τXSchema: Support for Data- and Schema-Versioned XML Documents

Faiz Currim, Sabah Currim, Curtis E. Dyreson, Shailesh Joshi, Richard T. Snodgrass, Stephen W. Thomas, Eric Roeder

September 8, 2009

TR-91

A TIMECENTER Technical Report

Title	τXSchema: Support for Data- and Schema-Versioned XML Documents
Author(s)	Faiz Currim, Sabah Currim, Curtis E. Dyreson, Shailesh Joshi, Richard T. Snodgrass, Stephen W. Thomas, Eric Roeder
Publication History	September 2009, a TIMECENTER Technical Report

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Simonas Šaltenis, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Sudha Ram

Individual participants
Yun Ae Ahn, Chungbuk National University, Korea; Michael H. Böhlen, Free University of Bolzano, Italy; Curtis E. Dyreson, Utah State University, USA; Dengfeng Gao, IBM Silicon Valley Lab, USA; Fabio Grandi, University of Bologna, Italy; Vijay Khatri, Indiana University, USA; Nick Kline, Microsoft, USA; Gerhard Knolmayer, University of Bern, Switzerland; Carme Martín, Technical University of Catalonia, Spain; Thomas Myrach, University of Bern, Switzerland; Kwang W. Nam, Chungbuk National University, Korea; Mario A. Nascimento, University of Alberta, Canada; John F. Roddick, Flinders University, Australia; Keun H. Ryu, Chungbuk National University, Korea; Dennis Shasha, New York University, USA; Paolo Terenziani, University of Piemonte Orientale "Amedeo Avogadro," Alessandria, Italy; Vassilis Tsotras, University of California, Riverside, USA; Fusheng Wang, Siemens, USA; Jef Wijsen, University of Mons-Hainaut, Belgium; and Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.aau.dk/TimeCenter>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Contents

List of Figures

List of Tables

Listings

Abstract

The W3C XML Schema recommendation defines the structure and data types for XML documents. XML Schema lacks explicit support for time-varying XML documents or for time-varying schemas. An XML document evolves as it is updated over time or as it accumulates from a streaming data source. A temporal document records the entire history of a document rather than just its current state or snapshot. Capturing a document's evolution is vital to providing the ability to recover past versions, track changes over time, and evaluate temporal queries. Capturing the evolution of a document's schema is similarly important. To date, users have to resort to ad hoc, non-standard mechanisms to create schemas for time-varying XML documents and to deal with evolving schemas.

This report presents a data model and architecture, called τXSchema, for constructing and validating temporal XML documents through the use of a temporal schema. A temporal schema guides the construction of a temporal document and is essential to managing, querying, and validating temporal documents. The temporal schema consists of a non-temporal (conventional) schema, logical annotation(s), and physical annotation(s). The annotations specify which portion(s) of an XML document can vary over time, how the document can change, and where timestamps should be placed. These components can themselves individually evolve over time. The advantage of using annotations to denote the time-varying aspects is that logical and physical data independence for temporal schemas can be achieved while remaining fully compatible with both existing XML Schema documents and the XML Schema recommendation. This report also describes how to construct a temporal document by "gluing" individual snapshots into an integrated history.

This technical report is divided into three parts: concerning *instance versioning*, extending to *schema versioning*, and reviewing the entire τXSchema language. The first two parts have a parallel structure. Each begins by discussing relevant related work before providing a motivating example that illustrates the challenges of instance and schema versioning, respectively, then lists design decisions made in τXSchema concerning that challenge. Theoretical considerations (separately for instance and schema versioning), architectural considerations, and implementation details are discussed in that order in each of the two parts. Each part ends with full example schema and instance documents. The third part completes the picture with a discussion of related work and research topics to be considered in the future.

1 Introduction

XML is becoming an increasingly popular language for documents and data. XML can be approached from two quite separate orientations: a *document-centered* orientation (e.g., HTML) and a *data-centered orientation* (e.g., relational and object-oriented databases). Schemas are important in both orientations. A schema defines the building blocks of an XML document, such as the types of elements and attributes. An XML document can be *validated* against a schema to ensure that the document conforms to the formatting rules for an XML document (is well-formed) and to the types, elements, and attributes defined in the schema (is valid). A schema also serves as a valuable guide for querying and updating an XML document or database. For instance, to correctly construct a query, e.g., in XQuery, a user will (usually) consult the schema rather than the data. Finally, a schema can be helpful in query optimization, e.g., in constructing a path index [54].

Several schema languages have been proposed for XML [49]. From among these languages, XML Schema is the most widely used. The syntax and semantics of XML Schema 1.0 are W3C recommendations [80, 78].

Time-varying data naturally arises in both document-centered and data-centered orientations. Consider the following wide-ranging scenarios. In a university, students take various courses in different semesters. At a company, job positions and salaries change. At a warehouse, inventories evolve as deliveries are made and good are shipped. In a hospital, drug treatment regimes are adjusted. And finally at a bank, account balances are in flux. In each scenario, querying the current state is important, e.g., "how much is in my account right now", but it also often useful to know how the data has changed over time, e.g., "when has my account been below $200".

A temporal document records the evolution of a document over time, i.e., all of the versions of the document. Capturing a document's evolution is vital to supporting time travel queries that delve into a past version [70] and incremental queries that involve the changes between two versions.

In this report we consider how to accommodate time-varying data within XML Schema. An obvious approach would have been to propose changes to XML Schema to accommodate time-varying data. Indeed, that has been the approach taken by many researchers for the relational and object-oriented models [56, 62, 71]. As we will discuss in detail, that approach inherently introduces difficulties with respect to document validation, data independence, tool support, and standardization. So in this report we advocate a novel approach that retains the non-temporal XML Schema for the document, utilizing a series of separate schema documents to achieve data independence, enable full document validation, and enable improved tool support, while not requiring any changes to the XML Schema standard (nor subsequent extensions of that standard; XML Schema 1.1 is in development).

We present a system, called *Temporal XML Schema*, or τXSchema, for constructing and validating temporal documents. τXSchema reuses XML Schema while adding the ability to designate which element types can vary over time. A temporal schema also describes how to associate time-varying elements across snapshots and provides some temporal constraints that broadly characterize how an element can change over time.

Coupled with the τXSchema schema specification language is an architecture and associated tools for constructing schemas for temporal XML documents. A temporal document records the evolution of a document over time, i.e., all of the versions of the document. τXSchema has a three-level architecture for specifying a schema for time-varying data . The first level is the schema for an individual version, called the *conventional schema* or *snapshot schema* . The snapshot schema is a conventional XML Schema document. The second level is the *logical annotations* of the conventional schema, which identify which elements can vary over time. For those elements, the logical annotations also effect a temporal semantics to the various integrity constraints (such as uniqueness) specified in the conventional schema. The third level is the *physical annotations*. The physical annotations describe how the time-varying aspects are represented.

The conventional schema and logical and physical annotations are collected together in an XML document termed a temporal schema. Similarly, the individual time slices are combined into one XML document, termed the temporal document, which serves as the XML instance. A temporal validator takes a temporal schema and a temporal document and validates both.

Each annotation can be independently changed, so the architecture exhibits *logical and physical data independence* [14]. Data independence allows XML documents using one representation to be automatically converted to a different representation while preserving the semantics of the data. τXSchema is accompanied with a suite of auxiliary tools to manage temporal documents and schemas. There are tools to convert a temporal document from one physical representation to a different representation, to extract a time slice from that document (yielding a conventional static XML document), and to create a temporal document from a sequence of static documents, in whatever representation the user specifies.

As mentioned, τXSchema *reuses* rather than extends XML Schema. τXSchema is consistent and compatible with both XML Schema and the XML data model. In τXSchema, a temporal validator augments a conventional validator to more comprehensively check the validity constraints of a document, especially temporal constraints that cannot be checked by a conventional XML Schema validator. We describe a means of validating temporal documents that ensures the desirable property of snapshot validation subsumption. We show in Section 18.7 how a temporal document can be smaller and faster to validate than the associated XML snapshots.

τXSchema focuses on both *instance versioning* (representing a time-varing sequence of XML instance documents) and *schema versioning* (representing a time-varying schema document [33, 69]). The schema can describe which aspects of an instance document change over time; this schema can itself be a temporal (time-varying) document. The temporal schema references or contains these annotations. All three components, (1) the conventional schema, (2) the logical annotations, and (3) the physical annotations, can change over time. The temporal validator and associated tools are able to contend with both instance and schema versioning.

Intensional XML data (also termed dynamic XML documents [1]), that is, parts of XML documents that consist of programs that generate data [58], are gaining popularity. Incorporating intensional XML data is beyond the scope of this report.

While this report concerns temporal XML Schema, we feel that the general approach of separate logical and physical annotations is applicable for introducing temporal aspects to other data models, such as UML [61]. The contribution of this report is two-fold: (1) introducing a three-level approach for logical data models and (2) showing in detail how this approach works for XML Schema in particular, specifically concerning a theoretical definition of snapshot validation subsumption for XML, validation of temporal XML documents, and implications for tools operating on realistic XML schemas and data, thereby exemplifying in a substantial way the approach. While we are confident that the approach could be applied to other data models, designing the annotation specifications, considering the specifics of data integrity constraint checking, and ascertaining the impact on particular tools for a different data model remain challenging (and interesting) tasks.

This technical report is divided into three parts. The first part concerns *instance versioning*; the second part extends the approach to support *schema versioning*; and the last part summarizes the entire τXSchema language.

The first two parts have a parallel structure. Each begins by discussing relevant related work before providing a motivating example that illustrates the challenges of instance and schema versioning, respectively, then lists design decisions made in τXSchema concerning that challenge. Theoretical considerations (separately for instance and schema versioning), architectural considerations, and implementation details are discussed in that order in each of the two parts. Each part ends with full example schema and instance documents.

Part III completes the picture with a discussion of related work and research topics to be considered in

the future. It ends with a summary of the τXSchema language design, detailing all of the new elements and attributes defined in the language or in instance documents that are instance- or schema-varying.

An appendix gives the four XML schemas that in concert with the tools comprise τXSchema. A summary of these schemas and the semantics of their items is in the τXSchema Reference.

Part I
Supporting Temporal XML Documents

In this part, we consider how to support *data* versioning, that is, a time-varying XML document. The next part will consider *schema versioning*, in which the schema also varies over time.

We first provide a motivating example, that of the history of the Winter Olympic games. Such a document changes through the adding of new information valid at a later time and the correction of previous information.

We then consider prior work in schemas for XML and in temporal databases.

Section 4 provides terminology, design desiderata and goals, and comprehensive discussion of the overarching design decisions, illustrated with a Company XML document.

We then turn to a deeper discussion of the language design, first through a theoretical framework and then through a detailed examination of extending constraints in XML schemas into temporal constraints. Section 7 considers adding transaction time to the mix and the following sections describe the overall architecture and the various tools that support τXSchema and the spectrum of possible representations of time-varying data.

We end with full listings of the WinOlympic and Company examples.

2 Motivation

Over a decade of work has been invested in the development of XML Schema. Before we undertake the task of presenting an approach to extend XML Schema, we feel it is important to consider its ability to support temporal data. In this section, we discuss whether conventional XML Schema is appropriate and satisfactory for time-varying data. We present an example that illustrates how a time-varying document differs from a conventional XML document. We then pinpoint some of the limitations of the XML Schema in supporting temporal documents and data. This allows us to motivate desired properties of schemas for time-varying documents. We end with a discussion of some real world applications that would benefit from document and schema versioning supported by the τXSchema framework.

Assume that the history of the Winter Olympic games is described in an XML document called `winter.xml`. The document has information about the athletes that participate, the events in which they participate, and the medals that are awarded. Over time the document is edited to add information about each new Winter Olympics and to revise incorrect information. Assume that information about the athletes participating in the `2002` Winter Olympics in Salt Lake City, USA was added on `2002-01-01`. On `2002-03-01` the document was further edited to record the medal winners. Finally, a small correction was made on `2002-07-01`.

To depict some of the changes to the XML in the document, we focus on information about the Norwegian skier Kjetil Andre Aamodt. On `2002-01-01` it was known that Kjetil would participate in the games and the information shown in Listing 1 was added to `winter.xml`. Kjetil won a medal; so on `2002-03-01` the fragment was revised as shown in Listing 2. The edit on `2002-03-01` incorrectly recorded that Kjetil won a silver medal in the Men's Combined; Kjetil won a gold medal. Listing 3 shows the correct medal information.

Listing 1: A fragment of `winter.xml` on `2002-01-01`

```
...
 <athlete>
   <athName>Kjetil Andre Aamodt</athName>
 </athlete>
...
```

Listing 2: Kjetil won a Silver medal, as of `2002-03-01`

```
...
 <athlete>
   <athName>Kjetil Andre Aamodt</athName> won a medal in
   <medal mtype="silver">Men's Combined</medal>
 </athlete>
...
```

Listing 3: Kjetil won a Gold medal, as of `2002-07-01`

```
...
 <athlete>
   <athName>Kjetil Andre Aamodt</athName> won a medal in
   <medal mtype="gold">Men's Combined</medal>
 </athlete>
...
```

A *t*ime-varying document records a *v*ersion history, which consists of the data in each version, along with the timestamps indicating the lifetime of that version. Listing 4 shows a fragment of the time-varying document that captures the history of Kjetil. The fragment is *compact* in the sense that each edit results in a small, localized change to the document. In Listing 4 the *transaction-time* lifetimes of each element are represented with an optional `<tv:timestamp_TransExtent>` sub-element. If the timestamp is missing,

the element has the same lifetime as its enclosing element. For example, there are two `<athlete>` elements with different lifetimes since the content of the element has changed. The last version of `<athlete>` has two `<medal>` elements because the medal information is revised. There are many different ways to represent the versions in a time-varying document; the methods differ in which elements are timestamped, how the elements are timestamped, and how changes are represented (e.g., perhaps only differences between versions are represented).

Listing 4: Snippet of a Temporal Document

```
...
<athlete_RepItem>
  <athlete_Version>
    <tv:timestamp_TransExtent begin="2002-01-01" end="2002-03-01"/>
      <athlete>
        <athName>Kjetil Andre Aamodt</athName>
      </athlete>
  </athlete_Version>
  <athlete_Version>
    <tv:timestamp_TransExtent begin="2002-03-01" end="9999-12-31"/>
      <athlete>
        <athName>Kjetil Andre Aamodt</athName>won a medal in
        <medal_RepItem>
          <medal_Version>
            <tv:timestamp_TransExtent begin="2002-03-01" end="2002-07-01"/>
            <medal mtype="silver">Men's Combined</medal>
          <medal_Version>
          <medal_Version>
            <tv:timestamp_TransExtent begin="2002-07-01" end="9999-12-31"/>
            <medal mtype="gold">Men's Combined</medal>
          <medal_Version>
        </medal_RepItem>
      </athlete>
  </athlete_Version>
</athlete_RepItem>
...
```

Keeping the history in a document or data collection is useful because it provides the ability to recover past versions, track changes over time, and evaluate temporal queries [36]. But it also changes the nature of validation against a schema. Assume that the file `winOlympic.xsd` contains the *conventional schema* for `winter.xml`. The conventional schema is the schema for an individual version. The conventional schema is a valuable guide for editing and querying individual versions. A fragment of the schema is given in Listing 5. Note that the schema describes the structure of the fragment shown in Listing 1, Listing 2, and Listing 3. The problem is that although individual versions conform to the schema, the time-varying document does not. So `winOlympic.xsd` cannot be used (directly) to validate the time-varying document of Listing 4.

Listing 5: `winOlympic.xsd`

```
...
<element name="athlete">
  <complexType mixed="true">
    <sequence>
      <element name="athName" type="string"/>
      <element ref="medal" minOccurs="0" maxOccurs="unbounded"/>
      <element name="birthPlace" type="string" minOccurs="0"
        maxOccurs="1"/>
    </sequence>
    <attribute name="age" type="nonNegativeInteger" use="optional"/>
  </complexType>
</element>
...
```

The conventional schema could be used *indirectly* for validation by individually reconstituting and validating each version. But validating every version can be expensive if the changes are frequent or the document is large (e.g., if the document is a database). While the Winter Olympics document may not change often, contrast this with, for instance, a Customer Relationship Management database for a large company. Thousands of calls and service interactions may be recorded every day. This would lead to a very large number of versions, making it expensive to instantiate and validate each individually. The number of versions could further be increased by the presence of both valid and transaction time.

To validate a time-varying document, a new, different schema is needed. The schema for a time-varying document should take into account the elements (and attributes) and their associated timestamps, specify the kind(s) of time involved, provide hints on how the elements vary over time, and accommodate differences in version and timestamp representation. Since this schema will express how the time-varying information is *represented*, we call it the *representational schema*. The representational schema will be related to the underlying conventional schema (Listing 5), and will allow the time-varying document to be validated using a conventional XML Schema validator (though not fully, as discussed in the further sections). The representational schema will also be important in constructing, evaluating, and optimizing temporal queries. Both the person who is formulating a query and the database need to know which elements in the document are time-varying elements since additional operations, like temporal slicing, are applicable to the temporal elements. Thus the schema language should have some capability of designating time-varying elements.

Finally, time-varying elements can have additional constraints. For instance, it might be important to stipulate that an athlete can win only a single medal in an event, although the existence and/or type of medal may change over time (for instance if the athlete is disqualified). The *valid time* component of this constraint is that only one medal appears in an `<athlete>` element at any point in time. But the *transaction time* component of the constraint is that multiple versions can be present (as the element is modified). A schema language for a temporal document needs to have some way of specifying and enforcing such constraints.

The conventional XML Schema validator is also *incapable* of fully validating a time-varying document using the representational schema. First, XML Schema is not sufficiently expressive to enforce *temporal constraints*. For example, XML Schema cannot specify the following (desirable) schema constraint: the transaction-time lifetime of a `<medal>` element should always be contained in the transaction-time lifetime of its parent `<athlete>` element. Second, a conventional XML Schema document augmented with timestamps to denote time-varying data cannot, in general, be used to validate a snapshot of a time-varying document. A snapshot is an instance of a time-varying document at a single point in time. Consider a temporal document with timestamps for the lifespan of the parent and child elements. If the schema asserts that a child element is mandatory (`minOccurs=1`), there is no way to ensure that the element is in every snapshot given that the element's timestamp may indicate that it has a shorter lifetime than its parent (resulting in times during which the element is not present, violating this integrity constraint); XML Schema provides no mechanism for reasoning about the timestamps.

Even though the representational and conventional schemas are closely related, there are no existing techniques to automatically derive a representational schema from a conventional schema (or vice-versa). The lack of an automatic technique means that users have to resort to ad hoc methods to construct a representational schema. Relying on ad hoc methods limits data independence. The designer of a schema for time-varying data has to make a variety of decisions, such as which elements should be time-varying, whether to timestamp with periods or with *temporal elements* [76] (which are sets of non-overlapping periods). By adopting a tiered approach, where the snapshot XML Schema, logical annotations, and physical annotations are separate documents, individual schema design decisions can be specified and changed, often without impacting the other design decisions, or indeed, the processing of tools. For example, a tool that computes a snapshot should be concerned primarily with the conventional schema; the logical and physical aspects of time-varying information should only affect (perhaps) the efficiency of that tool, not its correctness. With physical data independence, only a few applications that are concerned with representational

details would need to be changed.

To summarize, an improved tool support for representing and validating time-varying information is needed. Creating a time-varying XML document and representational schema for that document is potentially labor-intensive. Currently a user has to manually edit the time-varying document to insert timestamps indicating when versions of XML data are valid (for valid time) or are present in the document (for transaction time). The user also has to modify the conventional schema to define the syntax and semantics of the timestamps. The entire process would be repeated if a new timestamp representation were desired. It would be better to have automated tools to create, maintain, and update time-varying documents when the representation of the timestamped elements changes.

3 Background

Our work considers XML documents that are validated against a schema, specifically an XML Schema document. In providing temporal augmentations with the τXSchema approach, we consider the rich tradition of research in temporal data management, particularly in the relational field. This section provides an introduction to both XML Schema and key concepts from temporal databases.

3.1 XML

The extensible markup language XML has emerged as a standard for information exchange over the Internet. Its usage of plain text provides a platform-independent means to represent data. It has gained popularity across many classes of data including structured documents, heterogeneous and semi-structured records, data from scientific experiments and simulations, digitized images, and protocol exchange for web services. Since XML data is self-describing, XML is considered one of the most promising means to define semi-structured data, which is expected to be ubiquitous in large volumes from diverse data sources and applications on the web. XML allows users to make up any new tags for descriptive markup for their own applications. Such user-defined tags on data elements can identify the semantics of data. The relationships between elements can be defined by nested structures and references.

In the relational data model, a *schema* defines the structure of each relation in a database. Each relation has a very simple structure: a relation is a list of attributes, with each attribute having a specified data type. The schema also includes integrity constraints, such as the specification of primary and foreign keys. In a similar manner, an XML Schema document defines the valid structure for an XML document. But an XML document has a far more complex structure than a relation. A document is a (deeply) nested collection of elements, with each element potentially having (text) content and attributes.

There are various XML schemas that have been proposed in the literature and in the commercial arena. We chose to extend XML Schema [81] in τXSchema because it is backed by the W3C and supports most major features available in other XML schemas [49]. It would be relatively straightforward to apply the concepts in this paper to develop time support for other XML schema languages; less straightforward but possible would be to apply our approach of temporal and physical annotations to other data models, such as UML [61] (to produce temporally augmented class diagrams, for example). Previously, we have extended the Unifying Semantic Model, a conceptual model that extends the ER Model, to utilize annotations [48], very similar to what we propose here.

An XML schema is a description of a type of XML document, typically expressed in terms of constraints on the structure and content of documents of that type, beyond the basic syntax constraints imposed by XML itself. Thus an XML schema provides a view of the document type at a relatively high level of abstraction. The XML Schema language is also referred to as XML Schema Definition (XSD). The Document Type Definition (DTD) language [83], which is native to the XML specification, was being used as a schema language before XML Schemas were introduced. XML Schema language was introduced in order to overcome some of the limitations of DTDs like a different syntax from that of XML, limited data type capability, and limited type compatibility with databases.

XML Schema has many advancements over DTDs. Schemas are written in the same syntax as the instance documents. They have more than 44 built-in data types available, compared to only 10 data types for DTDs. A schema designer can also create his/her own data types if required. XML 1.1 introduced object-oriented data types that support inheritance and can extend or restrict a type. It also has a support for different keys like primary key and referenced key as opposed to only ID and IDREF support in DTDs.

The process of checking to see if an XML document conforms to a schema is called *validation*, which is separate from XML's core concept of syntactic well-formedness. All XML documents must be well-formed, but it is not required that a document be valid unless the XML parser is "validating", in which case the

document is also checked for the conformance with its associated schema. A well formed document obeys the basic rules of XML established for the structural design of a document. Moreover a valid document also respects the rules dictated by its corresponding XML Schema.

The parser provides an interface to an XML document, exposing its contents through a well-specified API. At present, two major API specifications define how XML parsers work: SAX [64] and DOM [82]. The DOM specification defines a tree-based approach to navigating an XML document. It processes XML data and creates an object-oriented hierarchical representation of the document that can be navigated at run-time. The tree-based W3C DOM parser creates an internal tree based on the hierarchical structure of the XML data. It can be navigated and manipulated from the software, and it stays in memory until it is released. DOM uses functions that return parent and child nodes, giving programmer full access to the XML data and providing the ability to interrogate and manipulate these nodes.

The SAX specification defines an event-based approach whereby a parser scans through XML data, calling handler functions whenever certain parts of the document (e.g., text nodes or processing instructions) are found. In SAX's event-based system, the parser does not create any internal representation of the document. Instead, the parser calls handler functions when certain events (defined by the SAX specification) take place. These events may include the start and the end of the document, finding a text node, finding child elements, or hitting a malformed element.

We now turn to time-varying XML documents.

Methods to represent temporal data and documents on the web have been actively researched. This research has covered a wide range of issues that include architectures for collecting document versions [31], strategies for storing versions [19], studies on the frequency of data change [19], temporal query languages [36, 60] and using events to trigger actions [39]. Techniques to capture the semantics of variants (alternatives of an element that can co-exist at a point in time) are orthogonal to our work, but have also been discussed [38, 87, 16]. The logical representation of deltas between the versions and the aspects of physical storage policy for storing those versions have been proposed so as to maximize the space utilization [53]. Grandi and Mandreoli [42] sketch an infrastructure for adding valid-time timestamps to a web document, and formatting timeslices from the document using XSLT. They give an XML Schema definition for the timstamps, as we do in τXSchema for our timestamps. Temporal and physical annotations are not discussed, nor are temporal constraints. Grandi has created a bibliography of previous work in this area [40]. More recent papers on version control include [44, 86, 85]. Iwaihara et al. [44] discuss a versioned temporal model in the context of access control. The model represents changes between versions with a "delta graph," which logically induces a "version graph" (essentially a timeslice-based representation). The focus of the paper is an access control language for versions, unlike τXSchema, there is no ability to specify which elements are to be versioned, the time domain of the versioning, or the (logical) representation of the versions. Wond and Lam [86] present a version management system for XML data. The system stores a document's history as a combination of some complete versions and deltas. The deltas are edit scripts, and can be used to construct a version from a nearby complete version. They also present part of a query language to retrieve desired versions. The focus of the paper is on efficient storage and retrieval of versions, whereas our focus is on fine-grained control of versioning. Wang and Zaniolo [85] present a comprehensive system for concisely representing a temporally-grouped XML version history. They also give a query language to retrieve past versions. Their extensions, like ours, require no changes to current standards to support versioning. Unlike τXSchema, everything is versioned and there is no support for temporal constraints in the versioning. Temporal and physical schema annotations are not discussed.

In context of time-varying documents, Garcia-Molina and Cho [20] provide evidence that some web resources change frequently (though not specifically XML resources). Nguyen et al. [59] describe how to detect changes in XML documents that are accessible via the web. In the Xyleme system [90], the XML Alerter module periodically (with a periodicity specified by the user) accesses the XML document and compares it with a cached version of the document. The result is a sequence of static documents, each with

an associated existence period. Dyreson [27] describes how a web server can capture some of the versions of a time-varying document, by caching the document as it is served to a client, and comparing the cached version against subsequent requests to see if anything has changed. Amagasa et al. [2] classify the methods used to access XML documents into two general categories: (i) using specialized APIs for XML documents, such as DOM, and (ii) directly editing documents, e.g., with an editor. In the former case, to access and modify temporal XML documents, DOM can be extended to automatically capture temporal information (and indeed, we have implemented such functionality in τDOM). Franceschet et al. [32] have also adopted this approach, but their approach requires the user to specify a valid ER schema and it only supports limited temporal data validation. It is also possible to capture transaction time information in the documents through change analysis, as discussed above and elsewhere [7, 22, 91]. Inconsistencies arise when the documents can be edited directly and methods need to be designed to resolve them [15]. Issues related to checking the validity of temporal documents (e.g., not allowing a child element or attribute to exist outside the lifespan of its parent) have been brought up [67]. We address this in more depth in Section 4 where we lay out our design goal regarding document validity.

Most previous approaches, irrespective of their methods to access XML documents, assume that timestamps are present on every time-varying element [51, 32, 43] (whereas our approach enables the schema designer to specify the physical location of the timestamps). There has been a lot of interest in the representation schemes for time-varying documents. Some version control tools have been developed for data varying XML documents (e.g., [53, 52]). Chien, Tsotras and Zaniolo [19] have researched techniques for compactly storing multiple versions of an evolving XML document. Chawathe et al. [18] described a model for representing changes in semi-structured data and a language for querying over these changes. A related option, the diff based approach [7, 22] focuses on an efficient way to store time-varying data and can be used to help detect transaction time changes in the document at the physical level. Buneman et al. [12, 13] provide another means to store a single copy of an element that occurs in many snapshots. Grandi and Mandreoli [41] propose a <`valid`> tag to define a validity context that is used to timestamp part of a document. Mandreoli et al. [51] utilize native support, in which an XML document is encoded using inverted lists of tuples with additional position and level numbers. Assuming a data document were stored in this representation, their slicing implementation could be used to implement unsquash efficiently. Finally, Chawathe et al. [18], Dyreson et al. [30], Mendelzon et al. [57] and Tang et al. [75] discuss timestamps on edges (instead of document nodes) in a semi-structured data model.

Recently there has been interest in incremental validation of XML documents [5, 63, 9, 4]. These consider validating a snapshot that is the result of updates on the previous snapshot, which has already been validated. In a sense, this is the dual to the problem we consider, which is validating a (compressed) temporal document all at once, rather than once per snapshot (incrementally or otherwise).

None of the approaches above focus on the extensions required in XML Schema to adequately specify the nature of changes permissible in an XML document over time, and the corresponding validation of the extended schema. In fact, some of the previous approaches that attempt to identify or characterize changes in documents do not consider a schema. As our emphasis is on logical and physical data modeling, we assume that a schema is available from the start, and that the desire is for that schema to capture both the static and time-varying aspects of the document. If no schema exists, tools can derive the schema from the base documents [6], but the details of that is beyond the scope of this paper. Our research applies at the logical view of the data, while also being able to specify the physical representation. Since our approach is independent of the physical representation of the data, it is possible to incorporate the diff-based approach and other representational approaches [13] in our physical annotations.

3.2 Temporal Databases

Most applications of database technology are temporal in nature [46]. Some examples include financial applications such as banking and accounting; record-keeping applications such as personnel, and inventory management; scheduling applications such as airline, train, and hotel reservations; and scientific applications such as weather monitoring and forecasting. Applications such as these rely on temporal databases, which record time-referenced data.

A temporal database is a database with built-in support for time aspects, e.g., a temporal data model and a temporal version of a structured query language. In a regular database, there is no concept of time. The database has a current state, and that's all that can be queried. In a temporal database, the database includes information about when things happened.

More specifically the temporal aspects usually include two orthogonal time dimensions: valid time and transaction time. These two kinds together form *bitemporal data* [45].

Valid Time: Valid time associates with a fact the time period during which the fact is true with respect to the real world. Valid time thus captures the time-varying states of the mini-world. All facts have a valid time by definition. However, the valid time of a fact may not necessarily be recorded in the database, for any of a number of reasons.

Transaction Time: Transaction time associates with the fact the time period during which the fact is stored in the database. This enables queries that show the state of the database at a given time. Unlike valid time, transaction time may be associated with any database entity, not only with facts. Thus, all database entities have a transaction-time aspect. This aspect may or may not, at the database designers discretion, be captured in the database. The transaction-time aspect of a database entity has a duration: from insertion to deletion, with multiple insertions and deletions being possible for the same entity. Transaction time captures the time-varying states of the database, and applications that demand accountability or "traceability" rely on databases that record transaction time.

Bitemporal Relation: A bi-temporal relation contains both valid and transaction time. Thus, it provides both temporal rollback and historical information.

Consider the following example emphasizing the use of both valid time and transaction time in a database table:

Name	ValidBegin	ValidEnd	TransactionStart	TransactionStop
Joe	1/1/2002	Forever	1/2/2002	UntilChanged

Joe was born on Jan 1^{st}, 2002. His father happily registered his son's birth-date on Jan 2^{nd}, 2002. In the Citizen table, two columns `ValidBegin` and `ValidEnd` would be present to record the date when a citizen is alive. Although the registration was done on Jan 2^{nd}, the database states that the information is valid since Jan 1^{st}. So `ValidBegin` contains Jan 1^{st}. Joe's record is valid while he is alive. So, `ValidEnd` contains an infinity value. To keep a track of the date when the record was inserted into the table two more fields are added to the Citizen table: `TransactionStart` and `TransactionStop`. `TransactionStart` is the time a transaction inserted that data, and `TransactionStop` is the time that a transaction *superseded* that data (or "until changed" if it has not yet been superseded). For this record, the `TransactionStart` would contain Jan 2^{nd} while `TransactionStop` would contain "until changed".

What happens if the data entry operator enters Joe's birth date as Jan 1^{st}, 2001 instead of Jan 1^{st}, 2002? When this is realized e.g., on Jan 10^{th}, 2002, the old transaction that started on Jan 2^{nd}, 2002 containing `ValidBegin` date as Jan 1^{st}, 2001 would be terminated and a new record containing correct birth date in

ValidBegin column would be inserted. The TransactionStop column for this record would have a value Jan 10^{th}, 2002.

Name	ValidBegin	ValidEnd	TransactionStart	TransactionStop
Joe	1/1/2001	Forever	1/2/2002	1/10/2002
Joe	1/1/2002	Forever	1/10/2002	UntilChanged

In the above example, the Citizen table is a bitemporal table, since it maintains both valid and transaction times for a every record. Thus, it is possible to rollback a particular record to a past date. In addition, it also provides all historical information about a record. We discuss bitemporal support for τXSchema in Section 7.

4 Design Goals and Design Decisions

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

We start out with some terminology that will be used throughout this document, including conventional and temporal (XML) documents, and conventional and temporal (XML) schemas (which are also XML documents themselves). Also defined is the annotation document and slice (which are also XML documents themselves). We then present some high level design desiderata and goals that motivate the specific decisions listed in Section 4.4. This includes decisions relevant to the temporal schema, annotations, and the temporal document. We conclude by presenting a brief example to illustrates the usage of the τXSchema language.

4.1 Terminology

This section defines terms relevant to τXSchema.

Conventional Document[1] A standard XML document that has no temporal aspects.

Temporal Document[2] A standard XML document that represents a sequence of conventional documents (i.e., slices). It may be user-created or the result of the SQUASH tool and has the root element `<temporalRoot>`.

Conventional Schema[3] A standard XML Schema document that describes the structure of the conventional document(s). The root element is `<schema>`.

Temporal Schema[4] A standard XML document that ties together the conventional schemas and the annotations. In our temporal system, the temporal schema is the logical equivalent to the XML Schema of the conventional world; it describes the rules and format of the temporal documents. The root element is `<temporalSchema>`.

Annotation Document A standard XML document that specifies a variety of characteristics (e.g., logical, physical, etc.) of a conventional document. For example, *logical* characteristics specify whether an element or attribute varies over valid time or transaction time, whether its lifetime is described as a continuous state or a single event, whether the element itself may appear at certain times (and not at others), and whether its content changes; *physical* characteristics specify the timestamp options for the representation, such as where the timestamps are placed and their kind (e.g., valid time or transaction time) and the kind of representation.

Slice A version of a temporal document at a given point in time. For example, if a temporal document is comprised of two conventional documents d_1 and d_2, which occur at time t_1 and t_2, respectively, then the slice at time t_2 is d_2.

[1] We also considered the terms "non-temporal document" (dismissed since this term focuses only on the absence of one aspect (temporal), but it could lack other aspects), "slice document" (dismissed since the term "slice" could refer to any type of document), and "base document" (dismissed since the term "base" could be confused with other contexts.

[2] We also considered "time-varying document," but dismissed it since the term "temporal" is more consistent with the rest of the terminology.

[3] We also considered the terms "non-temporal schema" (dismissed for the same reason as non-temporal document) and "base schema" (dismissed since a schema could really be composed of several base schemas.

[4] We also considered "temporal bundle" but dismissed since this term doesn't capture as cleanly the idea that this document acts as the schema for a temporal document.

4.2 Desiderata

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.
- Accommodate a variety of physical representations for time-varying data.
- Accommodate different kinds of time, such as indeterminate times, unknown times, the current time, and times at a variety of temporal granularities.
- 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.

Note that while ad hoc representational schemas may meet the last five desiderata, they certainly don't meet the first four.

In the following sections, we refine these desiderata into *design goals* and then into *design decisions.*

4.3 Design Goals

This section defines a set of high-level goals for the structure and organization of the documents and τXSchema language.

(a) *Upward compatibility* with established XML designs, techniques, and tools is the most important goal that drives the rest of the design. Conventional documents and conventional schemas should work within τXSchema. As an example, Figure 1 shows a conventional document and conventional schema being validated by XMLLINT [50]. τXMLLINT should be able to produce the same output as the conventional tool given the same input.

(b) *No changes* should be required for conventional documents. That is, conventional documents and schemas should not be aware of the fact that they are being used in τXSchema; instead, they should have standard syntax and be valid with conventional tools. Although this goal is just an expansion of goal **(a)**, it is worth mentioning specifically for emphasis.

(c) *Convenience* and *intuition* should be stressed. It is important to make migrating from a conventional system to τXSchema as easy as possible. Whenever possible, we should adopt existing XML formats, naming schemes, and methodologies; see Listings 6 and 7 for examples.

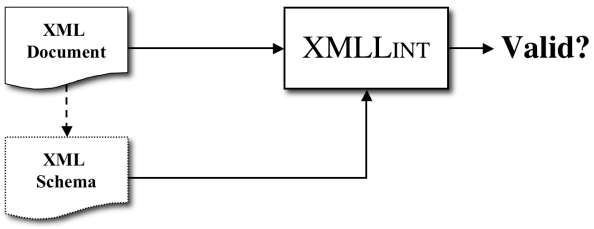


Figure 1: An XML document, which references an XML Schema, being validated by XMLLINT. The solid lines going into XMLLINT indicate that the documents are explicitly input into the tool.

Listing 6: Conventional XML Schema syntax to include a portion of one schema into another.

```
<include schemaLocation="otherSchema.xsd">
```

Listing 7: τXSchema syntax to include a portion of one temporal schema into another.

```
<include schemaLocation="otherTemporalSchema.xml">
```

(d) Adding temporal documents and schemas should be *easy*. Specifying that one or more documents vary over time should require little effort from the user. Further, the impact to the entire design and organization should be small.

(e) *Substitutability* of the various artifacts of the system is another primary goal. If there is more than one way to describe a temporal artifact, then each way should be permitted anywhere any other way is permitted. For example, both Listings 8 and 9 below contain the same information; the former lists a sequence of conventional documents and their lifetimes while the latter is the squashed version of these same documents.

Listing 8: One way to represent two conventional documents.

```
<temporalRoot>
  <sliceSequence>
    <slice location="version1.xml" begin="2008-01-02">
    <slice location="version2.xml" begin="2008-03-17">
  </sliceSequence>
</temporalRoot>
```

Listing 9: Another way to represent two conventional documents.

```
<temporalRoot  begin="2008-01-02">

  <Company_RepItem isItem="y" originalElement="Company">
    <Company_Version>
      <Company name="International Business Machines" />
      ...
    </Company_Version>
    <Company_Version>
      <Company name="IBM" />
      ...
    </Company_Version>
  </Company_RepItem>

</temporalRoot>
```

Thus, either method should be allowed to appear in place of the other.

(f) Temporal data can occur at *any level* of the system. This includes the temporal schema and annotation document. For example, temporal schemas should be allowed to reference other temporal schemas, which in turn reference other temporal (or conventional) schemas. The schemas for these schemas should be allowed to be temporal or conventional. Eventually a conventional document or schema is reached, and the process completes; however, no limits or constraints should be placed on the amount or location of temporal data. As another example, Listing 10 shows a temporal schema that references an annotation document that itself is temporal.

Listing 10: A temporal schema references an annotation document that is itself temporal.

```
<temporalSchema>
  ...
  <annotationSet>
    <!-- Note: annotations.xml is a time-varying document (e.g., it has several slices) -->
    <include schemaLocation="annotations.xml" />
  </annotationSet>
  ...
</temporalSchema>
```

(g) *Namespaces* should be preserved in the validation. If more than one namespace is used throughout the conventional schemas and in the conventional documents, then the validation should use this information in the validation process.

(h) Simple cases should be *simple* for the user to create. For example, if only one conventional schema is present, then the temporal schema is only required to list the URI for this schema, and no further markup is needed for the tools to work correctly. Listing 11 shows such a temporal schema.

Listing 11: The temporal schema should be as simple as possible.

```
<temporalSchema>
   <conventionalSchema>
      <include schemaLocation="Company.xsd" />
   </conventionalSchema>
</temporalSchema>
```

Even more simply, if a temporal document references a schema, that schema can be a conventional schema, interpreted by the tools as a temporal schema, as just discussed.

4.4 Design Decisions

This section outlines the design decisions that resulted from the design goals. We first describe general decisions that apply to all of τXSchema, and then discuss the decisions that apply specifically to temporal schemas, temporal documents, and annotation documents. The XML Schema schemas that define the sytax of these documents are provided in Appendix A.

4.4.1 General Decisions

(1) τXMLLINT will use XMLLINT (a conventional parser) as its internal engine for validating conventional documents. This decision fulfills goal **(a)** in that we can use a conventional parser on conventional documents, thus achieving full upward compatibility. It also fulfills (in fact, requires) goal **(b)** in that in order for the conventional parser to work correctly, no changes can be introduced into the conventional documents.

(2) The characterization of a temporal document will be achieved by using a `<sliceSequence>` element; each individual conventional document's URI and its associated lifetime will be directly listed

in this element via a `<slice>` element and its `location`, `begin`, and optional `end` attributes. See Listing 8 for a simple example. This decision fulfills goal **(f)** because the `<sliceSequence>` element can occur anywhere; thus allowing temporal data to occur anywhere. It also fulfills goal **(d)** because this method allows a simple way to add new versions of documents to the system.

(3) The schema of a conventional document must be a conventional schema. This satisfies goal **(a)** by promoting consistency with existing XML designs and practices and goal **(b)** by not requiring any changes to the conventional documents or schemas.

(4) In all cases, the default logical annotation is "anything can change" and the default physical annotation is "timestamp is located at root." This decision satisfies goal **(h)** by supplying a useful default annotation set if the user doesn't supply one. See Listing 11 above for an example of a schema that does not specify any annotations.

(5) When given a temporal document and temporal schema, τXMLLINT will by default only validate the current version of the document; the user must supply additional parameters/arguments to invoke validation over time.

(6) Implicit constraints on well-formedness apply to each slice separately of a temporal document. Specifically, each slice of a temporal document must satisfy the standard XML *well-formedness* constraints. Well-formedness constraints specify the logical and physical structure of an XML document and require that entities are properly nested: no start-tag, end-tag, empty-element tag, element, comment, processing instruction, character reference, or entity reference can begin in one entity and end in another.

As a comparison, Rizzolo and Vaisman's temporal extension to XML [67] specifies (in Definition 3 on page 1184) six conditions for a valid temporal document in their model. It is useful to see how these conditions translate to our model.

Their first condition is "The union of the temporal labels of the containment edges outgoing from a node is contained in the lifespan of that node." Containment edges represent time-varying subnodes, attribute values, or textual components of elements. This effectively says that a contained component cannot exist outside of its container within any snapshot. The second, "The temporal labels of the containment edges incoming to a node are consecutive," is specific to their encoding. The third is "For any time instant t, the sub-graph composed by all containment edges e_c such that $t \in T_{e_c}$ is a tree with root r. We call this subgraph a *snapshot* of the document at time t, denoted $\mathcal{D}(t)$." This is equivalent to "each snapshot is a tree." The fourth says, "the ID of a node remains constant for all the snapshots of a document." However, the ID of a node is not defined. It seems that this is specific to their encoding.

The fifth of Rizzolo and Vaisman's conditions says, "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 6.1.2, in our model a non-temporal referential integrity constraint is mapped in a temporal document to one that applies in each snapshot. Here we differ with Rizzolo and Vaisman, as what they define is what in our design is a *non-sequenced referential integrity constraint* (also discussed in Section 6.1.2). Our design is more uniform in that we utilize a per-snapshot semantics for *all* non-temporal constraints when applied to a temporal document.

The last of their conditions states, "Let $e_r(n_i, n_j, T_{e_r})$ be a reference edge. Then, $T_{e_r} \subseteq$ *lifespan*(n_j) holds." This states that a reference edge applies in a subset of the snapshots in which the destination node exists, which is a quite specific kind of non-sequenced constraint. Again, we prefer a

per-snapshot semantics for referential integrity, as with all other explicit non-temporal integrity constraints.

(7) A given conventional XML Schema constraint for a slice implies a *sequenced* constraint for the temporal document. See Section 6.2 for more information. This is a logical extension of the previous design decision.

4.4.2 Temporal Document Decisions

(8) The data stored in a temporal document may change over time. The two ways that a node in an XML document can vary with time are (1) in its content or (2) in its existence. Some nodes, especially those containing loose text, will change their content. Some nodes will exist in one version of an XML instance document but will not be present in another version. Other nodes will have both their content and existence change over time.

(9) A temporal document is defined as a document that has the root element `<temporalRoot>` and references a temporal schema. τXSchema tools will look for both of these conditions to determine if a document is temporal.

(10) A temporal document will have an attribute `<temporalSchemaLocation>` within `temporalRoot` that will specify the URI of the temporal schema. See Listing 12 for an example.

(11) The root element may have a `<sliceSequence>` element to list a sequence of conventional documents (i.e., slices). We choose the term "sequence" here since the ordering of the slices is important; they must be listed from earliest to latest. See Listing 12.

Listing 12: `exampleTemporalDocument.xml`

```
<temporalRoot>
  <temporalSchemaSet>
    <temporalSchema location="temporalSchema1.xml"/>
    <temporalSchema location="temporalSchema2.xml"/>
  </temporalSchemaSet>
  <sliceSequence>
    <slice location="version1.xml" begin="2008-01-03" />
    <slice location="version2.xml" begin="2008-06-27" />
    <slice location="version3.xml" begin="2008-08-11" />
  </sliceSequence>
</temporalRoot>
```

This decision satisfies goal **(h)** by providing an easy way to SQUASH the documents, goal **(d)** by providing a simple mechanism for adding slices, and goal **(e)** by allowing either the `<sliceSequence>` element or the SQUASH representation to appear in the temporal document. Note that this last point implies that a temporal document may have many different representations of the same information—it can have a `<sliceSequence>` to list the slices individually or it can be squashed into a full tree that represents the temporal data. We discuss representations further in Section 18.

4.4.3 Temporal Schema Document Decisions

(12) The root element of a temporal schema document will be `<temporalSchema>`.

(13) The child elements will be a single required `<conventionalSchema>` and a single (optional) `<annotationSet>`.

4.4.4 Annotation Document Decisions

(14) The root element of an annotation document will be `<annotationSet>`. We choose the term "Set" here since the ordering of the annotations within the document is unimportant. The root element can then have a number of subelements; one for each aspect possible. Currently we concentrate on the logical and physical aspects, and thus have defined `<logical>` and `<physical>` subelements.

(15) An *item* is a collection of XML elements that represent the same real-world entity. Items are in the temporal schema and elements are in the temporal document.

(16) The `<logical>` subelement will contain a set of `<item>` subelements (one for each logical constraint). Each `<item>` element specifies whether an element or attribute varies over valid time or transaction time, whether its lifetime is described as a continuous state or a single event, whether the element itself may appear at certain times (and not at others), and whether its content changes. The following shows an example logical annotation that specifies a single element.

```
<logical>
  <item target="Company/Person/FirstName">
    <transactionTime existence="constant"/>
    <itemIdentifier name="personID" timeDimension="transactionTime">
      <field path="./text()"/>
    </itemIdentifier>
  </item>
</logical>
```

Section 8 provides a complete specification of logical annotations.

(17) The `<physical>` subelement will contain a set of `<stamp>` subelements (one for each desired timestamp). These `<physical>` elements specify the timestamp options for the representation, such as where the timestamps are placed and their kind (e.g., valid time or transaction time) and the kind of representation. The following shows an example physical annotation that specifies a single timestamp.

```
<physical>
  <stamp target="Company/Person" dataInclusion="expandedVersion">
    <stampKind timeDimension="transactionTime" stampBounds="step"/>
  </stamp>
</physical>
```

Section 8 provides a complete specification of physical annotations.

