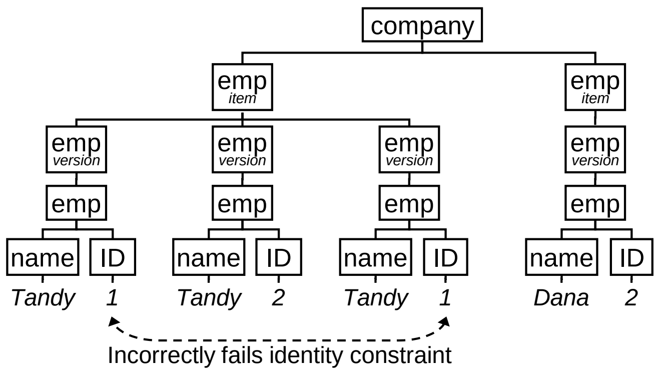


Fig. 4. The three <company> slices squashed.

would also cause incorrect behavior, because the conventional validator must be able to detect that the `ID` of Tandy and Dana are the same at times t_3–t_7.

The problem we are seeing is not in the individual value of any attribute, but rather in a complex interaction between values and times. Referential integrity constraints are similar: the interaction between values and time cannot be modeled in XML Schema, and thus it is easy to conceive scenarios that are logically invalid but undetectable within XML Schema and vice-versa. Considering cardinality constraints, versions of elements can have arbitrary start and ending times, and there is no way in XML Schema to determine how many versions exist at any given slice of time. In contrast, datatype constraints can be expressed in the representational schema since, by the definition of sequenced constraints, the type must be constant throughout all times and thus the complex interaction doesn't occur. The transformation from the conventional schemas to the representational schema is trivial in all cases.

We conclude that XML Schema lacks the expressive power to directly state some flavors of temporal constraints. Such constraints must be enforced instead by procedural code within the temporal constraint validator, τXMLLINT.

9.2 τXMLLINT Implementation

We have implemented τXMLLINT in Java and DOM. Our tool supports the entire constraint language presented here, including sequenced and non-sequenced constraints and all of the constructs summarized in Tables 1–5. In this section, we summarize our approach; the online Appendix provides the detailed algorithms.

τXMLLINT first reads the temporal document, creating a DOM tree. It then reads the temporal schema, including the logical and physical annotations. All the DOM nodes that are irrelevant to the constraints are removed. The removal is performed only once and the consequent validation steps are carried out based on a quite smaller DOM tree, significantly improving the performance of constraint checking.

For sequenced constraints, τXMLLINT performs conventional validation with the help of the `validate()` method provided by the `Validator` class in the Java Platform. Specifically, τXMLLINT invokes a *slicing* routine to extract each slice from the temporal document. For each slice (which is represented as a DOM object), the `validate()` method is called to evaluate that slice against its conventional schema. τXMLLINT indicates that the temporal document is valid only if `validate()` returns true for every slice.

For non-sequenced constraints, τXMLLINT provides its own validation algorithm for each of the type of constraint. As described in Section 3, the logical annotations provide the constraint definitions. τXMLLINT extracts all the defined constraints from the annotation file and checks them individually.

Although the validation of the constraint types vary, there are several common steps. These include the evaluation window, slide size, and their interaction with applicability (see Table 1). For identity constraints, τXMLLINT collects all the unique values valid within the specified applicability window into a list and then iterates through this list to look for offending duplicates. For cardinality constraints (e.g., Example 2 in Section 6.3), the validation is more complicated. τXMLLINT first collects for each target (designated by the `target` XPath expression) the items (designated by the `field` XPath expression) within the stated applicability window, and then groups these items by different group identifiers (designated by the `group` XPath expression). Each such collection will have a cardinality that must be checked. (If group is not specified, the `field` XPath expression is sufficient.) Finally, for data type constraints, the focus is instead on each individual data type, to ensure that it respects the requirements imposed by the constraint.

