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 nonsequenced 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 XMLEXIT, 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. Second, 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 subelements <name> and <ssn>, and attributes ID and email. An <order> is a subelement 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., XMLEXIT [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.2.xml, data.3.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.

Our design of an upward-compatible extension of XML Schema, τXSchema [4] addresses the first two concerns emphasized in the previous paragraph. τXSchema supports 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].

Listing 1. company.xsd
The challenge addressed by the present paper is how to accommodate both conventional XML integrity constraints, including the identity, referential, cardinality, and data type 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.)

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 time stamps 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).

Listing 2. data.xml
Listing 3. data.2.xml
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 are 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 data type 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 $\tau$XSchema. 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 compatibility 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
Another tool called SCHEMAMAPPER. This schema, instead generated automatically from the temporal schema by are described with a "Representational Schema" (box 9), separate document. The versions of the temporal document in the nontemporal data, which represents each slice as a elements. This removes a great deal of redundancy found

Temporal granularity, the resolution level at which each annotations also provide a user the means to specify 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 time stamp representation options chosen by the user. These annotations define where the physical time stamps will be placed (versioning level). The location of the time stamps 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 time stamp 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 time stamps 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 time stamp is maintained.

The temporal schema (box 4) ties the schema, logical annotations and physical annotations together. This document contains subelements that associate a series of conventional 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 time stamp representation options chosen by the user. These annotations define where the physical time stamps will be placed (versioning level). The location of the time stamps 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 time stamp 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 time stamps 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 time stamp is maintained.

The temporal schema (box 4) ties the schema, logical annotations and physical annotations together. This document contains subelements that associate a series of conventional schema with logical and physical annotations, along with the time span during which the association was in effect.

The figure shows a tool called SQUASH that can render a temporal document (box 8) consistent with the logical and physical annotations. Hence, the time stamps are spread out across the document, associated with versions of the elements. This removes a great deal of redundancy found in the nontemporal 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 reused by another employee to avoid confusion or problems redirecting emails after the second change). Using XML Schema alone, we cannot specify nor validate such constraints.

Instead, the designer can utilize torchXMLSchema LOCAL as a tool 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). torchXMLINT 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 whether it is valid. Analogously, torchXMLINT takes as input a single temporal document referencing a temporal schema. torchXMLINT validates the temporal document and reports either success or the errors encountered.

The validation using torchXMLINT 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 torchXMLINT should also report an error at time t.

With this high-level overview of torchXMLSchema (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 data type, cardinality, identity, and referential integrity constraints. These are conventional constraints and restrict specific XML document. In this paper, we extend these constraints in turn with temporal semantics.

Data type constraints restrict the content of the corresponding element or attribute. A data type 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> subelements 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 subelements 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 Toward 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, nonsequenced, and current [27]. All combinations are appropriate and useful. One could contemplate, for example, a sequenced cardinality constraint or a nonsequenced 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 XSchema 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: “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 ≤ 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 and 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 nonsequenced 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 nonsequenced constraint: ss ≤ 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). Nonsequenced 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 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 time stamped 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 time stamp function that maps an XML data model instance to a time stamp (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 (Sliced). Let \(D = (X, T, S, A)\) be an instance of a temporal XML model. Then for \(t \in T\), \(\text{slice}(t, D) = X_t\), where \(X_t \in X\) and \(t \in S(X_t)\).

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 nonsequenced 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 nonsequenced 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 \(\text{Eval}(n, E, X)\) 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 \(\text{Eval}(n, L, X) = [\text{Eval}(n, E_1, X), \ldots, \text{Eval}(n, E_k, X)]\).

Since an XPath expression evaluates to a list of nodes, \(\text{Eval}(n, L)\) 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 \(<\text{emp}\>\) 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].
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 \( \text{Eval}(x, L, \text{slice}(i, D)) = \text{Eval}(y, L, \text{slice}(j, D)) \) 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 \( \text{Eval}(z, L, \text{slice}(k, D)) = \text{Eval}(x, L, \text{slice}(i, D)) \).