(18) Previously we introduced a language, which we called *SchemaPath*, for locating element definitions in a snapshot schema [25, 28]. Recently, the W3C extended XML Schema to support element definition inheritance and introduced a new, simpler mechanism for specifying which element definitions can be annotated, i.e., a subclass needs to specify its superclass element definition. In the interest of reusing as much of XML Schema as possible, we decided to co-opt this new method, though it is less expressive than SchemaPath since only named element definitions can be annotated. This decision satisfies goal **(c)** by adopting an XML methodology.

In the new scheme an annotation is attached to an element definition through an element that names the "target" element definition as follows. Suppose that in the Company schema there is an element defintion for "company."

```
  <xs:element name="Company">
    <xs:complexType mixed="true">
      ...
  </xs:element>
```

To annotate the company element definition, a physical annotation would specify the name of the definition as its target, as illustrated below.

```
<physical>
  <stamp target="Company" ...>
  ...
  </stamp>
  ...
</physical>
```

Note that for this scheme to work the target must be unambiguously defined. Hence, every element definition name must be unique and only named elements can be annotated.

(19) An item specifies how an element in the temporal document may vary in its content and its existence. For the former, there are three possible alternatives. The first is "varying with gaps", which means that each of its corresponding data nodes may be present in some versions of the XML instance document and absent in others. A second, more restrictive form is "varying without gaps." The data node is not required to always be present. When it is present there may not be any gaps in its existence. The third value is constant. Then the corresponding data node is either always present or never present.

(20) The content may change in an element in the temporal document if the corresponding item specifies content as varying. There are restrictions on how a data node's content may change over time when the corresponding item specifies content as constant. The restrictions are different for each of the type of content (e.g., elements, attributes and loose text).

(21) Comments provide a way for a programmer to communicate with other programmers who use the XML document. Processing instructions provide a way for the programmer to communicate with XML-aware applications. Comments and processing instructions are not considered part of the XML document's content. XML Schema does not validate comments and processing instructions. Hence τXSchema does not validate them either. This design decision satisfies goal **(c)** by utilizing XML Schema directly.

Comments and processing instructions are always permitted to vary in content and existence. There are no annotations for comments and processing instructions and they are not considered to be part of an element's content.

Comments and processing instructions may appear outside the root of an XML document. To solve this, τXSchema introduces a node called `temporalRoot` that wraps the entire temporal document. Any comments or processing instructions that appear outside of the roots of individual snapshots will be wrapped by `temporalRoot`.

(22) Text is the simplest type of content. In XML, text content is contained in an element. Text cannot exist without an enclosing data element. When an item specifies (text) content as varying, the text content in the corresponding data element may change to any permitted value. The text may disappear (be empty) in one snapshot and return in a later one. For an item that specifies content as constant, the text content in each version must remain the same.

Text content is often distributed in several text nodes. We do not concatenate the text in the text nodes to consider changes in text content. If the distribution of the text changes, then the content is considered to have changed even if the concatenated value is the same. A node's content is only its own loose text, not that of its children (if any).

(23) The content of a data element consists of all of its loose text, attributes and direct child elements. A data element's content is considered for our purpose to not include any comment or processing instruction nodes, descendants of its direct children or the content of its direct children.

Suppose an item specifies content as varying. Its corresponding data element's content may change over time. On the other hand, if the item specifies content as constant, the corresponding data element's content must remain constant. The content and existence dimensions are orthogonal, so an item that specifies content as constant could have a child item that specifies existence as varying. In this case, the data element corresponding to the child item can vary in its existence, but must always appear in the same position when it exists.

When an item specifies existence as constant, the corresponding data element must either always be present or never be present. If the item specifies existence as "varying without gaps" the corresponding data element does not have to always exist overtime, but it is required to not have any gaps in its existence. When the defining element specifies existence as "varying with gaps", there are no restrictions on the existence of the corresponding data element.

(24) Data elements exist within the context of an XML document and must have a parent element. The data element's parent may vary with time. The scope of a data element's existence is limited to the time when its parent exists. An element and it child can both specify existence behavior independently. However, the behavior of the data element corresponding to the child-defining element will be affected by both the parent and child defining elements. Within the scope of the parent's existence, the child data element is affected by only the child-defining element. However, from above the parent, the child data element's existence behavior is affected by both the parent and child-defining element.

When the parent's existence is specified as constant there is no effect on the child. However, we need to consider the case where an item is specified as constant but the parent(s) of its constituent data element(s) is not. There are three possibilities for constant existence:

(a) Any item so designated must exist in every document snapshot.

(b) Any element associated with an item so designated must exist in every snapshot in which its parent element exists (i.e., the parent cannot exist without the child). The child can however switch parents over time.

(c) The third option is like the second, but the child cannot switch parents.

(25) A root node is a special data element node. There can only be one root node in an XML document and it is the node that contains all other nodes. When a root is time-varying, then the entire document is time-varying. A root node follows the same rules for varying content as any other node and may be specified as an item. A root node may have varying existence, either with or without gaps. The document can only exist when the root exists. When the root has a gap in its existence, the document also has a gap in its existence.

Any well-formed document has a root. The root may be different from one snapshot to the next. The root has no special restrictions when specified as an item.

(26) Attributes can vary over time but cannot be specified as items. An attribute's enclosing data element can be part of an item. There are two ways to specify how an attribute may change over time. The first is with current XML Schema constraints. The second is by specifying how the enclosing data element may vary with time.

Attributes will be specified as either required or constant. The attribute may be specified as required, optional or prohibited. The default is optional. If required, it must always be present. If prohibited, it can never be present. If specified as fixed, the attribute must always have the same value when it is present. An attribute cannot be both fixed and required/prohibited. The only two things that can't be specified with conventional XML Schema are existence as "varying with gaps" and both existence as constant and content as constant.

(27) All attributes exist within a data element and are part of its content. This places two additional constraints on the attribute. First, the attribute can exist only when the data element is present. Second, when the item corresponding to the data element specifies content as constant, the attribute's existence cannot change. The attribute is part of the data element's content, and the data element's content cannot change. The attribute's value may change if the item specifies content as varying.

(28) If no logical annotations are specified for a given element, then we are agnostic to the content of the element over time—it may or may not vary. Adding logical annotations to some elements does not affect this default behavior for other elements; it only defines the behavior for the specified elements.

(29) The default timestamp is placed at the root, and this timestamp always remains present. Specifying physical annotations only adds additional timestamp locations; it does not remove the default timestamp at the root. This approach is necessary in order to implement decision **(28)** above, since we must capture all varying elements whether or not there exists an associated logical or physical timestamp for that element.

(30) We extend the notion of DOM equivalence by also ensuring that the children of a given node are also DOM equivalent, and their children are DOM equivilant, ..., recursively. This "deep" versioning allows us to capture all changes to a subtree, whereas the standard DOM equivalence might miss changes that occur lower in the tree.

The rest of this section presents examples that compare the three methods and justify the method that we chose. The rule for "varying without gaps" is consistent with the rule for constant. In other words, during the time when the data element exists it follows the rule for constant. The difference being that it does not always have to exist. "Varying with gaps" has no restriction. Such a data element may always switch parents.

The following three tables contain a series of examples. Each example snapshot is a simple XML document with elements "A", "B" and "C", each of which are designated as items in the logical annotation for all three tables and all use attribute n as their item identifier.

The examples vary the values of existence for the C element and the element that encloses it to highlight the differences between the three possibilities. The A element, IBM, is constant in every example. Table 1 shows snapshots when the B element is constant. Table 2 has B elements that are "varying with gaps." The Monday and Tuesday snapshots use p1 and the Wednesday and Thursday snapshots use p2. This is done to illustrate the difference between possibilities 2 and 3. Table 3 has both p1 and p2 as "varying with gaps." In Table 1 all the examples are valid no matter which possibility we use for constant. Table 2 illustrates the difference between the three possibilities. Cells with note (1) are invalid using possibility 1 because element C, that is Bob, does not exist. They are valid using the other possibilities. Cells with note (2) are invalid with possibility 3 since Bob now has a different enclosing element. There are cells where it does not matter whether Bob exists. Some of these are labeled with note (3) and are the same for all three possibilities. Cell (5) is valid since an item can change enclosing elements when it is "varying with gaps." All the examples are valid only when using possibility 2. The definition of "varying without gaps" is illustrated by example 1 and example 2. In example 1 it doesn't matter if Bob exists on Friday. Either way there isn't a gap. In example 2 Bob does not exist in cell (4a). Therefore, Bob cannot exist in the cell labeled (4b) as this would create a gap in its existence.

In Table 3 cells with note (1) are invalid using possibility 1 but are valid using the other possibilities. Cells with note (2) are invalid with possibility 3 since Bob now has a different parent. The cell with note (6) is an error with possibility 1 since there is a gap in Bob's existence. This is because the rule for "varying with gaps" is kept consistent with the rule for constant. Cell (6) is valid for possibilities 2 and 3. Again, only with possibility 2 are all examples valid. Possibility 1 makes constant too restrictive. Possibilities 2 and 3 are similar, but 2 gives a bit more flexibility. Thus, we choose to adopt 2 as the semantics for constant.

Bob's Existence	*XML Snapshots*				
	Monday	*Tuesday*	*Wednesday*	*Thursday*	*Friday*
constant	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`
varying without gaps	`<A n="IBM"> <B n="p1"> </B> </A>`	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> </B> </A>`
varying with gaps	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> </B> </A>`	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>`	`<A n="IBM"> <B n="p1"> </B> </A>`

Table 1: `p1` is constant

4.5 Company Example

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

- Only transaction time is considered.
- The example does not use default namespaces for τXSchema files (e.g., temporal schemas) in order to emphasize which namespace is being used. However, conventional documents make use of default namespaces for brevity.
- As file contents are changed over time, a version number embedded in the name will also change so that the reader can more easily keep track of the changes. The version number for each file begins at 0 and is constructed as follows.
  - `Company.`S`.xsd` for conventional schemas, where $S = \{$`A, B, C,` $...\}$ indicates the version of the schema, e.g., `Company.A.xsd`.
  - `data.`S`.`D`.xml` for conventional documents, where S indicates the version of the schema being used and D indicates the version number of the conventional document, e.g., `data.A.0.xml`.
  - `temporalDocument.`S`.`D`.xml` for temporal documents, where S indicates the version of the temporal schema being used and D indicates the version number of the latest conventional document, e.g., `temporalDocument.0.3.xml`.
  - `temporalSchema.`D`.xml` for temporal schemas, where D indicates the version number of the temporal schema, e.g., `temporalSchema.0.xml`.

Bob's Existence	*XML Snapshots*				
	Monday	*Tuesday*	*Wednesday*	*Thursday*	*Friday*
constant	<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>	<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> (2) </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> (2) </A>	<A n="IBM"> (1) </A>
varying without gaps	<A n="IBM"> <B n="p1"> (3) </B> </A>	<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> (2) </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> (2) </A>	<A n="IBM"> (3) </A>
varying without gaps	<A n="IBM"> <B n="p1"> (3) </B> </A>	<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> (2) </A>	<A n="IBM"> <B n="p2"> (4a) </B> </A>	<A n="IBM"> (4b) </A>
varying with gaps	<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>	<A n="IBM"> <B n="p1"> </B> </A>	<A n="IBM"> <B n="p2"> </B> </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> (5) </A>	<A n="IBM"> </A>

Table 2: `p1` and `p2` are "varying without gaps"

Bob's Existence	*XML Snapshots*				
	Monday	*Tuesday*	*Wednesday*	*Thursday*	*Friday*
constant	<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> (2) </A>	<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>	<A n="IBM"> (1) </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> (2) </A>
varying without gaps	<A n="IBM"> <B n="p1"> </B> </A>	<A n="IBM"> <B n="p2"> </B> </A>	<A n="IBM"> <B n="p2"> </B> </A>	<A n="IBM"> (6) </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> (2) </A>
varying with gaps	<A n="IBM"> <B n="p1"> <C n="Bob"> </B> </A>	<A n="IBM"> <B n="p2"> </B> </A>	<A n="IBM"> <B n="p1"> </B> </A>	<A n="IBM"> </A>	<A n="IBM"> <B n="p2"> <C n="Bob"> </B> </A>

Table 3: `p1` and `p2` are "varying with gaps"

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

In this example, each time the user modifies the conventional schema, a new file is created. He must then modify the conventional document to reference this new, modified schema. In practice, this is awkward and would rarely happen. In a more realistic situation, the user would reuse the same filename by just modifying the file in place, and editors would be responsible for automatically retaining previous versions. Also, in practice the conventional document would change much more frequently than the conventional schema. Figure 2 depicts the overall scenario.

4.5.1 Initial Configuration

Consider the following scenario which begins on 2008-01-01. The user has a conventional schema which defines a `<Person>` element, which itself has a `<Name>` element, an `<SSN>` element, and an `ID` attribute (see Listing 13).
He also has a conventional document conforming to the schema (see Listing 14).

Together, these documents form a conventional system which can be validated with conventional validation tools (e.g., XMLLINT). Of course, τXMLLINT will also validate this conventional system. In

Listing 13: `Company.A.xsd`

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

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

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

</xsd:schema>
```

Listing 14: `data.A.0.xml`

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

  <Person ID="1">

    <Name>Steve</Name>
    <SSN>111-22-3333</SSN>
  </Person>


</Company>
```

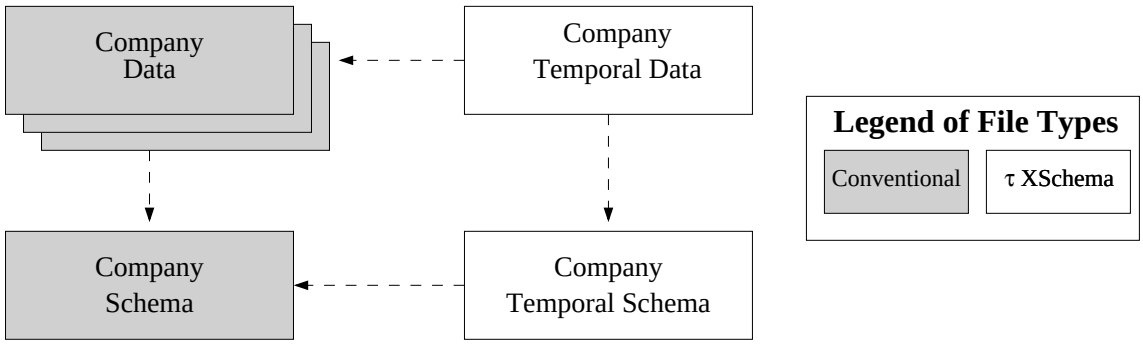


Figure 2: An overview of the end-state of the Company example.

the following sections, we will add new versions of the conventional document, add new versions of the conventional schema, break up the conventional schema into multiple subschemas, and specify logical annotations. Figure 2 shows the relationship between all the documents in the system. Note that Company schema (for details on this schema and example documents, please see Section 10) will import and include two subschemas: `Person` and `Product`. In this example, both the Person and Product schemas will change over time. Each time there is a new slice created, the Company schema must be updated to reference the new slice. There are other mechanisms available to the user for handling this scenario, as described in the document beginning at Section 11

4.5.2 Adding Temporal Data

On 2008-03-17, the user corrects the `<SSN>` element in the conventional document to produce a new version (see Listing 15). The user can now use τXSchema to create temporal documents and use the τXSchema tools to validate these documents.

Listing 15: `data.A.1.xml`

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

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

</Company>
```

The user creates a temporal document that lists both slices of the conventional document with their associated timestamps (see Listing 16).

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

Section 12 continues this example when multiple conventional schemas are employed as well as when each individual schema varies over time.

Listing 16: `temporalDocument.0.1.xml`

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

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

</td:temporalRoot>
```

5 Theoretical Framework

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

5.1 Snapshot Validation Subsumption

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

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

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

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

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

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

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

Intuitively, the property asserts that a good representational schema will validate only those time-varying documents for which every snapshot conforms to the conventional schema. The subsumption property is depicted in Figure 3.

5.2 Content and Existence Variance

The data stored in XML documents may change over time. It is useful to be able to validate the way data can change. The XML Schema standard provides a way to validate XML documents, but does not define

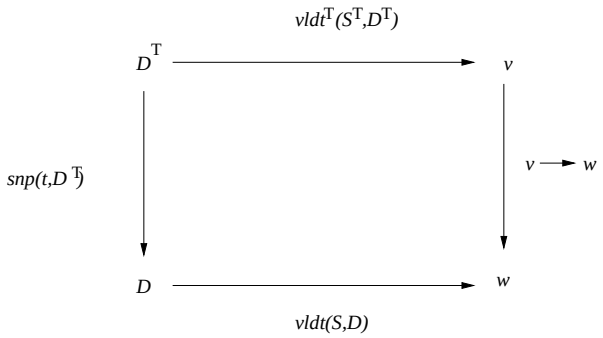


Figure 3: Snapshot Validation Subsumption

how an XML document is allowed to change with time. To meet this need, τXSchema was created as an extension of the XML standard that validates time-varying XML documents.

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

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

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

The other aspect an item may specify is content. The content of a data node depends on its node type. The content may change in the data node at any time if the corresponding item specifies content as varying. There are restrictions on how a data node's content may change over time when the corresponding item specifies content as constant. The restrictions are different for each of the type of content (e.g., elements, attributes and loose text). The detailed explanation of the restrictions can be found in Section 8.

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

5.3 Items

In order to create a temporal document it is important to identify which elements persist across various transformations of the document. This section discusses how to find and associate elements in different

snapshots of a temporal XML document. When elements are temporally-associated, an *item* is created. An item is a collection of XML elements that represent the same real-world entity. An item is a logical entity that evolves over time through various versions.

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

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

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

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

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

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

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

- `<athlete>` $\Rightarrow$ `[athName/*]`
- `<medal>` $\Rightarrow$ `[../athName/*, ./* ]`

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

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

Definition [Temporal association] Let x be an element of type T in the i^{th} snapshot of a temporal document. Let y be an element of type T in the j^{th} snapshot of the document. Finally let L be the

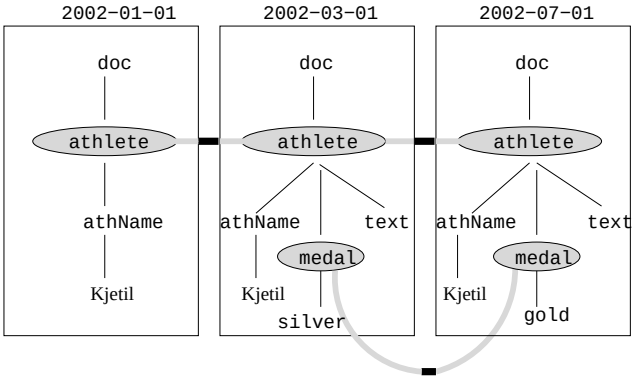


Figure 4: Items and Versions

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

A temporal association relates elements that are adjacent in time and that belong to the same item. For instance, the `<athlete>` element in Listing 1 on page 7 is temporally associated with the `<athlete>` element in Listing 2 but not the `<athlete>` element in Listing 3 (though the `<athlete>` element in Listing 2 is temporally related to the one in Listing 3).

5.4 Versions

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

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

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

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

As an aside, we observe that DOM equivalence in a temporal XML context is akin to value equivalence in a temporal relational database context [45]. DOM equivalence is used to determine versions of an item, as follows.

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

A version's lifetime is extended when the element from the next snapshot (or a future snapshot) is DOM equivalent (the lifetime can have gaps or holes, although having a gap may violate a schema constraint as described in Section 5.2). A new version is created when a temporal association is not DOM equivalent.

Figure 4 depicts the items and versions in the example. An abstract representation of the DOM for each snapshot of the document is shown. The items in the sequence of snapshots are connected within each shaded region. There is one athlete item and one medal item. The athlete item has two versions; the transition between versions is shown as a black stripe between the regions.

6 Extending Temporal XML Schema Constraints

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

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

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

6.1 XML Schema Constraints

We give the syntax and semantics of constraints in conventional XML Schema, using the `company` example. The root of this schema is the `company` entity. Under that, there are `products` and `supplier` elements. A `product` is a sub-element of `products` and an `order` is considered a sub-element of `suppliers` (with a reference from `order` to `productNumber` (within `product`) for data integrity).

6.1.1 Identity Constraints

Identity constraints restrict uniqueness of elements and attributes in a given document. As with the relational model, XML Schema allows users to define both `key` and `unique` constraints (we use the term *identity constraint* to refer to the union of the `key` and `unique` constraint types. The distinction between these two constraint types is that the evaluation of the `key` constraint should always yield a valid tuple (value) for all of the component fields (none of the fields should be empty), while the fields in a `unique` constraint are allowed to be absent.

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

An identity constraint may be named, and this name can then be used when defining a referential integrity constraint (similar to foreign keys in the relational model). A sample XML Schema identity constraint is in Listing 17.

Formally, we can define `unique` and `key` constraints as follows. Let n be the context node being validated (under which the identity constraint is defined). Let sel be the element named by the `selector` of the identity constraint (*sel* is an XPath expression for the selector relative to n), with the list of corresponding field expressions $F = (f_1, \ldots, f_m)$. Using the example in Listing 17, n corresponds to `products` and `selector` corresponds to `product`. Then a `unique` constraint can be formally defined as follows.

$$\text{unique}(n, \textit{sel}) = \forall i, j \in \textit{Eval}(n, \textit{sel})\ [\textit{Eval}(i, F) = \textit{Eval}(j, F) \Rightarrow i = j]$$

For a key constraint, the only change is that no field can evaluate to the empty list.

Listing 17: Sample Identity Constraint Definition

```
...
  <xs:element name="products">
    ...
    <xs:element name="product" minOccurs="0" maxOccurs="unbounded">
      ...
    </xs:element>
    ...
    <xs:key name="productKey">
      <xs:selector xpath="product" />
      <xs:field xpath="@productNo" />
    </xs:key>
    ...
 </xs:element>
...
```

There are some similarities between the functionality of `key` and `unique` constraints and the XML 1.0 `ID` definitions (and the equivalent `ID` simple type in XML Schema). However, the XML Schema `key` and `unique` constraints have a number of advantages over use of `ID`. The advantages allow us to define more powerful constraints at a temporal level (in Section 6.2.1).

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

- XML Schema keys can be applied to both elements and attributes. Since `ID` is an attribute (in DTDs; in XML Schema an element's type can be defined as `xs:ID`), it cannot be applied to other attributes.

- Using `key` and `keyref` allows the specification of the scope within which uniqueness applies (done by the `selector` element; i.e., it is "contextual uniqueness") while the scope of an XML ID is the whole document. Thus using a `key` constraint one can enforce: "within each order, the part numbers should be unique", to ensure that each order line has a different part number. This cannot be done using XML `ID`s.

- Finally, XML Schema enables the creation of a `key` or a `keyref` constraint from combinations of element and attribute content and does not restrict the possible datatypes for valid keys. XML `ID`s consist of single attribute content, and must be of the `ID` datatype.

6.1.2 Referential Integrity Constraints

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

A sample definition of a referential integrity constraint in XML Schema to specify that particular constraint follows. Note that a `keyref` uses a key via `refer`. Each `field` in the `keyref` must have a corresponding `field` in the associated `key` named by `refer`.

Listing 18: Sample Referential Integrity constraint

```
...
  <xs:element name="supplier" minOccurs="0" maxOccurs="unbounded">
    ...
    <xs:element name="order minOccurs="0" maxOccurs="unbounded">
      ...
      <xs:keyref name="ordersProductRef" refer="productKey">
        <xs:selector xpath="order" />
        <xs:field xpath="oProductNo" />
      </xs:keyref>
      ...
    </xs:element> % end order
    ...
  </xs:element> % end supplier
...
```

Formally, we can define the `keyref` constraint as follows. Let n_r be the context node being evaluated with a `keyref` constraint defined within it. Let L_r be the a list of XPath expressions defined by $[sel_r/f_r1, sel_r/f_r2, \ldots, sel_r/f_rn]$, where the `selector` element of the `keyref` constraint is represented by sel_r, and the corresponding `field` expressions by f_rn.

Let $Eval(n_r, L_r)$ denote the result of evaluating the list L_r relative to a `context` node n_r. Let e_r be an element from the list defined by $Eval(n_r, L_r)$ (e.g., one of the products listed in an order). Similarly, let $Eval(n_k, L_k)$ (see Section 6.1.1) denote the result of evaluating the referenced `key` constraint, and $e_k \in Eval(n_k, L_k)$. The `keyref` constraint is satisfied when $\forall e_r$ there exists a e_k (in the document) such that $e_r = e_k$.

6.1.3 Cardinality Constraints

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

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

Listing 19: Cardinality definitions using XML Schema

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

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

6.1.4 Datatype Restrictions

Datatype definitions in XML Schema can restrict the structure and content of elements, and the content of attributes. We currently consider datatypes defined using the XML Schema `simpleType` element. A

simple type is used to specify a value range. In the simplest case, a built-in XML Schema datatype (e.g., `integer`) imposes a value range. For more complicated requirements, a simple type can be derived from one of the built-in datatypes.

An example of an XML Schema datatype definition follows.

Listing 20: XML Schema data type definition

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

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

6.2 Temporal Augmentations to the XML Schema Constraints

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

There are two flavors of temporal constraints, *sequenced* and *non-sequenced*. A temporal constraint is sequenced with respect to a similar conventional constraint in the schema document, if the semantics of the temporal constraint can be expressed as the semantics of the conventional constraint applied at each point in time. A constraint is non-sequenced if it is evaluated over a temporal element as a whole (including the lifetime of the data entity) rather than at each point in time separately.

As discussed in Section 4.4, 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 zero and four website URLs for each supplier," has a sequenced equivalent of: "There should be between zero and four website URLs for each supplier *at each point in time*." We also allow the user to add a new sequenced constraint (e.g., with specific *applicability bounds*) in the logical annotation document.

The applicability bound, $B \subseteq T$, allow the user to restrict their consideration from the lifetime of the document to some desired subset they are interested in. For example, a constraint may only be valid between 1999–2005, at which time it is replaced by a new constraint. Applicability bounds are relevant for both sequenced and non-sequenced constraints. While the effect of an applicability bound (for a sequenced constraint) can be simulated by "removing" the constraint from the schema document (during some time slice), this restricts it to cases where the transaction time and valid time are identical.

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* [21]. We support current constraints by allowing the user to set the applicability bound of the sequenced constraint to *now*.

For the non-sequenced extension to constraints, we consider a window of evaluation, w, which can be a temporal element. The user specifies the window of evaluation (e.g., a day, or a Gregorian month). The user can also specify a slide size, ss, and applicability bound, B [26]. The default length for ss is a single granule interval corresponding to the granularity, $gran$, of the item within which it is defined. The default for B is the lifetime of the temporal document. We have established the following relationship among the components of a non-sequenced constraint: $gran \leq ss \leq w \leq B$.

Non-sequenced constraints are evaluated over a time interval rather than at a point in time. The window of evaluation must be within the applicability bound. So for non-sequenced constraints, we replace the

evaluation point t, where $t \in T$, with $w \in \mathcal{P}(B)$. When $size(w)$ is the same as $size(B)$, we term it a "fixed-window" constraint. For example, suppose the constraint requires there to be between 0 and 4 supplier URLs in the temporal document over a period of any calendar month. Let's say this constraint is applicable from 2009-03-01 to 2009-03-31. Here, w and B have the same size. If instead the applicability were (2009-03-01 to 2009-06-31), then we see a case of a "sliding-window" constraint (since the evaluation would take place during *each* month from March through June. Here, we see the the size of the slide is implicitly a *calendar month* as well. Let's say instead, the constraint evaluation window were a period of 30 days. Then the user may wish to restrict how this evaluation window would slide. For example, one may choose to evaluate it from March 1–30, then from March 2–31, and so on. Here, the size of the slide (ss) is a single day.

Non-sequenced constraints are listed in the logical annotations document. In a few cases (when we extend a particular XML Schema constraint for additional functionality), sequenced constraints are also listed in the logical annotation document.

In the case where schema versioning is permitted (we discuss schema versioning in detail in Part II), the constraints are evaluated only within a single schema-constant period. For example, let us take a cardinality constraint that restricts a maximum of 50 orders from a supplier in any calendar month. Let us further assume that on July 12, the schema for orders changed and the order `key` now was an `orderNumber` (instead of a combination of supplier number and date). Then the evaluation for the 50-order limit will be done separately for the first eleven days of July, and July 12-31.

We now proceed to discuss temporal enhancements to each of the XML Schema constraints described in Section 6.1. The general approach is to add non-sequenced extensions to each constraint (though for sequenced cardinality constraints, we add new semantics as well).

6.2.1 Identity Constraints

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

We have shown in Section 6.1.1 the advantages of XML Schema identity constraints over defining an element or attribute to have a type of `ID`. This motivates the following design decision: we extend the semantics of XML Schema identity constraints to support non-sequenced semantics, but do not do so for `ID` types. If an element or attribute in an XML Schema document is said to have a type of `ID`, then that only translates to a sequenced constraint.

Formally, we define a sequenced `key` constraint as follows. Let T be the set of time points associated with a temporal XML document over its lifetime. At all times t (where $t \in T$), we can extract a snapshot of the document. As with the conventional case, let n be a context node enclosing the associated identity constraint in the conventional schema. We represent the `selector` by sel and corresponding field XPath expressions $F = (f_1, \ldots, f_m)$. We define L as the list of XPath expressions $[sel/f_1, sel/f_2, \ldots, sel/f_n]$. Correspondingly, let $Eval(n, L)$ denote the result of evaluating the list of XPath expressions L from the context node n for the snapshot at point t. We denote the i^{th} element of $Eval(n, L)$ by e_i. Then $\forall i, j : e_i = e_j \Rightarrow i = j$. This predicate must be true for every snapshot of the document.

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

For both `key` and `unique` constraints we consider non-sequenced extensions. A non-sequenced `unique` (or `key`) constraint requires that the constrained element (or attribute) is unique over time (not just at a point in time). For example, if we wished to require that an employee's `SSN` were unique in a single conventional document as well as the temporal document, we could use a non-sequenced constraint. We will explain the sub-elements and attributes of these non-sequenced constraints shortly.

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. For example, given the (nontemporal) `productKey` definition in Listing 17 for the `product` element, the following snippets (Listings 21 and 22) reflect a perfectly legal change from one state to another for the `productNo` attribute (from `500` to `599`) within the first `<product>` element.

Listing 21: Initial State for `productNo` attribute

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

Listing 22: Changed State for `productNo` attribute

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

A non-sequenced identity constraint states that a field value combination is unique across time. This is both within and between nodes. Consider the conventional `unique` constraint defined in Listing 23. Suppose a non-sequenced uniqueness constraint is placed on the email address of an employee, with an evaluation window of a year (Listing 24). Then, no two employees can have the email address `jdoe@arizona.edu` (for example) in any year, nor can the same employee (e.g., John Doe) switch from `jdoe@arizona.edu` to `john.doe@arizona.edu` and back to `jdoe@arizona.edu` in a year. To specify a uniqueness constraint solely within a node, i.e., if we wished to only say that the same employee (e.g., John Doe) cannot switch from `jdoe@arizona.edu` to `john.doe@arizona.edu` and back to `jdoe@arizona.edu` in a single year, we would need to define an item at the level of a single employee (Listing 25). In this case, we do not re-use the conventional uniqueness constraint since its selector is different (and it is defined under `employees`, while we want it to be defined at the level of `employee` for a constraint at the *within* level). We leave for future work a discussion of (and specifying the syntax and semantics for) unique constraints that apply solely between nodes.

As seen in the example in Listing 22, a conventional identity constraint does not imply non-sequenced uniqueness. Thus, the same `productNo` (a conventional key) can be *re-used* for another product or changed between snapshots (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 (for example, a `productNo`) as unique across a temporal document (with snapshots coalesced across the window of evaluation).

Depending on how a non-sequenced identity constraint is specified, we may end up with two or more products with the same `productNo` over time. We would like to clarify that an attribute or element annotated with a non-sequenced unique (or key) constraint need not be used as the *itemIdentifier* for the

Listing 23: Conventional Uniqueness constraint for employee emails

```
...
<xs:element name="employees">
  ...
  <xs:element name="employee">
    ...
    <xs:attribute name="email" type="xs:string" use="optional" />
    ...
  </xs:element>
  ...
  <xs:unique name="unique_employee_email">
    <xs:selector xpath="employee" />
    <xs:field xpath="@email" />
  </xs:unique>
    ...
</xs:element>
...
```

Listing 24: Non-sequenced uniqueness constraint on employee emails

```
...
<item target="company/employees">
  ...
  <nonSeqUnique name="nsu_employee_email" conventionalIdentifier="unique_employee_email"
       evaluationWindow="year" slideSize="day" />
  ...
</item>
...
```

Listing 25: Non-sequenced uniqueness constraint within a single employee

```
...
<item target="company/employees/employee">
  ...
  <nonSeqUnique name="nsu_employee_email" evaluationWindow="year" slideSize="day" >
     <applicability begin="2007-01-01" />
     <selector xpath="." />
     <field xpath="@email" />
  </nonSeqUnique>
  ...
</item>
...
```

corresponding `item` (introduced in Part II for schema versioning). An `item` may have multiple (non-sequenced) unique and key constraints defined under it.

By specifying `productNo` as non-temporal, no change can be made to its value. A different (or new) `productNo` value indicates an instance of a distinct product. The simplest way to designate a time-invariant key is by specifying the item as non-temporal (`content="constant"`) in the logical annotations and combining this with a `key` constraint on the element or attribute.

A *non-sequenced unique constraint* specifies that a value (of an element or attribute) should not be re-used at a later time within an evaluation window. This is specified in the logical annotations through one of the following elements: `<nonSeqUnique>`, `<nonSeqKey>` or `<uniqueNullRestricted>` (a sub-element of `item`). We adopt the usual distinction in semantics between `key` and `unique` (i.e., the permissibility of *null* values).

With the refinements introduced in Section 6.2, we define a `nonSeqKey` constraint as follows. Let the `item` containing the `nonSeqKey` definition be denoted by n. Let L be the list of XPath expressions $[sel/f_1, sel/f_2, \ldots, sel/f_n]$ where sel is the `selector` and f_i are the `field` expressions.

Define n^w to be a version of n whose timestamp overlaps w. Evaluating the expression list L with respect to n^w returns a list of versions.

The union of all such versions overlapping the window of evaluation w is denoted by: For each window (a time period) w, $\forall i, j, e_i \in Union_{n^w}(Eval(n^w, L))$ and $e_j \in Union_{n^w}(Eval(n^w, L)), e_i = e_j \Rightarrow i = j$.

The effect of the slide size is to determine the start point for each successive w.

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

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

A more powerful version of the `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 a non-sequenced key constraint. We term this constraint as a *value cardinality constraint*, but do not explore it for now since it has no XML Schema equivalent.

We now proceed to discuss the different attributes and sub-elements for the `uniqueConstraint` (summarized in Table 21; the sub-elements are indented).

`name`: This allows the user to name the constraint and is useful in case the constraint is referenced elsewhere (e.g., in a referential integrity constraint).

`conventionalIdentifier`: Specifies the name of the identifier in the conventional schema document. If this is not specified, then it implies a new constraint is being defined and the `selector` and `field` sub-elements should not be empty.

`nullCountMin`: Used only in conjunction with the `uniqueNullRestricted` constraint to specify how many nulls are allowed over the non-sequenced time extent (minimum).

`nullCountMax`: Used in conjunction with the `uniqueNullRestricted` constraint to specify how many nulls are allowed over the non-sequenced time extent (maximum).

`dimension`: Specifies the dimension in which the unique constraint applies and is one of `validTime`, `transactionTime`, or `bitemporal`. The default is assumed to be `validTime` since that is closely related to capturing real world restrictions, rather than restrictions on data entry.

`evaluationWindow`: Specifies the time window over which the constraint should be checked. The purpose is to allow uniqueness to be specified for an interval, e.g., year. This is useful when, for example, a particular key value should not be re-used for a period of a year. The value then must be "unique over any period of a year". By default the evaluation window is the lifetime of the time varying XML document. Assuming we use XML Schema to specify the datatypes for time intervals, we can extend that with a union of the string: `lifetime`. This will allow us to set the time interval for `evaluationWindow` (and other attributes) to a value of `lifetime` (indicating a temporal element equivalent to the lifetime of the XML document). In the constraint examples that follow, we assume this datatype extension is done and use the keyword `lifetime` when needed. Strictly speaking, the evaluation window could be defined as a union of intervals. For example, a constraint could require *no more than ten orders are placed each month, in the first and third weeks of the month*. For simplicity we use an `xs:string` datatype and specify only a single attribute for the evaluation window (rather than sub-elements). The window corresponds to an interval. Thus, if just a granularity is given, such as "year", this is interpreted as the interval "1 year". If slideSize is not provided, then it is assumed to be one granule, or the temporal granularity of the underlying item being constrained.

`slideSize`: Specifies the size of the slide (an interval); must be used in conjunction with an evaluation window. By default, it takes the temporal granularity of the underlying item being constrained.

`applicability` : The applicability of a constraint specifies when it was valid. (Note: the `applicability` attribute applies to the valid time dimension; transaction-time applicability would concern when a constraint exists in the schema document.) Thus a key constraint may be enforced between 2005 and 2010. Strictly, the applicability need not be a single range, and may be a temporal element, which is why we specify the applicability as (`begin`, `end`) attribute pairs within a wrapping sub-element called `applicability`. If nothing is specified, the default is assumed to be the lifetime of the document. The applicability and evaluation window of the constraint are related. Defining an evaluation window that exceeds the applicability of the constraint is not really meaningful, as it cannot be checked beyond the constraint applicability. In such a case, a warning should be returned and the evaluation window should be shortened to the maximum allowable within the constraint applicability limits.

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

single `xpath` attribute. `selector` and `field` are needed to specify a new constraint (i.e., those that were not defined as identifiers in the conventional schema). If a new constraint is defined, the `conventionalIdentifier` attribute should not be used. A new constraint can be either defined as either a `key` or a `unique` constraint.

We now describe some non-sequenced identity constraint examples.

1. *The combination of supplier name and city is unique. 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 in a given business (calendar) year, we require that reuse should not occur for at least one year after discontinuation. Part numbers on the other hand may not be re-used later. These constraints are applicable between 2005 and 2009.*

```
...
<item target="company/suppliers">
  ...
  <nonSeqKey name="idSupplierNo" dimension="validTime" evaluationWindow="year"
      slideSize="day">
    <applicability begin="2005-01-01" end="2009-12-31">
    <selector xpath="supplier" />
    <field xpath="supplierName" />
    <field xpath="supplierCity" />
  </nonSeqKey>
  ...
</item>
...
<item target="company/products">
...
  <nonSeqKey name="idPartNo" dimension="validTime" evaluationWindow="lifetime">
    <applicability begin="2005-01-01" end="2009-12-31" />
    <selector xpath="product" />
    <field xpath="productNo" />
  </nonSeqKey>
...
</item>
...
```

2. *A product's key (in both valid and transaction time) is its RFID number. The constraint is applicable from 2007 onwards.*

```
...
<item target="company/products">
  ...
  <nonSeqKey name="product_RFID" dimension="bitemporal"
       evaluationWindow="lifetime">
     <applicability begin="2007-01-01" />
     <selector xpath="product" />
     <field xpath="@RFID" />
  </nonSeqKey>
  ...
</item>
...
```

3. *Employee email addresses are optional. If they do exist, they should be unique and should not be re-used for a two-year period. An existing unique constraint (conventional) exists on employee emails with an name of* `unique_employee_email`. *The assumption is that it is defined under employees, i.e., has the same context as the definition of the item with a target of company/employees. Employee pager numbers are also unique; but not all employees have them. No more than fifty employees should be without a pager in any given calendar year.*

```
...
<item target="company/employees">
  ...
  <nonSeqUnique name="nsu_employee_email" conventionalIdentifier="unique_employee_email"
       evaluationWindow="2 years" slideSize="day" />
  <uniqueNullRestricted name="unr_employee_pager"
       nullCountMax="50" dimension="bitemporal"
       evaluationWindow="year" slideSize="year" >
     <applicability begin="2007-01-01" />
     <selector xpath="employee" />
     <field xpath="@pager" />
  </uniqueNullRestricted>
  ...
</item>
...
```

6.2.2 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.

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

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

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

Intuitively, a non-sequenced `keyref` constraint should refer to the definition of an identifier that does not permit re-use. Without the restriction of not permitting re-use, the semantics of the referential integrity may not be well defined. For example, if the order number could be re-used, then a customer's largest order may end up referencing an order that was not originally placed by that customer. Not permitting re-use of order numbers however is a strict constraint that can be omitted if the `largestOrderNo` has a valid-time timestamp. Then, the non-sequential reference can be understood to be for a specific order number that was valid at the begin time of the `largestOrderNo`.

We represent a non-sequenced referential integrity constraint using a `nonSeqKeyref` element in the logical annotations. Next, we proceed to discuss the different attributes and sub-elements for the `nonSeqKeyref` (summarized in Table 24; the sub-elements are indented).

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

dropped.

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

`applicability`: A non-sequenced keyref can be associated with a particular `applicability` that specifies when it was in effect. If the applicability is not specified, the default is assumed to be the lifetime of the document. As with uniqueness constraints, the applicability can be a temporal element.

`selector` and `field`: These two sub-elements have a usage identical to their `uniqueConstraint` counterparts, but are needed to specify a new constraint (i.e., those that were not defined as referential integrity constraints in the conventional schema).

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

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

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

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

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

6.2.3 Cardinality Constraints

As discussed in Section 6.1.3, the cardinality of elements in conventional documents is restricted by using `minOccurs` and `maxOccurs`, and that of attributes by using `optional` and `required`. These automatically induce sequenced constraints in the temporal document.

Non-sequenced constraints can be used to restrict the cardinality over time. Consider the example of an `order` element in Listing 26. We see that the `deliveredOn` element may not always be present in a specific document snapshot. Let us further say, that while it may be empty at the time the order was placed, we require it to be appear at some point (say within three months of the order being placed). So,

even though `minOccurs="0"` is satisfactory for a conventional document, we may desire the equivalent `minOccurs="1"` for a temporal document.

Listing 26: Orders with an optional `deliveredOn`

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

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

Another refinement that may be desired for a non-sequenced cardinality constraint is the *aggregation level* at which the count is being performed. Let's consider the schema in Listing 27. A non-sequenced cardinality constraint can be used to place a limit of one hundred orders from a supplier in any given year. In this case, `order` is the direct child of suppliers, and the conventional `maxOccurs` constraint (on `order`) would be used to restrict the number of child `order` elements a `supplier` can have. Suppose, we wished to further constrain the number of orders for the `company` across the all suppliers to 1500 per month. In other words, the number of `order` elements that were descendants of `company`) should be ≤ 1500 in any calendar month. The conventional cardinality constraints are not designed to handle this. This is our motivation behind introducing the `aggLevel` option for a cardinality constraint.

Listing 27: Considering Aggregation Levels for an `order`

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

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

- sel be a context node (defined by `selector` in a cardinality constraint), with the list of corresponding `field` expressions $F = (f_1, \ldots, f_m)$.
- $ancestorOf(n, a)$ return `true` if a is an ancestor of node n, and `false` otherwise.
- For two lists L_1 and L_2, let $L_1 \uplus L_2$ return the result of appending the members of L_2 to L_1.
- $Eval^w(sel, F)$ be the result returned by the evaluation of F relative to sel (over the evaluation window w). There may be many nodes corresponding to a given sel (e.g., many `supplier` nodes), and each such (`supplier`) node can have many children. Therefore $eval^w(sel, F)$ returns a list of lists.

- $childList_i \in eval^w(sel, F)$ be the i^{th} member (also a list) of the result.
- $childListAggr = aggr(childlist_i, aggLevel)$ be the aggregation of the various $childList_i$ at the ancestor level of $aggLevel$; i.e., $childListAggr = (childList_1 \uplus \ldots \uplus childList_i \uplus \ldots \uplus childList_n)$, where $\forall i, ancestorOf(sel, aggLevel)$ is `true`.

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

The definition for `childList` can be modified for set semantics, `childSet`, by only considering distinct elements (i.e., duplicates are not considered). This brings up the issue of how to determine if two nodes are duplicates. One option is to go with the understanding of DOM-equivalence defined in Section 5.4. Another option is to consider two nodes equivalent if the values of their corresponding item identifiers match. When defining the `itemIdentifierRef` attribute in `cardConstraint`, the `selector` XPath for the cardinality constraint (i.e., the `field` XPath expression in cardinality) should be compatible with the referred item identifier.

Formally, we let:

- $childSet_i \in eval^w(sel, F)$ be the set of child nodes returned by the evaluation of the XPath expressions F relative to sel (over the evaluation window w).
- $childSetAggr = aggr(childSet_i, aggLevel)$ be the aggregation of the $childSet_i$ result at the ancestor level of $aggLevel$; i.e., $childSetAggr = (childSet_1 \cup \ldots \cup childSet_i \cup \ldots \cup childSet_n)$, where $\forall i, ancestorOf(sel, aggLevel)$ is `true`.

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

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

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

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

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

8. *There should be a maximum of 250 potential suppliers for the company across all products. We assume there exists an item identifier— on the potential supplier's* `supplierNo` *attribute. This constraint is to be enforced during 2009. [Sequenced constraint; use of* `childSet`*]*

   This is a sequenced constraint. However it cannot be enforced by a combination of a `minOccurs` and `maxOccurs`.

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

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

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

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

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

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

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

We now proceed to discuss the different attributes and sub-elements for the `cardConstraint` (summarized in Table 25; the sub-elements are indented).

`name`: This allows the user to name the constraint and is useful in case the constraint is referenced later.

`restrictionTarget`: Cardinality constraints can restrict the `childList`, `childSet`, `valueList`, and `valueSet` counts of elements and attributes.

`childList` refers to the actual number of sub-elements that can appear over time, and is analogous to the conventional minOccurs and maxOccurs for sequenced constraints. The difference between `childList` and `childSet` is similar, in that duplicate sub-elements are not counted for `childSet`. Duplication is determined using by referencing an applicable uniqueness constraint (which in terms specifies the fields to be evaluated).

`itemIdentifierRef`: The name of the item identifier. Used along with `childSet` to eliminate duplicates.

`dimension`: Specifies the dimension in which the cardinality constraint applies and is one of `validTime`, `transactionTime`, or `bitemporal`.

`evaluationWindow`: Associated with a non-sequenced cardinality constraint is the time window over which the constraint should be checked. This allows cardinality minimum and maximum ranges to be specified for an interval, e.g., year. This is useful, for example, when a restriction needs to be put on how many orders suppliers can handle in any given period. By default the time window is the lifetime of the XML document.

`slideSize`: Associated with the time window of evaluation. By default it is the granularity of the underlying data type.

`sequenced`: Denotes if the constraint is sequenced or not (using either `true` or `false`). This is allowed in the constraint specification since XML Schema only allows `minOccurs` and `maxOccurs` to be aggregated at the target parent level. Allowing a different aggregation level is useful, for example, if instead of restricting the number of potential suppliers for a product (assuming `<potential_suppliers>` is a child element of `<product>`), we wish to restrict the total number of potential suppliers the company maintains relationships with at any time. If a constraint is specified as sequenced, the `evaluationWindow` attribute must not be used.

`aggLevel`: Specifies the level at which the aggregation is performed for cardinality constraints; by default it is at the level of the target's parent. This is also the reason why we allow sequenced cardinality

specifications. For a sequenced constraint to be useful, the aggregation level should not be the target's parent.

`min` and `max`: Specify the minimum and maximum cardinality respectively.

`selector` and `field`: These two sub-elements have a usage identical to XML Schema usage for conventional constraints.

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

6.2.4 Datatype Restrictions (Constraints)

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

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

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

We now proceed to discuss the different attributes and sub-elements for the `transitionConstraint` (summarized in Table 26; the sub-elements are indented).