9.3 Empirical Evaluation

We now study the performance of τXMLLINT, focusing on how the validation time of a temporal document with τXMLLINT compares with validation time of multiple slices, each a conventional XML document, with XMLLINT. To do so, we use a benchmark dataset defined in τBench, which is a benchmark for temporal data [35]. τBench is based on XBench, which is a family of non-temporal benchmarks with XML documents, XML Schemas, and associated XQuery queries [36]. One of the benchmarks in XBench, called the *document-centric/single document* benchmark, defines a book store catalog with a series of books (`<item>`s), their authors and publishers, and related books. τBench runs a temporal simulation that randomly changes `<item>` elements at each point in time, based on user-supplied parameters, such as how many elements to change and how often to change them.

We used Dataset 4 from τBench, which consists of a temporal document and the slices at each point in time. We varied the number of slices from 50 to 800, to examine the scaling characteristics of the tools. Since the temporal simulation in τBench adds and deletes elements with equal frequency, the size of each slice remains roughly constant over time, at

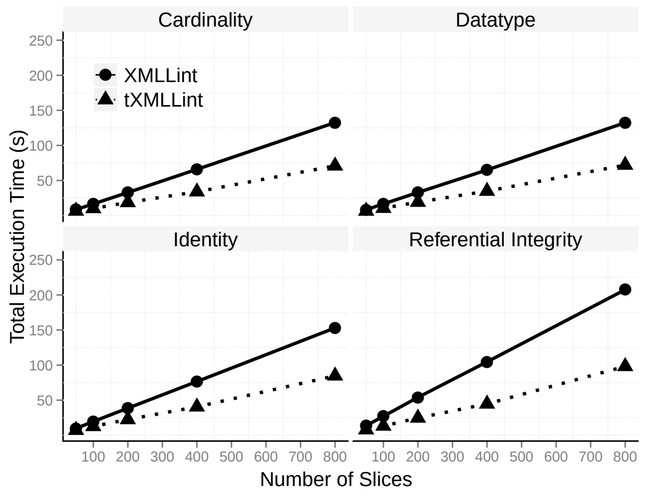


Fig. 5. Total execution time of sequenced constraints.

6.5MB, for a total size of 325MB (50 slices) to 5.2GB (800 slices). The temporal document ranged in size from 13MB (50 slices) to 31MB (800 slices). This document exhibited primarily linear growth (though at a rate much less than the conventional slices), with a small quadratic component arising from timestamps at different levels. The compression ratio increases from 25 for 50 slices to 167 for 800 slices.

We conducted two studies: a performance comparison between τXMLLINT and XMLLINT in validating sequenced constraints, and a performance evaluation of validating non-sequenced constraints with τXMLLINT. The second study was to examine the behavior of non-sequenced checks (τXMLLINT being the first such validator to implement constraints). We performed both studies on a machine running Ubuntu 9.10 with a 2.8GHz 16-core CPU and 64GB of memory. We evaluated each type of constraint; the online Appendix gives the actual constraint definitions used.

In the study of the sequenced constraints, we measured the *total execution* time of the tools, which is the wall-clock time taken from process invocation to process termination, including I/O and constraint validation. Since XMLLINT can only operate on a single slice at a time, we iteratively applied XMLLINT on every slice and report the aggregated total execution time. We applied τXMLLINT just to the temporal documents and report the total execution time.

Fig. 5 shows that for all four types of sequenced constraints, τXMLLINT is more efficient (has a lower total execution time) than XMLLINT. Moreover, as the number of slices increases, the performance benefits of applying τXMLLINT becomes even more significant. This is primarily due to the fact that the space requirement for storing all the slices grows faster than storing the single temporal document, thus the I/O overhead is inherently higher for XMLLINT to operate. The CPU overhead is also reduced because τXMLLINT removes irrelevant DOM nodes prior to slicing.