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, i.e., 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 constrained 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 type 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 nonsequenced 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 nonsequenced 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 \( X_t = \text{slice}(t, D) \)

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

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 nonsequenced <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 subelements within an <item> annotation). We adopt the usual distinction between key and unique constraints. The subelements and attributes of these nonsequenced constraints are listed 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 nonsequenced <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 nonsequenced constraint. We envision nonsequenced constraints being used in three ways.

1. Between—Consider the conventional unique constraint defined in Listing 1. Suppose a nonsequenced unique constraint is placed on the email address of an employee, with an evaluation window of a year
Then, no two employee items can have the same email address (for example) in any year, but the same employee (e.g., Dana) can switch from dana@txschema.com to ddeo@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 ddeo@txschema.com and back in a single year, we would need to define a nonsequenced 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 reuse an email, both constraints (Listing 6 and 7) are specified.

![Listing 6. Non-seq. constraint “between” employees](image)

Listing 7. Non-seq. constraint “within” each employee

A conventional identity constraint does not imply nonsequenced uniqueness (it only implies that there are no duplicates in a slice). Thus, the same productNo (a conventional key) can be reused for another product or changed between slices (for the same product, as long as it remains unique). To place nonsequenced restrictions on elements or attributes, we use nonsequenced unique and nonsequenced 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 be the item containing the <nonSeqKey> definition, let be the list of XPath expressions where is a field expression, let be the selector, and let is a temporal document. Then, for each window (a time period) , define where for each slice in the window.

![Listing 7. Non-seq. constraint “within” each employee](image)

### TABLE 1: Common Attributes and Subelements for Temporal Constraints

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the constraint</td>
<td>optional</td>
</tr>
<tr>
<td>dimension</td>
<td>validTime, transactionTime, or bitemporal (default: validTime)</td>
<td>optional</td>
</tr>
<tr>
<td>evaluationWindow</td>
<td>Time window over which the constraint should be checked (default: lifetime of document)</td>
<td>optional</td>
</tr>
<tr>
<td>slideSize</td>
<td>Size of the slide for successive evaluation windows (default: granularity of constrained data type); only used in conjunction with evaluationWindow</td>
<td>optional</td>
</tr>
<tr>
<td>&lt;applicability&gt;</td>
<td>When the constraint is applicable (default: lifetime of document)</td>
<td>[0:1] optional</td>
</tr>
<tr>
<td>(begin, end)</td>
<td>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</td>
<td></td>
</tr>
<tr>
<td>&lt;selector&gt;</td>
<td>For the definition of a new constraint. It is similar to the &lt;selector&gt; sub-element in the &lt;uniqueConstraint&gt; definition. It must be a relative XPath expression.</td>
<td>[0:1] optional</td>
</tr>
<tr>
<td>&lt;field&gt;</td>
<td>For the definition of a new constraint. It is similar to the &lt;field&gt; sub-element in the &lt;uniqueConstraint&gt; definition</td>
<td>[0:U] optional</td>
</tr>
</tbody>
</table>

### TABLE 2: Attributes for Temporal Unique Constraints <nonSeqUnique>, <nonSeqKey> and <uniqueNullRestricted>

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>conventionalIdentifier</td>
<td>The referenced conventional identifier of the conventional constraint being annotated (if present, the &lt;selector&gt; and &lt;field&gt; are omitted)</td>
<td>optional</td>
</tr>
<tr>
<td>scope</td>
<td>Either within or between (which is the default) semantics can be specified</td>
<td>optional</td>
</tr>
<tr>
<td>nullCountMin</td>
<td>The minimum number of null values allowable (used only within &lt;uniqueNullRestricted&gt;)</td>
<td>optional</td>
</tr>
<tr>
<td>nullCountMax</td>
<td>The maximum number of null values allowable (used only within &lt;uniqueNullRestricted&gt;)</td>
<td>optional</td>
</tr>
</tbody>
</table>
Definition (nonSeqKey, Within Semantics). To define a nonSeqKey within constraint, we replace the constraint given above with the following:

\[
\forall (e_i, t_i), (e_j, t_j) \in U(e, w) : [ \\
(\text{Eval}(e_i, F, X_i) = \text{Eval}(e_j, F, X_j) \land \\
\text{item}(e_i) = \text{item}(e_j)) \Rightarrow \\
\exists (e_k, t_k) \in U(e, w) : [t_i < t_k < t_j \land \\
\text{Eval}(e_i, F, X_i) \neq \text{Eval}(e_k, F, X_k)]],
\]

where \( X_i = \text{slice}(t_i, D) \), \( X_j = \text{slice}(t_j, D) \), and \( X_k = \text{slice}(t_k, D) \). 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 nonsequenced 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 of null values by allowing the user to specify a finite number (one or more) across time; the default number of null values by allowing the user to specify a finite number (one or more) across time. Setting the number of nulls allowed across time to 0 is equivalent to specifying a nonsequenced 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 reused at any later time. These constraints are applicable between 2005 and 2010.

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 nonsequenced 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 referential integrity constraint is applicable from 2008 to 2012, and no corresponding conventional constraint exists.
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.

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.

   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 grouping <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).

   Nonsequenced cardinality constraints can be used to restrict the cardinality over time. Consider the example of an <order> element in Listing 13. 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 nonsequenced 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.

Listing 8. Orders with an optional <deliveredOn>

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

   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 nonsequenced cardinality constraint can be used to place a limit of less than or equal to 1,500 <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

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>conventionalConstraint</td>
<td>The conventional referential integrity constraint being extended via this annotation (if present, the &lt;selector&gt; and &lt;field&gt; are omitted)</td>
<td>optional</td>
</tr>
<tr>
<td>refer</td>
<td>The referenced identifier; the corresponding dimension attribute must be compatible.</td>
<td>optional</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>newOnly</td>
<td>specifies whether only changes to “new” values should be considered (default &quot;false&quot;); only for &lt;nonSeqCardinality&gt;</td>
<td>optional</td>
</tr>
<tr>
<td>minOccurs</td>
<td>default: &quot;0&quot;</td>
<td>optional</td>
</tr>
<tr>
<td>maxOccurs</td>
<td>default: &quot;unbounded&quot;</td>
<td>optional</td>
</tr>
<tr>
<td>&lt;group&gt;</td>
<td>The level at which grouping is performed relative to the &lt;selector&gt; (default: self)</td>
<td>optional</td>
</tr>
</tbody>
</table>
four are distinct. To count each change, the user would set 

changes="newOnly," * otherwise all changes are counted.

We represent a nonsequenced 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 
<br>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 

<selector>). Combinations of <field>s are counted for each 
<br>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.

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 to 2011.

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 

\[
A(c, w) = \{(t, \text{item}(x), \text{item}(y), \text{Eval}(y, F, \text{slice}(t, D)))| t \in w \\land x \in \text{Eval}(c, S, \text{slice}(t, D)) \land y \in \text{Eval}(x, G)\}.
\]

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, \}, \ldots, (t_k, s_k, g_k, v_k)\}. From this set we can extract tuples that 
represent a change as follows:

\[
\text{Changes}(A(c, w)) = \{(t, s, g, v)|(t, s, g, v) \in A(c, w) \\land \neg \exists k=[t-1] \land (k, s, g, v) \in A(c, w)\}.
\]

While Changes(A(c, w)) extracts all changes, we are some-
times 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 

\[
\text{NewChanges}(A(c, w)) = \{(t, s, g, v)|(t, s, g, v) \in A(c, w) \\land \neg \exists k=[t-1] \land (k, s, g, v) \in A(c, w)\}.
\]

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

\[
\text{Count}(A(c, w)) = \{(s, \text{card}(|\{(t, s, g, u)\}|)\} | (t, s, g, u) \in \text{Changes}(A(c, w))\},
\]
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:

\[
\neg \exists s \forall x: (s, x) \in \text{Count}(A(c, w)) \land (x < \text{min} \lor \text{max} < x).\]

A nonsequenced cardinality constraint differs from a se-
quenced 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 nonsequenced cardinality constraint, 

but changes resulting in null values are counted rather 
than all changes.