`name`: This allows the user to name the constraint and is useful in case the constraint is referenced elsewhere.

`dimension`: Specifies the dimension in which the unique constraint applies and is one of `validTime`, `transactionTime`, or `bitemporal`. The default is `validTime` since a cardinality constraint on transaction time is akin to specifying how many "data entry changes" can be made to an element or attribute.

`selector` and `field`: These two sub-elements have a usage identical to their conventional XML Schema counterparts.

`valuePair`: This is used to list possible pairs for discrete changes. The pairs themselves are specified as `<old>` and `<new>` sub-elements. valuePair cannot be used simultaneously with `valueEvolution`. The values listed here should be within the range of values defined for the conventional `simpleType` datatype.

`valueEvolution`: This sub-element lists the direction for continuous changes. Only one of `valuePair` and `valueEvolution` should be used. The values listed here should be within the range of values defined for the conventional `simpleType` datatype. Continuous changes of the following direction are currently supported:

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

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

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

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

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

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

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

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

7 Support for Bitemporal Data

Up to this point, all the examples we have seen consider only a single dimension of time. But as explained in Section 3.2, both transaction and valid time play an important role in modeling entities which need to maintain the historical information. If an entity needs to maintain both the historical information as well as the history of changes, bitemporal support is needed. In this section, we consider a conceptual extension of τXSchema to provide support for bitemporal data and procedure for squashing the conventional documents along both time dimensions.

For illustration, we consider a modified example from Chapter 10 of the book **Developing Time-Oriented Database Applications in SQL** [72].

Nykredit is a major Danish mortgage bank. It maintains the information about properties and customers into bitemporal tables for historical information and to provide tracking support. Traditionally, its been using relational database tables to maintain this information. If this information needs to be migrated to XML, τXSchema with the support for bitemporal data would be useful.

In their database, the information about Property, Customers and their relationship is maintained in the following three tables.

```
Property (property_number, address, VT_Begin, VT_End, TT_Start,
          TT_End)
Customer (name, VT_Begin, VT_End, TT_Start, TT_End)
Prop_Owner (customer_number, property_number, VT_Begin, VT_End,
            TT_Start, TT_End)
```

Let us assume that, the information about the property is represented in XML using the schema given in Listing 28. For simplicity, only `property_number` and `address` attributes of the `Property` are considered. Property is associated with a owner by the `owner_name` attribute of the `<property>` element. To simplify the things a little, we assume that the owner is uniquely represented by the `owner_name` attribute.

Corresponding logical and physical annotations are given in Listings 29 and 30. As can be seen from the temporal annotation, the `<property>` element is content varying both in transaction-time and valid-time.

To illustrate the process of gluing in two dimensions, we consider the history, over both valid time and transaction time, of a flat (apartment) in Aalborg, at Skovvej 30 for the month of January 2008. All its transactions are listed below in the chronological order of transaction-time. The corresponding bitemporal-time diagrams and snippets of conventional XML documents are also given for understanding.

Assume that, initially, the mortgage for the flat was being handled by some other company. So, although Nykredit maintained the property information, no information about the owner is stored in the database. We

Listing 28: `property.xsd`

```
...
<element name="property">
  <complexType mixed="true">
    <sequence>
      <element name="address" type="string" minOccurs="1" maxOccurs="1" />
    </sequence>
    <attribute name="property_number" type="nonNegativeInteger" use="required"/>
    <attribute name="owner_name" type="string" use="optional"/>
  </complexType>
</element>
...
```

Listing 29: `property_logical_annotation.xml`

```
...
<item target="property">
  <transactionTime content="varying" existence="constant" />
  <validTime content="varying" existence="constant" />
  <itemIdentifier name="property_number" timeDimension="bitemporal">
    <field path="@property_number"/>
  </itemIdentifier>
</item>
...
```

Listing 30: `property_physical_annotation.xml`

```
...
<stamp target="property">
  <stampKind timeDimension="bitemporal" stampBounds="extent"/>
</stamp>
...
```

also assume that the flat exists in Nykredit's database from January 1. The snippet of the conventional document corresponding to this period is shown in Figure 5.

Transaction Time [01-01, UC) (Will be altered to [01-01, 01-10))

- Valid Time [01-01, Forever)

Listing 31: Property information, no owner details

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

Figure 5: Mortgage being handled by other company. No customer

On January 10, this flat was purchased by Eva Nielsen. We record this information at a current valid-time (`01-10`), current transaction-time (`01-10`). The snippets of the conventional documents corresponding to this transaction period starting on `01-10` are shown in Figure 6.

This information is valid starting now, and was inserted now. We will see that the transaction-time extent of *all* modifications is "now" to "until changed," which we encode as "forever" and express as `9999-12-31` in the XML document.

The interplay between valid time and transaction time can be confusing, so it is useful to have a visualization of the information content of a bitemporal table. Figure 7 shows the *bitemporal time diagram*, or simply *time diagram*, corresponding to the above insertion.

In this figure, the horizontal axis tracks transaction time and the vertical axis tracks the valid time. Information about the owners associated with the property are depicted as two-dimensional polygonal regions in the diagram. Arrows extending rightward denote "until changed" in transaction time; arrows extending upward denote "forever" in valid time. Here we have but one region, associated with Eva Nielsen, that starts at time 10 (January 10) in transaction time and extends to "until changed," and begins also at time 10 in valid time and extends to "forever." The arrow pointing upward extends to the largest valid time value ("forever"); the arrow pointing to the right extends to "now," that is, it advances day by day to the right (a

Transaction Time [01-10, UC) (Will be altered to [01-10, 01-15))

- Valid Time [01-01, 01-10)

Listing 32: Data corresponding to Valid time of Jan 1 - 10

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-10, Forever)

Listing 33: Data corresponding to Valid time of Jan 10 onwards

```
...
<property property_number="7797" owner_name="Eva">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

Figure 6: Eva purchased the flat on January 10

transaction time in the future is meaningless).

On January 15 Peter Olsen buys this flat; this legal transaction transfers ownership from Eva to him. Figure 8 illustrates how this update impacts the time diagram. The valid-time extent of this modification is always "now" to "forever," so from time 15 on, the property is owned by Peter; at the rest of the time, from time 10 to 15, the property was owned by Eva. Both regions extend to the right to "until changed." This time diagram captures two facts: Eva owning the flat and Peter owning the flat, each associated with a bitemporal region.

The snippets of the conventional documents corresponding to this transaction are shown in Figure 9.

On January 20, we find out that Peter has sold the property to someone else, with the mortgage again being handled by another mortgage company. From Nykredit's point of view, the property no longer has a owner as of (a valid time of) January 20.

Figure 10 shows the resulting time diagram. If we now request the valid-time history as best known, we will learn that Eva owned the flat from January 10 to January 15, and Peter owned the flat from January 15 to January 20. All prior states are retained. We can still time travel back to January 18 and request the valid-time history, which will state that on that day we thought that Peter still owned the flat.

The snippets of the conventional documents corresponding to this transaction are shown in Figure 11.

On January 23, we find out that Eva had purchased the flat not on January 10, but on January 3, a week earlier. So we insert those additional days, to obtain the time diagram shown in Figure 12. Corresponding snippets of the conventional documents are given in Figure 14

We learn on January 26 that Eva bought the flat not on January 10, as initially thought, nor on January 3, as later corrected, but on January 5. We specify a period of applicability of January 3 through 5, with the result shown in the time diagram in Figure 13. Corresponding conventional snippets are given in Figure 15

Finally, we learn on January 28 that Peter bought the flat on January 12, not on January 15 as previously thought. This change requires a period of applicability of January 12 through 15, setting the `owner_name` to Peter, which results in the time diagram in Figure 16. Effectively, the ownership must be transferred from Eva to Peter for those three days, resulting in the conventional documents given in Figure 17.

Gluing elements in two dimensions involves gluing them along one dimension (e.g., valid-time) followed by their gluing along the other dimension (e.g., transaction-time). The last timing diagram on January

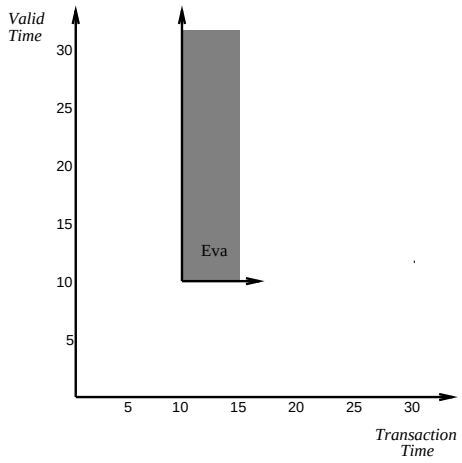


Figure 7: A bitemporal time diagram corresponding to Eva purchasing the flat, performed on January 10

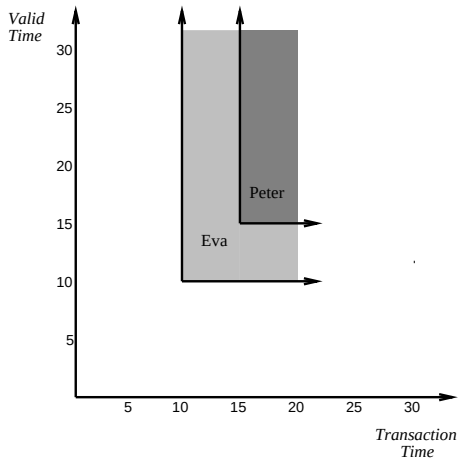


Figure 8: Peter buys the flat, performed on January 15

28 in Figure 16 could be divided into 7 time-periods along the transaction time dimension as shown in Figure 18, i.e., `[01-01 - 01-10)`, `[01-10 - 01-15)`, `[01-15 - 01-20)`, `[01-20 - 01-23)`, `[01-23 - 01-26)`, `[01-26 - 01-28)`, `[01-28 - UC)`.

All the above conventional documents are first squashed along valid-time dimension as explained soon to give seven temporal documents corresponding to each of the above periods. The sample sample representation of these documents corresponding to periods `[01-10, 01-15)`, `[01-20, 01-23)`, `[01-26, 01-28)` are given below in Listings 53, 54, and 55, respectively. These documents are temporal documents themselves.

Other representations are also possible for these documents. As an example, the document in Figure 54 could also be represented as shown in Figure 56. In this representation, multiple DOM-equivalent versions of the `<property>` are merged into a single version and their time periods are represented as a single time-varying element, i.e., a set of periods.

These temporal documents then act as conventional documents while performing squashing along transaction-time dimension. When squashed along transaction-time dimension, they give the final temporal document

Transaction Time [01-15, UC) (Will be altered to [01-15, 01-20))

- Valid Time [01-01, 01-10)

Listing 34: Transaction Time `[01-15, UC)`, Valid Time `[01-01, 01-10)`

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-10, 01-15)

Listing 35: Transaction Time `[01-15, UC)`, Valid Time `[01-10, 01-15)`

```
...
<property property_number="7797" owner_name="Eva">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-15, Forever)

Listing 36: Transaction Time `[01-15, UC)`, Valid Time `[01-15, F)`

```
...
<property property_number="7797" owner_name="Peter">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

Figure 9: Peter buys the flat, performed on January 15

shown in Listings 57, 58 and 59.

When we were concerned with only valid-time or only transaction-time in earlier examples, the coalescing of content-constant versions was done by lengthening the version periods. But when the interplay of two dimensions comes into picture, the periods in a single dimension generalize to *regions* in the time diagram, which are considerably more involved than one-dimensional periods. In terms of time diagram, an item version with two valid-time instants, `VT_Begin` and `VT_End`, and two transaction-time instants, `TT_Start` and `TT_Stop`, encodes a *rectangle* in bitemporal space. Such two rectangle can be coalesced when either their valid-time instants `VT_Begin` and `VT_End` match or their transaction-time instants `TT_Start` and `TT_Stop` match.

While representing these regions in the XML document, they could be split with the vertical lines (termed as *transaction-time splitting* shown in Figure 19) or horizontal lines (termed as *valid-time splitting*). Due to the semantics of transaction time, regions are often split with vertical lines in the timing diagram.

The temporal document in Figures 57–59 uses the first approach, since it minimizes the representation of the document.

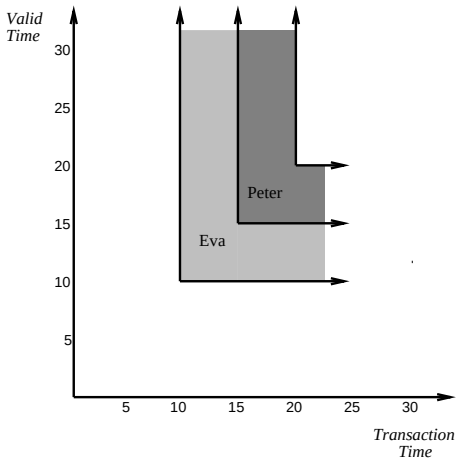


Figure 10: Peter sells the flat, performed on January 20

Listing 53: Transaction Time `[01-10, 01-15)`

```
...
<property_RepItem>
  <property_Version>
    <timestamp_ValidExtent begin="2008-01-01" end="2008-01-10" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-10" end="9999-12-31" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>
</property_RepItem>
...
```

Transaction Time [01-20, UC) (Will be altered to [01-20, 01-23))

- Valid Time [01-01, 01-10)

Listing 37: Transaction Time `[01-20, UC)`, Valid Time `[01-01, 01-10)`

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-10, 01-15)

Listing 38: Transaction Time `[01-20, UC)`, Valid Time `[01-10, 01-15)`

```
...
<property property_number="7797" owner_name="Eva">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-15, 01-20)

Listing 39: Transaction Time `[01-20, UC)`, Valid Time `[01-15, 01-20)`

```
...
<property property_number="7797" owner_name="Peter">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-20, Forever)

Listing 40: Transaction Time `[01-20, UC)`, Valid Time `[01-20, F)`

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

Figure 11: Peter sells the flat, performed on January 20

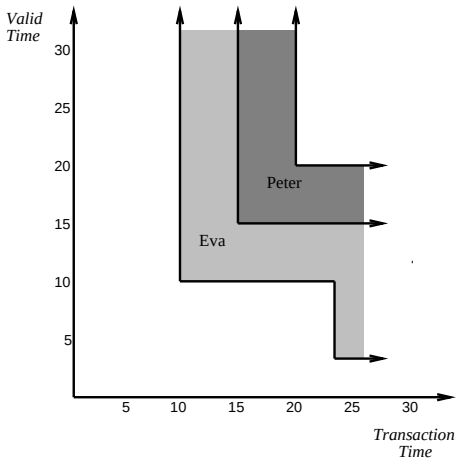


Figure 12: Discovered on January 23: Eva actually purchased the flat on January 3

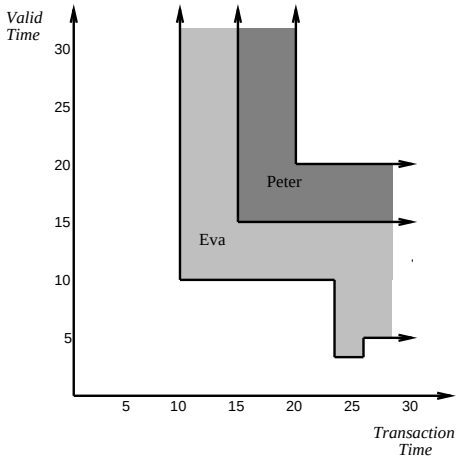


Figure 13: Discovered on January 26: Eva actually purchased the flat on January 5

Transaction Time [01-23, UC) (Will be altered to [01-23, 01-26))

- Valid Time [01-01, 01-03)

Listing 41: Transaction Time `[01-23, UC)`, Valid Time `[01-01, 01-03)`

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-03, 01-15)

Listing 42: Transaction Time `[01-23, UC)`, Valid Time `[01-03, 01-05)`

```
...
<property property_number="7797" owner_name="Eva">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-15, 01-20)

Listing 43: Transaction Time 23rd - UC, Valid Time 15th - 20th

```
...
<property property_number="7797" owner_name="Peter">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-20, Forever)

Listing 44: Transaction Time `[01-23, UC)`, Valid Time `[01-20, F)`

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

Figure 14: Discovered on January 23: Eva actually purchased the flat on January 3

Transaction Time [01-26, UC) (Will be altered to [01-26, 01-28))

- Valid Time [01-01, 01-05)

Listing 45: Transaction Time `[01-26, UC)`, Valid Time `[01-01, 01-05)`

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-05, 01-15)

Listing 46: Transaction Time `[01-26, UC)`, Valid Time `[01-05, 01-15)`

```
...
<property property_number="7797" owner_name="Eva">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-15, 01-20)

Listing 47: Transaction Time `[01-26, UC)`, Valid Time `[01-15, 01-20)`

```
...
<property property_number="7797" owner_name="Peter">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-20, Forever)

Listing 48: Transaction Time `[01-26, UC)`, Valid Time `[01-20, F)`

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

Figure 15: Discovered on January 26: Eva actually purchased the flat on January 5

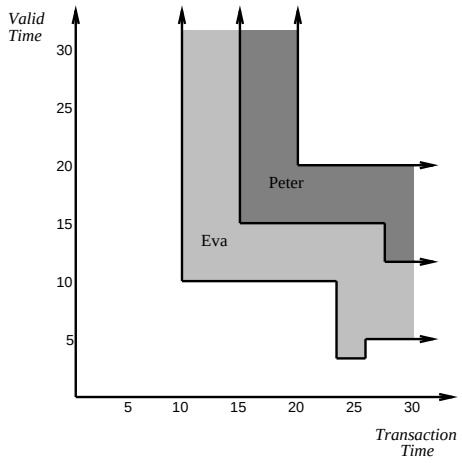


Figure 16: January 28: Peter actually purchased the flat on January 12

Listing 54: Transaction Time `[01-20, 01-23)`

```
...
<property_RepItem>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-01" end="2008-01-10" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-10" end="2008-01-15" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-15" end="2008-01-20" />
    <property property_number="7797" owner_name="Peter">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-20" end="9999-12-31" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

</property_RepItem>
...
```

Transaction Time [01-28, UC)

- Valid Time [01-01, 01-05)

Listing 49: Transaction Time `[01-28, UC)`, Valid Time `[01-01, 01-05)`

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-05, 01-12)

Listing 50: Transaction Time `[01-28, UC)`, Valid Time `[01-05, 01-12)`

```
...
<property property_number="7797" owner_name="Eva">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-12, 01-20)

Listing 51: Transaction Time `[01-28, UC)`, Valid Time `[01-12, 01-20)`

```
...
<property property_number="7797" owner_name="Peter">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

- Valid Time [01-20, Forever)

Listing 52: Transaction Time `[01-28, UC)`, Valid Time `[01-20, F)`

```
...
<property property_number="7797">
  <address> Skovvej 30, Alborg </address>
</property>
...
```

Figure 17: January 28: Peter actually purchased the flat on January 12

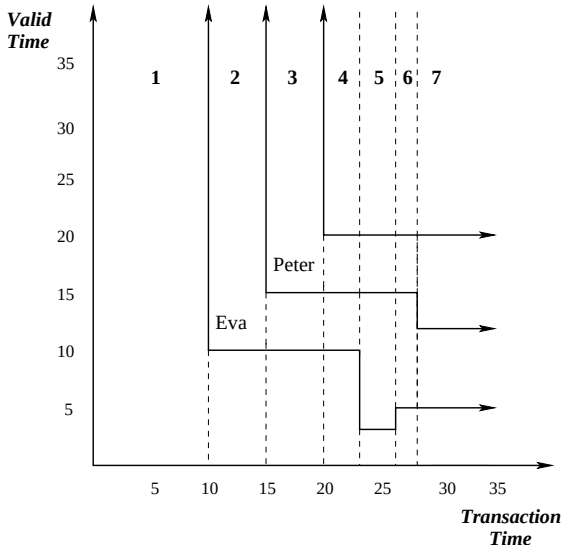


Figure 18: Transaction Time Regions

Listing 55: Transaction Time `[01-26, 01-28)`

```
...
<property_RepItem>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-01" end="2008-01-05" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-05" end="2008-01-12" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-12" end="2008-01-20" />
    <property property_number="7797" owner_name="Peter">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-20" end="9999-12-31" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

</property_RepItem>
...
```

Listing 56: Transaction Time `[01-20, 01-23)`

```
...
<property_RepItem>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-01" end="2008-01-10" />
    <timestamp_ValidExtent begin="2008-01-20" end="9999-12-31" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-10" end="2008-01-15" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="2008-01-15" end="2008-01-20" />
    <property property_number="7797" owner_name="Peter">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

</property_RepItem>
...
```

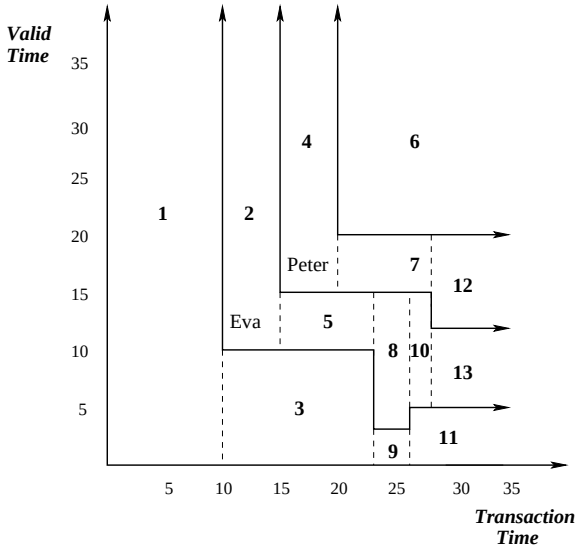


Figure 19: Transaction-time splitting of regions

Listing 57: Temporal Document along both valid-time and transaction-time

```
...
<property_RepItem>

  <property_Version>
    <timestamp_TransExtent start="2008-01-01" stop="2008-01-10" />
    <timestamp_ValidExtent begin="2008-01-01" end="9999-12-31" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="2008-01-10" stop="2008-01-15" />
    <timestamp_ValidExtent begin="2008-01-10" end="9999-12-31" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="2008-01-10" stop="2008-01-23" />
    <timestamp_ValidExtent begin="2008-01-01" end="2008-01-10" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="2008-01-15" stop="2008-01-20" />
    <timestamp_ValidExtent begin="2008-01-15" end="9999-12-31" />
    <property property_number="7797" owner_name="Peter">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="2008-01-15" stop="2008-01-23" />
    <timestamp_ValidExtent begin="2008-01-10" end="2008-01-15" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>
  ...
```

Listing 58: Temporal Document along both valid-time and transaction-time. **Continued**

```
<property_Version>
  <timestamp_TransExtent start="2008-01-20" stop="9999-12-31" />
  <timestamp_ValidExtent begin="2008-01-20" end="9999-12-31" />
  <property property_number="7797">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>

<property_Version>
  <timestamp_TransExtent start="2008-01-20" stop="2008-01-28" />
  <timestamp_ValidExtent begin="2008-01-15" end="2008-01-20" />
  <property property_number="7797" owner_name="Peter">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>

<property_Version>
  <timestamp_TransExtent start="2008-01-23" stop="2008-01-26" />
  <timestamp_ValidExtent begin="2008-01-03" end="2008-01-15" />
  <property property_number="7797" owner_name="Eva">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>

<property_Version>
  <timestamp_TransExtent start="2008-01-23" stop="2008-01-26" />
  <timestamp_ValidExtent begin="2008-01-01" end="2008-01-03" />
  <property property_number="7797">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>

<property_Version>
  <timestamp_TransExtent start="2008-01-26" stop="2008-01-28" />
  <timestamp_ValidExtent begin="2008-01-05" end="2008-01-15" />
  <property property_number="7797" owner_name="Eva">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>
```

Listing 59: Temporal Document along both valid-time and transaction-time. **Continued**

```
    <property_Version>
      <timestamp_TransExtent start="2008-01-26" stop="9999-12-31" />
      <timestamp_ValidExtent begin="2008-01-01" end="2008-01-05" />
      <property property_number="7797">
        <address> Skovvej 30, Alborg </address>
      </property>
    </property_Version>

    <property_Version>
      <timestamp_TransExtent start="2008-01-28" stop="9999-12-31" />
      <timestamp_ValidExtent begin="2008-01-12" end="2008-01-20" />
      <property property_number="7797" owner_name="Peter">
        <address> Skovvej 30, Alborg </address>
      </property>
    </property_Version>

    <property_Version>
      <timestamp_TransExtent start="2008-01-28" stop="9999-12-31" />
      <timestamp_ValidExtent begin="2008-01-05" end="2008-01-12" />
      <property property_number="7797" owner_name="Eva">
        <address> Skovvej 30, Alborg </address>
      </property>
    </property_Version>

  </property_RepItem>
  ...
```

In order to support bitemporal data, we anticipate following architectural and implementational changes to the existing tools.

SCHEMAMAPPER : SCHEMAMAPPER would need very little change. As the representation of a temporal document is going to remain the same, it needs to add both transaction and valid-time elements from the `TVSchema` for the elements from physical annotation which are time-varying along both the dimensions.

τXMLLINT : τXMLLINT would also need little change to support bitemporal data. Since the representation of items in a XML document is not going to change, the gluing procedure, which is the first part of the τXMLLINT algorithm, would remain the same. Next step is to validate the individual items identified during gluing. In the existing `Item` class, the validation procedure for the item needs to be extended to perform the validation of items varying along both valid and transaction time.

SQUASH : To perform squashing of bitemporal data we anticipate a need of a wrapper class, e.g., `DoBitemporalSquashing`, to the existing architecture. This class would use the existing `DoSquashing` class to perform the squashing of documents along valid-time for identified transaction-time periods. This will generate the series of temporal documents, which will act as conventional documents for squash along transaction-time. The existing `DoSquashing` class and other primitive functions will not be able to handle these temporal documents, since they were not designed anticipating the existence of items in the conventional documents. Thus the `DoSquashing` class would need some changes to handle these documents. Also, although the conceptual algorithms for the primitive functions remains the same, some implementation level changes would be needed. The existing `Item` class has the support for bitemporal time. But the coalescing algorithm handles only time-periods. It does not handle regions. The current `coalesce` function needs an extension to perform coalescing of regions.

UNSQUASH : UNSQUASH tool would also need some changes similar to the SQUASH tool. A new wrapper class (e.g., `DoBitemporalUnSquashing`) could be added. This class would first unsquash the given bitemporal document along the transaction-time dimension to give multiple temporal documents along valid-time. Each of these documents need to be unsquashed along the valid-time dimension giving multiple conventional documents. Existing `UnSquash` would work without any changes for performing unsquashing along the valid-time dimension. Some modifications would be needed to `UnSquash` class to perform the unsquashing along the transaction-time dimension.

Thus, although the tools would be based on the existing classes, addition of some new classes and modifications to the primitive functions would be necessary in order to provide the support for bitemporal data.

8 Architecture

In this section we describe the overall architecture of τXSchema and illustrate with an example. The design and implementation details of the tools are explained further in Section 9.

A visual depiction of the architecture of τXSchema is illustrated in Figure 20. This figure is central to our approach, so we describe it in detail and illustrate it with examples. We note that although the architecture has many components, only those components shaded in the figure are specific to an individual time-varying document and need to be supplied by a user. New time-varying schemas can be quickly and easily developed and deployed. We also note that the representational schema, instead of being the only schema in an ad hoc approach, is merely an artifact in our approach, with the conventional schema, logical annotations, and physical annotations being the crucial specifications to be created by the designer.

The designer annotates the conventional schema with logical annotations (box 5). The logical annotations together with the conventional schema form the logical schema. Listing 60 provides an extract of the logical annotations on the WinOlympic schema. The logical annotations specify a variety of characteristics such as whether an element or attribute varies over valid time or transaction time, whether its lifetime is described as a continuous state or a single event, whether the item itself may appear at certain times (and not at others), and whether its content changes. For example, `<athlete>` is described as a state element, indicating that the `<athlete>` will be valid over a period (continuous) of time rather than a single instant. Annotations can be nested, enabling the target to be relative to that of its parent, and inheriting as defaults the kind, `contentVarying`, and `existenceVarying` attribute values specified in the parent. The attribute `existenceVarying` indicates whether the element can be absent at some times and present at others. As an example, the presence of `existenceVarying` for an athlete's phone indicates that an athlete may have a phone at some points in time and not at other points in time. The attribute `contentVarying` indicates whether the element's content can change over time. An element's content is a string representation of its *immediate content*, i.e., text, sub-element names, and sub-element order.

As discussed in Section **(4)**, if *no annotations* are provided whatsoever, the default annotation is that *anything can change*. However, once we begin to annotate the conventional schema, the semantics we adopt are that elements that are not described as time-varying are static. Thus, they must have the same content and existence across every XML document in box 7. For example, we have assumed that the birthplace of an athlete will not change with time, so there is no annotation for `<birthPlace>` among the logical annotations. The schema for the logical annotation document is given by ASchema (box 2).

The next design step is to create the physical annotations (box 6). In general, the physical annotations specify the timestamp representation options chosen by the user. An excerpt of the physical annotations for the WinOlympic schema is given in Listing 61. Physical annotations may also be nested, inheriting the specified attributes from their parent; these values can be overridden in the child element.

Physical annotations play two important roles.

- They help to define where the physical timestamps will be placed (versioning level). The location of the timestamps is independent of which components vary over time (as specified by the logical annotations). Two documents with the same logical information will look very different if we change the location of the physical timestamp. For example, although the elements `phone` and `athName` are time-varying, the user may choose to place the physical timestamp at the `athlete` level. Whenever any element below athlete changes, the entire `athlete` element is repeated.

- The physical annotations also define the type of timestamp (for both valid time and transaction time). A timestamp can be one of two types: `step` or `extent`. An extent timestamp specifies both the start and end instants in the timestamp's period. In contrast a step-wise constant (step) timestamp represents only the start instant. The end instant is implicitly assumed to be just prior to the start of

Listing 60: Sample WinOlympic Logical Annotation

```
<?xml version="1.0" encoding="UTF-8"?>
<logical
   xmlns="http://www.cs.arizona.edu/tau/tauXSchema/ASchema"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://www.cs.arizona.edu/tau/tauXSchema/ASchema
                ASchema.xsd">
  <defaultTimeFormat>
    <format plugin="XMLSchema" granularity="gDay"/>
  </defaultTimeFormat>
...
  <item target="winOlympic/country/athleteTeam">
    <validTime content="constant" existence="varyingWithGaps">
      <maximalExistence begin="1924-01-01" />
    </validTime>
    <itemIdentifier name="teamName" timeDimension="transactionTime">
      <field path="./teamName"/>
    </itemIdentifier>
  </item>
...
  <item target="winOlympic/country/athleteTeam/athlete/medal">
    <validTime/>
    <transactionTime/>
    <itemIdentifier name="medalId1" timeDimension="bitemporal">
      <field path="./text()"/>
      <field path="../athName"/>
    </itemIdentifier>
  </item>
...
</logical>
```

Listing 61: Sample WinOlympic Physical Annotation

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

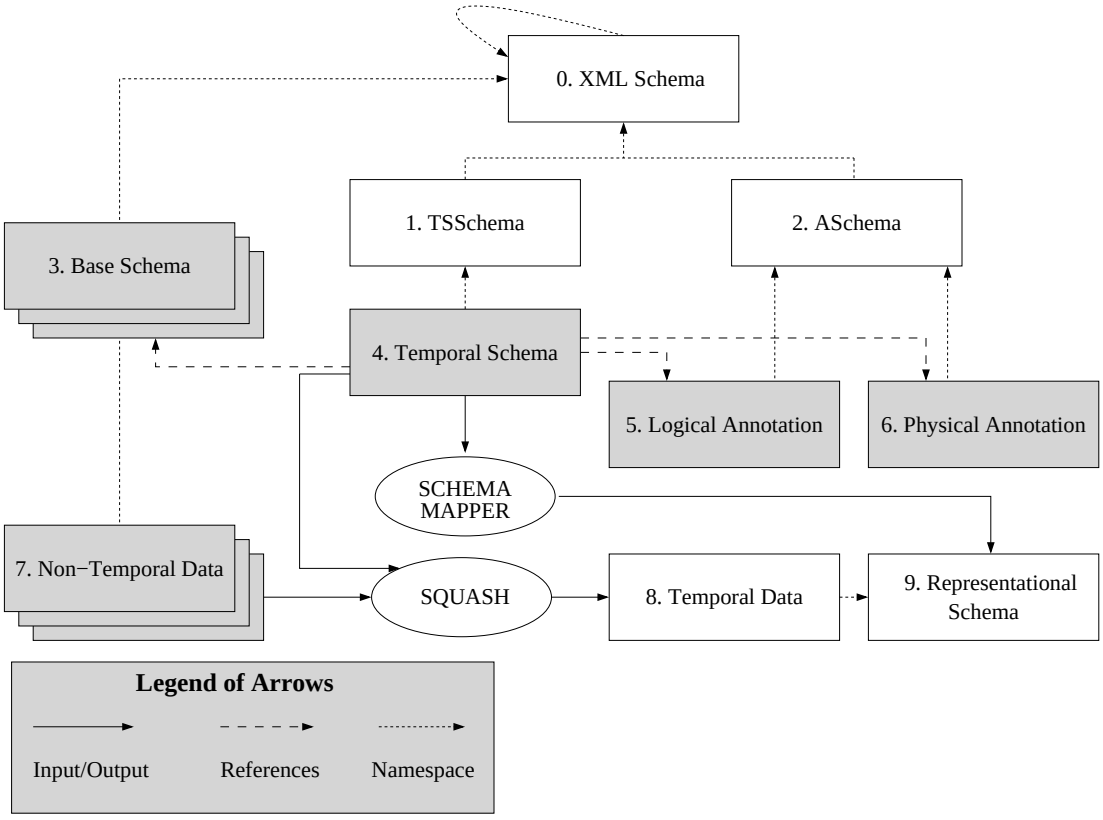


Figure 20: Overall Architecture of τXSchema

the next version, or *now* for the current version. However, one cannot use `step` timestamps when there might be "gaps" in time between successive versions. `extent` timestamps do not have this limitation. Changing even one timestamp from `step` to `extent` can make a big difference in the representation.

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

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

The temporal schema (box 4) ties the schema, logical annotations and physical annotations together. This document contains sub-elements that associate a series of conventional schema with logical and physical annotations, along with the time span during which the association was in effect. The schema for the temporal schema document is TSSchema (box 1).

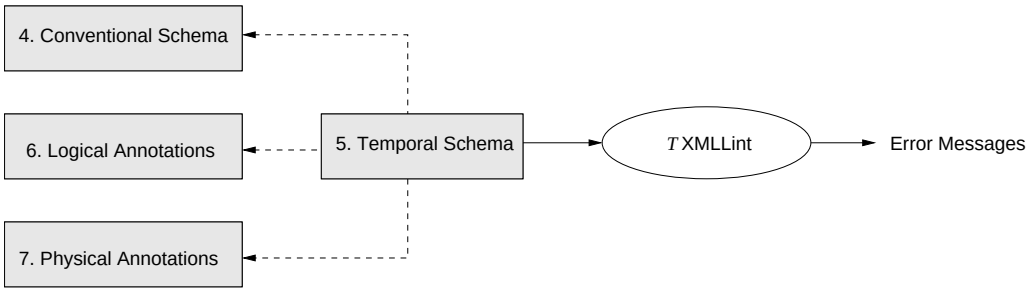


Figure 21: τXMLLINT: Checking the schemas

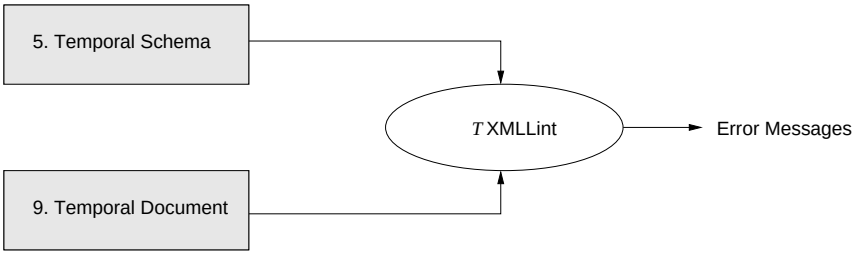


Figure 22: τXMLLINT: Checking the instance

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

Let's now turn our attention to the tools that operate on these various specifications. The temporal schema document (box 4) is passed through τXMLLINT (see Figure 21) which checks to ensure that the temporal and physical annotations are consistent with the conventional schema. The temporal validator (τXMLLINT) utilizes the conventional validator (e.g., XMLLINT) for many of its checks. For instance, it validates the logical annotations against the ASchema. But it also checks that the logical annotations are not inconsistent. Similarly, the physical annotation document is passed through τXMLLINT to ensure consistency of the physical annotations. The temporal constraint checker then evaluates the temporal constraints expressed in the schema (see Section 15 for more details). Finally, the temporal validator reports whether the temporal document was valid or invalid.

Once the annotations are found to be consistent, the *Schema Mapper* (software oval, Figure 20) generates the *representational schema* (box 9) from the original conventional schema and the logical and physical annotations. The representational schema is needed to serve as the schema for a time-varying document/data (box 8). The time-varying data can be created in four ways:

1. automatically from the non-temporal data (box 7) using τXSchema's Squash tool (described in Section 9.4),

2. automatically from the data stored in a database, i.e., as the result of a "temporal" query or view,

3. automatically from a third-party tool, or

4. manually.

The time-varying data is validated against the representational schema in two stages. First, a conventional XML Schema validating parser is used to parse and validate the time-varying data since the representational schema is an XML Schema document that satisfies the snapshot validation subsumption property. But as emphasized in Section 2, using a conventional XML Schema validating parser is not sufficient due to the limitations of XML Schema in checking temporal constraints. For example, a regular XML Schema validating parser has no way of checking something as basic as "the valid time boundaries of a parent element must encompass those of its child". These types of checks are implemented in τXMLLINT. So the second step is to pass the temporal data to τXMLLINT as shown in Figure 22. A temporal XML data file (box 8) is essentially a timestamped representation of a sequence of non-temporal XML data files (box 7). The namespace is set to its associated XML Schema document (i.e., representational schema). The timestamps are based on the characteristics defined in the logical and physical annotations (boxes 5 and 6). The temporal validator, τXMLLINT, by checking the temporal data, effectively checks the non-temporal constraints specified by the conventional schema simultaneously on all the instances of the non-temporal data (box 7), as well as the constraints between snapshots, which cannot be expressed in a conventional schema.

To reiterate, the conventional approach to storing timestamped data would require the user start with a representational schema (box 9) and use it to validate the temporal data (box 8). Both these documents become very complex if time varying data and schema are to be handled, and are non-intuitive to work with directly. Our proposed approach is to have the user design a conventional schema, add logical and physical annotations (boxes 5 and 6), leading to the representational schema (and temporal data) being automatically generated. In the second part of this technical report (Section 11 onwards), we discuss the user specification of the temporal schema (box 4), which is only needed if the conventional schema (box 3) and annotation documents (boxes 5 and 6) themselves can vary.

9 Tools and Algorithms

Our three-level schema specification approach enables a suite of tools operating both on the schemas and the data they describe. This section gives an overview of the suite of tools and the algorithms used by them.

The tools are open-source and beta versions are available [65]. The tools have been implemented in Java using the DOM API [82]. The DOM API was chosen over SAX API due to its ability to create an object-oriented hierarchical representation of the XML document that can be navigated and manipulated at run-time. The primitives explained below use this ability of the DOM API to easily manipulate the document-tree.

We first describe the details of the implementation primitives pushUp, pushDown and coalesce. These primitives are used by τXMLLINT, SQUASH, UNSQUASH, and RESQUASH tools for manipulating XML trees. SCHEMA MAPPER, a logical-to-representational mapper, is introduced next. This tool takes as input the conventional schema, logical and physical annotations, and generates a representational schema. This representational schema is used by τXMLLINT to validate the given temporal document using a conventional XML Schema validator. τXMLLINT does the actual temporal schema and data validation. Temporal data validation is a several-step process, a major part of this process being gluing elements to form items. The items are then validated individually.

Other tools in the suite squash, unsquash and resquash the documents. Given a temporal schema and a set of conventional documents, SQUASH combines all of the conventional documents into a single temporal document. UNSQUASH performs the opposite operation, breaking the single temporal document into multiple conventional documents. RESQUASH is just a combination of UNSQUASH and SQUASH; given a temporal document, an old physical annotation and a new physical annotation, RESQUASH changes the representation of the given document as per the new physical annotation.

9.1 Implementation Primitives

As mentioned earlier, the logical and physical annotations are orthogonal in nature; a user can change the location of timestamps, independent of specifying the temporal characteristics of a particular element. The representation of the temporal document will change accordingly. Thus, two documents having a single logical annotation can have different physical annotations and hence different representations.

While processing a temporal document, one of the most frequently needed operations on the temporal document moves the timestamps *up* or *down* in the hierarchy of XML elements defined by original snapshot schema. Another operation needed by both τXMLLINT and SQUASH utilities *coalesce*s the adjacent versions from a given item. We decided to write primitive functions for these operations so that they could be reused for building the tools with minimum efforts.We now describe the primitive functions representing these operations.

9.1.1 The pushUp Function

Although logical and physical annotations are orthogonal in nature, one restriction on the physical annotation is that, at least a single timestamp should be located at or above the topmost time-varying element in the XML schema hierarchy. If a given physical annotation has timestamps at locations other than the time-varying elements, the pushUp function moves the timestamps up in the hierarchy after coalescing the items.

Consider the conventional schema in Listing 62 and corresponding logical annotation (Listing 63) and physical annotation (Listing 64). Figures 23–26 depict step by step working of the pushUp function when applied to a temporal document having timestamps at the time-varying elements.

The first tree representation in Figure 23 represents the original document before applying the pushUp function. The timestamps are present at element <B>, which is temporal in nature (i.e., present in the logical annotation). The pushUp function moves the timestamp to element <A>, which is present in the physical annotation. It results in the three copies of element <A> corresponding to the three versions of item B. Elements <A>, <C> and <D> are non-temporal in nature. Thus their contents are the same and hence are duplicated in all the three versions.

Listing 62: Conventional Schema

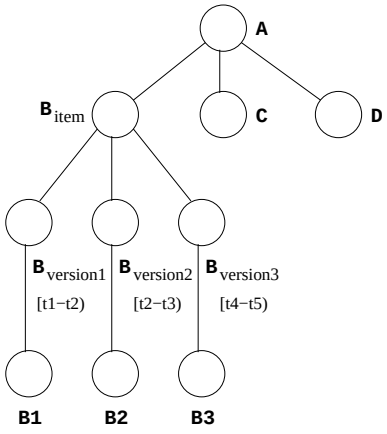
```
...
<element name="A">
  <complexType mixed="true">
    <sequence>
      <element name="B" type="string"/>
      <element name="C" type="string"/>
      <element name="D" type="string"/>
    </sequence>
  </complexType>
</element>
...
```

Listing 63: Logical Annotation

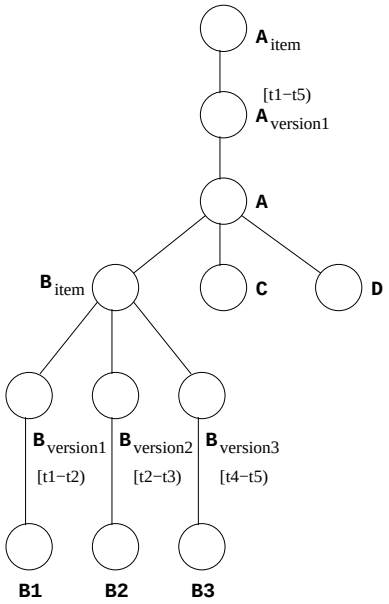
```
...
<item target="/A/B">
  <transactionTime/>
  <itemIdentifier name="A_id" timeDimension="transactionTime">
    <field path="./text"/>
  </itemIdentifier>
</item>
...
```

Listing 64: Physical Annotation