To study the performance of checking non-sequenced constraints, we applied τXMLLINT to the temporal documents and report both the total execution time and the *constraint validation* time, which is only the time required to validate

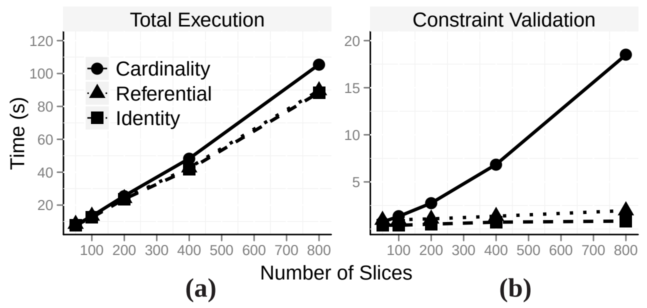


Fig. 6. Execution time of non-sequenced constraints.

the given constraint (i.e., I/O is omitted). For such constraints, Fig. 6**(a)** suggests that the running time is dominated by I/O for non-sequenced constraints as well. Fig. 6**(b)** emphasizes that cardinality constraints require greater CPU time. This is due to the fact that for the other non-sequenced constraints, the evaluation window was set at all time, whereas for the cardinality constraint it was set at one year. This implies that the number of evaluation windows increased from 1 (for 50 slices) to 16 (for 800 slices), effecting a quadratic growth in terms of number of slices.

10 Conclusion and Future Work

We have shown here how to smoothly include both conventional XML integrity constraints as well as new temporal constraints to XML documents whose content varies across time and even whose schema varies across time. This is done by replacing the schema with a (possibly time-varying) temporal schema and replacing the document with a temporal document, both of which are upward compatible with conventional XML and with conventional tools such as XMLLINT. Our approach accommodates all three kinds of temporal constraints, that is, current, sequenced, and non-sequenced, and reinterprets existing non-temporal constraints as sequenced in the presence of time-varying data. We have developed an implementation that utilizes a separate temporal validator component to evaluate most of the temporal constraints, those that cannot be expressed in the representational schema; this implementation is more efficient than evaluating the sequenced constraints independently for each slice with XMLLINT.

One area of future work concerns optimization and efficiency. It would be useful to consider the impact of timestamp placement (physical annotation) and impact of parameters (logical annotation) such as evaluation window size on efficiency (document size, I/O time, and CPU time for validation). New representations can be evaluated to improve the space-efficiency of temporal documents and the time taken to validate constraints. In particular, it is well known that DOM-based implementations suffer from a memory bottleneck for huge documents. We would like to explore SAX-based temporal constraint validation techniques to avoid loading a complete document history into memory. Any DOM application can be converted to a SAX implementation by having the latter cache any information that is needed that is not directly within the node currently being handled. So for example a SAX implementation of our temporal constraint checker, τXMLLINT, could cache the list of nodes computed (incrementally) by the *Eval()* function defined in Section 6.1.

We would also like to consider specific extensions to the temporal constraint annotations described in this paper. A more powerful version of the `<nonSeqUnique>` (or `<nonSeqKey>`) constraint would permit the user to specify exactly how many times any key (or unique) value other than null can appear across time. The default is 1, in which case it is identical to a non-sequenced unique or key constraint. We term this constraint as a *value cardinality constraint*, but leave it for future work as it has no XML Schema equivalent. Similarly, we leave for later consideration transition constraints on non-adjacent states [37], other varieties of cardinality constraints [25] with no XML Schema equivalent, and incorporating temporal indeterminacy [38] into constraint representation and evaluation.

In this paper we consider only the case where at most one `<item>` annotates each element type definition. It would be interesting to relax this restriction to permit several `<item>`s for an element type definition. Recall that an item represents the gluing of elements across slices; it is a logical rather than a physical construct, and logically, elements could be glued in more than one way. The relaxation would potentially allow us to combine "within" and "between" constraints into a single kind of contraint.

Finally, many optimizations could be applied to the validator. For example, checking constraints of the same type, such as `nonSeqUnique` can be scheduled together. Also, checking constraints with higher violation probability can be scheduled earlier. The order of the violation likelihood of the constraints can be inferred by the temporal document. For instance, the `transitionConstraint` is more likely to be violated if the temporal document contains many state change records.

REFERENCES

[1] "An act to protect investors by improving the accuracy and reliability of corporate disclosures made pursuant to the securities laws, and for other purposes (brief title: Sarbanes-oxley act of 2002)," July 2002.