6.4 Data Type 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 nonsequenced augmentations of such 
simple types. A nonsequenced equivalent of this type of 
constraint can be considered either at the schema level (i.e., 
evolution of the data type 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 data type 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 nonsequenced 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 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 data type 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.

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

### TABLE 5

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;valuePair&gt;</td>
<td>Sub-element listing possible pairs for discrete changes (only if &lt;valueEvolution&gt; not present)</td>
<td>(0,\infty)</td>
</tr>
<tr>
<td>&lt;old&gt;, &lt;new&gt;</td>
<td>Sub-elements of valuePair</td>
<td>(1,1)</td>
</tr>
<tr>
<td>&lt;valueEvolution&gt;</td>
<td>Sub-element specifying direction of continuous changes (only if &lt;valuePair&gt; not present)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(direction)</td>
<td>can be one of: &quot;LT&quot; (less than), &quot;GT&quot; (greater than), &quot;GE&quot; (greater or equal), &quot;LE&quot; (less or equal), &quot;EQ&quot; (equal) and &quot;NE&quot; (not equal)</td>
<td>required</td>
</tr>
</tbody>
</table>

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 \( \gamma \)XSchem. 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 now 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
If the schema changed on July 1 for the second half.

Concerning Section 6.3: new constraints. For example, consider the first example in deletes all the old constraints and then adds them back as on any such schema change. In effect the schema change constraint. The nonsequenced constraints are "restarted" schema change to any of the items involved in the constant period in which it is defined, if there were a to apply a nonsequenced constraint only within the schema-defined.

We considered two alternatives for the applicability of a nonsequenced 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 nonsequenced 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 nonsequenced 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 nonsequenced 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 \( e_c(n_i, n_j, T_e) \), if \( n_j \) is an attribute of type REF, such that there exists a reference edge \( e_r(n_j, n_k, T_e) \), then \( T_e = T_r \) holds.” As discussed in Section 5.1, in our model a nontemporal 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 nonsequenced referential integrity constraint (also discussed in Section 5.1). Our design is more uniform in that we utilize a per-slice semantics for all nontemporal constraints when applied to a temporal document, permitting the user to specify additional nonsequenced variants of such nontemporal 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 \( e_c(n_i, n_j, T_e) \) be a reference edge. Then, \( T_e \subseteq lifespan(n_j) \) holds.” This states that a reference edge applies in a subset of the slices in which the designation node exists, which is a quite specific kind of nonsequenced referential-integrity constraint. Again, we prefer a per-slice semantics for referential integrity, as with all other explicit nontemporal integrity constraints, while allowing the user to specify additionally nonsequenced 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

\( \tau \)Schema provides tools to construct and validate temporal documents, including an extension of XMLLint [3]. As discussed in Section 3, to validate a temporal document, \( \tau \)XMLLint 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 nonsequenced 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 representa-
tional schema at all [34].

Consider, for example, the (sequenced) employeeID-
Key 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 represen-
tational 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 \( t_3 \), 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 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, data type constraints can be expressed in
the representational schema since, by the definition of
sequenced constraints, the type must be constant through-
out 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, \( \tau \)XMLLINT.

9.2 \( \tau \)XMLLINT Implementation

We have implemented \( \tau \)XMLLINT in Java and DOM. Our
tool supports the entire constraint language presented here,
including sequenced and nonsequenced constraints and all
of the constructs summarized in Tables 1, 2, 3, 4, and 5. In
this section, we summarize our approach; the online
Appendix, which can be found on the Computer Society
Digital Library at http://doi.ieeecomputersociety.org/
10.1109/TKDE.2011.74, provides the detailed algorithms.