```
...
<stamp target="/A" dataInclusion="expandedVersion">
  <stampKind timeDimension="transactionTime" stampBounds="extent"/>
</stamp>
...
```

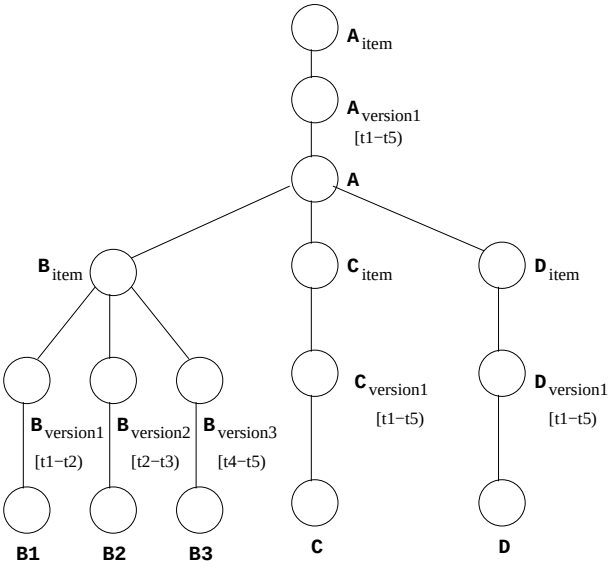


Original Document

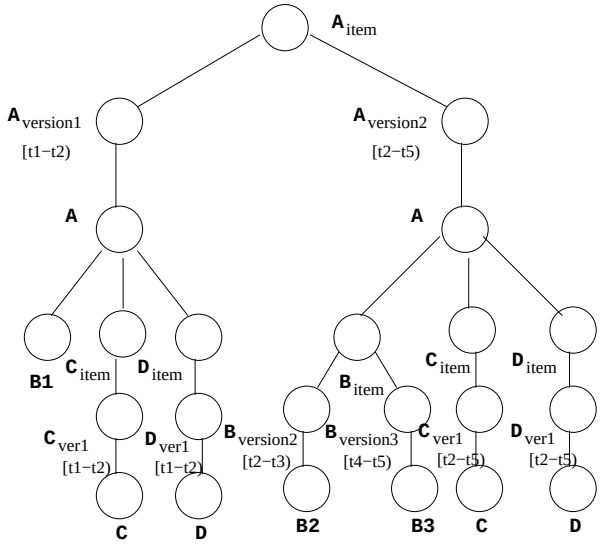


Before call pushUp *(A item , physicalAnnotation)*

Figure 23: Example of pushUp

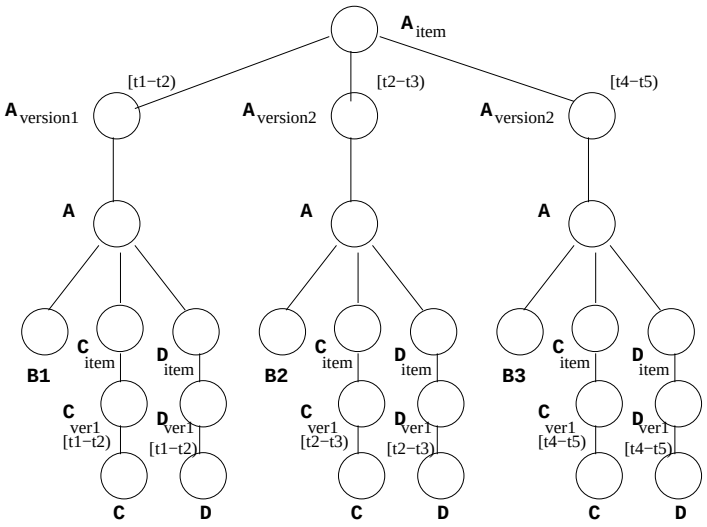


Before call to function splitChildVersions *(A item , physicalAnnotation)*



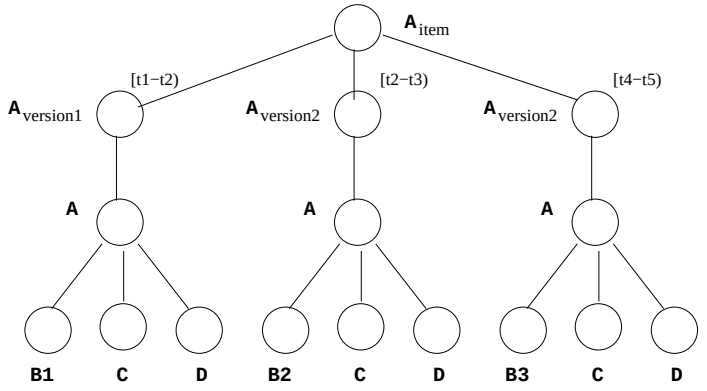
After first iteration of third **for** loop in function splitChildVersions

Figure 24: Example of pushUp: Continued



After second iteration of third **for** loop in function splitChildVersions

Figure 25: Example of pushUp: Continued



Final Result

Figure 26: Example of pushUp: Continued

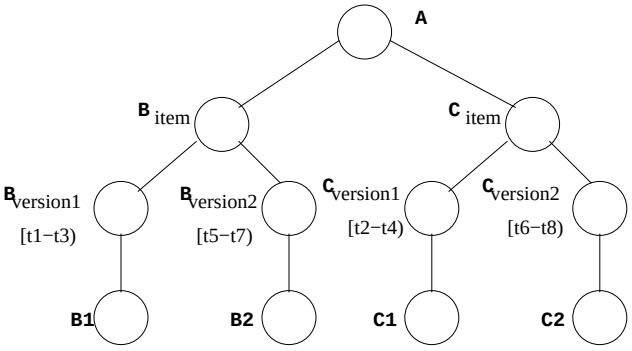
The pushUp function is used in SQUASH and RESQUASH tools. These tools first construct the temporal document with the timestamps located at the time-varying elements. The timestamps are then moved up in the hierarchy to the elements present in the physical annotation.

The recursive algorithm for pushUp is given in Figure 28. The function accepts an item representation of an XML element as one of its parameters. The algorithm is called on the root item in the temporal XML document. If the root element is not an item, it is converted into an item using createItem function before pushUp is called. The pushUp function recurses until it reaches the bottom of the XML tree. At that point, it moves timestamps up in the hierarchy by using the function splitChildVersions. The nested **for** loop in the function splitChildVersions may multiply the existing versions of the item by splitting them depending upon its versions's overlap with its child items' versions. The child items from the versions of the parent item are replaced by the child items' versions removing the child items not present in the physical annotation. The timestamp is thus pushed one level up in the hierarchy, closer to the elements present in the physical annotation.

Other helper functions used in the algorithm are as follows.

- isItem (*e*): The function checks whether the given XML element *e* has a representation of an item.
- createItem (*e*, *timePeriod*): The function creates a new XML element with the representation of an item and adds the given element *e* as the (single) version of newly create item with the time period of the version being *timePeriod*
- replace (*src*, *target*): The function replaces the *src* element with the *target* element.
- getTimePeriod (*itm*): The function returns the complete time-period of an item. i.e., The time-period with start time equal to the start time of the first version and end time equal to the end time of the last version of an item.

Figure 27 shows a slightly more complicated case, where two time-varying elements are siblings of each other. In this case, movement of timestamps *Up* in the hierarchy could result in the multiplication of the total number of versions depending upon the time overlap of individual versions from the sibling items. In this case, two versions of <B> and two versions of <C> give six versions of <A> after the application of pushUp function.



t1 < t2 < t3 < t4 & t5 < t6 < t7 < t8

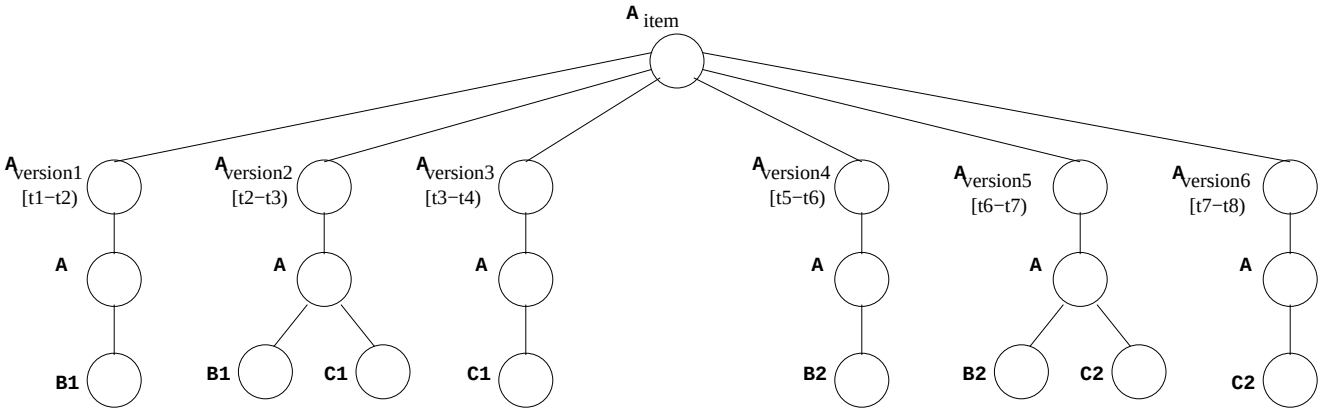


Figure 27: Example of pushUp

Figure 28: Algorithm: pushUp

```
//Inputs
// itm - An element from a temporal document which is an item
// physicalAnnotation - Parsed physical annotation document
//Output
// Modified itm element
function pushUp (itm, physicalAnnotation):
    for each version v of itm do
        for each child element c of v do
            if isItem(c)
                replace(c, pushUp(c, physicalAnnotation))
            else
                ci ← createItem(c, getTimePeriod(itm))
                replace(c, pushUp(ci, physicalAnnotation))
    splitChildVersions(itm, physicalAnnotation)
    return itm

//Inputs
// itm - An element from a temporal document which is an item
// physicalAnnotation - Parsed physical annotation document
function splitChildVersions (itm, physicalAnnotation):
    for each version v of itm do
        for each child element ci of v do
            if ci not in physicalAnnotation
                for each version cv of ci do
                    tpChild ← timePeriod(cv)
                    for each version v' of itm do
                        tp ← timePeriod(v')
                        if tpChild coincides with tp
                            ci' ← the child item of v' corresponding to cv
                            replace(ci', cv)
                        else if tpChild and tp overlap
                            partition tp and tpChild
                            tp' and tpChild' ← the partitions that coincide
                            v" ← the version corresponding to tp'
                            ci' ← the child item of v" corresponding to cv
                            replace(ci', cv)
```

9.1.2 The pushDown Function

The pushDown function behaves exactly opposite of the pushUp function. If a given physical annotation has timestamps at locations above the time-varying elements, the pushDown function moves these timestamps down the hierarchy. After executing this function on the temporal document, timestamps will be located at the time-varying elements. At this point, since the temporal characteristics and the representation coincide, it becomes easier to perform coalescing on the resultant temporal document.

Consider the example in Figures 23–26. According to the physical annotation in Figure 64, the tree-structured representation of the temporal document is given in Figure 26. Although <B> is a time-varying element, timestamp is present at the element <A> higher up in the hierarchy. This results in the duplication of elements <A>, <C> and <D>. When pushDown function is applied to the above document, the timestamps are moved down the hierarchy, the redundancy is eliminated and the final document looks as shown in the first tree of Figure 23. At this point, the user might be wondering, what if the elements <C> and <D> are not the same in three different versions of <A> in the given temporal document. This would not happen, since the elements <C> and <D> are not defined to be time-varying in the temporal annotation; so they better be the same. If they are different, the algorithm would report this as an error.

The recursive algorithm for the pushDown function is given in Figure 29. The algorithm is called on the root element in the temporal XML document. If the root element is not an item, it is first converted to an item element using function createItem function. The algorithm moves the timestamps down the hierarchy one level at a time. If an item is not a time-varying element and if it has multiple versions (e.g., element <A> of Figure 25), it is converted into a single version by using the mergeVersions function. The function groups corresponding child elements having the same item-identifier from its different versions into the same child item. The child element from the first version is then replaced by its corresponding child item XML element. After merging, since the parent item has only single version, the item is replaced by its single version.

Other helper functions used in the algorithm are as follows.

- isTimeVarying (*itm*, *temporalAnnotation*): The function returns **true** if *itm* definition is present in the logical annotation.
- versionCount (*itm*): The function returns the number of versions present in the given *itm* element.
- GetVersion (*itm*, *n*): The function returns the *n*th version of the given *itm* element.

Figures 32, 33 and 34 depict the stepwise working of function pushDown. For the given tree, element <D> is temporal in nature but the timestamp is present at the element <A> which is two levels up in the hierarchy. In the first step, the timestamp is moved to element <B>, while in the next step, the timestamps are moved to element <D>, which is actually a time-varying element.

9.1.3 The coalesce Function

As explained in Section 5, elements in two snapshots of a temporal XML document can be temporally-associated. If the elements are DOM-equivalent and the snapshot periods are contiguous, those two elements could be replaced by a single element with the time period extending from the start time of the first element to the stop time of the last element. This process is termed *coalescing* and is an integral part of SQUASH to compact the document.

After the snapshots are glued and the items are formed, coalesce is called for each item. The algorithm for coalesce is given in Figure 31. The algorithm compares the time-periods of the two contiguous versions. If they meet, and if the contents of the two versions are the same (i.e., if they are DOM-Equivalent as

Figure 29: Algorithm: pushDown

```
//Inputs
// itm - An element from a temporal document which is an item
// temporalAnnotation - Parsed logical annotation document
//Output
// Modified itm element
function pushDown (itm, temporalAnnotation):
    if isTimeVarying(itm, temporalAnnotation)
        processChildElements(itm)
        return itm
    else
        if versionCount(itm) = 1
            processChildElements(itm)
            return GetVersion(itm, 1)
        else
            mergeVersions(itm, temporalAnnotation)
            processChildElements(itm)
            return GetVersion(itm, 1)

//Input
// itm - An element from a temporal document which is an item
function processChildElements (itm):
    for each version v of itm do
        childElementList ← {}
        for each child element c of v do
            if isItem(c)
                c' ← pushDown(c, temporalAnnotation)
            else
                ci ← createItem(c, getTimePeriod(itm))
                c' ← pushDown(ci, temporalAnnotation)
            childElementList ← childElementList ∪ c'
        for each child element c of v do
            replace(c, c')
```

Figure 30: Algorithm: mergeVersions

```
//Inputs
// itm - An element from a temporal document which is an item
// temporalAnnotation - Parsed logical annotation document
function mergeVersions (itm, temporalAnnotation):
    let v1 ← GetVersion(itm, 1)
    for each child c of v1 do
        if isTimeVarying(c, temporalAnnotation)
            ci ← createItem(c, getTimePeriod(itm))
            replace(c, ci)
        else
            retain c
    for each version v of itm starting from GetVersion(itm, 2) do
        for each child c of v do
            if isTimeVarying(c, temporalAnnotation)
                evaluate item-identifier for c
                add c as a version to item ci from v1
        remove version v from itm
```

Figure 31: Algorithm: coalesce

```
//Input
// itm - An element from a temporal document which is an item.
function coalesce(itm):
    let v1 ← GetVersion(itm, 1)
    for each version v of itm starting GetVersion(itm, 2) do
        v2 ← v
        if (v1.time-period meets v2.time-period and DOM-Equivalent(v1, v2))
            v1.time.end ← v2.time.end
            remove version v2 from itm
        else
            v1 ← v2
```

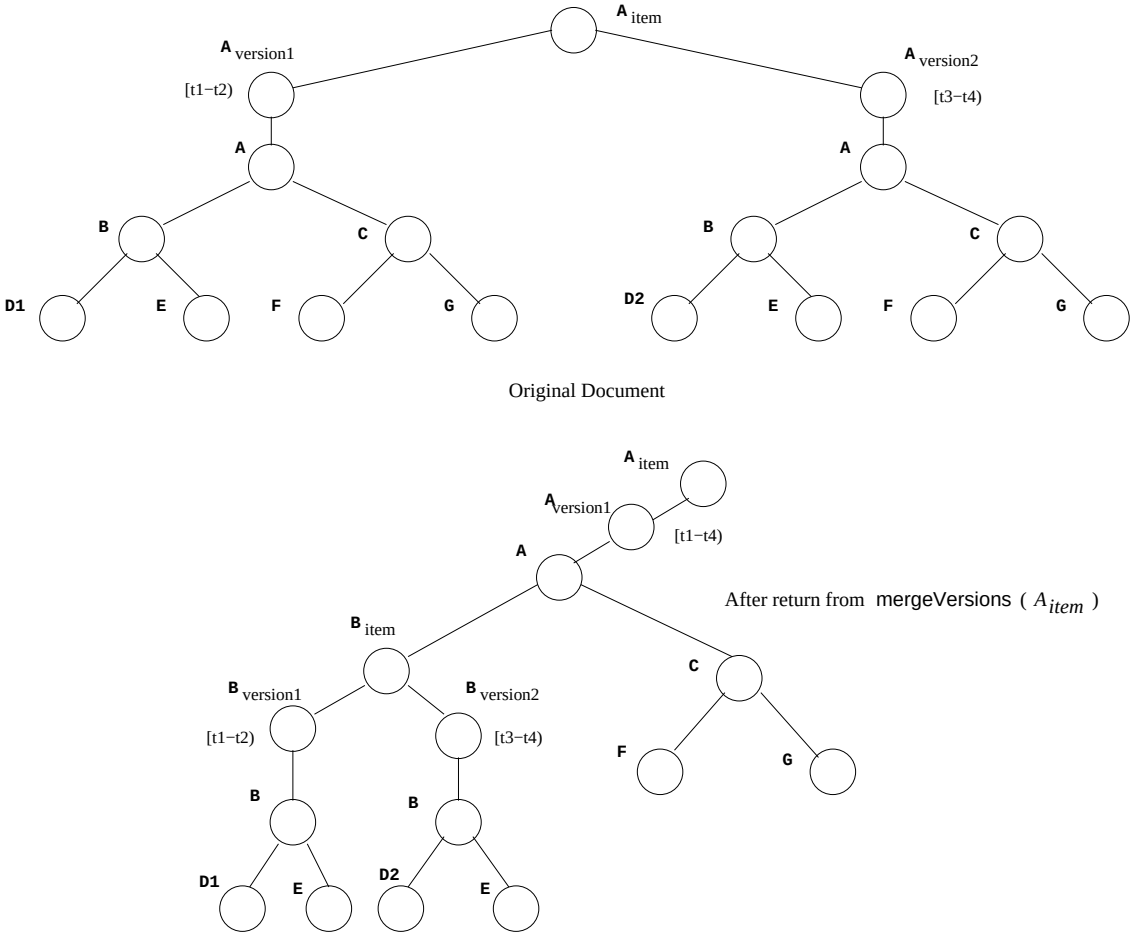


Figure 32: Example of pushDown

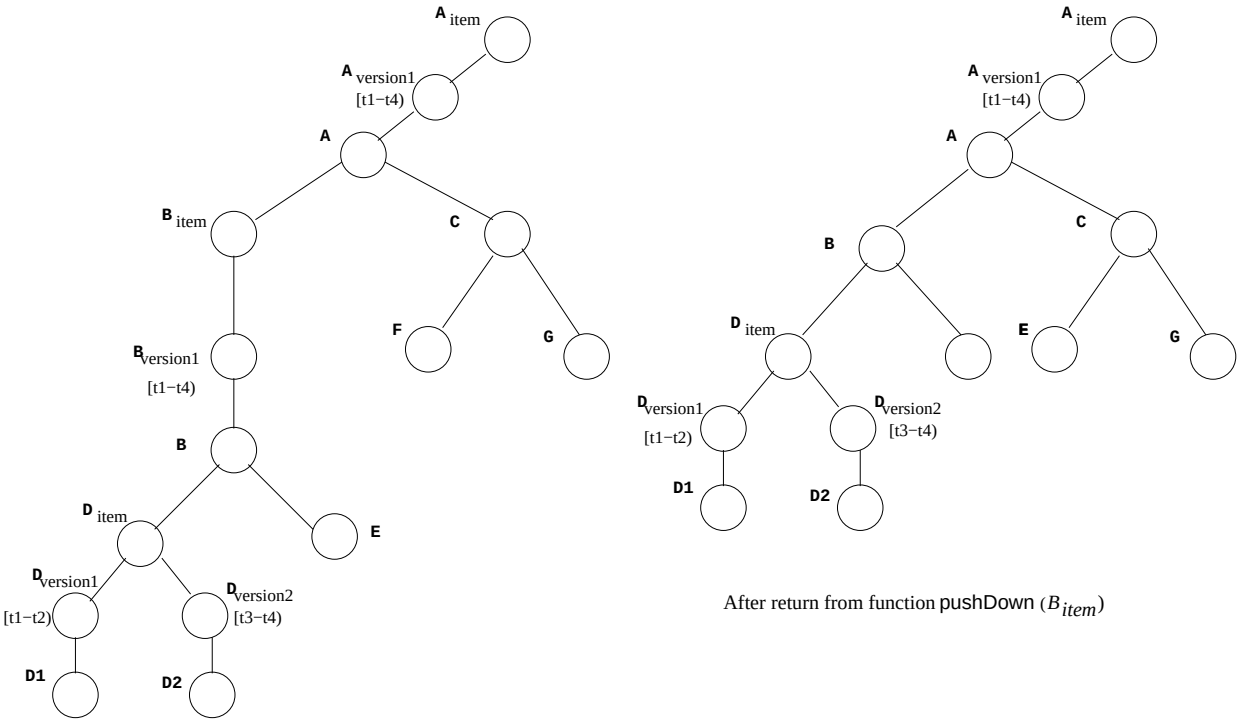


Figure 33: Example of pushDown: Continued

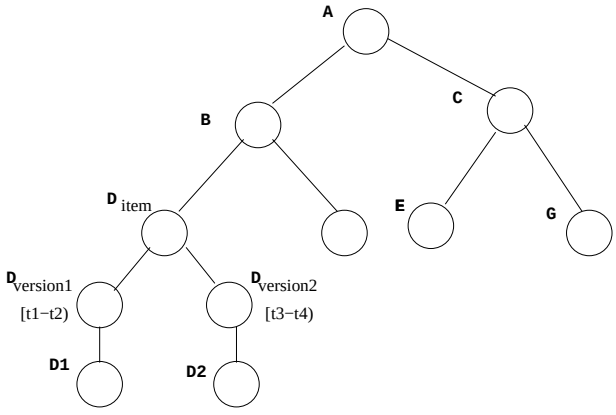


Figure 34: Example of pushDown: Continued

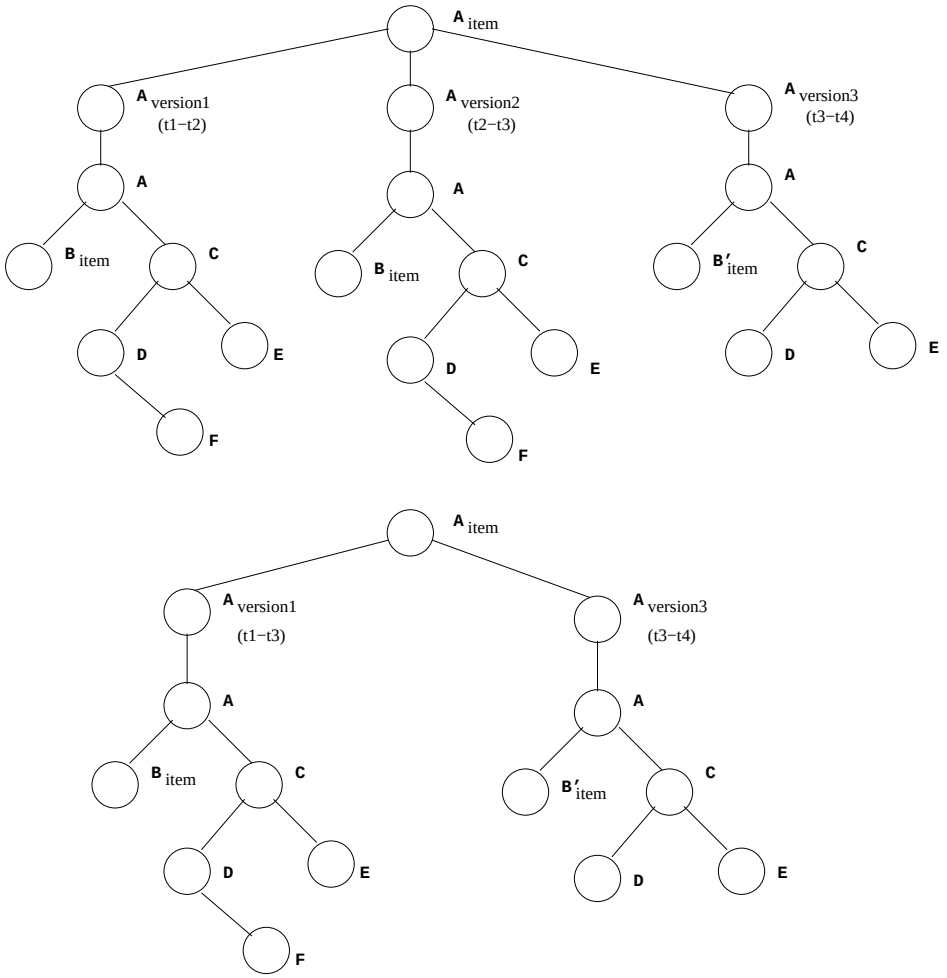


Figure 35: Example of coalesce

explained in Section 5.4), the stop time of the first version is then extended to the stop time of the second version.

Figure 35 shows the process of applying coalescing on Item A. In the tree-representation of the document, versions `A1 [t1-t2)` and `A2 [t2-t3)` are contiguous. They are also DOM-Equivalent (Section 5.4). Thus the two versions are replaced by a single version with time period `(t1-t3)`. After merging `A1` and `A2`, although the resulting version is contiguous with the next version `A3 [t3-t4)`, they are not merged, as they are not DOM-Equivalent. Thus, in the resulting document, there remain two versions `A1` and `A2`.

9.2 SCHEMA MAPPER

Once the annotations are found to be consistent, the logical-to-representational mapper generates the representational schema from the original conventional schema and the logical and physical annotations. The representational schema is needed to serve as the schema for a time-varying document.

Every time-varying element is given a timestamp for the valid time and/or the transaction time as appropriate. Non-time-varying elements and attributes are translated as is. The process of converting a conventional schema into the representational schema is explained in the next few paragraphs.

An XML Schema specification defines the types of elements and attributes that could appear in a document instance. More generally, the specification can be viewed as a (tree) grammar. The grammar consists of productions of the following form for each element type.

$$S \Rightarrow \texttt{<}S\texttt{>}\alpha\texttt{</}S\texttt{>}$$

In the above production, 'α' describes the contents of elements of type *S*.

A temporal schema denotes that some of the element types are time-varying. To construct a representational schema, several productions are added to the conventional schema for each time-varying element. No productions are removed from the non-temporal schema though some are modified. Since only elements can be temporal, this section focuses on the element-related components of a schema. The construction process consists of several steps. We will illustrate the process by describing what is done for a single, representative time-varying element type, *S*.

The first step is to add a production to indicate that the element type *S* is time varying, i.e., an item. The production has following form:

$$\textit{SItem} \Rightarrow \texttt{<}\textit{SItem}\ \texttt{itemId="}n\texttt{"}\ \texttt{>}\ \textit{SVersion}^{+}\ \texttt{</}\textit{SItem}\texttt{>}$$

An item has a unique `itemId` value, and consists of a list of versions. The third step is to add a production to specify each version of type *S*. The production for a version of an element of type *S* has the following form:

$$\textit{SVersion} \Rightarrow \texttt{<}\textit{SVersion}\texttt{>}\ t\ S\ \texttt{</}\textit{SVersion}\texttt{>}$$

where *t* is the definition of timestamp element and *S* is the non-temporal definition of the element's type. We do not impose a particular schema for a timestamp, rather we assume that the schema is given separately and adopted by the temporal document's schema. Each timestamp can have either or both of the following forms.

$$t \Rightarrow \texttt{<transactionTime start="..." stop="..."/>}$$

OR

$$t \Rightarrow \texttt{<validTime begin="..." end="..."/>}$$

The next step is to modify the context in which a time-varying element appears. For each time-varying element type, *S*, that appears in the left-hand-side of a production, replace *S* with *SItem*. For example, assume that the schema has a production of the following form:

$$X \Rightarrow \texttt{<}X\texttt{>}\ \beta\ S\ \gamma\ \texttt{</}X\texttt{>}$$

where β and γ describe arbitrary content before and after *S*, respectively. The production is replaced by the following production.

$$X \Rightarrow \texttt{<}X\texttt{>}\ \beta\ \textit{SItem}\ \gamma\ \texttt{</}X\texttt{>}$$

Only the element type is replaced, any other constraints on the element are kept (e.g., `minOccurs` and `maxOccurs` are unaffected).

The final step is to relax the uniqueness constraint imposed by a DTD identifier or XML Schema key definition. Since the same identifiers and key values can appear in multiple versions of an element, such values are no longer unique in a temporal document, even though they are unique within each snapshot. In temporal relational databases, the concept of a temporal key, which combines a snapshot key with a time, has been introduced. Temporal keys can be enforced by a temporal validating parser, but not by a conventional parser. So constraints that impose uniqueness within a snapshot must be relaxed or redefined as follows. The value of each id type attribute in a time-varying element is rewritten to be a unique value. Finally, schema keys are rewritten to include itemIds and version start and end times, creating a temporal key.

The algorithm for SCHEMA MAPPER is shown in Figure 36. The algorithm uses the same procedure explained in the above paragraphs to create the representational schema from the conventional schema. The helper function isConsistent checks whether the physical annotation is consistent with the given conventional schema. As part of consistency, it checks whether all the targets in the physical annotation are present in the conventional schema.

Figure 36: Algorithm: SCHEMA MAPPER

```
//Inputs
// conventionalSchema - Parsed snapshot schema document
// physicalAnnotation - Parsed physical annotation document
//Output
// Modified conventionalSchema document
function doSchemaMapping (conventionalSchema, physicalAnnotation):
    if isConsistent(conventionalSchema, physicalAnnotation)
        for each element e in physicalAnnotation do
            add following definitions to conventionalSchema

            <xs:element name="eItem">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="eVersion">
                            <xs:complexType>
                                <xs:sequence>
                                    <tv:element ref="timeStamp"/>
                                    <xs:element ref="e" />
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                    <xs:attribute name="itemID" type="ID"/>
                </xs:complexType>
            </xs:element>

            for each reference of e do
                replace <xs:element ref="e" /> with <xs:element ref="eItem" />

        add following definition to the conventionalSchema
        <xs:element name="temporalRoot">
            <xs:complexType>
                <xs:element ref="currentRoot" />
            </xs:complexType>
        </xs:element>

        return modified conventionalSchema
    else
        display error
```

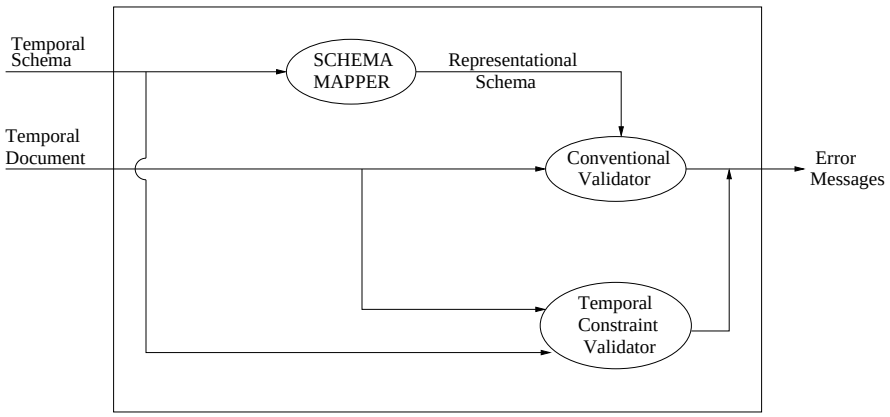


Figure 37: Validating a document with Time-Varying Data

9.3 τXMLLINT

In Section 8, we introduced the various components of the τXSchema architecture, including the validator. In this section, we explain τXMLLINT component in detail. Figure 37 provides the validation procedure used by τXMLLINT . The temporal schema document (box 4 of Figure 20) is passed through the τXMLLINT which first checks to ensure that the logical and physical annotations are consistent with the conventional schema and with each other. Once the annotations are found to be consistent, the logical-to-representational Mapper (SCHEMA MAPPER) generates the representational schema (box 9) from the original conventional schema and the logical and physical annotations. The representational schema is needed to serve as the schema for a time-varying document and is used to validate the temporal document using conventional validator.

Once the representational schema is ready, a conventional validator is used to parse and validate the time-varying data. τXMLLINT utilizes the conventional validator for many of its checks. For instance, it validates the logical and physical annotations against the ASchema. However, using a conventional XML Schema validating parser is not sufficient due to the limitations of XML Schema in checking temporal constraints. So the second step is to pass the temporal data to *Temporal Constraint Validator Module*. The module, by checking the temporal data, effectively checks the non-temporal constraints specified by the conventional schema simultaneously on all the instances of the non-temporal data (box 7), as well as the (non-sequenced temporal) constraints between snapshots, which cannot be expressed in a snapshot schema.

Figures 38 and 39 depict the two tasks performed by the τXMLLINT: (i) validating the consistency of a temporal schema and (ii) validating the instance of a temporal document against the temporal schema. Section 8 describes further details of how time-varying data is validated against the representational schema.

τXMLLINT has a *gluing component* that creates all the items and their item identifiers. Two elements with the same item identifiers should be glued together. It concatenates all of the fields together. It creates one string that is the schema for all the fields and a second string that is the value of all the fields. Element and attribute names cannot contain the '|' symbol since it is used to separate each field string in the concatenated string. The fields are concatenated in the order specified in the item identifier.

τXMLLINT maintains a hash map to hold all the items. Each item contains a reference to each of its constituent elements. Two elements are glued if their item identifiers match exactly. Both the schema and instance strings must be equal. Even the amount and location of white spaces in a field elements loose text must be identical. For every time-varying element, the gluing component determines whether to create a new item or to glue this element to an existing item.

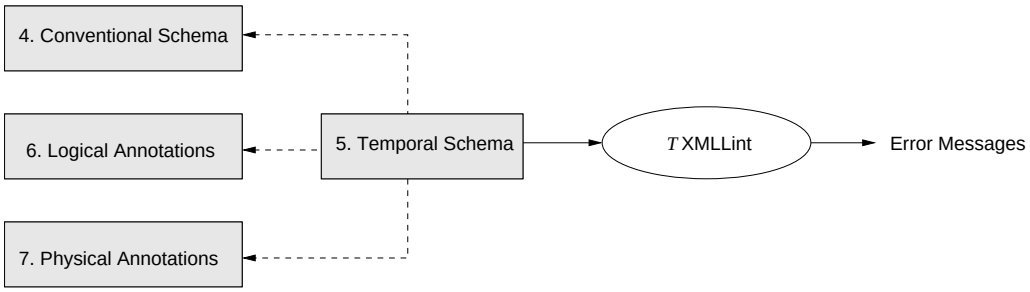


Figure 38: τXMLLINT – Checking the Schema

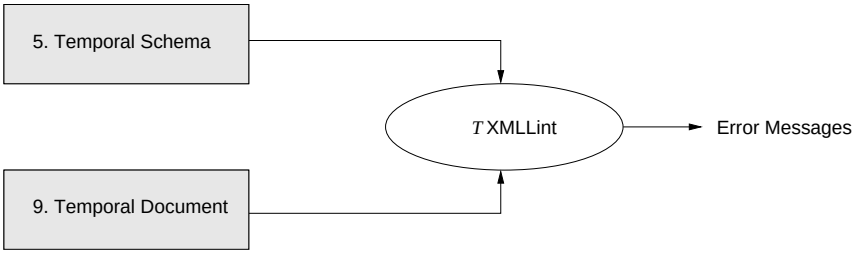


Figure 39: τXMLLINT – Checking the Instance

Once the items are created, the *Temporal Constraint Validator Module* validates individual item to check whether it satisfies the following constraints, if applicable to that item.

Content Constant: Content of an element cannot vary over time.

Existence Constant: The element cannot disappear and reappear again.

Content Varying Applicability: The contents of an item cannot change beyond the period specified by the `contentVaryingApplicability` element in the logical annotation.

Valid Time Frequency: The element cannot change more than specified number of times specified by the `frequency` element.

Maximal Existence Period: The element can exist only within the period specified by the `maximalExistence` element.

By checking the constraints on all the items, the module effectively checks for all the sequenced and non-sequenced constraints on the entire temporal document.

The algorithm for τXMLLINT is given in Figure 40. The algorithm uses a hash-map to maintain a mapping between item-identifier and the corresponding item. After checking the consistency of the schemas, the function creates a representational schema using the SCHEMA MAPPER. The given temporal document is parsed against this schema using the conventional validator. The for loop creates the items by gluing together the elements with the same item-identifier. Each item is then validated for sequenced and non-sequenced constraints explained in Section 6.

Figure 40: Algorithm: τXMLLINT

```
//Inputs
// conventionalSchema - Parsed snapshot schema document
// temporalAnnotation - Parsed logical annotation document
// physicalAnnotation - Parsed physical annotation document
// temporalDocument - Parsed temporal document
function doTemporalValidation (conventionalSchema, temporalAnnotation, physicalAnnotation,
                               temporalDocument):
    initialize a hash-table with item-identifier as key and item as hash value
    if Consistent(conventionalSchema, temporalAnnotation, physicalAnnotation)
        repSchema ← doSchemaMapping(conventionalSchema, physicalAnnotation)
        if conventionalValidator(temporalDocument, repSchema)
            for each element e in the temporalDocument do
                if isTimeVarying(e, temporalAnnotation)
                    evaluate the item-identifier
                    if item-identifier in hash-table
                        if the element is DOM-equivalent to some version in the item
                            coalesce the metadata with the version
                        else
                            create a new version
                    else
                        create a new item in hash-table, with one version
            for each item in hash-table do
                for each sequenced and non-sequenced constraint in temporalAnnotation do
                    if the constraint is not satisfied
                        display errors
        else
            display errors generated by the conventional validator
    else
        display errors
```

9.4 SQUASH

The SQUASH utility takes a sequence of XML documents, a logical annotation and a physical annotation as input and generates a temporal XML document consistent with the physical annotation.

The algorithm for SQUASH tool is given in Figure 41. It cleverly reuses pushUp, pushDown and coalesce primitives to create a compressed document from a set of conventional documents as per the given temporal schema.

The algorithm first checks for the consistency of the logical and the physical annotations with the conventional schema. It then creates a new XML document with `<temporalRoot>` as its root and attaches *root* elements of the conventional documents as its versions. At this point, the timestamps are present at the root level element. pushDown function then moves these timestamps down the hierarchy to the elements present in the logical annotation. Every item is then coalesced to create its compact representation. The pushUp function then moves the timestamps up in the hierarchy up to the elements present in the actual physical annotation.

9.5 UNSQUASH

The UNSQUASH utility performs the opposite operation of SQUASH. It takes a temporal XML document and a temporal schema and generates multiple non-temporal XML documents. It also provides the functionality of extracting a particular snapshot from the given temporal document using UNSQUASH utility. The algorithm for UNSQUASH is given in Figure 42.

The algorithm first checks for the consistency of the logical and physical annotations with the snapshot schema. It then constructs the representational schema using SCHEMA MAPPER and parses the given temporal document against the representational schema using the conventional validator. The pushDown function is first called on the given document to move the timestamps to the time-varying elements. A new physical annotation, containing only the root element, is created and passed to the function pushUp. The purpose is to move all the timestamps to the *root* element. At this moment every version of the *root* item element is a conventional document. These individual versions are then written to the separate files.

9.6 RESQUASH

The RESQUASH utility takes the temporal XML data and the two physical annotated schemas (the original schema and the target one) and converts the temporal XML document based on the target physical annotated schema. The algorithm for RESQUASH is given in Figure 43.

The algorithm first checks for the consistency of the logical annotation and the source and target physical annotations with the conventional schema. It then performs the operation pushDown on the given temporal document. The given temporal document has the representation as per the *srcPhysicalAnnotation*. The pushDown function moves all the timestamps to the actual time-varying elements as per the *temporalAnnotation*. The function pushUp is then called with the *targetPhysicalAnnotation* as its parameter, which then moves the timestamps up in the hierarchy to the elements mentioned in the new physical annotation.

Figure 41: Algorithm: SQUASH

```
//Inputs
// conventionalSchema - Parsed snapshot schema document
// logicalAnnotation - Parsed logical annotation document
// physicalAnnotation - Parsed physical annotation document
// snapshotSet - Set of snapshot documents
//Output
// temporalDocument - Temporal document created from snapshotSet
function doSquash (conventionalSchema, logicalAnnotation, physicalAnnotation, snapshotSet):
    if Consistent(conventionalSchema, logicalAnnotation, physicalAnnotation)
        repSchema ← doSchemaMapping(conventionalSchema, physicalAnnotation)
        create element <temporalRoot
                        beginDate="beginDate of first snapshot document"
                        endDate="endDate of last snapshot document">
        create element rootItm corresponding to root level element root
        for each snapshot in the set of snapshotSet do
            add root element root of snapshot as a version of rootItm
        root ← pushDown(rootItm, logicalAnnotation)
        for each item itm in temporalDoc do
            coalesce(itm)
        if isItem(root)
            rootItm ← root
        else
            rootItm ← createItem(root)
        rootItm ← pushUp(rootItm, physicalAnnotation)
        if rootItm not in physicalAnnotation
            replace(rootItm, getVersion(rootItm, 1))
        return temporalDoc
    else
        display errors.
```

Figure 42: Algorithm: UNSQUASH

```
//Inputs
// conventionalSchema - Parsed snapshot schema document
// logicalAnnotation - Parsed logical annotation document
// physicalAnnotation - Parsed physical annotation document
// temporalDocument - Temporal document created from above
//Output
// snapshotSets - Set of snapshots extracted from temporalDocument
function doUnSquash(conventionalSchema, temporalAnnotation, physicalAnnotation,
        conventionalDocument):
    if Consistent(conventionalSchema, logicalAnnotation,physicalAnnotation)
        repSchema ← doSchemaMapping(conventionalSchema,physicalAnnotation)
        if conventionalValidator(temporalDocument, repSchema)
            newPhysicalAnnotation ← root element definition of the conventionalSchema
            root ← temporalDocument.rootElement
            if isItem(root)
                rootItm ← root
            else
                rootItm ← createItem(root)
            root ← pushDown(rootItm, logicalAnnotation)
            if isItem(root)
                rootItem ← pushUp(root, newPhysicalAnnotation)
            else
                rootItem ← newItem(root)
                replace (root, pushUp(rootItem, newPhysicalAnnotation))
            snapshotSet ← {}
            for each version rootVer of rootItem do
                add element rootVer as a snapshot document to snapshotSet
            return snapshotSet
        else
            display errors generated by the conventional validator
    else
        display errors
```

Figure 43: Algorithm: RESQUASH

```
//Inputs
// conventionalSchema - Parsed snapshot schema document
// logicalAnnotation - Parsed logical annotation document
// temporalDocument - Temporal document to be resquashed
// srcPhysicalAnnotation - Parsed physical annotation document used for creating
                                        temporalDocument
// targetPhysicalAnnotation - Parsed physical annotation document to be used
                                        for creating new temporalDocument
//Output
// temporalDocument - resquashed temporal document
function doReSquashing (conventionalSchema, temporalAnnotation, srcPhysicalAnnotation,
                                targetPhysicalAnnotation,temporalDocument):
    if Consistent(conventionalSchema, logicalAnnotation, srcPhysicalAnnotation) and
            Consistent(conventionalSchema, temporalAnnotation, targetPhysicalAnnotation)
        root ← temporalDocument.rootElement
        if isItem(root)
            rootItem ← pushDown(root, logicalAnnotation)
        else
            rootItem ← newItem(root)
            replace(root, pushDown(rootItem, logicalAnnotation))
        rootItem ← pushUp(rootItem, targetPhysicalAnnotation)
        if rootItm not in physicalAnnotation
            replace(rootItm, getVersion(rootItm, 1))
        return temporalDocument
    else
        display errors
```

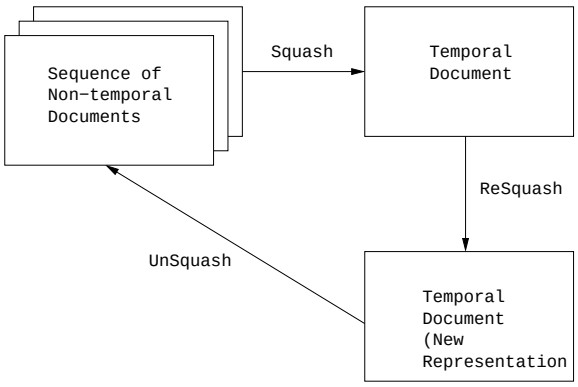


Figure 44: Squash/UnSquash/ReSquash Commutativity Diagram

It is also possible to work with two logical annotation documents (the original one and the target one) instead of physical ones, and convert the temporal XML document based on the target logical annotations. The only restriction with the logical annotations is that the data needs to be consistent with both sets of logical annotations. This constraint does not exist with the physical annotations because only the representation of a temporal document changes. This could be easily achieved by using the combination of UNSQUASH and SQUASH tools. The given temporal document will be unsquashed to retrieve the original conventional documents. These snapshot documents will then be squashed using the target logical annotation and the original physical annotation. Since the physical annotation remains the same, the new document will be the same as the original one. Although, while performing the squashing using the target logical annotation, the SQUASH tool would find out any violations of the sequenced and non-sequenced constraints enforced by the target logical annotation.

SQUASH, UNSQUASH and RESQUASH tools retain snapshot reducibility [8] in that the commutativity diagram in Figure 44 is maintained. Specifically, if we take a particular sequence of static XML documents, each associated with a time slice, and squash them into a temporal XML document, then resquash that into a separate temporal XML document, with a different physical schema, and then unsquash it again, we will get exactly the same sequence of static XML documents. This of course assumes that the static documents corresponding to the non-temporal schema provided and that the temporal XML documents are valid instances of the schema produced by the schema mapper.

10 Example Schema and Instance Documents

10.1 WinOlympic Example

Listing 65: Conventional schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

  <xs:element name="winOlympic">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="numEvents" type="xs:nonNegativeInteger"/>
        <xs:element ref="country" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="country">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element ref="athleteTeam"/>
      </xs:sequence>
      <xs:attribute name="countryName" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="athleteTeam">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element ref="athlete" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="numAthletes" type="xs:positiveInteger" use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="athlete">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="athName" type="xs:string"/>
        <xs:element ref="medal" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="phone" type="phoneNumType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="medal">
    <xs:complexType mixed="true">
      <xs:attribute name="mtype" type="medalType" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="medalType">
    <xs:restriction base="xs:string">
      <xs:pattern value="bronze|silver|gold"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="phoneNumType">
    <xs:restriction base="xs:string">
      <xs:length value="12"/>
      <xs:pattern value="\d{3}-\d{3}-\d{4}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Listing 66: Conventional document on 1 January 2002.

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="winOlympic.xsd">

  There are <numEvents>11</numEvents> events in the Olympics.
  <country countryName="Norway">
    <athleteTeam numAthletes="95">
      Athletes will take part in various events. The
      athletes participating are listed below
      <athlete>
        <athName>
          Kjetil Andre Aamodt
        </athName>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His telephone numbers are:
        <phone>123-402-0340</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

Listing 67: Conventional document on 1 March 2002.

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="winOlympic.xsd">

  There are <numEvents>11</numEvents> events in the Olympics.
  <country countryName="Norway">
    <athleteTeam numAthletes="95">
      Athletes will take part in various events. The
      athletes participating are listed below
      <athlete>
        <athName>
          Kjetil Andre Aamodt
        </athName>
        was the recipient of the
        <medal mtype="silver">Men's Combined</medal>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His telephone numbers are:
        <phone>123-402-0430</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

Listing 68: Conventional document on 1 July 2002.

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="winOlympic.xsd">

  There are <numEvents>11</numEvents> events in the Olympics.
  <country countryName="Norway">
    <athleteTeam numAthletes="95">
      Athletes will take part in various events. The
      athletes participating are listed below
      <athlete>
        <athName>
          Kjetil Andre Aamodt
        </athName>
        was the recipient of the
        <medal mtype="gold">Men's Combined</medal>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His telephone numbers are:
        <phone>123-402-0430</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

Listing 69: Temporal schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<temporalSchema xmlns="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">

  <conventionalSchema>
    <sliceSequence>
      <slice location="winOlympic.xsd" begin="2002-01-01" />
    </sliceSequence>
  </conventionalSchema>

  <annotationSet>
    <include schemaLocation="annotations.xml"/>
  </annotationSet>

</temporalSchema>
```

Listing 70: Annotation document.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationSet xmlns="http://www.cs.arizona.edu/tau/tauXSchema/ASchema">

  <physical>
    <stamp target="/winOlympic" dataInclusion="expandedVersion">
      <stampKind timeDimension="transactionTime" stampBounds="step"/>
    </stamp>
    <stamp target="/winOlympic/country" dataInclusion="expandedVersion">
      <stampKind timeDimension="validTime" stampBounds="extent"/>
    </stamp>
    <stamp target="/winOlympic/country/athleteTeam/@numAthletes"
           dataInclusion="expandedVersion">
      <stampKind timeDimension="validTime" stampBounds="extent">
```

```
        <format plugin="XMLSchema" granularity="gMonth"/>
      </stampKind>
    </stamp>
    <stamp target="/winOlympic/country/athleteTeam/athlete"
           dataInclusion="expandedVersion">
      <stampKind timeDimension="bitemporal" stampBounds="extent" />
    </stamp>
    <stamp target="/winOlympic/country/athleteTeam/athlete/medal"
           dataInclusion="expandedVersion">
      <stampKind timeDimension="bitemporal" stampBounds="extent" />
    </stamp>
    <stamp target="/winOlympic/country/athleteTeam/athlete/medal/medalType"
           dataInclusion="expandedVersion">
      <stampKind timeDimension="transactionTime" stampBounds="extent" />
    </stamp>
    <stamp target="/winOlympic/country/athleteTeam/athlete/phone"
           dataInclusion="expandedVersion">
      <stampKind timeDimension="bitemporal" stampBounds="extent" />
    </stamp>
  </physical>

  <logical>
    <item target="/winOlympic">
      <transactionTime/>
      <itemIdentifier name="olympicId1" timeDimension="transactionTime">
        <field path="//text"/>
      </itemIdentifier>
    </item>
    <item target="/winOlympic/country">
      <validTime kind="state" content="constant" existence="varyingWithGaps">
        <maximalExistence begin="1924-01-01" />
      </validTime>
      <itemIdentifier name="countryId1" timeDimension="validTime">
        <field path="@countryName"/>
      </itemIdentifier>
    </item>
    <item target="/winOlympic/country/athleteTeam">
      <attribute name="numAthletes">
        <validTime kind="state" content="varying"/>
      </attribute>
    </item>
    <item target="/winOlympic/country/athleteTeam/athlete">
      <validTime kind="state"/>
      <transactionTime/>
      <itemIdentifier name="atheleteId1" timeDimension="bitemporal">
        <field path="athName"/>
      </itemIdentifier>
    </item>
    <item target="/winOlympic/country/athleteTeam/athlete/medal">
      <validTime kind="event"/>
      <transactionTime/>
      <itemIdentifier name="medalId1" timeDimension="bitemporal">
        <field path="//text"/>
        <field path="../athname"/>
  <!-- Should not glue across winOlympic elements (i.e., across validTime).    -->
  <!-- Could have Kjetil winning the gold in Men's combined in 2002 and 2006. -->
      </itemIdentifier>
      <attribute name="medalType">
        <transactionTime />
      </attribute>
    </item>
    <item target="/winOlympic/country/athleteTeam/athlete/phone">
      <validTime kind="state" content="varying" existence="varyingWithGaps"/>
        <transactionTime/>
      <itemIdentifier name="phoneId1" timeDimension="bitemporal">
        <field path="//text"/>
      </itemIdentifier>
    </item>
```