[2] F. Grandi, "Introducing an annotated bibliography on temporal and evolution aspects in the world wid web," *SIGMOD Record*, vol. 33, pp. 84–86, June 2004.

[3] Libxml, "The XML C parser and toolkit of Gnome, version 2.7.2," 2008. http://xmlsoft.org/, Viewed February 5, 2009.

[4] F. Currim, S. Currim, C. Dyreson, S. Joshi, R. T. Snodgrass, S. W. Thomas, and E. Roeder, "τxschema: Support for data- and schema-versioned XML documents," *TimeCenter*, 2009. TR-91.

[5] C. Dyreson, R. T. Snodgrass, F. Currim, and S. Currim, "Schema-mediated exchange of temporal XML data," in *ER 2006: Proc. of the 25th Intl. Conf. on Conceptual Modeling*, Lecture Notes in Computer Science, Vol. 4215, pp. 212–227, Springer-Verlag, 2006.

[6] C. Dyreson, R. T. Snodgrass, F. Currim, S. Currim, and S. Joshi, "Weaving temporal and reliability aspects into a schema tapestry," *Data and Knowledge Engineering*, vol. 63, no. 3, pp. 752–773, 2007.

[7] R. T. Snodgrass and I. Ahn, "Temporal databases," *IEEE Computer*, vol. 19, no. 9, pp. 35–42, 1986.

[8] F. Currim, S. Currim, C. E. Dyreson, and R. T. Snodgrass, "A tale of two schemas: Creating a temporal XML schema from a snapshot schema with τxschema," in *9th Intl. Conf. on Extending Database Technology*, pp. 559–560, Springer Berlin/Heidelberg, 2004.

[9] S. Joshi, "τXSchema - support for data- and schema-versioned XML documents," Master's thesis, Computer Science Department, University of Arizona, August 2007.

[10] R. T. Snodgrass, C. Dyreson, F. Currim, S. Currim, and S. Joshi, "Validating quicksand: Temporal schema versioning in τxschema," *Data Knowledge Engineering*, vol. 65, no. 2, pp. 223–242, 2008.

[11] Z. Brahmia, R. Bouaziz, F. Grandi, , and B. Oliboni, "Schema versioning in τxschema-based multitemporal XML repositories," Tech. Rep. TR-93, TimeCenter, December 2010.

[12] S. S. Chawathe, S. Abiteboul, and J. Widom, "Managing historical semistructured data," *Theory and Practice of Object Systems*, vol. 5, pp. 143–162, August 1999.

[13] C. E. Dyreson, H. Lin, and Y. Wang, "Managing versions of web documents in a transaction-time web server," in *WWW '04: Proc. of the 13th Intl. Conf. on World Wide Web*, pp. 422–432, ACM, 2004.

[14] S. Y. Chien, V. J. Tsotras, and C. Zaniolo, "Efficient schemes for managing multiversionXML documents," *The VLDB Journal*, vol. 11, no. 4, pp. 332–353, 2002.

[15] D. Gao and R. T. Snodgrass, "Syntax, semantics, and evaluation in the τxquery temporal XML query language," Tech. Rep. TR-72, TimeCenter, 2003.

[16] K. Nørvåg, "Algorithms for temporal query operators in XML databases," in *EDBT Workshops*, pp. 169–183, 2002.

[17] J. Chomicki, "Efficient checking of temporal integrity constraints using bounded history encoding," *ACM Trans. on Database Systems*, vol. 20, no. 2, pp. 149–186, 1995.

[18] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati, "An access control model supporting periodicity constraints and temporal reasoning," *ACM Trans. on Database Systems*, vol. 23, pp. 231–285, 1998.

[19] A. U. Tansel, "Temporal data modeling and integrity constraints in relational databases," in *ISCIS 2004*, Lecture Notes in Computer Science, pp. 459–469, Springer, 2004.

[20] J. Chomicki and D. Niwinski, "On the feasibility of checking temporal integrity constraints," *Journal of Computer and System Sciences*, vol. 51, no. 3, pp. 523–535, 1995.