\( \tau \)XMLLINT 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
conducted based on a quite smaller DOM tree, significantly
improving the performance of constraint checking.

For sequenced constraints, \( \tau \)XMLLINT performs conven-
tional validation with the help of the validate() method
provided by the Validator class in the Java Platform.
Specifically, \( \tau \)XMLLINT 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.
\( \tau \)XMLLINT indicates that the temporal document is valid
only if validate() returns true for every slice.

For nonsequenced constraints, \( \tau \)XMLLINT provides its
own validation algorithm for each of the type of constraint.
As described in Section 3, the logical annotations provide
the constraint definitions. \( \tau \)XMLLINT extracts all the
declared 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 evalua-
tion window, slide size, and their interaction with applic-
ability (see Table 1). For identity constraints, \( \tau \)XMLLINT
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. \( \tau \)XMLLINT 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 \( \tau \)XMLLINT, focusing on
how the validation time of a temporal document with
\( \tau \)XMLLINT compares with validation time of multiple
slices, each a conventional XML document, with XMLLINT.
To do so, we use a benchmark data set defined in \( \tau \)Bench,
which is a benchmark for temporal data [35]. τBench is based on XBench, which is a family of nontemporal 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 Data set 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 6.5 MB, for a total size of 325 MB (50 slices) to 5.2 GB (800 slices). The temporal document ranged in size from 13 (50 slices) to 31 MB (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 time stamps 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 nonsequenced constraints with τXMLLint. The second study was to examine the behavior of nonsequenced checks (τXMLLint being the first such validator to implement constraints). We performed both studies on a machine running Ubuntu 9.10 with a 2.8 GHz 16-core CPU and 64 GB of memory. We evaluated each type of constraint; the online Appendix, available in the online supplemental material, 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 nonsequenced 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 the given constraint (i.e., I/O is omitted). For such constraints, Fig. 6a suggests that the running time is dominated by I/O for nonsequenced constraints as well. Fig. 6b emphasizes that cardinality constraints require greater CPU time. This is due to the fact that for the other nonsequenced 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 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 nonsequenced, and reinterprets existing nontemporal 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, $\tau$XMLLINT, 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 $\langle$nonSeqUnique$\rangle$ (or $\langle$nonSeqKey$\rangle$) 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 nonsequenced 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 nonadjacent 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 $\langle$item$\rangle$ annotates each element type definition. It would be interesting to relax this restriction to permit several $\langle$items$\rangle$ 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 constraint.

Finally, many optimizations could be applied to the validator. For example, checking constraints of the same type, such as $\langle$nonSeqUnique$\rangle$ 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


Faiz A. Currim received the PhD degree from the University of Arizona, and was a professor at the University of Iowa prior to returning to Arizona. He is with the Department of Management Information Systems at the University of 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 A. Currim received the PhD degree from the University of Arizona. She is a senior data warehouse analyst in the Mosaic Project at the University of Arizona. Her research interests include conceptual data modeling, learning, database design and management, data warehouse, XML Schema management and IT Governance. She is a member of the IEEE.

Curtis E. 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, he 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 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 ACM SIGMOD Conference. He received the 2004 Outstanding Contribution to ACM Award and the 2002 ACM SIGMOD Contributions Award. He is currently 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 evaluation, storage structures, and database design. He is an ACM fellow. 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 and PhD degree in computer science from the University of Arizona in 2009. He is currently working toward the PhD degree in computer science from Queen’s University in Canada. His research interests include temporal databases, query language design, query evaluation, storage structures, and database design. He is a member of the IEEE.

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 working toward the PhD degree in the Department of Computer Science at the University of Arizona. His interests include database technologies and XML processing.