```
  </logical>
</annotationSet>
```

Listing 71: Temporal document.

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

  <sliceSequence>
    <slice location="slice1.xml" begin="2002-01-01" end="2002-03-01" />
    <slice location="slice2.xml" begin="2002-03-01" end="2002-07-01" />
    <slice location="slice3.xml" begin="2002-07-01"/>
  </sliceSequence>

</temporalDocument>
```

10.2 Company Example

Listing 72: Conventional schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xs:element name="company">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element ref="companyData"/>
        <xs:element ref="supplier" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="product"  minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="productKey">
      <xs:selector xpath="company/product"/>
      <xs:field xpath="@productNo"/>
    </xs:key>
    <xs:keyref name="oProductKey" refer="productKey">
      <xs:selector xpath="company/order"/>
      <xs:field xpath="oProductNo"/>
    </xs:keyref>
  </xs:element>
  <xs:element name="companyData">
    <xs:complexType mixed="true">
      <xs:all>
        <xs:element name="companyName" type="xs:string" minOccurs="1"
          maxOccurs="1"/>
        <xs:element name="cURL" type="xs:string" minOccurs="0"
          maxOccurs="1"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="supplier">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="sURL" type="xs:string" minOccurs="0"
          maxOccurs="unbounded"/>
        <xs:element name="sRating" type="xs:string" minOccurs="0"
          maxOccurs="1"/>
        <xs:element name="order" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType mixed="true">
```

```
            <xs:sequence>
              <!-- orderNo is unique within a supplier -->
              <xs:element name="orderNo" type="xs:integer" minOccurs="1"/>
              <xs:element name="oProductNo" minOccurs="1" maxOccurs="unbounded"/>
              <xs:element name="oQty"    type="xs:integer" minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="orderType" type="orderType" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="supplierNo"   type="xs:integer" use="required"/>
      <xs:attribute name="supplierName" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="product">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:choice>
          <xs:element name="priceinDollars" type="xs:float" minOccurs="1" maxOccurs="1"/>
          <xs:element name="priceinPounds"  type="xs:float" minOccurs="1" maxOccurs="1"/>
          <xs:element name="priceinEuros"   type="xs:float" minOccurs="1" maxOccurs="1"/>
        </xs:choice>
        <xs:element name="qtyOnHand" type="xs:integer" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="productNo"   type="xs:integer" use="required"/>
      <xs:attribute name="productName" type="xs:string"  use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="orderType">
    <xs:restriction base="xs:string">
      <xs:pattern value="normal|rush"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Listing 73: Conventional document on 29 March 2004.

```
<?xml version="1.0" encoding="UTF-8"?>
<company
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="company.xsd">

  <companyData>
    <companyName>IBM</companyName>
    <cURL>http://www.ibm.com</cURL>
  </companyData>

  <supplier supplierNo="1" supplierName="Seagate" >
    <sURL>http://seagate.com</sURL>
    <sRating>AAA</sRating>
    <order orderType="normal">
      <orderNo>1</orderNo>
      <oProductNo>2</oProductNo>
      <oQty>50</oQty>
    </order>
  </supplier>

  <supplier supplierNo="2" supplierName="Wistron Corporation" >
    <sURL>http://www.wistron.com</sURL>
    <sRating>AA</sRating>
  </supplier>

  <supplier supplierNo="3" supplierName="small_supplier_1" >
  </supplier>

  <product productNo="1" productName="hard disk 73 GB 7200rpm">
    <priceinDollars>100</priceinDollars>
    <qtyOnHand>100</qtyOnHand>
```

```
  </product>

  <product productNo="2" productName="SCSI hard disk 147 GB 10000rpm">
    <priceinDollars>150</priceinDollars>
    <qtyOnHand>100</qtyOnHand>
  </product>

</company>
```

Listing 74: Conventional document on 30 March 2004.

```
<?xml version="1.0" encoding="UTF-8"?>
<company
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="company.xsd">

  <companyData>
    <companyName>IBM</companyName>
    <cURL>http://www.ibm.com</cURL>
  </companyData>

  <supplier supplierNo="1" supplierName="Seagate" >
    <sURL>http://seagate.com</sURL>
    <sRating>AAA</sRating>
    <order orderType="normal">
      <orderNo>1</orderNo>
      <oProductNo>2</oProductNo>
      <oQty>50</oQty>
    </order>
    <order orderType="rush">
      <orderNo>2</orderNo>
      <oProductNo>1</oProductNo>
      <oQty>100</oQty>
    </order>
  </supplier>

  <supplier supplierNo="2" supplierName="Wistron Corporation" >
    <sURL>http://www.wistron.com</sURL>
    <sRating>AA</sRating>
    <order orderType="normal">
      <orderNo>1</orderNo>
      <oProductNo>2</oProductNo>
      <oQty>10</oQty>
    </order>
  </supplier>

  <supplier supplierNo="3" supplierName="small_supplier_1" >
  </supplier>

  <product productNo="1" productName="hard disk 73 GB 7200rpm">
    <priceinDollars>100</priceinDollars>
    <qtyOnHand>40</qtyOnHand>
  </product>

  <product productNo="2" productName="SCSI hard disk 147 GB 10000rpm">
    <priceinDollars>125</priceinDollars>
    <qtyOnHand>80</qtyOnHand>
  </product>

</company>
```

Listing 75: Conventional document on 31 March 2004.

```
<?xml version="1.0" encoding="UTF-8"?>
<company
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="company.xsd">

  <companyData>
    <companyName>IBM</companyName>
    <cURL>http://www.ibm.com</cURL>
  </companyData>

  <supplier supplierNo="1" supplierName="Seagate" >
    <sURL>http://seagate.com</sURL>
    <sRating>AAA</sRating>
    <order orderType="normal">
      <orderNo>1</orderNo>
      <oProductNo>2</oProductNo>
      <oQty>50</oQty>
    </order>
    <order orderType="rush">
      <orderNo>2</orderNo>
      <oProductNo>1</oProductNo>
      <oQty>100</oQty>
    </order>
    <order orderType="normal">
      <orderNo>3</orderNo>
      <oProductNo>2</oProductNo>
      <oQty>25</oQty>
    </order>
  </supplier>

  <supplier supplierNo="2" supplierName="Wistron Corporation" >
    <sURL>http://www.wistron.com/indexNew.html</sURL>
    <sRating>AA</sRating>
    <order orderType="normal">
      <orderNo>1</orderNo>
      <oProductNo>2</oProductNo>
      <oQty>10</oQty>
    </order>
  </supplier>

  <supplier supplierNo="3" supplierName="small_supplier_1" >
  </supplier>

  <product productNo="1" productName="hard disk 73 GB 7200rpm">
    <priceinDollars>105</priceinDollars>
    <qtyOnHand>120</qtyOnHand>
  </product>

  <product productNo="2" productName="SCSI hard disk 147 GB 10000rpm">
    <priceinDollars>125</priceinDollars>
    <qtyOnHand>70</qtyOnHand>
  </product>

</company>
```

Listing 76: Temporal schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<temporalSchema xmlns="http://www.cs.arizona.edu/tau/tauXSchema/TS">

  <conventionalSchema>
    <sliceSequence>
      <slice location="company.xsd" begin="2004-03-29" />
    </sliceSequence>
  </conventionalSchema>

```

```
  <annotationSet>
    <include schemaLocation="annotations.xml" />
  </annotationSet>

</temporalSchema>
```

Listing 77: Annotation document.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationSet xmlns="http://www.cs.arizona.edu/tau/tauXSchema/ASchema">

  <physical>
    <stamp target="//company" dataInclusion="expandedVersion">
      <stampKind timeDimension="transactionTime" stampBounds="step"/>
    </stamp>
    <stamp target="/company/product" dataInclusion="expandedVersion">
      <stampKind timeDimension="validTime" stampBounds="step"/>
    </stamp>
    <stamp target="/company/supplier" dataInclusion="expandedVersion">
      <stampKind timeDimension="validTime" stampBounds="extent">
        <format plugin="XMLSchema" granularity="gMonth"/>
      </stampKind>
    </stamp>
    <stamp target="//order" dataInclusion="expandedVersion">
      <stampKind timeDimension="validTime" stampBounds="extent" />
    </stamp>
  </physical>

  <logical>
    <item target="/company/supplier">
      <validTime kind="state" content="varying" existence="varyingWithGaps"/>
      <transactionTime/>
      <itemIdentifier name="SupplierId1" timeDimension="bitemporal">
        <field path="@supplierNo"/>
      </itemIdentifier>
        <attribute name="supplierName">
        <validTime kind="state" content="varying"/>
      </attribute>
    </item>
    <item target="/company/product">
      <validTime kind="state" content="varying" existence="varyingWithGaps"/>
      <transactionTime/>
      <itemIdentifier name="ProductId1" timeDimension="bitemporal">
        <keyref refName="productKey" refType="snapshot"/>
      </itemIdentifier>
      <attribute name="productName">
        <validTime kind="state" content="varying"/>
        <transactionTime/>
      </attribute>
    </item>
    <item target="/company/supplier/order">
      <validTime kind="event"/>
      <transactionTime/>
      <itemIdentifier name="OrderId1" timeDimension="bitemporal">
        <field path="orderNo"/>
    </itemIdentifier>
      <attribute name="otype">
        <validTime kind="state" content="varying"/>
      </attribute>
    </item>
    <item target="/company/supplier/sURL">
      <validTime kind="state" existence="varyingWithGaps"/>
      <itemIdentifier timeDimension="validTime">
        <field path="."/>
      </itemIdentifier>
    </item>
    <item target="/company/supplier/sRating">
      <validTime kind="state" content="varying" existence="varyingWithoutGaps"/>
```

```
      <transactionTime/>
      <itemIdentifier timeDimension="validTime">
        <field path="."/>
      </itemIdentifier>
    </item>
    <item target="/company/product/qtyOnHand">
      <validTime kind="state" content="varying"/>
      <transactionTime/>
      <itemIdentifier timeDimension="bitemporal">
        <field path="."/>
      </itemIdentifier>
    </item>
    <item target="/company/product/priceinDollars">
      <validTime kind="state" content="varying"/>
      <itemIdentifier timeDimension="validTime">
        <field path="."/>
      </itemIdentifier>
    </item>
    <item target="/company/product/priceinPounds">
      <validTime kind="state" content="varying"/>
      <itemIdentifier timeDimension="validTime">
        <field path="."/>
      </itemIdentifier>
    </item>
    <item target="/company/product/priceinEuros">
      <validTime kind="state" content="varying"/>
      <itemIdentifier timeDimension="validTime">
        <field path="."/>
      </itemIdentifier>
    </item>
  </logical>
<annotationSet>
```

Listing 78: Temporal document.

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

  <sliceSequence>
    <slice location="slice1.xml" begin="2004-03-29" end="2002-03-30" />
    <slice location="slice2.xml" begin="2004-03-30" end="2002-03-31" />
    <slice location="slice3.xml" begin="2004-03-31"/>
  </sliceSequence>

</temporalRoot>
```

Listing 79: Squashed document.

```
<?xml version="1.0" encoding="UTF-8"?>
<tv_root
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema"
  xsi:noNamespaceSchemaLocation="repCompany.xsd">

  <rep:company_RepItem>
    <!-- Version 1 of 1 for company -->
    <rep:company_Version begin="2004-03-29">
      <rep:company>
        <rep:companyData>
          <rep:companyName>IBM</rep:companyName>
          <rep:cURL>http://www.ibm.com</rep:cURL>
        </rep:companyData>

        <rep:supplier_RepItem>
```

```
      <!-- Version 1 of 1 for supplier 1 (Seagate) -->
      <rep:supplier_Version begin="2004-03-29">
        <rep:supplier supplierNo="1" supplierName="Seagate">
          <rep:sURL>http://seagate.com</rep:sURL>
          <rep:sRating>AAA</rep:sRating>

          <rep:order_RepItem>
            <!-- Version 1 of 1 for order 1 -->
            <rep:order_Version begin="2004-03-29">
              <rep:order orderType="normal">
                <rep:orderNo>1</rep:orderNo>
                <rep:oProductNo>2</rep:oProductNo>
                <rep:oQty>50</rep:oQty>
              </rep:order>
            </rep:order_Version>
          </rep:order_RepItem>

          <rep:order_RepItem>
            <!-- Version 1 of 1 for order 2 -->
            <rep:order_Version begin="2004-03-30">
              <rep:order orderType="rush">
                <rep:orderNo>2</rep:orderNo>
                <rep:oProductNo>1</rep:oProductNo>
                <rep:oQty>100</rep:oQty>
              </rep:order>
            </rep:order_Version>
          </rep:order_RepItem>

          <rep:order_RepItem>
            <!-- Version 1 of 1 for order 3 -->
            <rep:order_Version begin="2004-03-31">
              <rep:order orderType="normal">
                <rep:orderNo>3</rep:orderNo>
                <rep:oProductNo>2</rep:oProductNo>
                <rep:oQty>25</rep:oQty>
              </rep:order>
            </rep:order_Version>
          </rep:order_RepItem>

        </rep:supplier_Version>
      </rep:supplier_RepItem>


    <rep:supplier_RepItem>
      <!-- Version 1 of 2 for supplier 2 (Wistron) -->
      <rep:supplier_Version begin="2004-03-29" end="2004-03-31">
        <rep:supplier supplierNo="2" supplierName="Wistron Corporation">
          <rep:sURL>http://www.wistron.com</rep:sURL>
          <rep:sRating>AA</rep:sRating>

          <rep:order_RepItem>
            <!-- Version 1 of 1 for order 3 -->
            <rep:order_Version begin="2004-03-30">
              <rep:order orderType="normal">
                <rep:orderNo>1</rep:orderNo>
                <rep:oProductNo>2</rep:oProductNo>
                <rep:oQty>10</rep:oQty>
              </rep:order>
            </rep:order_Version>
          </rep:order_RepItem>

      </rep:supplier_Version>

      <!-- Version 2 of 2 for supplier 2 (Wistron) -->
      <rep:supplier_Version begin="2004-03-31">
        <rep:supplier supplierNo="2" supplierName="Wistron Corporation">
          <rep:sURL>http://www.wistron.com/indexNew.html</rep:sURL>
          <rep:sRating>AA</rep:sRating>
```

```
        <rep:order_RepItem>
          <!-- Version 1 of 1 for order 3 -->
          <rep:order_Version begin="2004-03-30">
            <rep:order orderType="normal">
              <rep:orderNo>1</rep:orderNo>
              <rep:oProductNo>2</rep:oProductNo>
              <rep:oQty>10</rep:oQty>
            </rep:order>
          </rep:order_Version>
        </rep:order_RepItem>
    </rep:supplier_Version>

</rep:supplier_RepItem>

<rep:supplier_RepItem>

  <!-- Version 1 of 1 for supplier 3 (small supplier) -->
  <rep:supplier_Version begin="2004-03-29">
    <rep:supplier supplierNo="3" supplierName="small_supplier_1"/>
  </rep:supplier_Version>

</rep:supplier_RepItem>


<rep:product_RepItem>

  <!-- Version 1 of 3 for product 1 (7200rpm hard disk) -->
  <rep:product_Version begin="2004-03-29" end="2004-03-30">
    <rep:product productNo="1" productName="hard disk 73 GB 7200rpm">
      <rep:priceinDollars>100</priceinDollars>
      <rep:qtyOnHand>100</qtyOnHand>
    </rep:product>
  </rep:product_Version>

  <!-- Version 2 of 3 for product 1 (7200rpm hard disk) -->
  <rep:product_Version begin="2004-03-30" end="2004-03-31">
    <rep:product productNo="1" productName="hard disk 73 GB 7200rpm">
      <rep:priceinDollars>100</priceinDollars>
      <rep:qtyOnHand>40</qtyOnHand>
    </rep:product>
  </rep:product_Version>

  <!-- Version 3 of 3 for product 1 (7200rpm hard disk) -->
  <rep:product_Version begin="2004-03-31">
    <rep:product productNo="1" productName="hard disk 73 GB 7200rpm">
      <rep:priceinDollars>105</priceinDollars>
      <rep:qtyOnHand>20</qtyOnHand>
    </rep:product>
  </rep:product_Version>

</rep:product_RepItem>


<rep:product_RepItem>

  <!-- Version 1 of 3 for product 2 (SCSI 10000rpm hard disk) -->
  <rep:product_Version begin="2004-03-29" end="2004-03-30">
    <rep:product productNo="2" productName="SCSI hard disk 147 GB 10000rpm">
      <rep:priceinDollars>150</rep:priceinDollars>
      <rep:qtyOnHand>100</rep:qtyOnHand>
    </rep:product>
  </rep:product_Version>

  <!-- Version 2 of 3 for product 2 (SCSI 10000rpm hard disk) -->
  <rep:product_Version begin="2004-03-30" end="2004-03-31">
    <rep:product productNo="2" productName="SCSI hard disk 147 GB 10000rpm">
      <rep:priceinDollars>125</rep:priceinDollars>
```

```
              <rep:qtyOnHand>80</rep:qtyOnHand>
            </rep:product>
          </rep:product_Version>

          <!-- Version 3 of 3 for product 2 (SCSI 10000rpm hard disk) -->
          <rep:product_Version begin="2004-03-31">
            <rep:product productNo="2" productName="SCSI hard disk 147 GB 10000rpm">
              <rep:priceinDollars>125</rep:priceinDollars>
              <rep:qtyOnHand>70</rep:qtyOnHand>
            </rep:product>
          </rep:product_Version>

        </rep:product_RepItem>

    </rep:company_Version>
  </rep:company_RepItem>

</tv_root>
```

Part II
Supporting Schema Versioning of XML Documents

The previous part concerned *data* versioning; this part concerns *schema* versioning, and follows a similar structure: motivation (including an extension of the Company example), review of related work, further design decisions, architecture, theoretical framework, implementation details, and the full WinOlympic example.

11 Introduction

Schema designers often edit their schemas, refining and adding element and attribute types. As an example, in `2003-01-01`, the designers of Winter Olympic schema realize that they also need the name of the sport in which the athlete has won the medal. And they decide to add that as a "required" attribute of the `<medal>` element. As new release of this schema is developed, all XML documents that were instances of its earlier version will be rendered invalid, with the maintainers responsible for updating their XML documents.

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.

Schema versioning should offer a solution to the above problem by enabling intelligent handling of any temporal mismatch between data and its schemas. A framework is needed that would retain past data and past schemas, while allowing the current data and schema to be extracted.

This work has many real-world applications. As an example, the Botanic Garden and Botanical Museum in Berlin-Dahlem (BGBM[5]) maintains a repository of XML Schemas[6] related to index terms, keywords, biodiversity data about specimens and observations, meta-level data about collections, organizations, and networks, and various wrapper and configuration files. Most of these XML schemas have had multiple versions over the last two to three years. The BioCASE Collection Profile is up to version 1.24; the Access to Biological Collection Data is up to version 2.06.

As another example, the *Pharmacogenetics Knowledge Base* (PharmGKB[7]) "contains genomic, phenotype and clinical information collected from ongoing pharmacogenetic studies." Its schema is up to version 4.0; its evolution is documented.[8] The PHARMGKB XML schema was designed conventionally, not utilizing an architecture that supports schema versioning. As new releases of this schema were developed (for example, on May 26, 2004 Version 4.0, the latest version, was released), all XML documents that were instances of this schema were rendered invalid, with the maintainers responsible for updating their XML documents. The architecture proposed in this report retains past data and past schemas, while always allowing the current data and schema to be extracted, for tools that are not schema-versioning aware.

[5] http://www.bgbm.org

[6] http://www.bgbm.org/biodivinf/schema/default.asp

[7] http://www.pharmgkb.org/

[8] http://www.pharmgkb.org/schema/history.html

12 Motivation

We now extend our design in Section 4.3 to include schema versioning. Here we must focus on the intricacies of changing namespaces from slice to slice, versioning of subschemas that are included or imported into the main schema, and versioning multiple main schemas. We briefly cover each in turn.

12.1 Company Example Extended

We now extend our example presented in Figure 2 in Section 4.5 to include schema versioning. We use the same conventions and naming schemes as before. Figure 45 depicts the scenario. Here, in addition to the Company Data document varying over time, the Company Schema, Person Schema, Product Schema, and Company Annotation documents will also vary over time. Note that those documents are depicted here as multiple slices.

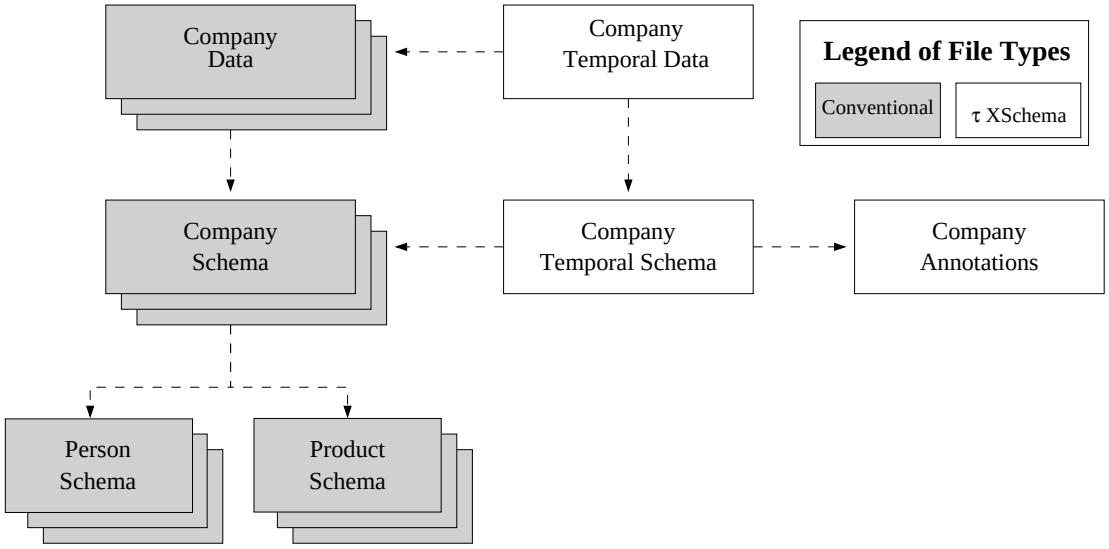


Figure 45: An overview of the end-state of the Company example.

12.2 Changing Schemas

On 2008-05-22, the user decides to make a change to the conventional schema. He changes the `<Name>` element to `<FirstName>`, resulting in a new version of the schema (Listing 80, line 19).

The user must therefore update the conventional document to conform to the new schema (see Listing 81 line 5).

The user now creates an explicit temporal schema to represent the changes to the conventional schema. He chooses to use method (a) as described in design decision **(31)** in Section 14 (see Listing 82). Note that since the user has yet to specify any annotations, the defaults are still in effect.

The user must also update the temporal document to include the new slice of the conventional document (Listing 83, line 11).

Listing 80: `Company.B.xsd`

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

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

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

</xsd:schema>
```

Listing 81: `data.B.1.xml`

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

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

</Company>
```

Listing 82: `temporalSchema.0.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice location="Company.A.xsd" begin="2008-01-01" />
      <ts:slice location="Company.B.xsd" begin="2008-05-22" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

</ts:temporalSchema>
```

Listing 83: `temporalDocument.1.1.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD"
    temporalSchemaLocation="./temporalSchema.0.xml"/>

  <td:sliceSequence>
    <td:slice location="data.A.0.xml" begin="2008-01-01" />
    <td:slice location="data.A.1.xml" begin="2008-03-17" />
    <td:slice location="data.B.1.xml" begin="2008-05-22" />
  </td:sliceSequence>

</td:temporalRoot>
```

12.2.1 Introducing Subschemas

On 2008-07-11, the user decides to split the schema into several smaller subschemas while also adding a new element. The *main* schema (see Listing 84) no longer defines any elements itself, but instead uses the XML Schema `<import>` (line 9) and `<include>` (line 10) elements to reference two subschemas (see Listings 85 and 86).

Listing 84: `Company.C.xsd`

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

  <xsd:import namespace="http://www.person.org" schemaLocation="./Person.C.0.xsd" />

  <xsd:include schemaLocation="./Product.C.0.xsd" />

  <xsd:element name="Company">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" type="per:PersonType" maxOccurs="unbounded"/>
        <xsd:element name="Product" type="ProductType" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

Listing 85: `Person.C.0.xsd`

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

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

</xsd:schema>
```

Listing 86: `Product.C.0.xsd`

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

  <xsd:complexType name="ProductType">
    <xsd:sequence>
      <xsd:element name="Type" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

The user updates the instance document to conform to the new schema paradigm and adds a `<Product>` element (see Listing 87, lines 10–12).

Listing 87: `data.C.2.xml`

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

  <Person ID="1">

    <per:FirstName>Steve</per:FirstName>
    <per:SSN>111-22-3333</per:SSN>
  </Person>

  <Product>
     <Type>Widget</Type>
  </Product>


</Company>
```

Note that in this example, the `Company` schema is taking the so-called "heterogeneous" and "homogeneous" namespace approaches for the `Person` and `Product` subschemas, respectively [23]. That is, the Person subschema's namespace is exposed in the `Company` schema, while the `Product` subschema defines the same target namespace as the `Company` schema.

The user then changes the temporal schema (Listing 88, line 8) and the temporal document (Listing 89, lines 4 and 11) to reflect these modifications. Note that the temporal schema only references the `Company` schema, because that schema directly imports `Person` and includes `Product`.

Listing 88: `temporalSchema.1.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice location="Company.A.xsd" begin="2008-01-01" />
      <ts:slice location="Company.B.xsd" begin="2008-05-22" />
      <ts:slice location="Company.C.xsd" begin="2008-07-11" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

</ts:temporalSchema>
```

Listing 89: `temporalDocument.1.2.xml`

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

```
    temporalSchemaLocation="./temporalSchema.1.xml"/>

  <td:sliceSequence>
    <td:slice location="data.A.0.xml" begin="2008-01-01" />
    <td:slice location="data.A.1.xml" begin="2008-03-17" />
    <td:slice location="data.B.1.xml" begin="2008-05-22" />
    <td:slice location="data.C.2.xml" begin="2008-07-11" />
  </td:sliceSequence>

</td:temporalRoot>
```

12.2.2 Adding Logical Annotations

On 2008-08-04, the user decides to construct an annotation document. Here, the user creates a logical annotation (via the `<item>` element) that specifies that only the `<FirstName>` element can change (see Listing 90).

Listing 90: `annotations.0.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<as:annotationSet xmlns:as="http://www.cs.arizona.edu/tau/tauXSchema/AS">

  <as:logical>
    <as:item target="Company/Person/FirstName">
      <as:transactionTime existence="constant"/>
      <as:itemIdentifier name="personID" timeDimension="transactionTime">
        <as:field path="./text()"/>
      </as:itemIdentifier>
    </as:item>
  </as:logical>

</as:annotationSet>
```

The user must then update the temporal schema to include the annotation document (see Listing 91, lines 12–16) and the temporal document to point to the new version of the temporal schema (see Listing 92, line 4). τXMLLINT will now check this constraint between conventional document slices over time.

Listing 91: `temporalSchema.2.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice location="Company.A.xsd" begin="2008-01-01" />
      <ts:slice location="Company.B.xsd" begin="2008-05-22" />
      <ts:slice location="Company.C.xsd" begin="2008-07-11" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

  <ts:annotationSet>
    <ts:sliceSequence>
      <ts:slice location="annotations.0.xml" begin="2008-08-04" />
    </ts:sliceSequence>
  </ts:annotationSet>

</ts:temporalSchema>
```

Listing 92: `temporalDocument.2.3.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD"
    temporalSchemaLocation="./temporalSchema.2.xml"/>
```

```
4
5   <td:sliceSequence>
6     <td:slice location="data.A.0.xml" begin="2008-01-01" />
7     <td:slice location="data.A.1.xml" begin="2008-03-17" />
8     <td:slice location="data.B.1.xml" begin="2008-05-22" />
9     <td:slice location="data.C.2.xml" begin="2008-07-11" />
10  </td:sliceSequence>
11
12 </td:temporalRoot>
```

12.2.3 Temporal Subschemas

On 2008-09-10, the user changes the `Person` subschema (see Listing 93, line 11) to include a `<LastName>` element.

Listing 93: `Person.D.1.xsd`

```
1  <?xml version="1.0"?>
2  <xsd:schema
3    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4    targetNamespace="http://www.person.org"
5    xmlns="http://www.person.org"
6    elementFormDefault="qualified">
7
8    <xsd:complexType name="PersonType">
9      <xsd:sequence>
10       <xsd:element name="FirstName" type="xsd:string"/>
11       <xsd:element name="LastName" type="xsd:string"/>
12       <xsd:element name="SSN" type="xsd:string"/>
13     </xsd:sequence>
14     <xsd:attribute name="ID" type="xsd:string"/>
15   </xsd:complexType>
16
17 </xsd:schema>
```

The user must update the `Company` schema (Listing 94, line 9) to reference the new version of the subschema.

Listing 94: `Company.D.xsd`

```
1  <?xml version="1.0"?>
2  <xsd:schema
3    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4    targetNamespace="http://www.company.org"
5    xmlns="http://www.company.org"
6    elementFormDefault="qualified"
7    xmlns:per="http://www.person.org" >
8
9    <xsd:import namespace="http://www.person.org" schemaLocation="./Person.D.1.xsd" />
10
11   <xsd:include schemaLocation="./Product.C.0.xsd" />
12
13   <xsd:element name="Company">
14     <xsd:complexType>
15       <xsd:sequence>
16         <xsd:element name="Person" type="per:PersonType" maxOccurs="unbounded"/>
17         <xsd:element name="Product" type="ProductType" maxOccurs="unbounded"/>
18       </xsd:sequence>
19     </xsd:complexType>
20   </xsd:element>
21
22 </xsd:schema>
```

The user must also update the conventional document (Listing 95, line 7) to include a `<LastName>` element.

Listing 95: `data.D.3.xml`

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

  <Person ID="1">

    <per:FirstName>Steve</per:FirstName>
    <per:LastName>Thomas</per:LastName>
    <per:SSN>111-22-3333</per:SSN>
  </Person>

  <Product>
     <Type>Widget</Type>
  </Product>


</Company>
```

The user then changes the temporal schema (Listing 96, line 9) and temporal document (Listing 97, lines 4 and 12) to reflect these modifications.

Listing 96: `temporalSchema.3.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice location="Company.A.xsd" begin="2008-01-01" />
      <ts:slice location="Company.B.xsd" begin="2008-05-22" />
      <ts:slice location="Company.C.xsd" begin="2008-07-11" />
      <ts:slice location="Company.D.xsd" begin="2008-09-10" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

  <ts:annotationSet>
    <ts:sliceSequence>
      <ts:slice location="annotations.0.xml" begin="2008-08-04" />
    </ts:sliceSequence>
  </ts:annotationSet>


</ts:temporalSchema>
```

Listing 97: `temporalDocument.3.3.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD"
    temporalSchemaLocation="./temporalSchema.3.xml"/>

  <td:sliceSequence>
    <td:slice location="data.A.0.xml" begin="2008-01-01" />
    <td:slice location="data.A.1.xml" begin="2008-03-17" />
    <td:slice location="data.B.1.xml" begin="2008-05-22" />
    <td:slice location="data.C.2.xml" begin="2008-07-11" />
    <td:slice location="data.D.3.xml" begin="2008-09-10" />
  </td:sliceSequence>

</td:temporalRoot>
```

12.2.4 Namespace Changes

One month later (2008-11-13), the user changes the target namespace of the main schema to `"steves-company.org"` (see Listing 98, lines 4 and 5). Since the `Product` subschema uses the homogeneous

namespace paradigm, it also must be updated to the new namespace (see Listing 99, lines 4 and 5). Of course, the user must also update the conventional document (see Listing 100, line 2).

Listing 98: `Company.E.xsd`

```
1  <?xml version="1.0"?>
2  <xsd:schema
3    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4    targetNamespace="http://www.stevescompany.org"
5    xmlns="http://www.stevescompany.org"
6    elementFormDefault="qualified"
7    xmlns:per="http://www.person.org" >
8
9    <xsd:import namespace="http://www.person.org" schemaLocation="./Person.D.1.xsd" />
10
11   <xsd:include schemaLocation="./Product.E.1.xsd" />
12
13   <xsd:element name="Company">
14     <xsd:complexType>
15       <xsd:sequence>
16         <xsd:element name="Person" type="per:PersonType" maxOccurs="unbounded"/>
17         <xsd:element name="Product" type="ProductType" maxOccurs="unbounded"/>
18       </xsd:sequence>
19     </xsd:complexType>
20   </xsd:element>
21
22 </xsd:schema>
```

Listing 99: `Product.E.1.xsd`

```
1  <?xml version="1.0"?>
2  <xsd:schema
3    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4    targetNamespace="http://www.stevescompany.org"
5    xmlns="http://www.stevescompany.org"
6    elementFormDefault="qualified">
7
8    <xsd:complexType name="ProductType">
9      <xsd:sequence>
10       <xsd:element name="Type" type="xsd:string" minOccurs="1" maxOccurs="1"/>
11     </xsd:sequence>
12   </xsd:complexType>
13
14 </xsd:schema>
```

Listing 100: `data.E.3.xml`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Company xmlns="http://www.stevescompany.org" xmlns:per="http://www.person.org">
3
4    <Person ID="1">
5
6      <per:FirstName>Steve</per:FirstName>
7      <per:LastName>Thomas</per:LastName>
8      <per:SSN>111-22-3333</per:SSN>
9    </Person>
10
11   <Product>
12      <Type>Widget</Type>
13   </Product>
14
15
16 </Company>
```

The user then changes the temporal schema (see Listing 101, line 10) and temporal document (see Listing 102, lines 4 and 13) to reflect his modifications.

Listing 101: `temporalSchema.4.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice location="Company.A.xsd" begin="2008-01-01" />
      <ts:slice location="Company.B.xsd" begin="2008-05-22" />
      <ts:slice location="Company.C.xsd" begin="2008-07-11" />
      <ts:slice location="Company.D.xsd" begin="2008-09-10" />
      <ts:slice location="Company.E.xsd" begin="2008-11-13" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

  <ts:annotationSet>
    <ts:sliceSequence>
      <ts:slice location="annotations.0.xml" begin="2008-08-04" />
    </ts:sliceSequence>
  </ts:annotationSet>

</ts:temporalSchema>
```

Listing 102: `temporalDocument.4.3.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD"
    temporalSchemaLocation="./temporalSchema.4.xml"/>

  <td:sliceSequence>
    <td:slice location="data.A.0.xml" begin="2008-01-01" />
    <td:slice location="data.A.1.xml" begin="2008-03-17" />
    <td:slice location="data.B.1.xml" begin="2008-05-22" />
    <td:slice location="data.C.2.xml" begin="2008-07-11" />
    <td:slice location="data.D.3.xml" begin="2008-09-10" />
    <td:slice location="data.E.3.xml" begin="2008-11-13" />
  </td:sliceSequence>

</td:temporalRoot>
```

12.2.5 Multiple Conventional Schemas

The user now (2008-11-27) wants to create a level of independence between each of the conventional schemas: when a subschema changes, he does not want to have to change the main schema. To do this, he creates a temporal schema for each of the conventional schemas and in the main conventional schema he references the temporal schema (as opposed to the conventional subschema).

Listings 103 and 104 show the new temporal schemas for the Product and Person subschemas, respectively. Listing 105, lines 9 and 12, shows the main conventional schema referencing the new temporal schemas while Listing 106 shows the new temporal schema. Finally, Listing 102 shows the temporal document for this new configuration.

Listing 103: `ProductTemporalSchema.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice filename="Product.C.0.xsd" begin="2008-07-11" />
      <ts:slice filename="Product.E.1.xsd" begin="2008-11-13" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

</ts:temporalSchema>
```

Listing 104: `PersonTemporalSchema.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice filename="Person.C.0.xsd" begin="2008-07-11" />
      <ts:slice filename="Person.D.1.xsd" begin="2008-09-10" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

</ts:temporalSchema>
```

Listing 105: `Company.F.xsd`

```
1  <?xml version="1.0"?>
2  <xsd:schema
3    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4    targetNamespace="http://www.stevescompany.org"
5    xmlns="http://www.stevescompany.org"
6    elementFormDefault="qualified"
7    xmlns:per="http://www.person.org" >
8
9    <xsd:import namespace="http://www.person.org"
10               schemaLocation="./PersonTemporalSchema.xml" />
11
12   <xsd:include schemaLocation="./ProductTemporalSchema.xml" />
13
14   <xsd:element name="Company">
15     <xsd:complexType>
16       <xsd:sequence>
17         <xsd:element name="Person" type="per:PersonType" maxOccurs="unbounded"/>
18         <xsd:element name="Product" type="ProductType" maxOccurs="unbounded"/>
19       </xsd:sequence>
20     </xsd:complexType>
21   </xsd:element>
22
23 </xsd:schema>
```

Listing 106: `temporalSchema.5.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice location="Company.A.xsd" begin="2008-01-01" />
      <ts:slice location="Company.B.xsd" begin="2008-05-22" />
      <ts:slice location="Company.C.xsd" begin="2008-07-11" />
      <ts:slice location="Company.D.xsd" begin="2008-09-10" />
      <ts:slice location="Company.E.xsd" begin="2008-11-13" />
      <ts:slice location="Company.F.xsd" begin="2008-11-27" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

  <ts:annotationSet>
    <ts:sliceSequence>
      <ts:slice location="annotations.0.xml" begin="2008-08-04" />
    </ts:sliceSequence>
  </ts:annotationSet>

</ts:temporalSchema>
```

Listing 107: `temporalDocument.5.3.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD"
    temporalSchemaLocation="./temporalSchema.5.xml"/>

  <td:sliceSequence>
    <td:slice location="data.A.0.xml" begin="2008-01-01" />
    <td:slice location="data.A.1.xml" begin="2008-03-17" />
    <td:slice location="data.B.1.xml" begin="2008-05-22" />
    <td:slice location="data.C.2.xml" begin="2008-07-11" />
    <td:slice location="data.D.3.xml" begin="2008-09-10" />
    <td:slice location="data.E.3.xml" begin="2008-11-13" />
  </td:sliceSequence>

</td:temporalRoot>
```

13 Review of Related Work

In this section, we review prior related work in the area of schema versioning. Version and source control for schemas and schema objects is needed, especially in complex, multi-enterprise development environments. The XML Schema working group at W3C has discussed desirable behaviors for use cases that involve schema versioning in XML [16]. Various techniques to support evolution of XML schemas, where they allow for extensibility in the original design have also been proposed [34]. The emphasis of the paper is to avoid changes to the existing applications, like [24] in temporal databases, by anticipating changes to the schemas and then designing them for evolution. This is typically achieved through a careful use of wildcards, allowing extensions through namespaces, allowing applications to ignore unknown objects, and forcing applications to understand unknown objects when no other option is available. This approach does not address the whole problem, as many schema changes cannot be expressed in their limited notations.

Schema versioning has been previously researched in the context of temporal databases [68, 24]. But an XML schema is a grammar specification, unlike a (relational) database schema, so new techniques are required. Although various XML schema languages have been proposed in the literature and in the commercial arena, none model schema changes nor provide versioning. We chose to base our research on XML Schema because it is backed by the W3C and is the most widely-used schema language.

Brahmia et al. propose a six-component taxonomy of schema change operations for use in supporting schema versioning across both valid and transaction time with XMLSchema [10].

Raghavachari and Shmueli consider a problem different from that considered in the present paper: can a nontemporal XML document D that is known to be valid according to nontemporal XML schema S be efficiently validated against a different schema S' [66]. However, their problem and proposed solutions are relevant to the validation for a temporal document against a temporal schema as considered in the present paper. As the schema evolves over time, the data is required to also evolve so that he data timestamped with a transaction time at the new time is consistent with the schema timestamped with that transaction time. It is possible for the tool constructing that temporal document, or for the SQUASH tool as it considers a schema change, to efficiently revalidate the data document currently in force against the new schema.

14 Design Decisions

In this section we outline the design decisions relating to temporal schemas. We use the same terminology and consider the same goals as presented in Section 4.

(31) A temporal schema will have the root element `<temporalSchema>` which belongs to the τXSchema namespace. The root element will have two subelements, only the first of which is required.

- `<conventionalSchema>`. In this element, the user will specify the conventional schema(s) that belongs to the system.
- `<annotationSet>`. In this element, the user will specify the annotation(s) that belongs to the system, if any.

Within each of the two elements there will be four separate ways to specify schemas and annotations.

(a) Listing the URI of each conventional document with a `<sliceSequence>` element (see Listing 8)

(b) Including a (conventional or temporal) document with a `<include>` element (see Listing 6 for an example and decision **(34)** for more information)

(c) Placing the text of a document directly in the element (see Listing 9)

(d) Omitting the element altogether. In this case, default behavior will be assumed (see, e.g., design decision **(4)**). Default behavior would only apply to annotation documents, as a default conventional schema (by perhaps automatically detecting and creating a schema based on the first conventional document) would be dangerous and might provide unintended semantics.

Providing these different mechanisms allows for substitutability (satisfying design goal **(f)** in Section 4.3), convenience (satisfying goal **(a)**), and simplicity (satisfying goal **(d)**).

(32) An `<include>` element will be used to include a document into the temporal schema; it has the same semantics of placing the entire actual text of the document into the schema. `<include>` can reference any kind of document, including a conventional schema, a temporal schema, and annotation documents. This element has the effect of removing the root of the included document; see Listings 108 and 109 for an example.

Listing 108: A schema using `<include>`.

```
<temporalSchema>
 ...
 <annotationSet>
  <include schemaLocation="anno.xml"/>
 </annotationSet>
 ...
</temporalSchema>
```

Listing 109: `anno.xml`

```
<annotationSet>
 ...
 <logical>
  ...
 </logical>
  ...
</annotationSet>
```

This decision satisfies goal **(e)** by allowing both the `<include>` element and the actual text of the document to have the exact same semantics and goal **(c)** by keeping consistent syntax as XML Schema `<include>` elements.

(33) Both conventional and temporal schemas can `<include>` any number of conventional and temporal schemas. See Listing 10 for an example. This decision satisfies goal **(f)** by permitting temporal data to occur at any level in the system.

(34) There can be one temporal schema for each independent conventional schema present in the system. In this way, each conventional schema can vary over time independently, as well as have their own logical and physical annotations. See Figure 46 for an example, and Section 4.5 for more information.

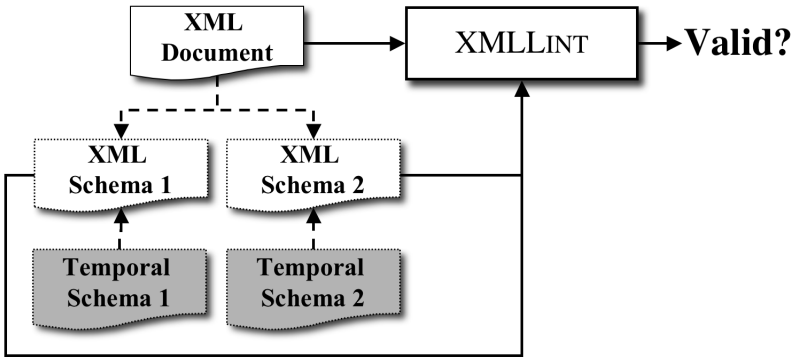


Figure 46: Each conventional schema has a separate corresponding temporal schema.

15 Approach

There are several key ideas to our solution. First, a *temporal schema* serves the analogous purpose of an XML Schema document for a static document. So we have a single point of reference for the schema of a temporal document. Of course, the temporal schema may itself contain versions within it. That means that the temporal documents *it* references must also have associated temporal schema as *their* schemas. The temporal schema is all the user needs for describing the temporal document, just as the conventional XML Schema is all the user needs for describing an XML document.

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

The third key idea, which we call *schema-constant periods*, first appeared in a paper by one of the authors on temporal aggregation [74]. We introduced the concept in Section 18, and explain how we use it. It is possible, even with versioned schemas having themselves versioned schemas, to identify contiguous periods of time when there are no schema changes, *anywhere*. Now, during such schema-constant periods the data may be (and probably is) versioned, but at least you have a fixed base schema and fixed logical annotations, each of which has a fixed schema. And since the physical annotations are fixed, the representation is also fixed, so it is possible to read and interpret the temporal document during that schema-constant period, and even to validate that portion of the document. (This is just the situation discussed in our Part I, of a single schema and versions of the data.) So a general temporal document can be viewed as a sequence of data-varying documents, each over a single schema-constant period. Since we can validate within each schema-constant period, given the approaches elaborated on earlier, all we have to do is validate *across* schema changes.

The final key idea first appeared in the original presentation of τXSchema [25]: the representational schema (a) is derivable solely from information in the temporal schema, (b) can be designed to enable some of the temporal integrity constraints to be checked by a conventional validator, and (c) can be computed and cached within τXMLLINT, completely unbeknown to the user.

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

In the remainder of this section, we introduce the architecture through a running example, then describe how the validator can be extended to validate documents in this seemingly precarious situation of data that changes over time, while its schema and even its representation are also changing over time.

All times mentioned in this paper are from the *transaction time* dimension [73], though τXSchema also supports *valid time* for data versioning. While schema versioning has been considered in the context of valid time [17], doing so is quite complex and in our opinion not worth this complexity. Thus in τXSchema schemas vary and are versioned only over transaction time.

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

15.1 Supporting Versioned Schemas

For convenience, we review some architectural diagrams from Section 8. Figure 47 illustrates the architecture of τXSchema. We now generalize the architecture to also support versioned schemas. As noted previously, the PHARMGKB schema has undergone a series of changes. (Our emphasis in this chapter is on

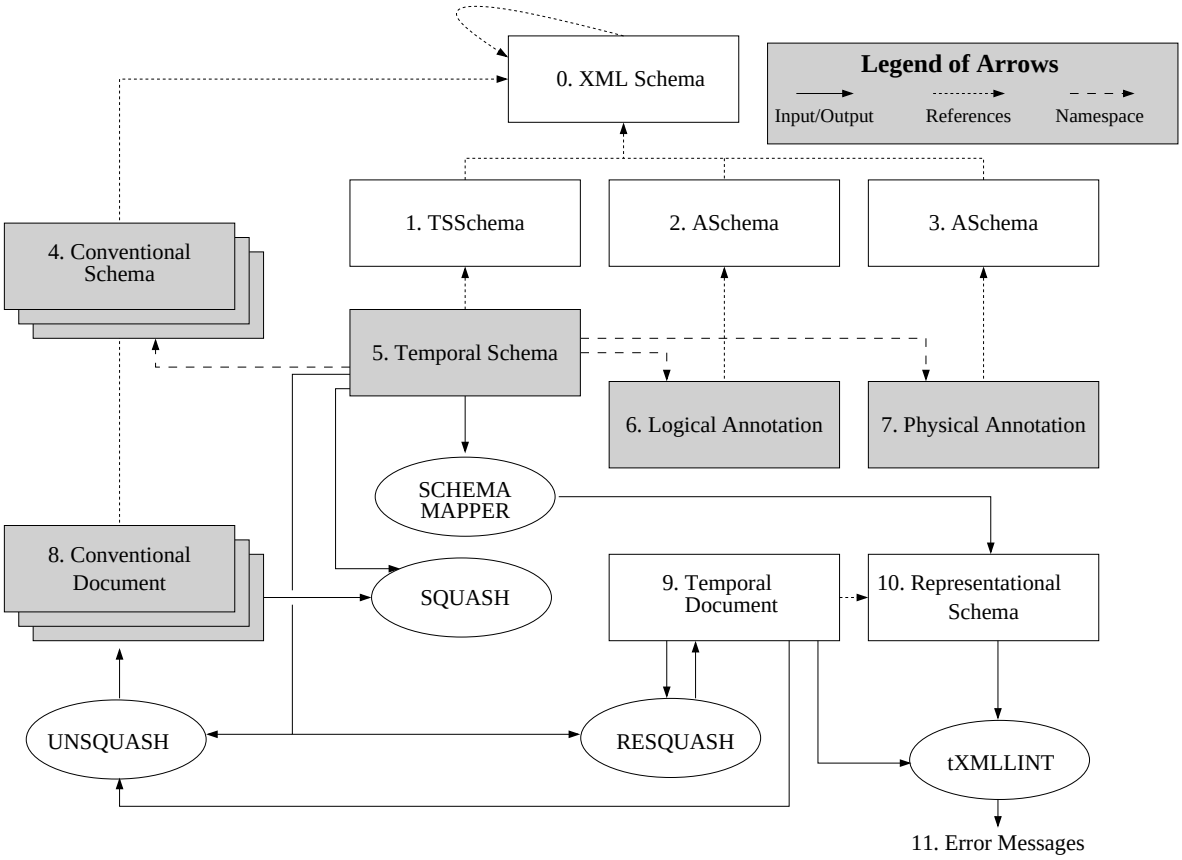


Figure 47: Overall Architecture of τXSchema

how to validate a time-varying document against a time-varying schema. For more discussion of τXSchema architecture per se, please consult Part I, Section 8).

This implies that box 3 is actually a *sequence* of base schemas, three of which are excerpted in Listings 110–112. Not only do these base schemas change over time, but the schemas included by them (e.g., `sequence.xsd`, `experiment.xsd`) can vary over time. Similarly, the temporal annotations (box 5) and those annotations included by them and the physical annotations (box 6) and those annotations included by *them* all can vary over time, resulting in multiple versions.

Listing 110: `<ExperimentClass>` element in version 3.1

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

```
              type="NonEmptyTokenType"
              minOccurs="0" maxOccurs="1"/>
          ...
         </xsd:sequence>
        </xsd:extension>
       </xsd:complexContent>
      </xsd:complexType>
     </xsd:element>
     ...
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 111: `<ExperimentClass>` element in version 3.1

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

Listing 112: `<ExperimentClass>` element in version 4

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

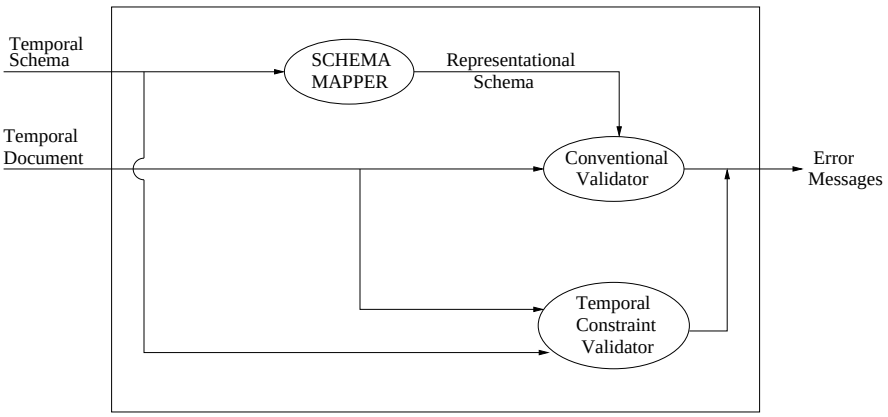


Figure 48: Validating a Document with Time-Varying Data

This versioning is handled by timestamping the `<temporalRoot>` element, and adding periods to specify when that annotation element became applicable. So our PHARMGKB schema would have many annotation elements, with version 3.1 becoming applicable on April 25, 2003, version 3.2 on May 21, 2003, and version 4.0 on May 12, 2004.