[21] J. Chomicki and D. Toman, "Implementing temporal integrity constraints using an active dbms," *IEEE Trans. on Knowledge and Data Engineering*, vol. 7, no. 4, pp. 566–582, 1995.

[22] J. F. Roddick, "Schema evolution in database systems: an annotated bibliography," *SIGMOD Rec.*, vol. 21, no. 4, pp. 35–40, 1992.

[23] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: the prism workbench," in *Very Large Data Base*, 2008.

[24] C. Combi, S. Degani, and C. S. Jensen, "Capturing temporal constraints in temporal er models," in *ER '08: Proc. of the 27th Intl. Conf. on Conceptual Modeling*, pp. 397–411, Springer-Verlag, 2008.

[25] F. Currim and S. Ram, "Modeling spatial and temporal set-based constraints during conceptual database design," *Information Systems Research*, forthcoming.

[26] A. Artale, C. Parent, and S. Spaccapietra, "Evolving objects in temporal information systems," *Annals of Mathematics and Artificial Intelligence*, vol. 50, no. 1-2, pp. 5–38, 2007.

[27] R. T. Snodgrass, *Developing time-oriented database applications in SQL*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2000.

[28] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass, "On the semantics of "now" in databases," *ACM Trans. on Database Systems*, vol. 22, no. 2, pp. 171–214, 1997.

[29] F. Currim and S. Ram, "Conceptually modeling windows and bounds for space and time in database constraints," *Commun. ACM*, vol. 51, no. 11, pp. 125–129, 2008.

[30] R. Snodgrass, "The temporal query language TQuel," *ACM Trans. on Database Systems*, vol. 12, no. 2, pp. 247–298, 1987.

[31] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo, "Coalescing in temporal databases," in *Proc. of the Intl. Conf. on Very Large Data Bases*, (Bombay, India), pp. 180–191, September 1996.

[32] R. T. Snodgrass, S. Gomez, and E. McKenzie, "Aggregates in the temporal query language tquel," *IEEE Trans. on Knowledge and Data Engineering*, vol. 5, no. 5, pp. 826–842, 1993.

[33] F. Rizzolo and A. A. Vaisman, "Temporal XML: modeling, indexing, and query processing," *The VLDB Journal*, vol. 17, no. 5, pp. 1179–1212, 2008.

[34] S. W. Thomas, "The implementation and evaluation of temporal representations in XML," Master's thesis, Computer Science Department, University of Arizona, March 2009.

[35] S. W. Thomas, R. T. Snodgrass, and R. Zhang, "τBench: Extending XBench with Time," Tech. Rep. TR-93, TimeCenter, December 2010.

[36] B. B. Yao, M. T. Ozsu, and J. Keenleyside, "XBench - A Family of Benchmarks for XML DBMSs." Technical Report CS-TR-2002-39, School of Computer Science, University of Waterloo, December 2002.

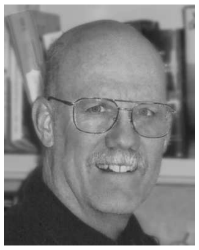
[37] E. Rose and A. Segev, "Toodm – a temporal object-oriented data model with temporal constraints," in *10th Intl. Conf. on the Entity-Relationship Approach*, pp. 205–229, 1991.

[38] C. E. Dyreson and R. T. Snodgrass, "Supporting valid-time indeterminacy," *ACM Trans. on Database Systems*, vol. 23, no. 1, pp. 1–57, 1998.

Faiz Currim is with the Department of Management Information Systems at the University of Arizona. He received his PhD from the University of Arizona, and was a professor at the University of Iowa prior to returning to Arizona. His research interests include applications in the areas of database design and management, conceptual data modeling, database constraints, spatial and temporal data, and XML Schema management.

Sabah Currim is a Senior Data Warehouse Analyst in the Mosaic Project at the University of Arizona. She received her PhD from the University of Arizona. Her research interests include conceptual data modeling, learning, database design and management, data warehouse, XML Schema management and IT Governance.

Curtis Dyreson is an Assistant Professor in the Department of Computer Science at Utah State University. He serves as the ACM SIGMOD DiSC Editor, the ACM SIGMOD Anthology Editor and the Information Director for ACM Transactions on Database Systems. His interests include temporal databases, native XML databases, data cubes, and providing support for proscriptive metadata. Prior to coming to Utah State University, Curtis was a professor at Washington State University, James Cook University, Aalborg University, and Bond University.



Richard T. Snodgrass received the BA degree in physics from Carleton College and the MS and PhD degrees in computer science from Carnegie Mellon University. He joined the University of Arizona in 1989, where he is a professor of computer science. He is an ACM fellow. Dr. Snodgrass was editor-in-chief of the ACM *Transactions on Database Systems*, was ACM SIGMOD Chair from 1997 to 2001, and has chaired the ACM Publications Board, the ACM History Committee, and the ACM SIG Governing Board Portal Committee. He served on the editorial boards of the *International Journal on Very Large Databases* and the IEEE *Transactions on Knowledge and Data Engineering*. He chaired the Americas program committee for the 2001 International Conference on Very Large Databases and the program committee for the 1994 ACM SIGMOD Conference. He received the 2004 Outstanding Contribution to ACM Award and the 2002 ACM SIGMOD Contributions Award. He currently is a member of the Advisory Board of ACM SIGMOD, and the Outstanding Contribution to ACM Award Committee. He chaired the TSQL2 Language Design Committee, edited the book, the TSQL2 Temporal Query Language (Kluwer Academic Press), and has worked with the ISO SQL3 committee to add temporal support to that language. He authored Developing Time-Oriented Database Applications in SQL (Morgan Kaufmann), was a coauthor of Advanced Database Systems (Morgan Kaufmann), and was a coeditor of Temporal Databases: Theory, Design, and Implementation (Benjamin/Cummings). He codirects TimeCenter, an international center for the support of temporal database applications on traditional and emerging DBMS technologies. His research interests include ergalics (the science of computation), temporal databases, query language design, query optimization and evaluation, storage structures, and database design. He is a senior member of the IEEE and the IEEE Computer Society.

Stephen W. Thomas received the BS degree in computer science from New Mexico State University in 2006 and the MS degree in computer science from the University of Arizona in 2009. He is currently pursuing the PhD degree in computer science from Queen's University in Canada. His research interests include temporal databases, text mining, and empirical software engineering.

Rui Zhang received the BEng degree in computer science from the Nanjing University of Technology in 2004, and the MSc degree in computer science from the University of Nebraska at Omaha in 2006. He is currently a PhD candidate in the Department of Computer Science at the University of Arizona. His interests include Database technologies and XML processing.

APPENDIX I ALGORITHMS

In this appendix we provide the major algorithms for checking sequenced and non-sequenced constraints. This material elaborates on the summary of the τXMLLINT implementation provided in the paper.

```
CheckSequencedConstraints($constraints$,
                          $temporal\_document$):
foreach $constraint$ in $constraints$ do
  $temporal\_document \leftarrow$
        $FilterTemporalDocument($
              $constraint, temporal\_document)$
  foreach $t$ in $ExtractTimePoints($
              $temporal\_document)$ do
      $slice \leftarrow ExtractSlice(t, temporal\_document)$
      if $\neg Validate(slice)$ then
          return $false$
      end if
  end
end
return $true$
```

Algorithm 1: Checking Sequenced Constraints

Algorithm 1 first shrinks the input $temporal\_document$ by removing irrelevant nodes in the $FilterTemporalDocument$ routine, resulting in a small fraction of the original document. The relevant nodes are determined by the constraints, particularly the values of the `selector` and `field` elements. The nodes in the document that are not referenced by these elements will not be kept in the filtered document. It then iterates through the time points, which are the times when changes occur in the document, computed by routine $ExtractTimePoints$.

```
ExtractSlice($at\_time$, $node$):
if $IsVersionNode(node)$ then
  foreach $child$ in $node.getChildNodes()$ do
      if $child.begin\_time \leq at\_time \wedge$
            $at\_time < child.end\_time$ then
          return $ExtractSlice(at\_time, child)$
      end
  end
  return $nil$
else
  $slice \leftarrow node$
  foreach $child$ in $node.getChildNodes()$ do
      $extract \leftarrow ExtractSlice(at\_time, child)$
      if $extract \neq nill$ then
          $slice.addChild(extract)$
      end if
  end
  return $slice$
end
```

Algorithm 2: Slicing

A recursive routine $ExtractSlice$, presented as Algorithm 2, is then invoked to extract the corresponding slice. The versions within a version node are assumed to be contiguous; hence, only one child will ever be extracted. If *nil* is returned from the root, then there is a problem with the timestamps and τXMLLINT will return false.