The schema annotation elements reference individual base schemas. One approach is to have a different document (file) for each version, similar to what is shown in box 7. So we might have files named `root.4.25.03.xsd`, etc., or perhaps `root.3.1.xsd`. etc. Each of these files would reference subsidiary schemas, such as `sequence.v3.1.xml.xsd` or `experiment.4.25.03.xsd`. As one can imagine, this becomes rather cumbersome. The problem with this approach is that whenever a subsidiary schema changes, a new version is produced, with its own URI, which requires the referencing schema document to be changed. So a new version of `experiment.xsd` requires a new version of `sequence.xsd`, which requires a new version of `root.xsd`.

Listing 113: A Temporal Schema for PHARMGKB: `temporalschema.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<temporalSchema
    xmlns="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema"
    xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema" >

        <conventionalSchema>
          <sliceSequence>
            <slice location="root.xsd" />
          </sliceSequence>
        </conventionalSchema>

        <annotationSet>
          <sliceSequence>
            <slice location="annotations.xml" />
          </sliceSequence>
        </annotationSet>

</temporalSchema>
```

While this approach is allowed, τXSchema also permits *temporal schemas*, in place of multiple versions of conventional schemas. Consider the sequence of root schemas: `root.1.0.xsd`, `root.2.0.xsd`, ... We write a simple temporal schema for these and invoke the SQUASH utility, which produces a single temporal document, `tv_snapshot.xml` which is then referenced by multiple schema annotation elements. Similarly, we use SQUASH to generate temporal schemas for `sequence.xsd` and `experiment.xsd`.

This rather involved state of affairs, with time-varying documents and time-varying schemas, is illustrated with a T Diagram in Figure 49. In this notation, first described over forty years ago [11], the input of a translator is given on the left arm of the "T" (for example, for SCHEMAMAPPER in the upper right-hand-side of the figure, the input is the temporal schema document, `temp_schema.xml`), the name of the translator is given at the base of the "T" (here, "Schema Mapper"), and the output of the translator is given on the right arm of the "T" (here, a representational schema, `rep.xsd`). The name of these diagrams was to the best of our knowledge given by McKeeman, Horning, and Wortman in their classic compiler book [55].

We extend these diagrams to allow multiple inputs, which unfortunately complicates them somewhat. As shown in Figure 49, SQUASH takes both a temporal schema and a sequence of conventional documents and produces a temporal document, and UNSQUASH does just the opposite (this is illustrated for the logical annotations, which are SQUASHed into a single `tv_anno.xml` document, then UNSQUASHed back into their constituent time slices).

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

15.2 Validating Against a Time-Varying Schema

To validate a time-varying document associated with a time-varying schema, τXMLLINT applies the conventional validator to the document, using the representational schema produced by SCHEMAMAPPER (see Figure 50). It then determines the times when the schema changes, thus determining the periods when the schema is constant, termed the *schema-constant periods*. These periods will be non-overlapping and continuous; between the periods are schema change *walls*. For each such period, the time-varying data checker is invoked to check the temporal integrity constraints over the time-varying data, with the *single* base schema, logical annotation, and physical annotation.

During this process, τXMLLINT treats each URI it encounters as the specification of a temporal *timeslice* operation to select the appropriate version. The timeslice is as of the time of the document or context that contains the URI. For example, consider the excerpt in Listing 114. `root.xsd` is a time-varying document, containing several schema versions. In this context, τXMLLINT will utilize the temporal context of "May 21, 2003" to extract a *single* version of the `root` schema. To do so, it calls UNSQUASH, passing it (a) the temporal schema, (b) the temporal document, and (c) a timestamp. It passes the same information for all the schemas included by that schema, such as `sequence` and `ExperimentClass`. The underlying semantics ensures that at any point in time, there is a single base schema, a single logical annotation, and a single physical annotation.

Listing 114: An excerpt from the time-varying Temporal Schema for PHARMGKB

```
<conventionalSchema>
    <include schemaLocation="root.xsd"/>
</conventionalSchema>

<annotationSet>
   <include schemaLocation="anno.xml"/>
</annotationSet>
```

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

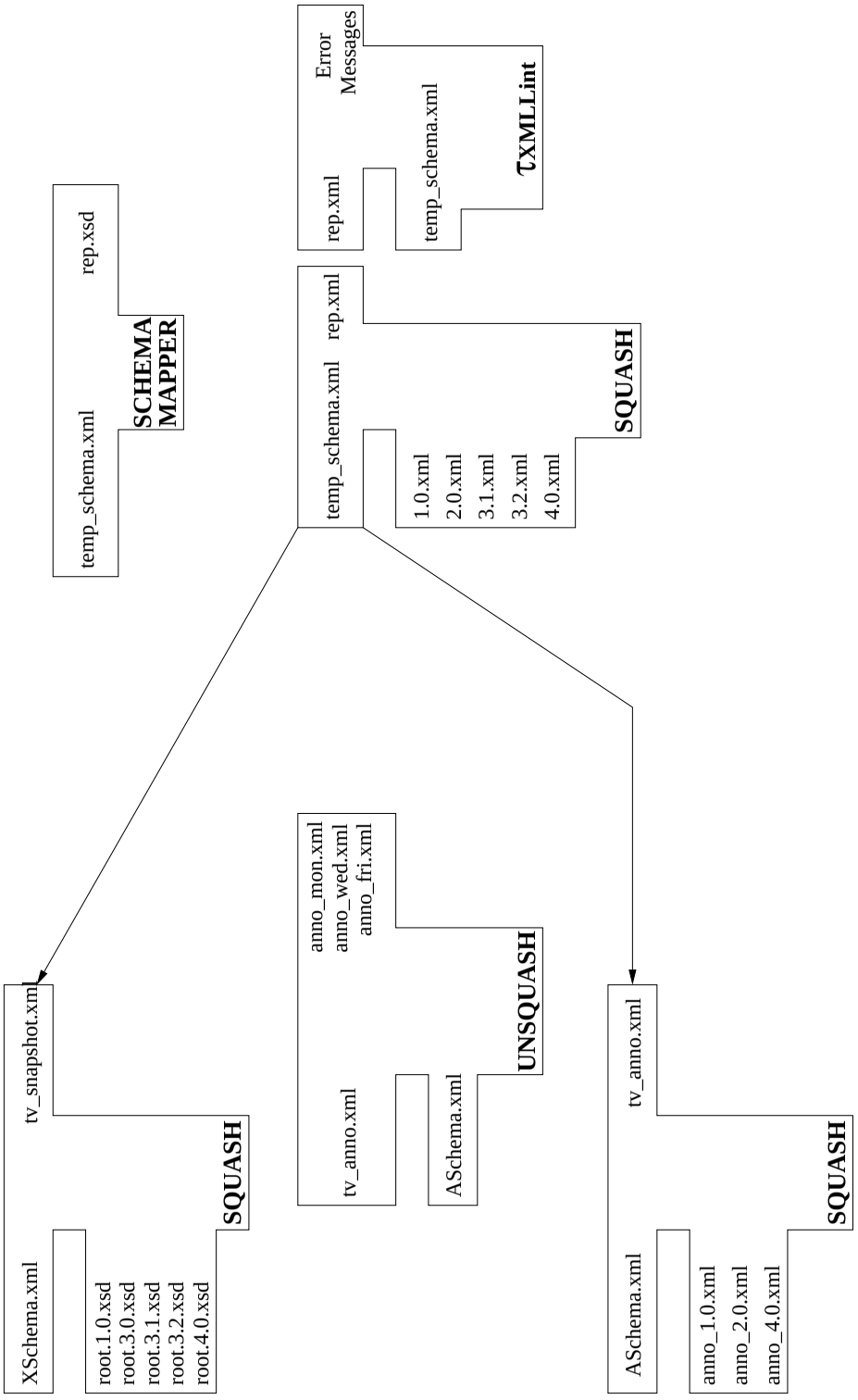


Figure 49: T Diagram of Validation

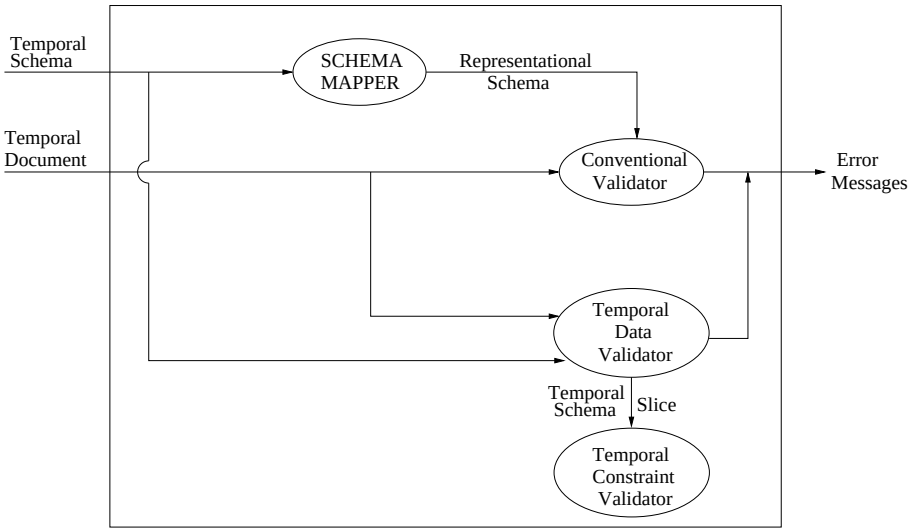


Figure 50: Validating a Document with a Time-Varying Schema

Let's examine how τXMLLINT depicted in Figure 50 could handle the versioned schema for PHARMGKB. Recall that prior to Version 3.2, the `<ExperimentClass>` element of PHARMGKB contained nested `<sampleSet>` elements (Listing 110). In Version 3.2, this was replaced with a `<sampleSetXref>` element (Listing 111), that just mentioned the unique identifier of the sample set, which was moved to the top of the document, with a `pharmgkbId` attribute.

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

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

What would the representational schema look like for this temporal document? We could see that schema directly by running SCHEMAMAPPER on our temporal schema. A portion of the temporal document is shown in Listing 115. Note that every change of the base schema (which is what occurred here) or in the physical annotation results in a new `<schemaVersion`*i*`>` element and `rep`*i* namespace within the time-varying root (with these names being generated by SCHEMAMAPPER). The conventional validator can thus check to ensure that prior to the schema change on May 25, `<ExperimentClass>` elements contained an `<sampleSet>` element, and afterward, an `<sampleSetXref>` element. (SQUASH will ensure that the appropriate `<version>` is used in the generated temporal document; τXMLLINT will also check this.)

Listing 115: A portion of a temporal document (`rep.xml`)

```
<?xml version="0.1" encoding-"UTF-8"?>
<sv_root ...>

<schemaItem>
  <schemaVersion0>
  <tTime>May 1, 2004</tTime>
    <rep0:tv_root>
      <tTime>May 1, 2004</tTime>
      <pharmGKB>
           ...
        <ExperimentClass>
             ...
          <sampleSet> ... </sampleSet>
             ...
         </ExperimentClass>
       </pharmGKB>
     </rep0:tv_root>
   </schemaVersion0>
   <schemaVersion1>
     <tTime>May 29, 2004</tTime>
     <rep1:tv_root>
       <pharmGKB>
          ...
          <ExperimentClass>
             ...
            <sampleSetXref>...</sampleSetXref>
             ...
          </ExperimentClass>
          <sampleSet>
             ...
          </sampleSet>
        </pharmGKB>
     </rep1:tv_root>
  <schemaVersion1>
  ...
</schemaItem>

</sv_root>
```

Continuing with the example, in Version 4.0 an `<ExperimentClass>` can now cross-reference more than one `<sampleSet>`. (Note the `unbounded` for `maxOccurs` in Listing 112.) Additionally, a `<sampleSet>` is now a set of `<sample>` instead of a set of `<subject>`. The latter change can be checked by the conventional validator because such sub-elements would themselves be enclosed in a new `<tv_version_3>` element. The former change, however, possibly cannot be checked by the conventional validator.

In context of constraints, we made reference to sequenced and non-sequenced distinctions (in Section 6). A temporal constraint is termed as *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 [72]. Given a conventional XML Schema constraint, we can define the corresponding temporal semantics in τXSchema in terms of a sequenced constraint. In the earlier schema, with a `maxOccurs` of 1, the temporal semantics of this integrity constraint is the sequenced constraint, "*at every point in time*, there can be a maximum of one such element." However, depending on the physical annotations, it may be that the `<sampleSet>` element is itself versioned, which implies that an `<ExperimentClass>` element could have several `<sampleSet>` elements, each resident at non-overlapping periods, so that at any one time, there wouldn't be more than one. In this case, this integrity constraint would need to be checked separately by the time-varying data checker component in τXMLLINT, which knows the temporal extent of the integrity constraint (from the temporal schema), and thus could check for a maximum of one only before Version 4.0 went into effect. In some cases, the representational

schema can be designed such that many sequenced constraints can be checked directly by the conventional validator.

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

16 Theoretical Framework

There are four aspects that do not show up with time-varying data, but rather are unique to versioned schemas: (1) an evolving definition of keys, (2) accommodating gaps in lifetimes, (3) the semantics of mixed data and schema changes, and (4) checking non-sequenced constraints across schema changes. We examine each in turn in this section.

16.1 Accommodating Evolving Keys

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

When a pair of elements is glued, an *item* is created. An item is an element that evolves over time through various versions. Determining which elements should be glued depends on two factors: the *type* of the element, and the *item identifier* for that element's type. The item identifiers and gluing of elements to form items is already explained in detail in Section 5.3.

When a schema-change wall is encountered, items across the wall need to be associated. This process is called as *cross-wall gluing*, or *bridging*. Figure 51 depicts the concepts of gluing and bridging.

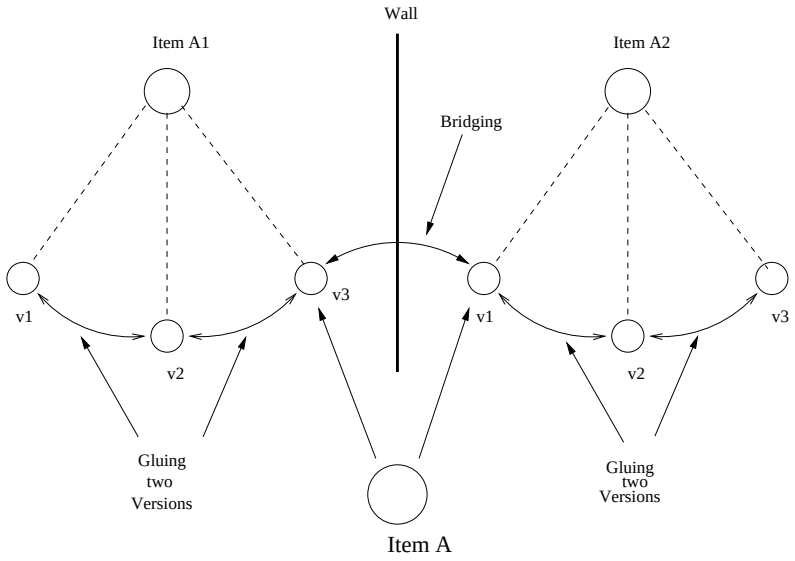


Figure 51: Gluing and Bridging

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

Gluing uses the item identifier to associate the first three elements with an item and likewise the next three elements. Bridging determines that the element that is version 3 of item A1 and the element that is version 1 of item A2 are actually versions of the same item, item A. So in fact item A has *six* versions, the three elements before the schema change and the three elements after the schema change. Gluing and bridging occur in different stages within the validator; both conspire to realize an item across schema

changes, which is the first step in checking the temporal constraints associated with that item's definition in the schema.

What is relevant for our purposes here is that item identifiers specified in the logical annotations, are usually the (conventional) key of the element type [12] given in the base schema, and are used by τXMLLINT to extract the items from the temporal document and then check the temporal constraints on those items.

What if either the conventional key (specified in the base schema) upon which an item identifier is defined, or if the item identifier itself (specified in a logical annotation) changes? This is a particularly insidious kind of quicksand. Even worse is when the underlying element type of an item changes. As an example, if the `<athlete>` element in the `winolympic.ver1.xsd` is replaced by `<player>` in the future versions, an item that was a particular `<athlete>` element before the schema change could be associated with a particular `<player>` element in the conventional document associated with the later schema.

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

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

This could be best described with an example. Say that in 2002 the item identifier is the `athID` attribute of the `<athlete>` element. In January 2005, this attribute is renamed `athNumber`; we specify a mapping type of `useBoth`. In March 2005, the item identifier is changed to the `athName` element, with a mapping type of `useNew`. (This attribute has been around since 2002, but it wasn't used as a key until January 2005.) Assume that, in June 2005 we add a new attribute, `athKey`, and specify that as the item identifier, with a mapping type of `useOld`. Finally, in July 2005, just before the beginning of the games, we replace the `<athlete>` element with a `<player>` element, with a `playerID` attribute as the item identifier and a mapping type of `replace`.

The gluing of elements into items is then done the following way. Before 2005, the `athID` is used for gluing. When the schema change occurs sometime in January 2005, we glue across the schema change by matching the `athID` value of the element before the schema change with the `athNumber` value after the change: these (integer) values must match for the two elements to be glued. In March 2005, we glue across the schema change by matching up old elements and new elements that have the same (string) value for their `athName` element, the new item identifier. The only difference is that before the schema change, that element was present but wasn't being used as a key. In a consistent fashion, in June we also glue using the `athName` element, which was the *old* item identifier.

July is the most complex. We need to glue an `<athlete>` element with an item identifier of `athKey` with a `<player>` element with an item identifier of `playerID`. For this, we use the `MappingLocation` attribute in the temporal schema to access a mapping table that provides a list of pairs, each with an `athKey` and a `playerID` value.

This list of pairs is termed a *replace mapping list*. As it is instance-based, containing as it does a list of key *values*, the replace mapping list should only be used as a last resort. Its role is to allow bridging for all

cases in which the other three mapping types, which have no need for storing instance information in the schema, are not appropriate.

Of course, the mapping location document can also be time-varying; τXMLLINT extracts the relevant timeslice with UNSQUASH.

16.2 Accommodating Gaps

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

Figure 52 shows three cases that may arise while bridging the items from consecutive schema-constant periods. It shows the data and schema changes along the transaction-time dimension, from left to right. The schema-change walls are shown as bold vertical lines. The horizontal lines depicts the evolution of a particular item (in this case, three separate items). The bridging process is shown by the jumpers over schema change walls. A dotted line indicates when the item did not exist in the database. The first item existed during the entire transaction time period depicted in this figure. There is a single gap in the existence time of the second item: it ceased to exist sometime during P_1 but reappeared in P_2. The third item had a much longer gap, reappearing only in P_3.

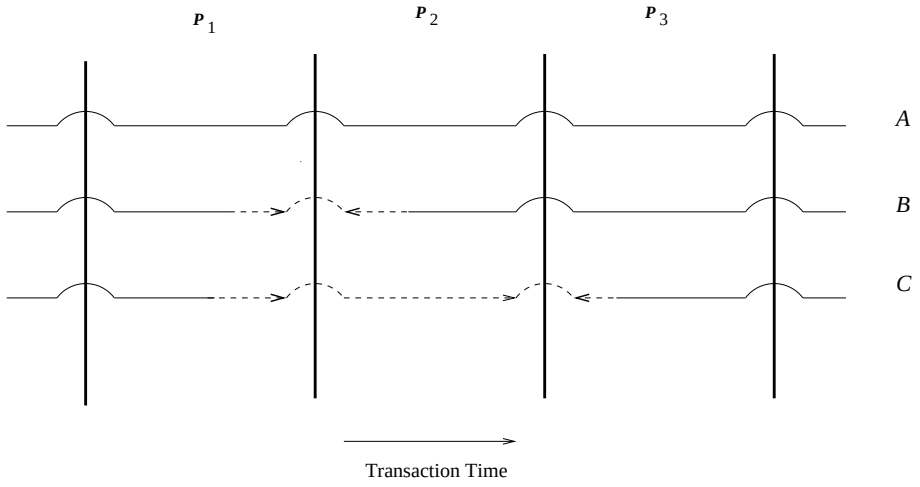


Figure 52: Cross Wall Gluing

We now now examine each item in turn.

1. The item A (the first horizontal line) is present throughout schema-constant periods P_1 and P_2. Thus the last snapshot of P_1 and the first snapshot of P_2 contains versions of item A. Here, no extra work is needed as the items can be bridged directly using one of the above four methods.

2. The item B (the second horizontal line) disappeared for some time in P_1 and reappeared about halfway through in P_2. Thus the last snapshot of P_1 and first snapshot of P_2 will not contain versions of item B. Bridging these two items in this case involves virtually extending period of item B's last version until the end of P_2 as if it were present during the last snapshot; and virtually extending its first version's period until the start of P_3; and then performing the bridging using one of the above four methods. Each virtual extension is depicted as a dashed line with an arrow indicating the direction the extension was made. In an implementation, this could be done by simply checking item's last version from P_1 and first version from P_2.

3. An item could also disappear for one or more schema-constant periods and then reappear again. Item C (the bottom horizontal line) was present for initial part of P_1. It then disappeared over entire period P_2 and again appeared in the later half of P_3. For such cases, bridging involves virtually extending the period of the item C's last version from P_1 over multiple schema-constant periods followed by bridging using one of above methods. So P_1's version is extended to the wall, then bridged to a virtual element over all of P_2, then bridged to the extended element in P_3.

Figure 53 illustrates the most complex situation of cross-gap gluing over multiple schema-constant periods. Documents in the top right part of the figure show the temporal schema corresponding to schema-change walls in March, May and July respectively. The two time lines correspond to an `<assay>` item. The top time line is that contained in the March document; the bottom one is that contained in the July document. The `replace` and `useNew` methods are used for item correspondences in July and September, respectively. The item identifier during this period is the attribute '`assayKey`' of `<assay>`. In July we replace the `<assay>` element with a `<genotypingAssay>` element, with a `genoID` attribute as the item identifier and a mapping type of `replace`. In September the item identifier is changed back to the `name` attribute, with a mapping type of `useNew`.

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

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

For the first case, no extra work is needed as the items can be bridged directly using one of the above four methods.

But, to handle cases 2 and 3, the following two approaches were considered.

- Associate the pieces of an item across a schema change wall by virtually extending period of versions of the item. As an example, in Figure 52, bridging the two pieces of item B involves virtually extending period of item B's last version until the end of P_2 as if it were present during the last snapshot; and virtually extending its first version's period until the start of P_3; and then performing the bridging using one of the above four methods. Similarly, for the item corresponding to the third line, bridging involves virtually extending the period of the item C's last version from P_1 over multiple schema-constant periods followed by bridging using one of above methods. So P_1's version is extended to the wall, then bridged to a virtual element over all of P_2, then bridged to the extended element in P_3.

- The second option is not to extend the "item" across a schema change wall if it does not exist. So the item matching semantics, e.g., "useNew" matches only those items that exist immediately before the wall with those that exist immediately after the wall. As an example, in Figure 52, bridging the two pieces of items B and C having gaps in their existence across the schema change walls is not possible.

We decided to take the second approach, since we couldn't really "know" a priori if an item that reappears is the same item or a different one from the earlier one.

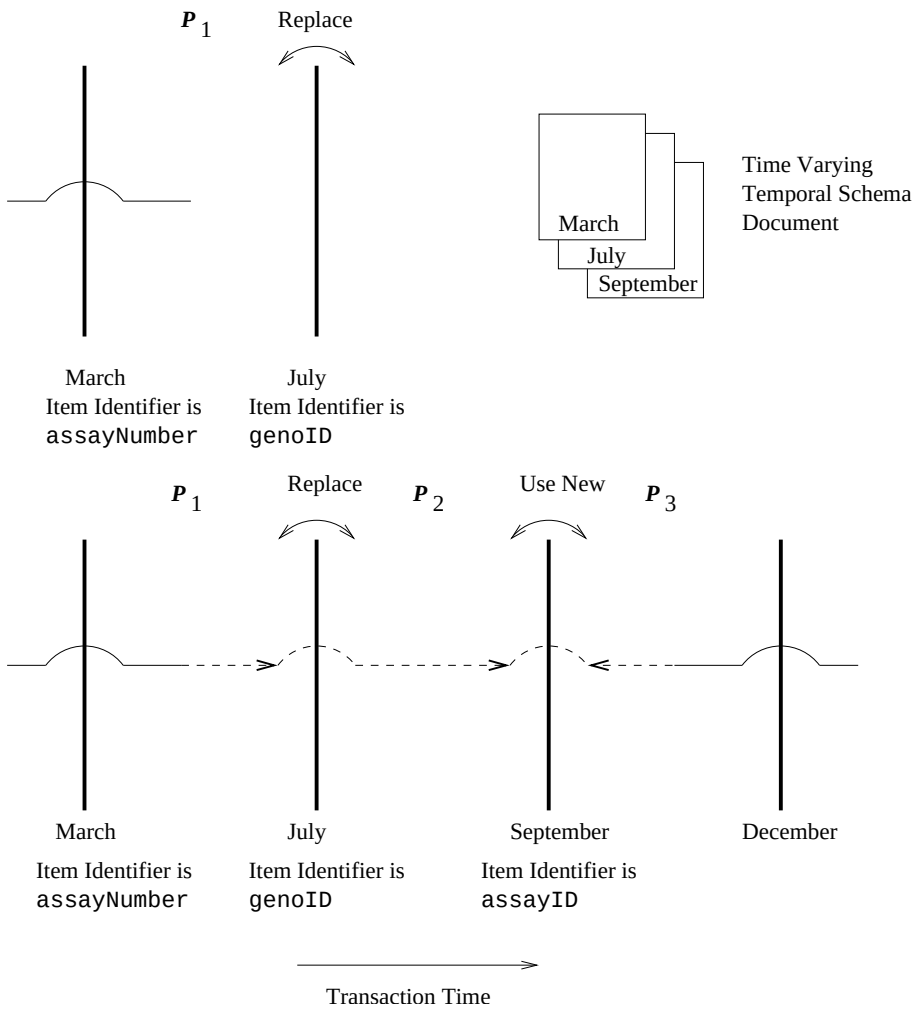


Figure 53: Cross-Gap Gluing

16.3 Semantics for mixed data and schema changes

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

We considered three ways to handle this situation.

1. Not allow any data change in a transaction containing schema changes. This is the most stringent option and makes the user's job more difficult, forcing him to split the task into multiple transactions. This may not be always feasible from real world point of view. Consider a situation where an element is modified to have a new 'required attribute', data change is mandatory in this case and hence cannot be separated from schema change. It could be argued that this is achievable with addition of a new 'optional' attribute, followed by required data changes and then making the attribute required. But it requires more work from the user's side.

2. Allow schema changes to coexist with data changes, except for schema changes to item identifier

fields. This will eliminate the need of replace mapping list and the bridging could always be done using one of the three options '`useNew`', '`useOld`', or '`useBoth`'.

3. Allow data changes to coexist with schema changes within a transaction without any restrictions.

We decided to go with the third approach, as it is the most general. A schema change for an element can consist of changes to its structure or its attributes or to the element definitions nested within it. Thus, given two schemas, it becomes very difficult to find the difference between the schemas and to validate the versions. So, we decided not to validate versions of an item across schema change walls if a schema change is detected for it.

16.4 Non-Sequenced Constraints

A constraint is *non-sequenced* if it is applied to a temporal item as a whole (including the lifetime of the data entity) rather than to individual time slices. They are defined in a logical annotation as an extension of conventional XML Schema constraints. An example of a non-sequenced (cardinality) constraint is: "An item cannot change more than three times in a year.". This type of constraint cannot be validated using the conventional validator and thus needs to be validated using the 'Temporal Constraint Checker' module of τXMLLINT.

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

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

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

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

Consider a situation shown in Figure 54. It maintains the same conventions as Figure 52. Changes to an item are shown by X's. A new non-sequenced constraint is introduced during third schema-constant period P_3 stating that "An item cannot change more than three times in a year." But the item has already undergone four changes during previous schema-constant periods P_1 and P_2.

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

According to the second alternative semantics, there is immediately a violation of the constraint, due to activity during the previous two schema-constant periods.

We decided on the first alternative: to apply a non-sequenced constraint only within the schema-constant period in which it is defined. Thus the non-sequenced constraints are "turned off" on any schema change. So for instance a constraint that says that the content must be constant is checked only up to the schema wall, and then checked within the new schema starting from the wall. In effect the schema change deletes all the old constraints and then adds them back as new constraints.

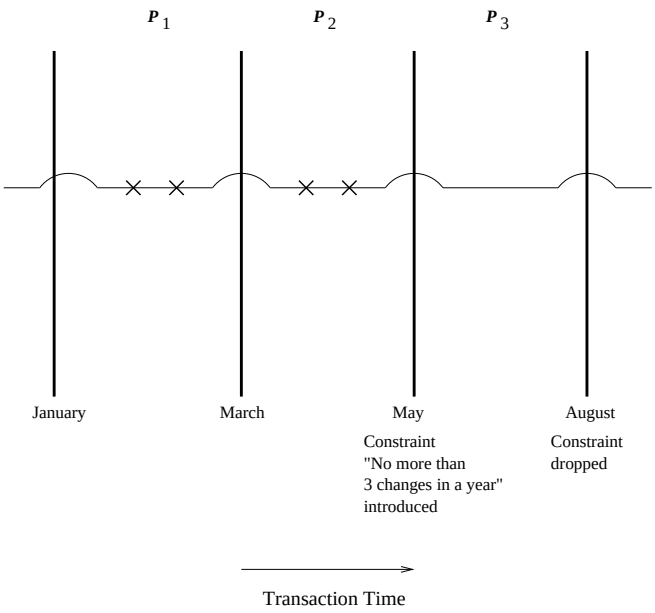


Figure 54: Non-Sequenced Constraints

17 Implementation

In this section we present the modifications and enhancements to the tools that have been implemented to support schema versioning. We also discuss the methodology used by τXMLLINT and its relationship to the representation used. These descriptions extend those presented in Section 9.

17.1 Overview

The tools are open-source and beta versions are available [65] and the full implementation and architecture is described by Joshi [47]. Figure 55 shows the overall architecture of the tools as a UML class diagram [61]. The architecture consists of three packages: `tau.xml` for the interface of each tool; `tau.time` for classes that handle time; and `tau.util` for utility classes common to all tools and classes.

The tools have been implemented in Java using the DOM API [82]. The DOM API was chosen over SAX API due to its ability to create an object-oriented hierarchical representation of the XML document in main memory which can be navigated and manipulated at run-time. This capability has proven to be extremely useful in all of the tools.

Figure 62 on page 173 shows the overall architecture of the tools as they manage XML documents and their schemas. A sequence of non-temporal documents is input into SQUASH to create a temporal representation; this document can then be validated using τXMLLINT and SCHEMAMAPPER. UNSQUASH can be used to reconstruct the original non-temporal documents from the temporal representation, while RESQUASH can be used to create a new representation (e.g., different timestamp locations) from a given representation.

17.2 τXMLLINT

Figure 63 provides the validation procedure used by τXMLLINT. The first step is to pass the temporal schema into τXMLLINT, which ensures that the logical and physical annotations are consistent with the conventional schema and with each other. Once the annotations are found to be consistent, SCHEMAMAPPER is invoked to generate a representational schema from the original conventional schema and the logical and physical annotations. The representational schema is then used as the schema for the temporal document and input into a conventional validator (in this case, XMLLINT). The next step is to pass the temporal document and the temporal schema to *Temporal Constraint Validator Module*. This step is to enforce temporal constraints that are not possible to be enforced by the representational schema alone.

The algorithm for τXMLLINT is given in Figure 58. τXMLLINT is able to check for the following types of temporal constraints.

Content Constant Content of an element cannot vary over time.

Cardinality Constant Cardinality of an element cannot vary over time.

Existence Constant The element cannot disappear and reappear again.

Content Varying Applicability The contents of an item cannot change beyond the period specified by the `contentVaryingApplicability` element in the annotation.

Valid Time Frequency The element cannot change more than specified number of times specified by the `frequency` element.

Maximal Existence Period The element can exist only within the period specified by the `maximalExistence` element.

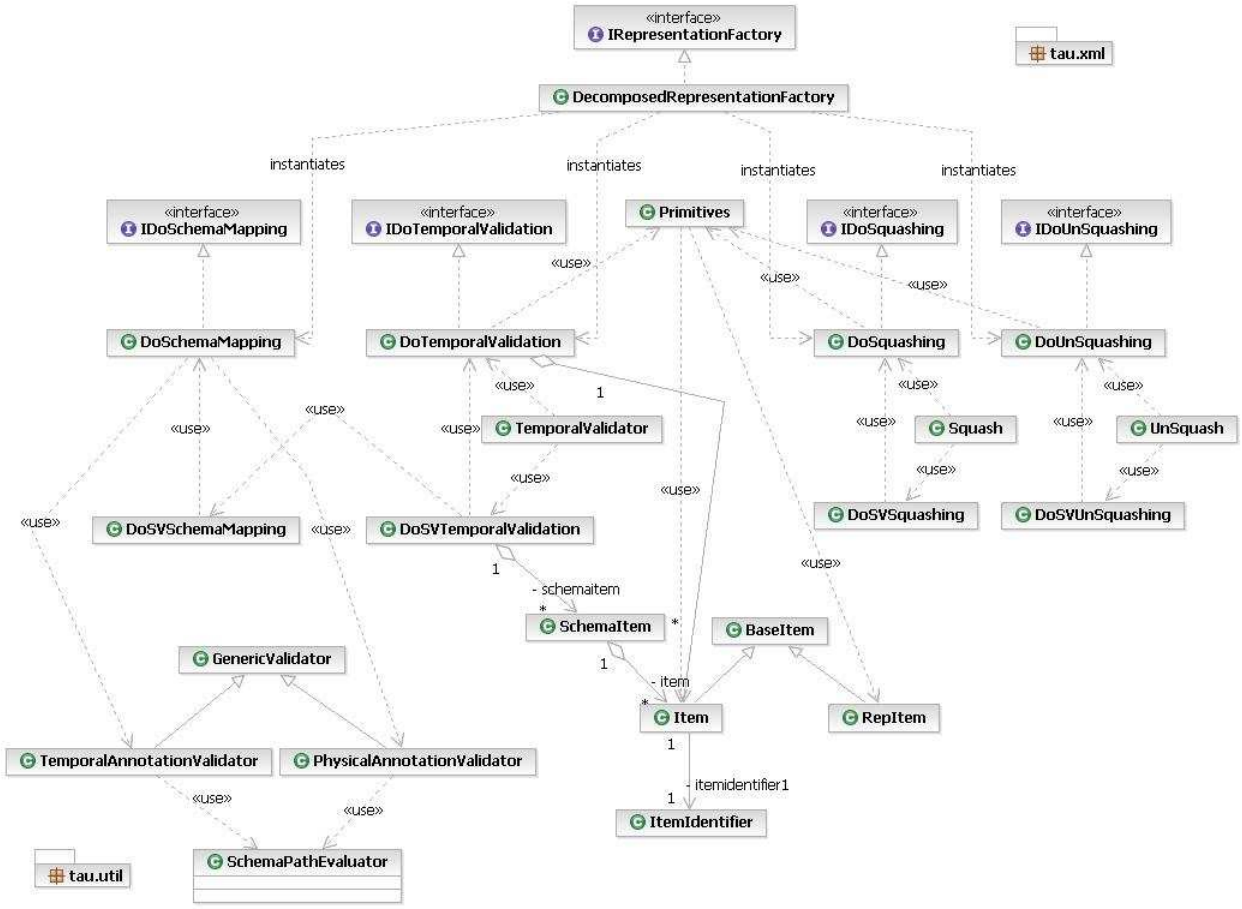


Figure 55: Overview class diagram for the tools

τXMLLINT enforces each temporal constraint using a simple brute-force approach. For example, for an item that has a content-constant constraint, the *Temporal Constraint Validator* loops through each version of the item and determines whether the nodes are DOM-equivalent.

17.3 Tool Modifications and Extensions

In order to implement different classes of representations (as described in Section 18), the tools were reorganized to abstract the details of the representation so that these details may vary freely without affecting the rest of the code. This was achieved by introducing abstract factory methods [35] in each of the tools (SQUASH, UNSQUASH, SCHEMAMAPPER, τXMLLINT) in place of the original methods; each abstract factory method would then call the appropriate concrete method based on the type of representation specified by the user. Figures 59 and 60 show the placement of the abstract factory method.

The changes to allow the edit-based representation to be used within the tools are described in detail in Section 18.4, and the changes to the item-based representation are described in Section 18.5.

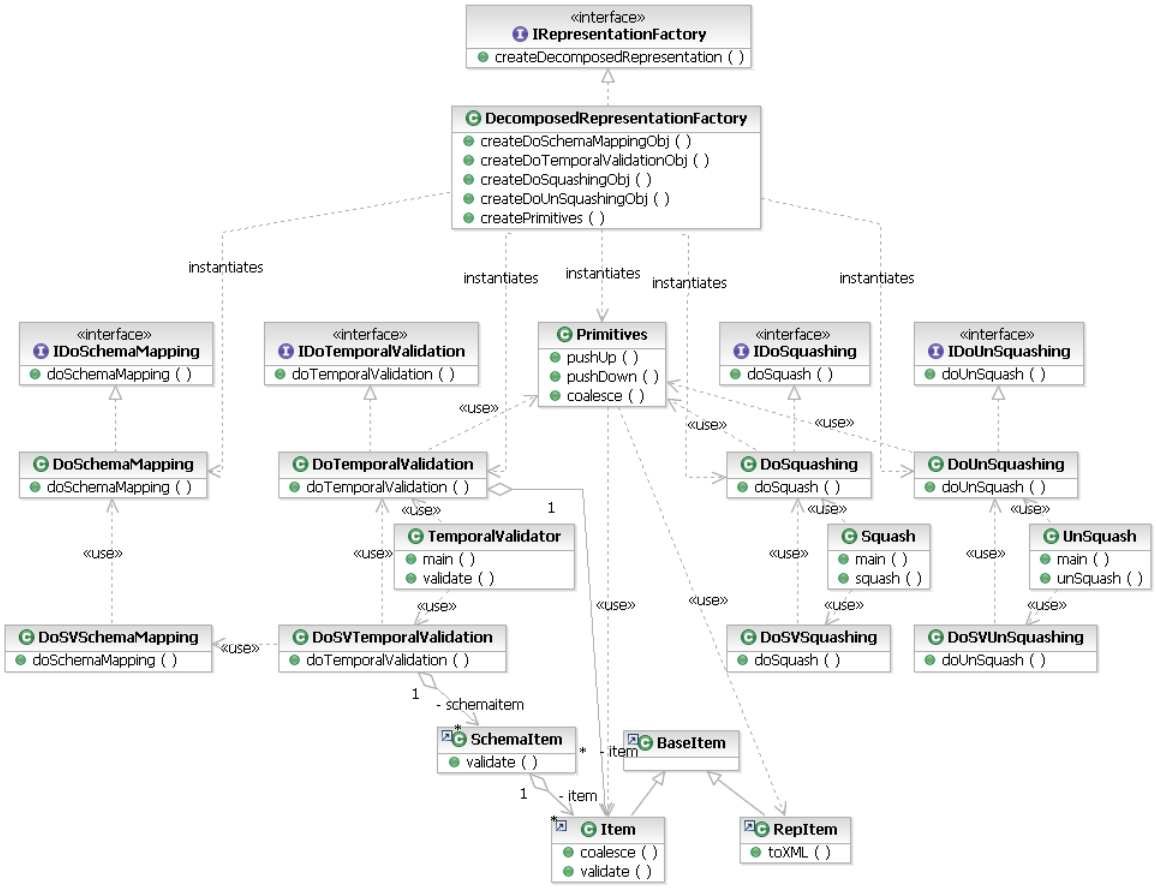


Figure 56: Detailed class diagram for `tau.xml`

17.4 Schema Versioning

To implement schema versioning, the representational schema had to be generalized so that each schema-constant period corresponds to a new namespace. Each namespace begins with the root element of the slice and describes all changes within the SCP in the same way as described in previous sections.

In particular, the tools are constructed in a such a way that schema versioning is handled by a different Java class than schema-constant cases. This allows flexibility in the way that schema-versioning is implemented and provides an abstraction to the developer. Both the schema-versioning and schema-constant classes implement the same interface so that the other parts of the tools do not need to be aware of the fact that schemas are changing. The schema-versioning classes make use of the functionality within schema-constant classes for each SCP; this again reduces code duplication and promotes software reuse.

Figure 61 shows the method for validating a temporal document with a time-varying schema. (This method was also presented in Section 18.6 as Figure 63.) To validate such a document, τXMLLINT applies the conventional validator to the document, using the representational schema produced by SCHEMAMAPPER. It then determines the times when the schema changes, thus determining schema-constant periods. For each such period, the time-varying data checker is invoked to check the temporal integrity constraints over the time-varying data, with the single base schema and annotations. Then the *temporal constraint checker* glues across the schema change walls and performs the temporal checks across these walls.

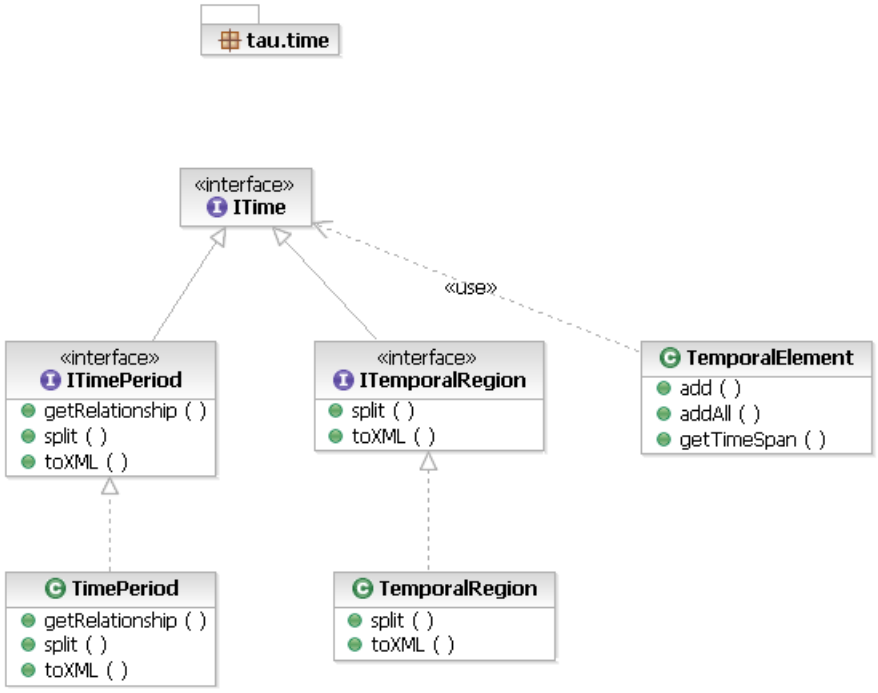


Figure 57: Detailed class diagram for `tau.time`

The framework for cross-wall validation is described in detail by Joshi [47] and our implementation closely follows his design. Briefly, the tools must consider the following issues that arise with schema versioning.

Accommodating evolving keys. When a schema-change wall is encountered, items across the wall need to be associated. This process is called *cross-wall* gluing or *bridging*. This becomes especially tricky if either the conventional key (specified in the base schema) upon which an item is defined, or if the item identifier itself (specified in the logical annotation) changes. The solution is to use an `<itemIdentifier-Correspondence>` element to determine the type of mapping desired, specifying how old item identifiers are to be mapped to new item identifiers. As described in Section 16.1 this element has four attributes: `oldRef`, a string naming an item that appears in the old schema, `newRef`, a string naming an item that appears in the new schema, `mappingType`, an XML Schema enumeration, and optionally a `mappingLocation`, which is a URI. We have defined four mutually exclusive mapping types.

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

Figure 58: Algorithm: τXMLLINT