After the slices are extracted, the routine $Validate$ is invoked to verify the sequenced constraints. In our current implementation, we utilize the DOM based validation facility provided by Java.

The number of $time\_periods$ (slices) and the size of each slice dominate the overall complexity of checking sequenced constraints. Since the $ExtractSlice$ routine traverses the entire document tree, its complexity in the worst case (when at_time is greater than all the $begin_time$s) is O(n), where n is the number of nodes in the document. Assuming the number of time points in the temporal document is m, the complexity of Algorithm 1 is O($n \cdot m$).

```
CheckNonSequencedConstraints($constraints$,
                             $temporal\_document$):
foreach $constraint$ in $constraints$ do
  $evaluation\_windows \leftarrow GetEvaluationWindows($
        $constraint.evaluation\_window\_size,$
        $constraint.slide\_size, temporal\_document)$
  foreach $eval\_window$ in $evaluation\_windows$ do
      $results \leftarrow ExtractNodes($
                  $eval\_window,$
                  $constraint.identifier,$
                  $constraint.xpath,$
                  $temporal\_document)$
      if $\neg ValidateSpecificConstraint($
              $constraint, results)$ then
          return $false$
      end
  end
end
return $true$
```

Algorithm 3: Checking Non-Sequenced Constraints

```
ExtractNodes($evaluation\_window, identifier,$
             $xpath\_query, temporal\_document$):
$result \leftarrow new\ map$
$candidate\_nodes \leftarrow XPath.Evaluate($
      $identifier, xpath\_query, temporal\_document)$
$candidate\_nodes \leftarrow FilterNodesbyEvalWindow($
            $evaluation\_window, candidate\_nodes)$
foreach $can\_node$ in $candidate\_nodes$ do
  $result.add(can\_node.identifier, can\_node.value)$
end
return $result$
```

Algorithm 4: Extracting Related Nodes for Non-Sequenced Constraints

Algorithm 3 validates all the non-sequenced constraints. For each constraint, its evaluation windows are first computed based on (i) the period of the temporal document, (ii) the evaluation window size (the length of each period in the temporal document during which this constraint applies), and

(iii) the slide size (the distance between the begin times of successive evaluation windows).

$ExtractNodes$ extracts only the nodes from the temporal documents that are relevant to a constraint, within each evaluation window. We discuss this routine shortly. $ValidateSpecificConstraint$ is then utilized to examine whether the extracted nodes violate the constraint. This routine, which checks all four types of non-sequenced constraints, is straightforward. For a cardinality constraint, the routine groups each distinct key and accumulates the count of occurrences of each key. Similarly, for a unique constraint, distinct keys are grouped. But in this case, if a key's count is more than one, this constraint is considered to be violated. In checking a referential constraint, the routine evaluates the XPath expression of the conventional constraint that is referenced by the temporal constraint. The existence of the values of the nodes that are being checked is examined against the values returned from evaluating the conventional constraint. Finally, in verifying a datatype constraint, each pair of consecutive values specified by the constraint are examined to determine whether the value transition rules are violated.

To provide the routine $ValidateSpecificConstraint$ with the proper input, routine $ExtractNodes$ (Algorithm 4) evaluates XPath expressions specified by the constraints and filters the document to produce the *items*, each identified by an *identifier*, to be validated by the non-sequenced constraints. As mentioned above, this algorithm performs XPath evaluation and document content filtering, returning a mapping from identifiers to the values associated with each identifier. By evaluating the XPath expressions from the constraints, a set of candidate nodes is extracted from the temporal document. These nodes are then processed to retain only the nodes in the current evaluation window. For each *candidate_node*, the distinct *identifiers* are grouped and stored in the *result* variable.