```
//Inputs
// conventionalSchema - Parsed conventional schema document
// logicalAnnotation - Parsed logical annotation document
// physicalAnnotation - Parsed physical annotation document
// temporalDocument - Parsed temporal document
function doTemporalValidation (conventionalSchema, logicalAnnotation, physicalAnnotation,
                               temporalDocument):
    initialize a hash-table with item-identifier as key and item as hash value
    if Consistent(conventionalSchema, logicalAnnotation, physicalAnnotation)
        repSchema ← doSchemaMapping(conventionalSchema, physicalAnnotation)
        if conventionalValidator(temporalDocument, repSchema)
            for each element e in the temporalDocument do
                if isTimeVarying(e, logicalAnnotation)
                    evaluate the item-identifier
                    if item-identifier in hash-table
                        if the element is DOM-equivalent to some version in the item
                            coalesce the metadata with the version
                        else
                            create a new version
                    else
                        create a new item in hash-table, with one version
            for each item in hash-table do
                for each sequenced and non-sequenced constraint in temporalAnnotation do
                    if the constraint is not satisfied
                        display errors
        else
            display errors generated by the conventional validator
    else
        display errors
```

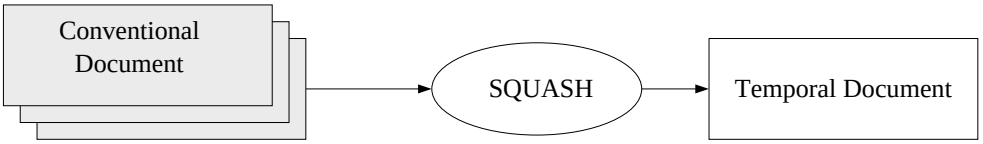


Figure 59: SQUASH before abstract factory methods were added.

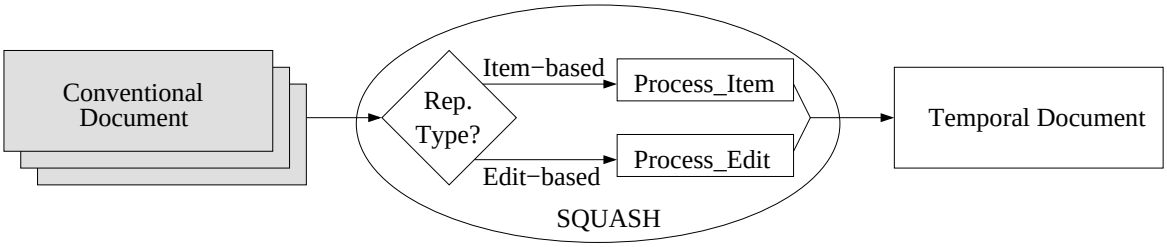


Figure 60: SQUASH after abstract factory methods were added.

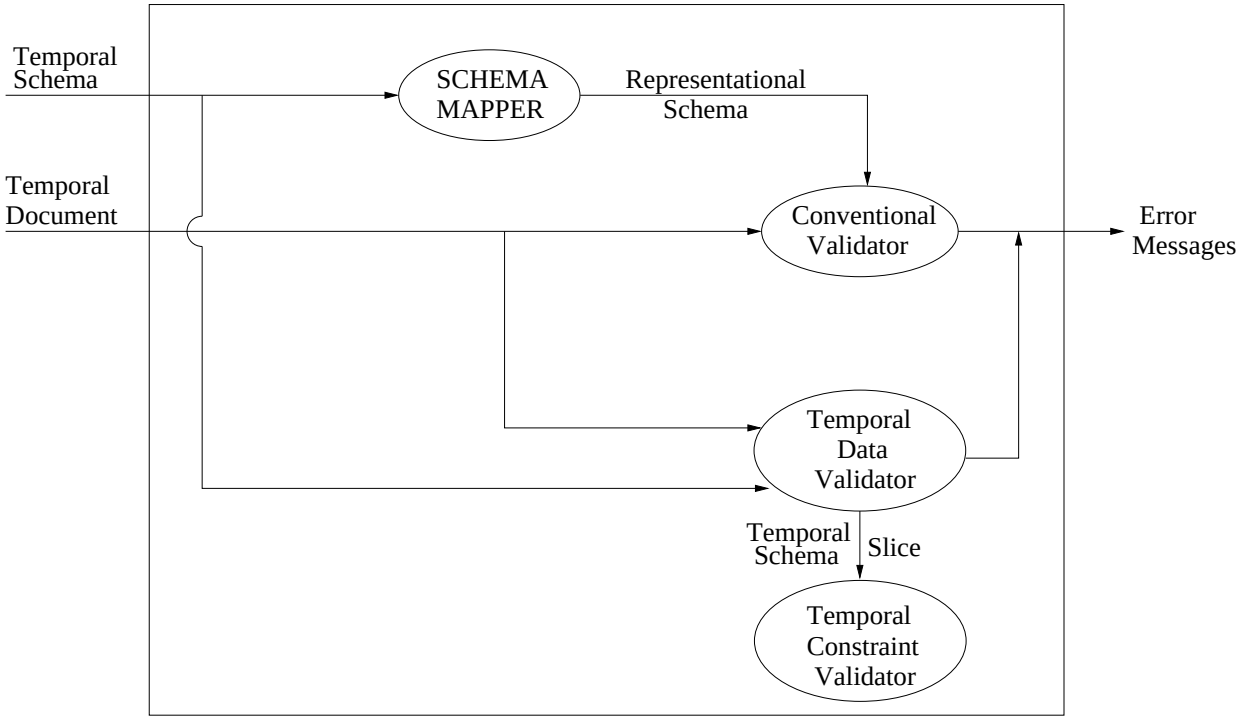


Figure 61: Validating a document with Time-Varying Data and a Time-Varying Schema.

This could be best described with an example. Say that in 2002 the item identifier is the `athID` attribute of the `<athlete>` element. In January 2005, this attribute is renamed `athNumber`; we specify a mapping type of `useBoth`. In March 2005, the item identifier is changed to the `athName` element, with a mapping type of `useNew`. (This attribute has been around since 2002, but it wasn't used as a key until January 2005.) Assume that, in June 2005 we add a new attribute, `athKey`, and specify that as the item identifier, with a mapping type of `useOld`. Finally, in July 2005, just before the beginning of the games, we replace the

<athlete> element with a <player> element, with a playerID attribute as the item identifier and a mapping type of replace.

The gluing of elements into items is then done the following way. Before 2005, the athID is used for gluing. When the schema change occurs sometime in January 2005, we glue across the schema change by matching the athID value of the element before the schema change with the athNumber value after the change: these (integer) values must match for the two elements to be glued. In March 2005, we glue across the schema change by matching up old elements and new elements that have the same (string) value for their athName element, the new item identifier. The only difference is that before the schema change, that element was present but wasn't being used as a key. In a consistent fashion, in June we also glue using the athName element, which was the *old* item identifier.

July is the most complex. We need to glue an <athlete> element with an item identifier of athKey with a <player> element with an item identifier of playerID. For this, we use the MappingLocation attribute in the temporal schema to access a mapping table that provides a list of pairs, each with an athKey and a playerID value.

Accommodating gaps. When gaps appear in the lifetime of an item, the process of finding the correspondence between the items from corresponding SCPs becomes more difficult. To handle the case of gaps, it has been decided to create a new item when the element reappears; that is, the first item is not virtually extended across a schema change wall, since it is difficult or impossible to know a priori if an item that reappears is the same item or a different item. Thus, when gaps appear, the tools create a new item and both the original item and new item are treated like every other item in the evaluation.

Semantics for mixed data and schema changes. When a data change co-exists with a schema change within a single transaction, τXSchema places no restrictions on the types of changes allowed; this is the most general and flexible design. However, given two schemas, it becomes very difficult to find the differences between them and validate the versions. Thus, versions of an item across schema change walls are not validated if a schema change is detected for it. This decision leads to no additional work needed from the tools.

Non-sequenced constraints. Here we consider temporal constraints such as *"An item can only be changed 3 times per year."* Note that it has been decided to consider non-sequenced constraints only within a SCP since the user might introduce unintended complexity when specifying new constraints in the middle of the year: should the constraint be checked only from that point forward or from the beginning of the document's lifetime? This means that the tools must verify each constraint within each SCP separately, which is relatively simple. For example, for the above temporal constraint, the tools must simply check the number of versions of an item and make sure that it is less than or equal to the specified amount. Note that only the theoretical design exists for some non-sequenced constraints; full implementation is left for future work.

We note that the complete set of features and functionality described above is not yet implemented by the tools.

17.5 Packages

Here we describe the packages that have been created to support the tools.

tau.xml This package contains interfaces and classes corresponding to tools τXMLLINT, SCHEMAMAPPER, SQUASH, and UNSQUASH. The details of the important classes used for data versioning are given below.

- `Item`: Provides an abstraction for a logical item. It contains methods for manipulating versions, their coalescing validation.
- `RepItem`: Provides an abstraction for actual representation item element in the XML document. It provides methods for conversion of an XML element to/from a logical item. Both these classes extend from the base class `BaseItem`, which provides common functionality.
- `ItemIdentifier`: Provides an abstraction for item-identifier.
- `Primitives`: Provides implementation for primitives explained in Section 9.1.
- `LogicalAnnotationValidator` and `PhysicalAnnotationValidator`: Provide checks for the consistency of logical and physical annotations with the conventional schema.
- `DoSchemaMapping`, `DoSquashing`, `DoUnSquashing`, `DoTemporalValidation`: Provide the implementation for the algorithms explained in Section 7. Each of the classes implement corresponding interfaces preceding their name by '`I`'. As an example, `DoSquashing` implements `IDoSquashing`.

The extended tools for schema versioning use these classes internally to manipulate schema-versionsed XML documents. The classes used for schema-versioning are `DoSVDataSquashing`, `DoSVDataUnSquashing`, `DoSVTemporalValidation`, and `DoSVSchemaMapping`, where '`SV`' stands for 'schema-versioned'. The implementation of these classes first identify schema-constant-periods and call corresponding data-versioning classes on individual schema-constant-periods.

The classes `Squash`, `UnSquash` and τXMLLINT provide commond-line tools for the end-user. These classes accept temporal schemas and temporal documents files as command line parameters and internally invoke schema-versioning or data-versioning tools depending upon whether the schema is versioned or not.

tau.time This package contains the implementation of classes to handle time. It provides implementation for both `TimePeriod` (used for single time dimension) and `TemporalRegion` (used for bitemporal elements).

tau.util This package contains utility classes. `SchemaPathEvaluator` provides abstraction for evaluating schemapath expressions. Given a target and reference element, the function checks for the consistency of the target according to the conventional schema. This functionality is used by both `LogicalAnnotationValidator`, `PhysicalAnnotationValidator` and `ItemIdentifier`.

As explained, the class for every tool implements its corresponding interface. Thus, it is easily possible to accommodate a new implementation of these tools for a new representations without necessitating many changes to the overall picture. Use of 'Abstract Factory' design pattern makes the integration and selection of the new representation seamless by addition of just a few lines of codes to the `RepresentationFactory` class.

To add a new representation, we need to add new classes implementing the new representation for each tool. Each class needs to implement the corresponding interface mentioned earlier. Once these classes are added, a small addition of code is needed to the `RepresentationFactory` class. Then, any representation can be easily selected by providing corresponding parameter to the `RepresentationFactory` class.

18 Representations

In this section we present the design space for temporal representations in XML. We first introduce and describe some aspects relating to schema versioning. We then characterize the general design space. Then we elaborate on the edit-based, item-based, and slice-based approaches.

18.1 Schema Versioning Considerations

In the following sections our focus is on the different approaches to representig temporal data. However, central to each approach is the method of handling schema versioning. This section briefly summarizes some concepts that are present in each method.

A key idea that first appeared in a paper on temporal aggregation [74] is that of *schema-constant periods (SCP)*. It is possible, even with versioned schemas having themselves versioned schemas, to identify contiguous periods of time when there are no schema changes, anywhere. These are termed as schema-constant periods. These periods are non-overlapping and continuous; between the periods are schema change *walls*. Now, during these periods the data may be (and probably is) versioned, but at least we have 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, so it is possible to read and interpret the temporal document during that schema-constant period, and even to validate that portion of the document. So a general temporal document can be viewed as a sequence of data-varying documents, each over a single schema-constant period. Since we can validate each schema-constant period, given the approaches outlined by Joshi [47], all we have to do is validate across schema changes.

For each of the representation classes below, we will show how schema change walls are handled. However, for simplicity, most examples will contain a single schema constant period without loss of generality.

18.2 Design Space

Researchers have proposed and evaluated many different temporal representations on an individual basis [12, 19, 38, 47, 52, 67]. We have found that all extant representations can be categorized into one of four categories depending on the decisions made to the following two considerations. The first consideration is whether the resulting representation will keep the entire content of a slice explicitly or use some sort of compression which requires slices to be reconstructed but eliminates data duplication. We call this decision *Direct* or *Indirect*. The second consideration is whether the resulting representation will explicitly capture the changes to the XML tree or will only capture changes to the file itself without any knowledge of XML. We call this decision *Itemed* or *Flat*. As these considerations are orthogonal, they induce four possible classes of representations, as shown in Table 4. The Flat classes do not consider XML structure and instead treat the file as a flat text file; the Itemed classes use XML structure within the representation. The Direct classes maintain full versions of each slice in the representation which may result in data duplication; the Indirect classes use some sort of compression which requires slices to be reconstructed but alleviates all data duplication. We now name and briefly describe each class in turn.

	Flat	**Itemed**
Direct	*Slice-based*	*Item-based*
Indirect	*Edit-based*	*Reference-based*

Table 4: The design space of temporal representations and the resulting classes.

The *slice-based* representation class maintains the full text content of each slice throughout the entire

history of a document and does not use knowledge of XML structure. Each new slice is simply appended to the representation with the appropriate timestamp. In this way, a full history of slices is maintained in a single representation. Two advantages of the slice-based scheme are its simplicity and its ease of implementation: no processing or logic is needed to add a new slice. Another advantage is the ease of reconstructing an arbitrary slice: one must simply find the requested timestamp and select the corresponding slice. The major disadvantage of this scheme is the size of the resulting representation: it will grow linearly with the size and number of slices and will thus require both a larger amount of disk space and a large amount of memory for processing. We elaborate on this in Section 18.3.

The *edit-based* representation class maintains the full text content of only the most recent slice, storing reverse edit scripts for each additional slice. It does not use knowledge of XML structure; instead it uses the well-known `diff` tool to compute the text differences between two slices. This scheme can often result in an extremely compact representation but requires extra processing to reconstruct and validate past slices. More detail is given in Section 18.4.

The *item-based* representation class creates and maintains an item for each time-varying XML element. It keeps each slice intact and uses knowledge of XML structure. An *item* is a collection of versions that in concert represent the same real-world entity. It is a logical entity that evolves over time through various slices. This scheme is compact and allows for fast validation. More detail is given in Section 18.5.

The *reference-based* representation class is similar to the item-based representation in that it uses the concept of items. However, the reference-based scheme depends on key identifiers (via XML Schema `<key>` elements) in each element node. Here, every item is present as a child of a top level element, and then each slice representation makes references to one or more of these items. Thus, this scheme factors out common data items to avoid duplication, but as a result the slices do not remain fully intact. This scheme provides similar size performance to the item-based scheme on average and performs similarly in validation time.

Listings 116–121 show two slices of an XML document and the resulting representation in each of the four classes of representations. In each case, SQUASH is used to transform the two slices into the single representation, and UNSQUASH can be used to recreate the original two slices from each of the four representations. Table 4 summarizes the design space.

Listing 116: Slice on 2008-01-01.

```
<!-- 2008-01-01 -->
<person>
  <fname>Steve</fname>
  <age>24</age>
</person>
```

Listing 117: Slice on 2008-03-17.

```
<!-- 2008-03-17 -->
<person>
  <fname>Steve</fname>
  <age>25</age>
</person>
```

Listing 118: Slice-based representation.

```
<tv_root>
  <timestamp begin="2008-01-01">
    <person>
      <fname>Steve</fname>
      <age>24</age>
    </person>
  </timestamp>
  <timestamp begin="2008-03-17">
    <person>
      <fname>Steve</fname>
      <age>25</age>
    </person>
  </timestamp>
</tv_root>
```

Listing 119: Edit-based representation.

```
<tv_root>
  <timestamp begin="2008-03-17">
    <person>
      <fname>Steve</fname>
      <age>25</age>
    </person>
  </timestamp>

  <timestamp begin="2008-01-11" >
    <change lines="3">
      &lt;age&gt;24&lt;/age&gt;
    </change>
  </timestamp>
</tv_root>
```

Listing 120: Item-based representation.

```
<tv_root>
  <person>
    <fname>Steve</fname>
    <age_Item>
      <age_Version begin="2008-01-01">
        <age>24</age>
      </age_Version>

      <age_Version begin="2008-03-17">
        <age>25</age>
      </age_Version>
    </age_Item>
  </person>
</tv_root>
```

Listing 121: Reference-based representation.

```
<tv_root>
  <person>
    <fname>Steve</fname>
    <age_Item itemRef="1"/>
  </person>
  <age_Item itemID="1"/>
    <age_Version begin="2008-01-01">
      <age>24</age>
    </age_Version>
    <age_Version begin="2008-03-17">
      <age>25</age>
    </age_Version>
  </age_Item>
</tv_root>
```

18.3 Slice-Based Representation

As mentioned previously, the *slice-based* representation class maintains the full text content of each slice throughout the entire history of a document and does not use knowledge of XML structure. Each new slice is simply appended to the representation with the appropriate timestamp. In this way, a full history of slices is maintained in a single representation, albeit with obvious data duplication.

Since any data change will result in a new copy of the entire XML tree, and any schema change will result in a new wall, we can expect the size of the representation to grow linearly with the number of slices. In some real-world scenarios where each slice is on the order of kilobytes and the number of slices is measured in the thousands, this size growth can become problematic for both disks (for storage) and memory (for parsing and processing). However, this approach is extremely simple and so it is often a researcher's initial idea.

This approach can be thought of as a special case of the item-based scheme with the physical timestamp placed at the root. With this strategy, the slice-based scheme can be trivially implemented into the τXSchema tools.

18.4 Edit-Based Representation

The *edit-based* scheme (also called *diff-based* or *delta-based*) is proposed and described in several research papers [13, 19, 53]. Briefly, this representation maintains the most recent version of the document and then only the edits necessary to transform each slice into the previous (see Listing 119).

The edit-based scheme has the potential of minimizing the representation size in some cases, and on average, will result in significantly smaller representation than the slice-based approach. Another advantage of the edit-based scheme is the relative simplicity of the construction of the representation.

However, the edit-based approach suffers from high processing overhead to reconstruct early versions, since the edit script has to be applied for every slice in between. It is also difficult to make time-traveling queries [37] since either the entire version history must first be reconstructed, or a complex analysis of the edit scripts must be performed. Also, it is important to note that the edit scripts are saved as text only and not XML trees; as a result, it is difficult or impossible to validate temporal constraints directly on the edit scripts.

We have implemented this representation in the τXSchema tools with the following approach. First, we take the most recent slice in its entirety and place it as a subelement of a `<timestamp>` element. Then, for each successive slice, we run `diff -e` to compute the difference. `diff` is a standard command line tool

that computes the difference between two files. The `-e` option formats the output into an `ed` script. We encode the output of `diff -e` as follows.

Listing 122: `diff` output.

```
1 6c
2     <quantity>2</quantity>
3 .
```

Listing 123: Edit-based encoding.

```
1 <change lines="6">
2     &lt;quantity&gt;2&lt;/quantity&gt;
3 </change>
```

We create a `<change>`, `<add>`, or `<delete>` element depending on the operation specified by `diff` (line 1 of Listing 122), along with the `lines` attribute. The text content of the new element is set to be the output of `diff` with special characters (e.g., angle brackets) encoded to retain valid XML syntax (i.e., "$<$" is replaced with"`<`"). To recreate an arbitrary slice, we iteratively apply the `patch` tool on the `diff` output.

One important but easily overlooked detail with this approach is the issue of white space. Since white space is an important and necessary characteristic of `diff` output, it should be captured and maintained in the resulting representation. However, the default behavior of DOM and other parsers is to ignore whitespace in elements that don't explicitly set the `xml:space` attribute to `preserved` [84], which could cause problems during the `patch`ing stage. For example, consider the simple document shown in Listing 124. After being parsed by DOM and then written to disk, the resulting file could look something similar to the document shown in Listing 125.

Listing 124: Original document.

```
<a>
  <b>foo</b>
</a>
```

Listing 125: Parsed and output by DOM.

```
<a><b>foo</b>
</a>
```

Since the edit-based representation outputs the latest slice in its entirety and then uses this as the starting point for the reverse edit scripts, it is crucial that newlines be preserved. To achieve this, we introduce a filter that is applied to the conventional documents before they are parsed by DOM; the filter encodes each newline in the original document with a `<tXnl>` element (meaning "τXSchema new line"). See Listing 126 and 127 for an example filter output.

Listing 126: Original document.

```
<a>
  <b>foo</b>
</a>
```

Listing 127: After filter and DOM mangling.

```
<a> <tXnl/><b>foo</b><tXnl/>
</a>
```

We then apply the reverse filter after writing the latest slice to disk (that is, the filter removes all newlines and then replaces all `<tXnl>` with newlines) to ensure that the newly written file matches the newline structure of the original, and thus the reverse edit scripts will work correctly.

To implement temporal validation of this representation, we use the following approach. For each schema-constant period, we UNSQUASH the temporal document into individual slices and then SQUASH them into a slice-based representation. We can then use the tools to validate the new slice-based representation. This allows us to use the same (unmodified) tools to validate all three representation classes, which promotes software reuse and reduces complexity. We argue that this approach is necessary in order to validate some temporal constraints, since it is impossible to do so using the edit scripts alone. However, there is some performance degradation due to the extra steps involved. This tradeoff is quantified in Section 18.7.

Note that the above approach must treat each schema-constant period separately, as opposed to the entire representation, due to the nature of the edit-based scheme. Section 18.4.2 below elaborates on this idea.

Through this implementation strategy, we are able to fully capture, reproduce, and validate the changes

between slices with the help of commonly available tools and create a practically useful edit-based representation.

18.4.1 Capturing Namespaces

Since the edit-based approach does not represent data changes in an XML format (rather, these changes are captured in encoded text within an XML element), the issue of capturing namespaces does not arise. The output of `diff` does not indicate what kind of change occurred on a line (e.g., namespace change versus element change), only what lines changed and the new values of those lines. Thus, when namespace changes occur, we cannot detect them explicitly and so we handle them in the same manner as any other schema change.

18.4.2 Schema Versioning

Changes to the schema are handled with the following approach. For each schema change, we represent a wall by inserting a new `<schemaVersion`X`>` element. Each `<schemaVersionX>` is populated with a set of one or more `<tv:timestamp>` sub-elements: the first such sub-element contains the entire version of the most recent slice in the current schema-constant period, and each additional sub-element contains the edit script produced by `diff`. Listing 128 shows an example edit-based representation with schema versioning.

Listing 128: Edit-based representation with schema versioning.

```
<sv_root>
<schemaItem>
  <schemaVersion0>
    <tv:timestamp begin="2008-03-17">
      <rep0:person>
            ...
      </rep0:person>
    </tv:timestamp>
    <tv:timestamp begin="2008-01-11" >
      <change lines="3">...</change>
    </tv:timestamp>
    <tv:timestamp begin="2008-01-07" >
      <change lines="8">...</change>
    </tv:timestamp>
    ...
  </schemaVersion0>
  <schemaVersion1>
    <tv:timestamp begin="2008-04-23">
      <rep0:person>
            ...
      </rep0:person>
    </tv:timestamp>
    <tv:timestamp begin="2008-04-10" >
      <change lines="3">...</change>
    </tv:timestamp>
    ...
  </schemaVersion1>
</sv_root>
```

The design of this representation is such that each SCP is exactly similar in syntax and format to the static schema case. The representational schema must then be constructed so that for each SCP the contents of the base schema are inserted as a sub-element of a timestamp element. Timestamp elements are also permitted to have `change`, `add`, or `delete` sub-elements to encode the edit scripts.

18.5 Item-Based Representation

The *item-based* scheme [47] creates and maintains an item for each time-varying element and was the original representation type implemented in τXSchema. An *item* is a collection of XML elements that in concert represent the same real-world entity. It is a logical entity that evolves over time through various slices. An item can occur at any level in the XML tree hierarchy, and is specified by the user via a physical annotation. Every occurrence of the actual time-varying element from the conventional document is replaced by its corresponding item. The item-based representation has the following features.

- Only elements can be time-varying and can have versions. The immediate content (text and attributes), is considered to be an integral part of an element and therefore does not have a separate time-varying lifetime.
- A version of an element is created when the immediate content or attributes of an element change. This includes text content, sub elements, comments, and processing instructions. The change must be observable through DOM: only changes observable through DOM create a new version. This implies that whitespace and attribute ordering does not create a new version. In contrast, since the edit-based representation uses `diff` to observe changes, whitespace and attribute ordering would create a new edit script in the representation.
- If an element is glued but remains unchanged, then the lifetime of the current version of the element is extended; no new version is created. This implies that versions are coalesced [47].
- The timestamp that represents the version's lifetime is a N-dimensional temporal element. It may include `now`, `until changed`, and/or indeterminate times.

We now extend the item-based representation to explicitly capture schema versioning and namespaces with elements. These tasks are accomplished with the use of additional elements and namespaces in the resulting representation. Briefly, each element e in the original document with namespace ns will take on the following form in the representation,

`<ns:e>` $\Rightarrow$ `<rep`X`_`Y`_ns:e>`

where X and Y are unique integers for each schema-constant period and namespace change, respectively. The following sections describe this production in more detail and provide examples.

18.5.1 Capturing Namespaces

A conventional document may have more than one namespace, with each namespace associated with a different schema. Further, these namespaces may change from slice to slice without a schema changing (i.e., the namespace mapping in the conventional document changes, but the conventional schema does not change). To capture and reproduce such situations, each namespace in each slice must be mapped by the tools to a unique namespace in the representation. Consider the two slices shown in Listings 129 and 130.

Listing 129: Slice on 2008-01-01.

```
<!-- 2008-01-01 -->
<a>
  <ns1:b>foo</ns1:b>
  <ns2:c>bar</ns2:c>
</a>
```

Listing 130: Slice on 2008-03-17.

```
<!-- 2008-03-17 -->
<a>
  <ns1:b>foo1</ns1:b>
  <ns2:c>blah</ns2:c>
</a>
```

In this simple example, three different namespaces are used and remain constant between slices (specifically, the default namespace, `ns1` and `ns2`). The resulting representation (Listing 131) has one corresponding namespace for each of the original namespaces. Here, the `rep` namespace corresponds to the default namespace used by `<a>` in the slices.

Listing 131: Item-based representation of Listings 129 and 130.

```
<tv_root>
  <rep:a>
    <!-- rep_ns1 is a newly created namespace -->
    <rep_ns1:b_RepItem>
      <rep_ns1:b_Version begin="2008-01-01">
        <ns1:b>foo</ns1:b>
      </rep_ns1:b_Version>
      <rep_ns1:b_Version begin="2008-03-17">
        <ns1:b>foo1</ns1:b>
      </rep_ns1:b_Version>
    </rep_ns1:b_RepItem>

    <!-- rep_ns2 is a newly created namespace -->
    <rep_ns2:c_RepItem>
      <rep_ns2:c_Version begin="2008-01-01">
        <ns2:c>bar</ns2:c>
      </rep_ns2:c_Version>
      <rep_ns2:c_Version begin="2008-03-17">
        <ns2:c>blah</ns2:c>
      </rep_ns2:c_Version>
    </rep_ns2:c_RepItem>
  </rep:a>
</tv_root>
```

Since no naming conflicts are present in this scenario, the unique integer Y is not necessary and is therefore omitted. Similarly, the unique integer X for the SCP is omitted. To illustrate a scenario where a naming conflict occurs, consider the following two slices.

Listing 132: Slice on 2008-01-01.

```
<!-- 2008-01-01 -->
<a>
  <ns1:b>foo</ns1:b>
  <rep_ns1:c>bar</rep_ns1:c>
</a>
```

Listing 133: Slice on 2008-03-17.

```
<!-- 2008-01-01 -->
<a>
  <ns1:b>foo1</ns1:b>
  <rep_ns1:c>bar</rep_ns1:c>
</a>
```

Since `<ns1:b>` is time-varying, we need to create an item and thus a new namespace for the item. We can not use `rep_ns1` in the resulting representation because that namespace already exists in the original slice and would cause confusion. Instead, we create a new namespace `rep_0_ns1`, as shown in Listing 134.

Listing 134: Item-based representation of Listings 132 and 133.

```
<tv_root>
  <rep:a>
    <rep_0_ns1:b_RepItem>
      <rep_0_ns1:b_Version begin="2008-01-01">
        <rep_0_ns1:b>foo</rep_0_ns1:b>
      </rep_0_ns1:b_Version>
      <rep_0_ns1:b_Version begin="2008-03-17">
        <rep_0_ns1:b>foo1</rep_0_ns1:b>
      </rep_0_ns1:b_Version>
    </rep_0_ns1:b_RepItem>

    <rep_rep_ns1:c>bar</rep_rep_ns1:c>
  </rep:a>
</tv_root>
```

In this scenario, the value 0 happens to be the first unique integer that relieves the naming conflict. If there already exists a namespace `rep_0_ns1` in the original document, then the representation would try `rep_1_ns1`. If `rep_1_ns1` already exists, this process would iterate until no conflicts remain.

18.5.2 Schema Versioning

To achieve schema versioning in the item-based representation class, we introduce one level of abstraction. For each schema-constant period, we create a new representational schema in the normal way and define a unique namespace for this schema of the form `<rep`X`_`Y`>` as described above. Then, in the main representational schema, we import the representational schema for each SCP and define a new element `<schemaItem>`. Sub-elements of this element correspond to each SCP and thus each `<rep`X`_`Y`>` namespace. Listings 135 and 136 below show an simple example of schema versioning and Listing 137 shows the representational schema.

Listing 135: Version 1 of a simple schema.

```
...
<element name="athlete">
  <complexType mixed="true">
    <sequence>
      <element name="athName"
               type="string"/>
    </sequence>
    <attribute name="athID"/>
    <attribute name="age" />
  </complexType>
</element>
...
```

Listing 136: Version 2 of a simple schema.

```
...
<element name="athlete">
  <complexType mixed="true">
    <sequence>
      <element name="athName"
               type="string"/>
    </sequence>
    <attribute name="athNumber"/>
    <attribute name="age" />
  </complexType>
</element>
...
```

Note that the files `rep0.xsd` and `rep1.xsd` correspond to the representational schemas created for the two SCPs. Each defines the namespace and elements corresponding to its SCP.

The temporal document then has one `<schemaVersion`X`>` element for each SCP and the representation proceeds in the normal way. Listing 138 shows the temporal document created in the scenario.

18.6 Functionality Placement: Schema vs. Tools

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

Before facing these issues, it is convenient to discuss the approach that the τXSchema tools take to validate temporal contraints. Figure 62 shows the overall architecture of the tools as they manage XML documents and their schemas. A sequence of non-temporal documents is input into SQUASH to create a temporal representation; this document can then be validated using τXMLLINT and SCHEMAMAPPER. UNSQUASH can be used to reconstruct the original non-temporal documents from the temporal representation, while RESQUASH can be used to create a new representation (e.g., different timestamp locations) from a given representation.

Figure 63 provides the validation procedure used by τXMLLINT. The first step is to pass the temporal schema into τXMLLINT, which ensures that the logical and physical annotations are consistent with the conventional schema and with each other. Once the annotations are found to be consistent, SCHEMAMAPPER is invoked to generate a representational schema from the original conventional schema and the logical and physical annotations. The representational schema is then used as the schema for the temporal document and input into a conventional validator (in this case, XMLLINT). The next step is to pass the temporal document and the temporal schema to *Temporal Constraint Validator Module*. This step is to enforce temporal constraints that are not possible to be enforced by the representational schema alone.

A key design decision during the validation of temporal constraints is the placement of functionality: should a constraint be implemented in the representational schema or within the temporal constraint validator? Implementing (or *expressing*, or *enforcing*) constraints in the representational schema may provide

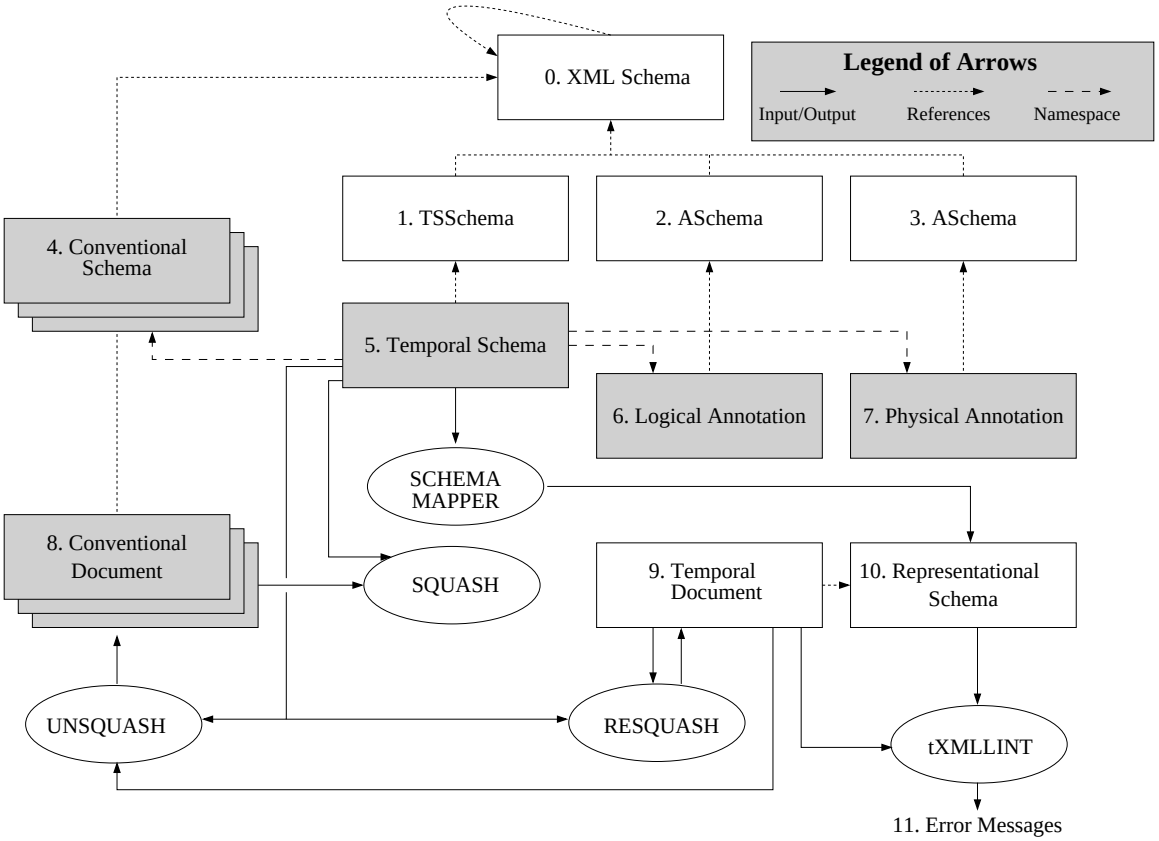


Figure 62: The overall architecture of τXSchema.

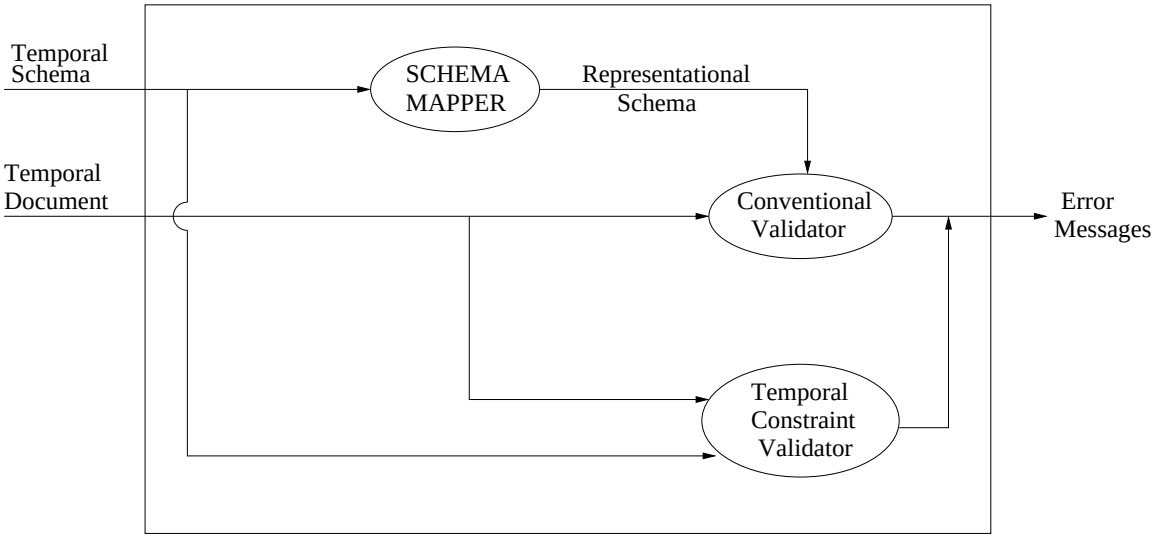


Figure 63: Validating a document with Time-Varying Data: τXMLLINT.

faster validation since a conventional validator can be invoked directly, but may result in increased size and complexity of the representational schema. Conversely, implementing constraints in the temporal validator

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

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

may yield small and compact schemas, but requires more time to perform the validation since the tools must perform checks across all slices sequentially and individually in the worst case.

In the following sections we explore two issues related to the placement of functionality. First, we determine which temporal constraints are possible to be expressed in a representational schema using the item-based representation class[9] and which can only be implemented in the temporal constraint validator. Second, for those constraints that can be implemented in both the schema and the temporal constraint validator, we provide a brief analysis of the tradeoffs between the two placements.

18.6.1 Constraints

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

Identity constraints These constraints restrict uniqueness of elements and attributes in a given document. Identity constraints are defined in the schema document using a combination of `<selector>` and `<field>` sub-elements within an `<key>` or `<unique>` element.

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

Cardinality constraints The cardinality of elements in XML documents is restricted by the use of `minOccurs` and `maxOccurs` in the XML Schema document. The cardinality of attributes is restricted using `optional`, `required`, or `prohibited`.

Datatype restrictions Datatype definitions can restrict the structure and content of elements, and the content of attributes. For example, a datatype definition can restrict the content of an element `<age>` to be between 0 and 100.

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

18.6.2 Sequenced Constraints

In this section we examine whether each class of sequenced constraints can be enforced by a representational schema. Given a conventional XML Schema constraint, we define the corresponding logical semantics in

[9]Section 18 introduces and describes the four kinds of representation classes and Section 18.6.4 outlines how the other three classes affect this analysis.

XML Schema in terms of a *sequenced constraint*. For example, a conventional (cardinality) constraint, "There should be between 0 and four 4 URLs for each supplier," has the following sequenced constraint: "There should be between 0 and 4 website URLs for each supplier *at every point in time*."

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

Identity Constraints We claim that identity constraints of elements and attributes *cannot* be enforced in a representational schema in the general case. To see this, consider the following example that begins with Listings 139 and 140. In this example, we require `<zip>` elements to have unique `code` attributes via an identity constraint named `zipUnique`.

Listing 139: XML Schema `<unique>`.

```
25 ...
26    <xs:unique name="zipUnique">
27      <xs:selector xpath="zip"/>
28      <xs:field xpath="@code"/>
29    </xs:unique>
30 ...
```

Listing 140: Unique `codes` (slice 1).

```
163 ...
164   <zip code="85721"> Tucson,  AZ </zip>
165   <zip code="85001"> Phoenix, AZ </zip>
166 ...
```

Now suppose the user were to change the `code` for Tucson to be the same as Phoenix (violating the conventional schema's identity constraint), and then back again (see Listings 141 and 142).

Listing 141: Slice 2 (invalid).

```
163 ...
164   <zip code="85001"> Tucson,  AZ </zip>
165   <zip code="85001"> Phoenix, AZ </zip>
166 ...
```

Listing 142: Slice 3 (valid).

```
163 ...
164   <zip code="85721"> Tucson,  AZ </zip>
165   <zip code="85001"> Phoenix, AZ </zip>
166 ...
```

Assuming that the physical annotations place the timestamps at the `<zip>` element level, the above actions would create an item-based representation similar to the one shown in Listing 143.

Listing 143: Squashed version of the three slices.

```
89  ...
90    <zip_RepItem>
91      <zip_Version begin="1" end="1">
92        <zip code="85721"> Tucson, AZ </zip>
93      </zip_Version>
94      <zip_Version begin="2" end="2">
95        <zip code="85001"> Tucson, AZ </zip>
96      </zip_Version>
97      <zip_Version begin="3">
98        <zip code="85721"> Tucson, AZ </zip>
99      </zip_Version>
100   </zip_RepItem>
101
102   <zip_RepItem>
103     <zip_Version begin="1">
104       <zip code="85001"> Phoenix, AZ </zip>
105     </zip_Version>
106   </zip_RepItem>
107  ...
```

With this representation, there is no way to create an identity constraint in XML Schema that can detect that both `code` values at time 2 are the same. If the constraint were constructed to restrict all

`./zip_RepItem/zip_Version/zip/@code` values[10] to be unique, this would fail since at times 1 and 3, Tucson has a `code` value of 85701, and this is legal in our temporal constraint. If the constraint were constructed to require all `./zip_Version/zip/@code` within each `./zip_RepItem` to be identical, this would also fail since the user is allowed to change the zip code from slice to slice.

One could imagine extending the constraint shown in Listing 139 to include key specification fields `begin` and `end` for the valid times associated with each version of the `zip` element, as shown in Listing 144 on lines 29 and 30, with the corresponding attributes in the element specification.

Listing 144: XML Schema `<unique>` with additional fields.