The complexity of Algorithm 4 is determined by the number of candidate nodes n in the document as well as the number of time points m. The overall complexity is thus O($n \cdot m$). The complexity of Algorithm 3 is determined primarily by the number of evaluation windows l as well as the complexity of Algorithm 4, and is thus O($l \cdot n \cdot m$).

This worst case behavior is consistent with the experimental results given in Section 9.3. Concerning Figure 5, for sequenced constraints, the number of nodes (n) and the number of evaluation windows (l) are both fixed, with the number of slices (m) varied on the x-axis, implying the observed linear growth in total execution time. Concerning Figure 6, for non-sequenced constraints, the number of nodes and number of evaluation windows are still fixed, except for cardinality constraints, which has a smaller window size, resulting in an increasing number of evaluation windows as the number of slices increases. Thus referential integrity and identity constraints exhibit linear behavior, while cardinality constraints exhibit quadratic behavior.

The implementation of the τXMLLINT tool can be downloaded (from http://cgi.cs.arizona.edu/apps/tauXSchema/).

APPENDIX
II CONSTRAINTS USED IN EVALUATION

During our evaluation, we used the following three non-sequenced constraints. To produce fair execution time results, when we evaluated one of the constraints, we deactivated the other two in the annotations document.

```
<!-- Non-sequenced Identify Constraint: -->
<!-- Item IDs are unique for books and may
<!-- not ever be re-used. -->
<item target="item">
  <nonSeqKey name="bookIDKey" dimension="validTime"
      evaluationWindow="36500">
    <selector xpath="." />
    <field xpath="@id" />
  </nonSeqKey>
</item>
```

```
<!-- Non-sequenced Referential Integrity: -->
<!-- A related item should refer to a valid
<!-- item (possibly not currently an item in print). -->
<item target="item">
  <nonSeqKeyref name="relatedItemRI" refer="itemID">
    <selector xpath="." />
    <field xpath="related_items//related_item//item_id" />
  </nonSeqKeyref>
  <itemIdentifier name="item_id"
      timeDimension="transactionTime">
    <field path="@id"/>
  </itemIdentifier>
</item>
```

```
<!-- Non-sequenced Cardinality: -->
<!-- In any calendar year, an item may have up
<!-- to 6 authors. -->
<item target="item">
  <nonSeqCardinality name="bookAuthorsNSeq" maxOccurs="6"
      dimension="validTime" evaluationWindow="365"
      slideSize="365">
    <selector xpath="." />
    <field xpath="authors//author/@author_id" />
  </nonSeqCardinality>
  <itemIdentifier name="item_id"
      timeDimension="transactionTime">
    <field path="@id"/>
  </itemIdentifier>
</item>
```

Additionally, we used the following four sequenced constraints. Again, to produce fair execution time results, when evaluating one of the constraints, we deactivated the other three in the schema.

```
<!-- Sequenced Cardinality: -->
<!-- An item must have between 1 and 4 authors. -->
<xs:element ref="author" minOccurs="1" maxOccurs="4"/>
```

```
<!-- Sequenced Identify Constraint: -->
<!-- Item ISBNs are unique. -->
<xs:unique name="ISBNUnique">
  <xs:selector xpath=".//item/attributes"/>
  <xs:field xpath="ISBN"/>
</xs:unique>
```

```
<!-- Sequenced Referential Integrity: -->
<!-- A related item should refer to a valid item -->
<xs:key name="itemID">
  <xs:selector xpath=".//item"/>
  <xs:field xpath="@id"/>
</xs:key>
<xs:keyref name="itemIDRef" refer="itemID">
  <xs:selector xpath=".//item/related_items/related_item"/>
  <xs:field xpath="item_id"/>
</xs:keyref>
```

```
<!-- Sequenced Datatype: -->
<!-- The number_of_pages must be of type short. -->
<xs:element name="number_of_pages" type="xs:short"/>
```