```
25 ...
26    <xs:unique name="zipUniqueAttempt">
27      <xs:selector xpath="zip"/>
28      <xs:field xpath="@code"/>
29      <xs:field xpath="@begin"/>
30      <xs:field xpath="@end"/>
31    </xs:unique>
32 ...
```

As long as the `begin` and `end` attributes are maintained in proper order (which can be checked by τXMLLINT), the keys will uniquely identify each key within each snapshot. Listing 145 below shows an example where the addition of such attributes will achieve the desired functionality. Here, the conventional validator would detect that the `zip` elements on lines 95 and 104 are in violation of the unique constraint, which is indeed correct.

However, this approach will not succeed in the general case because it only enforces uniqueness at the interval end points and not anywhere within the interval. For example, consider the excerpt from a squashed document shown in Listing 146. We see that the elements on lines 95 and 101 conflict our desired constraint, but since the `begin` attributes are distinct, XML does not detect an error.

Listing 146: Squashed document with multiple changes

```
89 ...
90   <zip_RepItem>
91     <zip_Version begin="1" end="1">
92       <zip code="85721" begin="1" end="1"> Tucson, AZ </zip>
93     </zip_Version>
94     <zip_Version begin="2" end="2">
95       <zip code="85001" begin="2" end="2"> Tucson, AZ </zip>
96     </zip_Version>
97   </zip_RepItem>
98
99   <zip_RepItem>
100    <zip_Version begin="1" end="2">
101      <zip code="85001" begin="1" end="2"> Phoenix, AZ </zip>
102    </zip_Version>
103  </zip_RepItem>
104  ...
```

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

Referential Integrity Constraints We claim that referential integrity constraints *cannot* be implemented in a representational schema. The argument is similar to that for identity constraints: there is no way to create a constraint in XML Schema that can both satisfy referential integrity and time issues. Consider the example shown in Listings 147 and 148.

[10] This is XPath code [77].

Listing 147: A referential constraint.

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

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

Listing 148: Squashed document.

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

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

Here, we have a constraint that says *"A city element's state attribute must match an existing state element's id attribute at every point in time.*" The squashed document shows that this constraint is satisfied at times 1 and 3, but violated at time 2 since Arizona will point to a non-existent state `id`. To construct an XML Schema that could describe this situation, one would need to be able to somehow discriminate between different `<regions_Version>` elements according to their `begin` and `end` attributes, but there is no such way to accomplish this without the help from a procedural language like XQuery [79]. We are again forced to conclude the XML schema lacks sufficient mechanisms to enforce a referential constraint.

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

Listings 149 and 150 below show an example constraint: "*The element* `<supplier>` *can occur exactly 1 or 2 times.*" We first assume that the physical timestamps—specified in the temporal schema—are placed at a predecessor or successor element of the `<supplier>` element. In this case the specification of the `<supplier>` element requires no modification in the representational schema.

Listing 149: Conventional schema 1.

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

Listing 150: Representational schema 1.

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

Listings 151 and 152 show the same example as above, except now the physical timestamps are located at the level of the `<supplier>` element. In this case, the transformation pushes the constraints up to the `<supplier_RepItem>` element.

Listing 151: Conventional schema 2.

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

Listing 152: Representational schema 2.

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

Datatype Constraints We claim that datatype definitions of both elements and attributes *can* be enforced in the representational schema. This can be achieved by copying the datatype definition for each element in the original schema into the representational schema. Since datatype restrictions are not affected by the location of timestamps, the transformation is trivial in all cases. See Listings 153 and 154 for an example of the datatype constraint: "*The element* `<age>` *must have a value between 0 and 100, inclusive, at all times.*" No changes to the constraint must be made in the transformation.

Listing 153: Datatype conventional schema.

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

Listing 154: Datatype rep. schema.

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

18.6.3 Non-sequenced Constraints

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

We claim that in general it is *not* possible to enforce non-sequenced constraints within a representational schema. Since non-sequenced constraints can reference arbitrary sections of time that don't necessarily cor-

respond to slice lifetimes or schema change (*schema wall*) boundaries, it is impossible to use XML Schema to isolate and thus validate these sections. For example, consider the simple non-sequenced cardinality constraint: "*There should be two or three unique suppliers in any given year.*" If the document were changed at intervals that were less than one year in duration, we could have a representation that looked similar to Listing 155.

Listing 155: Squashed version. One day equals one unit of time.

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

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

However, we do note that there exist specific circumstances in which non-sequenced constraints *may* be validated. Again consider the non-sequenced cardinality constraint: "*There should be 2 or 3 unique suppliers in any given year.*" Also suppose that the timestamps were placed at some element above the `<supplier>` element and that slices were created exactly once per year. The result will be a representation that closely mimics the individual slices. We see that it is possible to create a representational schema to enforce this constraint (Listings 156 and 157).

Listing 156: Item-based temporal representation #1.

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

Listing 157: Non-sequenced representational schema #1.

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

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

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

Listing 158: Item-based temporal representation #2.

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

Listing 159: Non-sequenced representational schema #2.

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

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

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

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

18.6.4 Functionality of Other Representation Classes

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

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

18.6.5 Placement of Functionality

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

Consider the model of validation used by τXMLLINT shown in Figure 63. First, the temporal document is validated against the representational schema using a conventional validator (i.e., XMLLINT). Then the *Temporal Constraint Validator Module* is invoked to explicitly and exhaustively check all temporal constraints. This module uses DOM to parse and traverse each slice and manually checks each constraint present in the logical annotation set. From the description of these steps we draw two simple observations. First, the conventional validator is always invoked on the temporal document, no matter which constraints are being implemented in the representational schema. Second, temporal constraints which are "hard" to implement are done so using DOM. Thus, since the conventional validator is empirically much faster than DOM, and is being invoked anyway, we argue that all constraints, when possible, should be implemented within the representational schema. This will provide much better performance in terms of time required, and as we have shown in the previous sections, will not greatly increase the complexity of the representational schema. Furthermore, SCHEMAMAPPER will not require extensive modifications in order to create a schema that can enforce these constraints, since the transformation is trivial in most cases and relatively simple in the rest.

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

18.7 Evaluation of Representation Classes

This section presents a detailed empirical analysis of each representation class in a variety of scenarios. The goal is to determine how each class performs with respect to four metrics: size of representation, time to construct the representation, time to validate the representation, and time to reconstruct an arbitrary slice. We first describe the motivation for this evaluation and present the methodology used in our experiments. We then analyze the results of the experiments and we conclude with general observations and recommendations.

18.7.1 Motivation

When choosing a representation for a temporal XML document, we consider several characteristics in the decision making process. Consider the following examples.

- A user wants to transmit a document across the country on the internet. Here, the *size of the representation* is the most important feature.
- A user makes frequent updates to a file, resulting in frequent creations of the representation. Here, the *time taken to create the representation* will be the most important feature.
- A user wants to frequently select different versions of a document, an operation called temporal slicing. Here, the *time taken to extract the original documents* is the most important feature.
- A user makes frequent updates to a file and must always validate the document to ensure correctness. Here, both the *time taken to create* and *validate the representation* are the most important features.

It is not clear whether any single representation class can best meet the needs of every user in every scenario. Our aim is to quantify the features of each class so that informed decisions can be made by the user, taking into account their particular needs.

18.7.2 Methodology

Given the above motivation, we will address the following questions about each representation class. Is the size of the representation linear in the number and size of slices, or does it provide some level of compression? Does the overhead of the representation result in a large amount of time required to squash and unsquash? Can we validate the representation quickly enough to allow practical use?

To answer these questions, we first extended the τXSchema tools to support each representation type[11] and then ran a set of experiments to test and evaluate each representation. In these experiments, we were interested in how the representation would respond to several independent variables: the amount of change from slice to slice, the types of changes within each slice, the number of slices in the system, and the size of each individual slice. To quantify the changes, we measured several dependent variables: the size of the resulting representation, the time taken to create the representation from the original documents, the time taken to extract the original documents from the representation, and the time taken to validate the representation against the original schema(s) and temporal constraints. Tables 6 and 7 summarize the experiment.

Independent Variables		
Name	Expressed as	Values
Slice Size	Number of elements	$10, 20, 40, 80, ..., 2000$
Number of Slices	Number of files	$10, 20, 40, 80, ..., 2000$
Amount of Change	Percentage of changed elements	$0, 2, 4, 8, 16, 32, 64$
Type of Change	Percent value change vs. new element	$(0, 100), (25, 75), (50, 50), (75, 25), (0, 100)$

Table 6: The independent variables considered in the experiments.

Dependent Variables	
Name	Measured by
Representation size	Kilobytes on disk of the representation
Time taken to squash files	Seconds of execution
Time taken to unsquash files	Seconds of execution
Time taken to validate files	Seconds of execution

Table 7: The dependent variables measured in the experiments.

We have created tools[12] to help build temporal cases dynamically based on the four experiment parameters and to automate the run process and output the results. The experiments were executed on a machine running Ubuntu 8.10 with a 2.83 GHz Intel Core2 quad-core processor and 8 GB main memory. The testing scripts were created in Perl and the data was analyzed in Matlab.

Note that all of the following experiments were run with the temporal constraint functionality placed at the tool level as opposed to the representational schema level as described in Section 18.6. Also note that schema-versioning is not considered in the following examples due to the complexity it adds to the creation of large, random scenarios. However, we believe that these results will provide a good initial understanding of the behavior of each representation class since each schema change results in a new wall, with the representation structure remaining the same within each wall. Thus, a schema change will have the same effect across all representation types.

[11] Due to time and tool constraints, we could not evaluate the reference-based representation. However, it is believed that this representation would perform similarly to the item-based representation under typical circumstances.

[12] See Appendix B for details.

We note here that the execution times presented in the following sections contain both I/O and execution times—the entire execution time of the process. Since caching was not disabled during the experiments, it is possible that some experiments unfairly report smaller execution times; this might happen if an experiment involved reading a file that was already in the O/S cache.

18.7.3 Initial Sensitivity to Parameters

Initial experiments were run to test the sensitivity of each representation type to the variables that controlled the amount and type of change in each slice. In particular, we performed runs with the amount of change set to 1%, 2%, 4%, ..., 64% and with the type of change set to (0% new version, 100% new item), (25%, 75%), (50%, 50%), (75%, 25%), and (100%, 0%). We also varied the size of each slice (values of 10 and 100) and number of slices (values of 10 and 200). For each combination of parameters, we ran 30 repetitions for each representation class and took the minimum result (since we are interested in the performance of the tools in isolation, but the experiments were conducted on a time-shared machine with background processes running). Interestingly, the results showed that all representation classes were relatively insensitive to the change-related parameters. This is likely because for each representation class, the amount of computation overhead for each additional change per file is small when compared to the execution of the entire tool (e.g., file I/O for each slice, DOM parsing of each file, etc). Appendix C provides the detailed results of these executions.

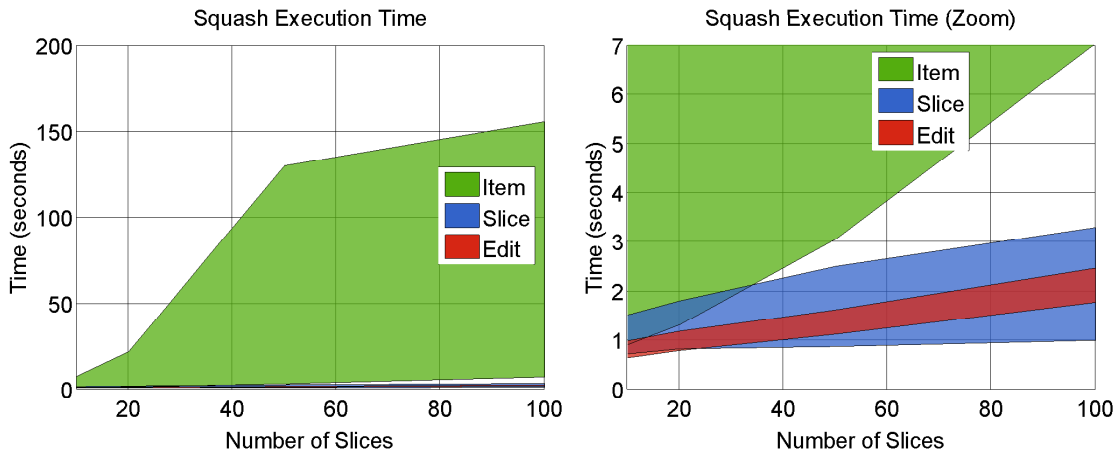
As a consequence of these results, for each of our experiments described in the following sections, we held the amount of change constant at 32% and the type of change constant at (75%, 25%), which represent intermediate values for each.

18.7.4 SQUASH Results

We first consider the amount of time required for each representation class to squash a temporal document. Figure 64 displays the results with all classes on the same plot (with a band stretching across the document size parameter for each class) while Figure 65(a)–65(c) shows each representation class individuallly. We immediately see that the item-based scheme is particularly sensitive to the parameters and even modest increases result in a large increase in time. We now briefly investigate why this happens.

Figures 66(a) and 66(b) show the execution traces of the SQUASH algorithm for the item-based and edit-based representation classes, respectively. Note that in each case, only the steps that required a significant portion of the execution time are shown. Table 8 summarizes the actual exectution time for each step in both classes for three simple scenarios. We immediately see that for the item-based representation, the bulk of the execution time is being spent on task 1.2.2 (physical to temporal conversion), specifically in the push down operation. The push down operation recursively calls itself to “push down” the items from the root node down to the `<part>` elements; this algorithm involves merging similar versions of an item into a single version. In these executions, the push down operation is called a total of 342, 642, and 1282 times for the three input sets, respectively. The edit-based scheme does not have the concept of items, and thus avoids the penalty of the push down operation. Further, the `diff -e` call that the edit-based scheme employs between each slice is relatively inexpensive: on the order of 0.001 seconds to compare two files with 20 elements each, 0.03 seconds to compare files with 1000 elements each, and 0.2 seconds to compare files with 10,000 elements each, and 1.65 seconds to compare files with 100,000 elements each.

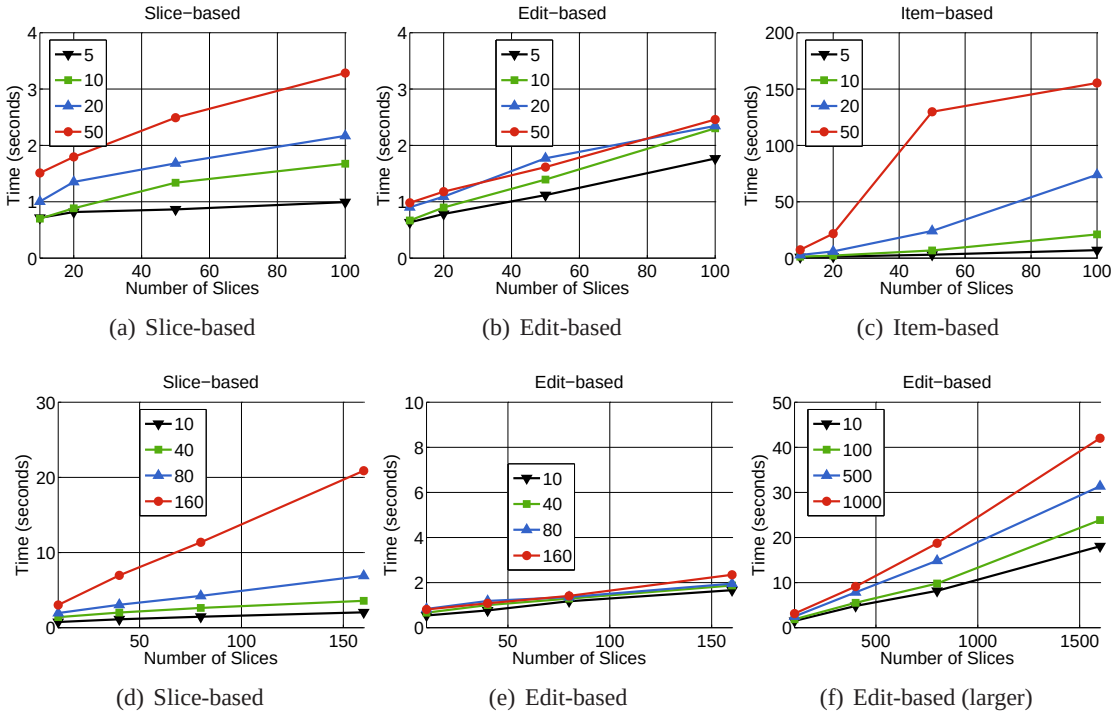
In order to further investigate the time required for the slice-based and edit-based schemes under heavier conditions we increased the magnitude of the parameters and the results are shown in Figures 65(d)–65(f). We see again that the edit-based scheme has better performance when compared to the slice-based scheme under the same parameters. Figure 65(f) shows the parameter set that first starts to stress the performance



(a) Full view. The item-based class requires orders of magnitude more time to squash than the slice-based and edit-based classes.

(b) Zoom view of the slice-based and edit-based classes.

Figure 64: Time required to squash a temporal document. The three band colors correspond to the different representation types. Each band stretches across $\{5, 10, 20, 50\}$ elements per slice.



(a) Slice-based

(b) Edit-based

(c) Item-based

(d) Slice-based

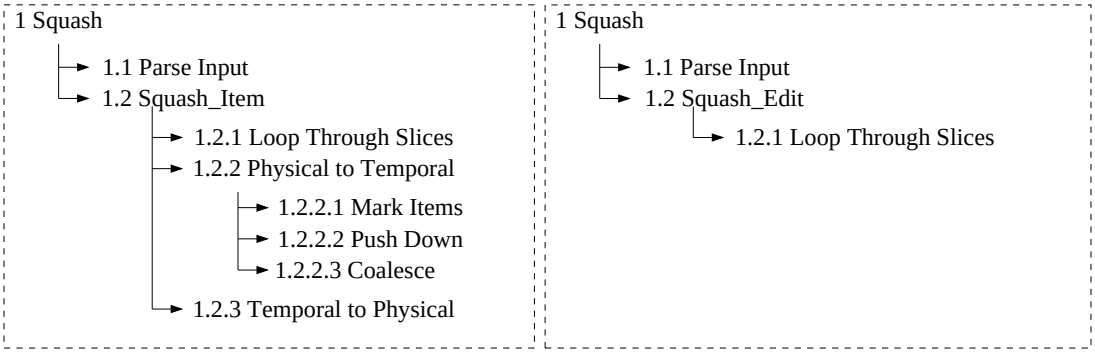
(e) Edit-based

(f) Edit-based (larger)

Figure 65: Time required to squash a temporal document. Here, the lines correspond to different document sizes, shown in number of elements.

of the edit-based scheme (i.e., the parameters that first cause the execution time to show larger than linear growth); these parameters are roughly 10x the parameters that stressed the slice-based scheme.

Figures 67(a)–67(c) show the size on disk of the resulting temporal document. As expected, the slice-

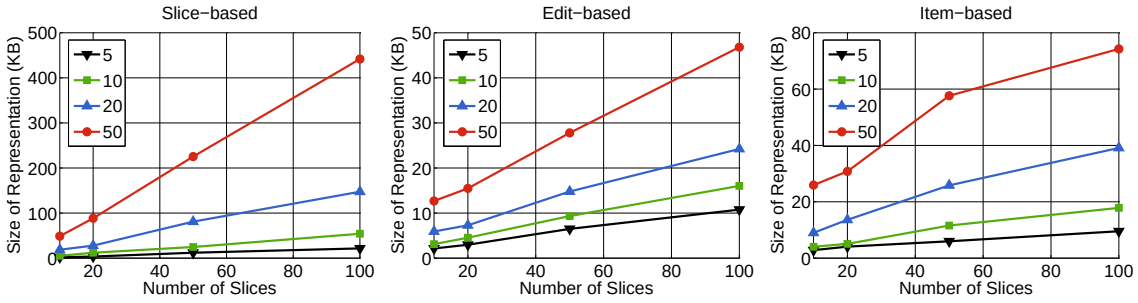


(a) Item-based (b) Edit-based

Figure 66: The main methods (in terms of time) entered during the execution of SQUASH.

Step #	Item-based			Edit-based			Slice-based		
	(10, 10)	(10, 20)	(20, 20)	(10, 10)	(10, 20)	(20, 20)	(10, 10)	(10, 20)	(20, 20)
1	2.15	3.99	9.59	0.85	0.99	1.09	0.91	1.13	1.70
1.1	0.02	0.01	0.02	0.02	0.02	0.01	0.01	0.01	0.01
1.2	1.63	3.47	9.04	0.33	0.46	0.56	0.39	0.59	1.15
1.2.1	0.06	0.10	0.14	0.28	0.43	0.51	0.07	0.10	1.14
1.2.2	1.43	3.21	8.46	-	-	-	0.19	0.24	0.14
1.2.2.1	0.57	1.06	2.38	-	-	-	0.01	0.02	0.04
1.2.2.2	0.80	2.08	5.67	-	-	-	0.03	0.05	0.05
1.2.2.3	0.04	0.05	0.09	-	-	-	0.14	0.17	0.19
1.2.3	0.10	0.11	0.39	-	-	-	0.08	0.18	0.61

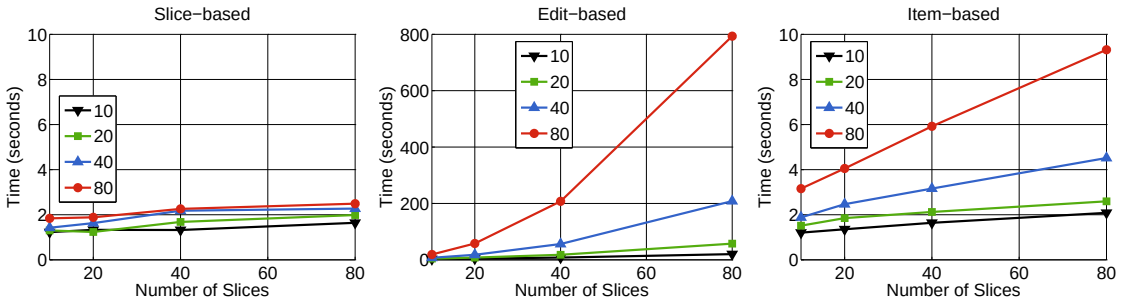
Table 8: The execution times (in seconds) in SQUASH for each task, broken up by representation type and shown for three different input sets. In these runs, the amount of change was set to .32 and the type of change was set to $(75\%, 25\%)$.



(a) Slice-based (b) Edit-based (c) Item-based

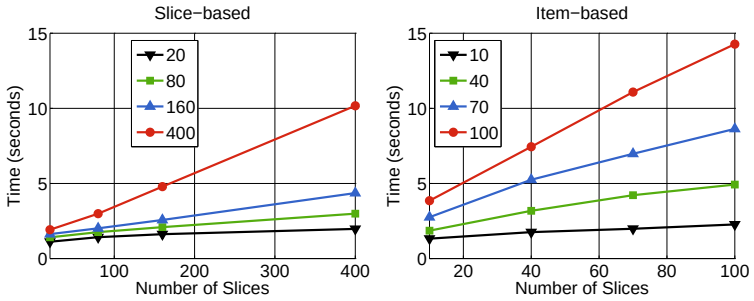
Figure 67: Size of the resulting temporal document. Note the different scales on the y-axis.

based scheme grows linearly with the number of slices and the size of the representation; this is because this scheme keeps the entire unmodified slice in the representation. However, the edit- and item-based schemes are able to provide a large amount of compression and keep the file size relatively low.



(a) Slice-based (b) Edit-based (c) Item-based

Figure 68: Time required to validate the temporal document. Note the different scales on the time axis; the edit-based scheme takes orders of magnitude longer.



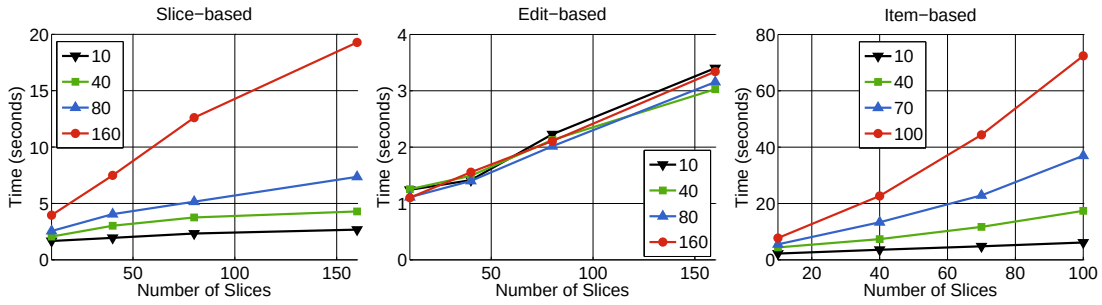
(a) Slice-based (b) Item-based

Figure 69: Time required to validate the temporal document. Note the different scales on the x-axis. The slice-based scheme can handle roughly four times the number of slices within the same time period.

18.7.5 τXMLLINT Results

Here we considered the validation time required for a single temporal cardinality constraint. Figures 68(a)–68(c) show the amount of time required to validate scenarios with modest number of slices and size for each slice. We see that while the slice-based and item-based schemes can handle these parameter ranges with relative ease, the edit-based scheme takes orders of magnitude more time. This is due to the nature of the current implementation of the edit-based scheme: the temporal document is first unsquashed into a series of slices; then these slices are squashed into an item-based representation with the timestamp at the root; finally, the item-based representation is validated in the normal way. Although this implementation benefits from the reuse of several existing software modules and is logically correct, it suffers from both high overhead and the bad performance of the item-based squashing module. These factors add up to significant values, with the majority of the time coming again from the push down operation of the item-based scheme (70% of the total execution time).

Figure 69 shows the slice-based and item-based schemes under heavier conditions to illustrate the magnitude of the parameters that first cause a noticable increase in execution time. We see that the slice-based scheme can handle roughly four times the number of slices as the item-based scheme within the same time period. This is because the slice-based scheme requires no preprocessing before validation can begin, while the item-based scheme must undergo a number of operations to be in the correct form for validation.



(a) Slice-based (b) Edit-based (c) Item-based

Figure 70: The amount of time required to extract all slices from a temporal document. Note the different x and y axes.

18.7.6 UNSQUASH Results

Figure 70 shows the amount of time required to extract all slices from the temporal document. We see that, like the SQUASH results, the edit-based scheme has the best performance, while the item-based scheme requires significantly more time than the other two schemes. An execution analysis similar to that for SQUASH was performed and the same conclusions were reached. In particular, during the unsquash operation, the item-based scheme must perform the opposite of merging items and pushing down timestamps: it must push up timestamps and unmerge items. These operations result in a huge number of recursive calls and loops for every item. In contrast, the edit-based scheme only needs to run the `patch` command for each edit script. The slice-based scheme must simply loop though the temporal document and extract each slice with a copy-and-paste-like method; no preprocessing is needed. Even still, it does not perform as well as the edit-based scheme.

18.7.7 Representation Conclusions and Recommendations

The above results show that, as anticipated, no representation scheme provides the best performance under all conditions. In particular, while the edit-based scheme shows the fastest time to squash and unsquash as well as the smallest representation size, it suffers from tremendous overhead during validation[13]. On the other hand, the slice-based scheme can be squashed, unsquashed, and validated quickly but the resulting representation is very large. Further, the item-based scheme results in a smaller representation and can be validated quickly, but suffers from a large amount of time to squash and unsquash. Table 9 summarizes the findings.

Our current recommendation would be to use the edit-based scheme for all activities that do not require temporal validation (although under this assumption, a conventional validator may still make sense to validate the last instance and the one representation in its entirety), and to use the slice-based scheme in all cases that do require temporal validation. However, if improvements can be made to the item-based squashing and unsquashing methodologies (i.e., the "push up" and "push down" operations) in terms of execution time, then the item-based scheme would become most attractive in all scenarios. In this case, additional analysis would be required to study the effects of using an item-based scheme in its original form versus a hybrid between the item-based and the edit-based. For example, one could imagine storing the representation on

[13]It should be noted that although the current implementation of edit-based validation is not very efficient, it would be difficult to find any implementation that would be. The problem lies in the inability to enforce complex temporal constraints by examining the edit scripts alone. This implies that the first step during validation would always have to be reconstructing the original slices. Only then could the validation process—whatever that may be—begin.

Representation	SQUASH Time		SQUASH Size		UNSQUASH Time		τXMLLINT Time	
	Rank	Ratio	Rank	Ratio	Rank	Ratio	Rank	Ratio
Slice-based	2	1.1	3	3.9	2	2.6	1	-
Edit-based	1	-	1	-	1	-	3	41.1
Item-based	3	15.7	2	1.5	3	13.0	2	1.6

Table 9: The overall results of the analysis. The *Rank* columns indicate the performance of this representation when compared to the other two (e.g., a rank of 2 means it was the second best). The *Ratio* column indicates how much worse this representation performed compared to the top ranking representation, measured as the average ratio between the two representations.

disk using the edit-based method, while converting into the item-based scheme for all in-memory operations (e.g, adding or extracting slices, enforcing temporal constraints). Further, one might consider using a mix of representation types for different parts of the timeline. For example, a user could use the slice-based scheme for the five most recent slices, the item-based scheme for the next 300 slices, and the edit-based scheme for all other slices. This might allow efficient extraction of recent files, efficient validation for all files in the recent past, and efficient storage for files not likely to be queried often. The time and space tradeoffs for these options are left for future work.

Listing 137: Representational schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema"
  xmlns="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema"
  xmlns:rep0="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema0"
  xmlns:rep1="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema1"
  xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema">

<xsd:import namespace="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema"
            schemaLocation="TVSchema.xsd" />
<xsd:import namespace="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema0"
            schemaLocation="rep0.xsd" />
<xsd:import namespace="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema1"
            schemaLocation="rep1.xsd" />

<xsd:element name="sv_root">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="schemaItem">
        <xsd:complexType>
          <xsd:sequence>

            <xsd:element maxOccurs="1" minOccurs="1" name="schemaVersion0">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element maxOccurs="1" minOccurs="1" ref="tv:timestamp" />
                  <xsd:element maxOccurs="1" minOccurs="1" ref="rep0:tv_root" />
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>

            <xsd:element maxOccurs="1" minOccurs="1" name="schemaVersion1">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element maxOccurs="1" minOccurs="1" ref="tv:timestamp" />
                  <xsd:element maxOccurs="1" minOccurs="1" ref="rep1:tv_root" />
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>

          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
        <xsd:attribute name="temporalSchema" type="xsd:string">
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Listing 138: Temporal document.

```
<?xml version="1.0" encoding="UTF-8"?>
<sv_root xmlns="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema"
  xmlns:rep0="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema0"
  xmlns:rep1="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema1"
  xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema">

<schemaItem>
  <schemaVersion0>
    <tv:timestamp begin="2002-01-01" end="2005-01-01" />
        <rep0:tv_root>
          <rep0:athlete>
            ...
          </rep0:athlete>
        </rep0:tv_root>
  </schemaVersion0>
  <schemaVersion1>
    <tv:timestamp begin="2002-01-01" end="2005-01-01" />
        <rep1:tv_root>
          <rep1:athlete>
            ...
          </rep1:athlete>
        </rep1:tv_root>
  </schemaVersion1>
 </schemaItem>

</sv_root>
```

Listing 145: Squashed document with multiple changes

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

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

19 Example Schema and Instance Documents

19.1 Conventional Schemas

Listing 160: Conventional schema on 1 January 2002.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xsd:element name="winOlympic">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element ref="country" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="country">
    <xsd:complexType mixed="false">
      <xsd:sequence>
        <xsd:element ref="athleteTeam"/>
      </xsd:sequence>
      <xsd:attribute name="countryName" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athleteTeam">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element name="teamName" minOccurs="1" maxOccurs="1" type="xsd:string"/>
        <xsd:element ref="athlete" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="numAthletes" type="xsd:positiveInteger" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athlete">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element name="athName" type="xsd:string"/>
        <xsd:element name="phone" type="phoneNumType" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:simpleType name="phoneNumType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="12"/>
      <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 161: Conventional schema on 1 January 2005.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xsd:element name="winOlympic">
    <xsd:complexType mixed="true">
      <xsd:sequence>
      <!--numEvents added on Wednesday-->
        <xsd:element name="numEvents" type="xsd:nonNegativeInteger"/>
        <xsd:element ref="country" minOccurs="0" maxOccurs="unbounded"/>
```

```
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="country">
    <xsd:complexType mixed="false">
      <xsd:sequence>
        <xsd:element ref="athleteTeam"/>
      </xsd:sequence>
      <xsd:attribute name="countryName" type="xsd:string" use="required"/>
      <xsd:attribute name="countryLead" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athleteTeam">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element name="teamName" minOccurs="1" maxOccurs="1" type="xsd:string"/>
        <xsd:element ref="athlete" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="numAthletes" type="xsd:positiveInteger" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athlete">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element name="athName" type="xsd:string"/>
        <xsd:element name="phone" type="phoneNumType" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:simpleType name="phoneNumType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="12"/>
      <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

19.2 Annotations

Listing 162: Annotation document on 1 January 2002.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationSet xmlns="http://www.cs.arizona.edu/tau/tauXSchema/PXSchema">

  <physical>
    <stamp target="/winOlympic">
      <stampKind timeDimension="transactionTime" stampBounds="extent"/>
    </stamp>

    <stamp target="/winOlympic/country">
      <stampKind timeDimension="transactionTime" stampBounds="extent"/>
    </stamp>

    <stamp target="/winOlympic/country/athleteTeam">
      <stampKind timeDimension="transactionTime" stampBounds="extent"/>
    </stamp>

    <stamp target="/winOlympic/country/athleteTeam/athlete">
      <stampKind timeDimension="transactionTime" stampBounds="extent"/>
    </stamp>
  </physical>

  <logical>
    <item target="/winOlympic">
    <transactionTime content="varying" existence="constant"/>
```

```
      <itemIdentifier name="olympicId1" timeDimension="transactionTime">
        <field path="./text"/>
      </itemIdentifier>
    </item>

    <item target="/winOlympic/country">
    <transactionTime content="varying" existence="varyingWithGaps"/>
      <itemIdentifier name="countryId1" timeDimension="transactionTime">
        <field path="./@countryName"/>
      </itemIdentifier>
    </item>

    <item target="/winOlympic/country/athleteTeam">
    <transactionTime content="varying" existence="varyingWithGaps"/>
      <itemIdentifier name="teamName" timeDimension="transactionTime">
        <field path="./teamName/text"/>
      </itemIdentifier>
    </item>

    <item target="/winOlympic/country/athleteTeam/athlete">
      <transactionTime content="varying" existence="varyingWithGaps"/>
      <itemIdentifier name="atheleteId1" timeDimension="transactionTime">
        <field path="./athName/text"/>
      </itemIdentifier>
    </item>
  </logical>

</annotationSet>
```

Listing 163: Annotation document on 1 January 2005.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationSet xmlns="http://www.cs.arizona.edu/tau/tauXSchema/PXSchema">

  <physical>
    <stamp target="/winOlympic">
      <stampKind timeDimension="transactionTime" stampBounds="extent"/>
    </stamp>

    <stamp target="/winOlympic/country">
      <stampKind timeDimension="transactionTime" stampBounds="extent"/>
    </stamp>

    <stamp target="/winOlympic/country/athleteTeam">
      <stampKind timeDimension="transactionTime" stampBounds="extent"/>
    </stamp>

    <stamp target="/winOlympic/country/athleteTeam/athlete">
      <stampKind timeDimension="transactionTime" stampBounds="extent"/>
    </stamp>
  </physical>

  <logical>
    <item target="/winOlympic">
    <transactionTime content="varying" existence="constant"/>
      <itemIdentifier name="olympicId1" timeDimension="transactionTime">
        <field path="./text"/>
      </itemIdentifier>
    </item>

    <item target="/winOlympic/country">
    <transactionTime content="varying" existence="varyingWithGaps"/>
      <itemIdentifier name="countryId1" timeDimension="transactionTime">
        <field path="./@countryName"/>
        <field path="./@countryLead"/>
      </itemIdentifier>
    </item>

```

```
    <item target="/winOlympic/country/athleteTeam">
    <transactionTime content="varying" existence="varyingWithGaps"/>
      <itemIdentifier name="teamName" timeDimension="transactionTime">
        <field path="./teamName/text"/>
      </itemIdentifier>
    </item>

    <item target="/winOlympic/country/athleteTeam/athlete">
      <transactionTime content="varying" existence="varyingWithGaps"/>
      <itemIdentifier name="atheleteId1" timeDimension="transactionTime">
        <field path="./athName/text"/>
      </itemIdentifier>
    </item>
  </logical>

</annotationSet>
```

19.3 Conventional Documents

Listing 164: Conventional document on 1 January 2002.

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="schema1.xsd">
  There are
  events in the Olympics.
  <country countryName="Norway">
    <athleteTeam numAthletes="95">
      <teamName>Norway_Army</teamName>
      Athletes will take part in various events. The athletes participating are listed below
      <athlete>
        <athName>
            Kjetil Andre Aamodt
        </athName>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His phone numbers are:
        <phone>123-402-0340</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

Listing 165: Conventional document on 1 January 2003.

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="winOlympic.ver1.xsd">
  There are
  events in the Olympics.
  <country countryName="Norway">
    <athleteTeam numAthletes="95">
      <teamName>Norway_Army</teamName>
      Athletes will take part in various events. The athletes participating are listed below
      <athlete>
        <athName>
```

```
            Kjetil Andre Aamodt
        </athName>
      </athlete>
      <athlete>
        <athName>
            Andre Agassi
        </athName>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His phone numbers are:
        <phone>123-402-0340</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

Listing 166: Conventional document on 1 January 2005.

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="schema2.xsd">
  There are
  <numEvents>11</numEvents>
  events in the Olympics.
  <country countryName="Norway" countryLead="Andre Agassi">
    <athleteTeam numAthletes="95">
      <teamName>Norway_Army</teamName>
      Athletes will take part in various events. The athletes participating are listed below
      <athlete>
        <athName>
            Kjetil Andre Aamodt
        </athName>
      </athlete>
      <athlete>
        <athName>
            Andre Agassi
        </athName>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His phone numbers are:
        <phone>123-402-0340</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

Listing 167: Conventional document on 1 January 2006.

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="schema2.xsd">
  There are
  <numEvents>11</numEvents>
  events in the Olympics.
  <country countryName="Norway" countryLead="Andre Agassi">
    <athleteTeam numAthletes="95">
      <teamName>Norway Army</teamName>
      Athletes will take part in various events. The athletes participating are listed below
      <athlete>
        <athName>
            Kjetil Andre Aamodt
        </athName>
      </athlete>
      <athlete>
        <athName>
            Andre Agassi
        </athName>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His phone numbers are:
        <phone>123-402-0340</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

19.4 Temporal Schema

Listing 168: Temporal Schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<temporalSchema xmlns="http://www.cs.arizona.edu/tau/tauXSchema/TS">

  <conventionalSchema>
    <sliceSequence>
      <slice location="schema1.xsd" begin="2002-01-01" />
      <slice location="schema2.xsd" begin="2005-01-01" />
    </sliceSequence>
  </conventionalSchema>

  <annotationSet>
    <sliceSequence>
      <slice location="annotations1.xml" begin="2002-01-01" />
      <slice location="annotations2.xml" begin="2005-01-01" />
    </sliceSequence>
  </annotationSet>

</temporalSchema>
```

19.5 Representational Schemas

Listing 169: Representational schema for 2002-01-01 to 2005-01-01.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema attributeFormDefault="unqualified"
  elementFormDefault="unqualified"
  targetNamespace="http://www.cs.arizona.edu/tau/RepSchema0"
  xmlns="http://www.cs.arizona.edu/tau/RepSchema0"
  xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsd:import namespace="http://www.cs.arizona.edu/tau/TVSchema" schemaLocation="TVSchema.xsd" />
  <xsd:simpleType name="phoneNumType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="12" />
      <xsd:pattern value="\d{3}-\d{3}-\d{4}" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="tv_root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="winOlympic_RepItem" />
      </xsd:sequence>
      <xsd:attribute name="begin" type="xsd:date" />
      <xsd:attribute name="end" type="xsd:date" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athleteTeam_RepItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="athleteTeam_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="athleteTeam">
                <xsd:complexType mixed="true">
                  <xsd:sequence>
                    <xsd:element maxOccurs="1"
                      minOccurs="1" name="teamName" type="xsd:string" />
                    <xsd:element
                      maxOccurs="unbounded" ref="athlete_RepItem" />
                  </xsd:sequence>
                  <xsd:attribute name="numAthletes"
                    type="xsd:positiveInteger" use="optional" />
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="isItem" type="xsd:string" />
      <xsd:attribute name="originalElement" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="country_RepItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="country_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="country">
                <xsd:complexType mixed="false">
                  <xsd:sequence>
                    <xsd:element
```

```
                    ref="athleteTeam_RepItem" />
                </xsd:sequence>
                <xsd:attribute name="countryName"
                  type="xsd:string" use="required" />
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="isItem" type="xsd:string" />
    <xsd:attribute name="originalElement" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="winOlympic_RepItem">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="1"
        name="winOlympic_Version">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="tv:timestamp_TransExtent" />
            <xsd:element name="winOlympic">
              <xsd:annotation>
                <xsd:documentation>
                  Schema for recording non
                  temporal country information
                </xsd:documentation>
              </xsd:annotation>
              <xsd:complexType mixed="true">
                <xsd:sequence>
                  <xsd:element
                    maxOccurs="unbounded" minOccurs="0" ref="country_RepItem" />
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="isItem" type="xsd:string" />
    <xsd:attribute name="originalElement" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="athlete_RepItem">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="1"
        name="athlete_Version">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="tv:timestamp_TransExtent" />
            <xsd:element name="athlete">
              <xsd:complexType mixed="true">
                <xsd:sequence>
                  <xsd:element name="athName"
                    type="xsd:string" />
                  <xsd:element
                    maxOccurs="unbounded" minOccurs="0" name="phone"
                    type="phoneNumType" />
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="isItem" type="xsd:string" />
```

```
      <xsd:attribute name="originalElement" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 170: Representational schema for 2002-01-01 to 2005-01-01.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema attributeFormDefault="unqualified"
  elementFormDefault="unqualified"
  targetNamespace="http://www.cs.arizona.edu/tau/RepSchema1"
  xmlns="http://www.cs.arizona.edu/tau/RepSchema1"
  xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsd:import namespace="http://www.cs.arizona.edu/tau/TVSchema" schemaLocation="TVSchema.xsd" />
  <xsd:simpleType name="phoneNumType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="12" />
      <xsd:pattern value="\d{3}-\d{3}-\d{4}" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="tv_root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="winOlympic_RepItem" />
      </xsd:sequence>
      <xsd:attribute name="begin" type="xsd:date" />
      <xsd:attribute name="end" type="xsd:date" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athleteTeam_RepItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="athleteTeam_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="athleteTeam">
                <xsd:complexType mixed="true">
                  <xsd:sequence>
                    <xsd:element maxOccurs="1"
                      minOccurs="1" name="teamName" type="xsd:string" />
                    <xsd:element
                      maxOccurs="unbounded" ref="athlete_RepItem" />
                  </xsd:sequence>
                  <xsd:attribute name="numAthletes"
                    type="xsd:positiveInteger" use="optional" />
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="isItem" type="xsd:string" />
      <xsd:attribute name="originalElement" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="country_RepItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="country_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="country">
```

```
                <xsd:complexType mixed="false">
                  <xsd:sequence>
                    <xsd:element
                      ref="athleteTeam_RepItem" />
                  </xsd:sequence>
                  <xsd:attribute name="countryName"
                    type="xsd:string" use="required" />
                  <xsd:attribute name="countryLead"
                    type="xsd:string" use="required" />
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="isItem" type="xsd:string" />
      <xsd:attribute name="originalElement" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="winOlympic_RepItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="winOlympic_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="winOlympic">
                <xsd:annotation>
                  <xsd:documentation>
                    Schema for recording non
                    temporal country information
                  </xsd:documentation>
                </xsd:annotation>
                <xsd:complexType mixed="true">
                  <xsd:sequence>
                    <!--numEvents added on Wednesday-->
                    <xsd:element name="numEvents"
                      type="xsd:nonNegativeInteger" />
                    <xsd:element
                      maxOccurs="unbounded" minOccurs="0" ref="country_RepItem" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="isItem" type="xsd:string" />
      <xsd:attribute name="originalElement" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athlete_RepItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="athlete_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="athlete">
                <xsd:complexType mixed="true">
                  <xsd:sequence>
                    <xsd:element name="athName"
                      type="xsd:string" />
                    <xsd:element
                      maxOccurs="unbounded" minOccurs="0" name="phone"
                      type="phoneNumType" />
```

```
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="isItem" type="xsd:string" />
      <xsd:attribute name="originalElement" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 171: Final Representational schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="unqualified"
  targetNamespace="http://www.cs.arizona.edu/tau/RepSchema"
  xmlns="http://www.cs.arizona.edu/tau/RepSchema"
  xmlns:rep0="http://www.cs.arizona.edu/tau/RepSchema0"
  xmlns:rep1="http://www.cs.arizona.edu/tau/RepSchema1"
  xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsd:import namespace="http://www.cs.arizona.edu/tau/TVSchema" schemaLocation="TVSchema.xsd" />
  <xsd:import namespace="http://www.cs.arizona.edu/tau/RepSchema0" schemaLocation="rep0.xsd" />
  <xsd:import namespace="http://www.cs.arizona.edu/tau/RepSchema1" schemaLocation="rep1.xsd" />
  <xsd:element name="sv_root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="schemaItem">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element maxOccurs="1" minOccurs="1"
                name="schemaVersion0">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element maxOccurs="1"
                      minOccurs="1" ref="tv:timestamp_TransExtent" />
                    <xsd:element maxOccurs="1"
                      minOccurs="1" ref="rep0:tv_root" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element maxOccurs="1" minOccurs="1"
                name="schemaVersion1">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element maxOccurs="1"
                      minOccurs="1" ref="tv:timestamp_TransExtent" />
                    <xsd:element maxOccurs="1"
                      minOccurs="1" ref="rep1:tv_root" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="temporalSchema" type="xsd:string">
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

19.6 Temporal Document

Listing 172: Temporal Document.

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

  <sliceSequence>
    <slice location="slice1.xml" begin="2002-01-01" end="2003-01-01" />
    <slice location="slice2.xml" begin="2003-01-01" end="2005-01-01" />
    <slice location="slice3.xml" begin="2005-01-01" end="2006-01-01" />
    <slice location="slice4.xml" begin="2006-01-01"/>
  </sliceSequence>

</temporalRoot>
```

Listing 173: Squashed document.

```
<?xml version="1.0" encoding="UTF-8"?>
<rep:sv_root xmlns:rep="http://www.cs.arizona.edu/tau/RepSchema"
  temporalSchema="winolympic_tempSchema.xml"
  xmlns:rep0="http://www.cs.arizona.edu/tau/RepSchema0"
  xmlns:rep1="http://www.cs.arizona.edu/tau/RepSchema1"
  xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema">

  <schemaItem>
    <schemaVersion0>
      <tv:timestamp_TransExtent begin="2002-01-01"
        end="2005-01-01" />
      <rep0:tv_root
        xmlns:rep0="http://www.cs.arizona.edu/tau/RepSchema0"
        begin="2002-01-01" end="2005-01-01">
        <rep0:winOlympic_RepItem isItem="y"
          originalElement="winOlympic">
          <winOlympic_Version>
            <tv:timestamp_TransExtent begin="2002-01-01"
              end="2005-01-01" />
            <winOlympic>
              There are events in the Olympics.
              <rep0:country_RepItem isItem="y"
                originalElement="country">
                <country_Version>
                  <tv:timestamp_TransExtent
                    begin="2002-01-01" end="2005-01-01" />
                  <country countryName="Norway">
                    <rep0:athleteTeam_RepItem
                      isItem="y" originalElement="athleteTeam">
                      <athleteTeam_Version>
                        <tv:timestamp_TransExtent
                          begin="2002-01-01" end="2003-01-01" />
                        <athleteTeam
                          numAthletes="95">
                          <teamName>
                            Norway_Army
                          </teamName>
                          Athletes will take part in various events. The athletes
                          participating are listed below
                          <rep0:athlete_RepItem
                            isItem="y" originalElement="athlete">
                            <athlete_Version>
                              <tv:timestamp_TransExtent
                                begin="2002-01-01" end="2003-01-01" />
                              <athlete>
                                <athName>
```

```
                    Kjetil Andre Aamodt
                  </athName>
                </athlete>
              </athlete_Version>
            </rep0:athlete_RepItem>
            <rep0:athlete_RepItem
              isItem="y" originalElement="athlete">
              <athlete_Version>
                <tv:timestamp_TransExtent
                  begin="2002-01-01" end="2003-01-01" />
                <athlete>
                  <athName>
                    Trine
                    Bakke-Rognmo
                  </athName>
                  His phone numbers are:
                  <phone>
                    123-402-0340
                  </phone>
                  <phone>
                    123-402-0000
                  </phone>
                </athlete>
              </athlete_Version>
            </rep0:athlete_RepItem>
            <rep0:athlete_RepItem
              isItem="y" originalElement="athlete">
              <athlete_Version>
                <tv:timestamp_TransExtent
                  begin="2002-01-01" end="2003-01-01" />
                <athlete>
                  <athName>
                    Lasse Kjus
                  </athName>
                </athlete>
              </athlete_Version>
            </rep0:athlete_RepItem>
          </athleteTeam>
        </athleteTeam_Version>
        <athleteTeam_Version>
          <tv:timestamp_TransExtent
            begin="2003-01-01" end="2005-01-01" />
          <athleteTeam
            numAthletes="95">
            <teamName>
              Norway_Army
            </teamName>
            Athletes will take part in various events. The athletes
            participating are listed below
            <rep0:athlete_RepItem
              isItem="y" originalElement="athlete">
              <athlete_Version>
                <tv:timestamp_TransExtent
                  begin="2003-01-01" end="2005-01-01" />
                <athlete>
                  <athName>
                    Kjetil Andre Aamodt
                  </athName>
                </athlete>
              </athlete_Version>
            </rep0:athlete_RepItem>
            <rep0:athlete_RepItem
              isItem="y" originalElement="athlete">
              <athlete_Version>
                <tv:timestamp_TransExtent
                  begin="2003-01-01" end="2005-01-01" />
                <athlete>
                  <athName>
```

```
                                Andre Agassi
                              </athName>
                            </athlete>
                          </athlete_Version>
                        </rep0:athlete_RepItem>
                        <rep0:athlete_RepItem
                          isItem="y" originalElement="athlete">
                          <athlete_Version>
                            <tv:timestamp_TransExtent
                              begin="2003-01-01" end="2005-01-01" />
                            <athlete>
                              <athName>
                                Trine
                                Bakke-Rognmo
                              </athName>
                              His phone numbers are:
                              <phone>
                                123-402-0340
                              </phone>
                              <phone>
                                123-402-0000
                              </phone>
                            </athlete>
                          </athlete_Version>
                        </rep0:athlete_RepItem>
                        <rep0:athlete_RepItem
                          isItem="y" originalElement="athlete">
                          <athlete_Version>
                            <tv:timestamp_TransExtent
                              begin="2003-01-01" end="2005-01-01" />
                            <athlete>
                              <athName>
                                Lasse Kjus
                              </athName>
                            </athlete>
                          </athlete_Version>
                        </rep0:athlete_RepItem>
                      </athleteTeam>
                    </athleteTeam_Version>
                  </rep0:athleteTeam_RepItem>
                </country>
              </country_Version>
            </rep0:country_RepItem>
          </winOlympic>
        </winOlympic_Version>
      </rep0:winOlympic_RepItem>
    </rep0:tv_root>
  </schemaVersion0>
  <schemaVersion1>
    <tv:timestamp_TransExtent begin="2005-01-01"
      end="9999-12-31" />
    <rep1:tv_root
      xmlns:rep1="http://www.cs.arizona.edu/tau/RepSchema1"
      begin="2005-01-01" end="9999-12-31">
      <rep1:winOlympic_RepItem isItem="y"
        originalElement="winOlympic">
        <winOlympic_Version>
          <tv:timestamp_TransExtent begin="2005-01-01"
            end="9999-12-31" />
          <winOlympic>
            There are
            <numEvents>11</numEvents>
            events in the Olympics.
            <rep1:country_RepItem isItem="y"
              originalElement="country">
              <country_Version>
                <tv:timestamp_TransExtent
                  begin="2005-01-01" end="9999-12-31" />
```

```
<country countryLead="Andre Agassi"
  countryName="Norway">
  <rep1:athleteTeam_RepItem
    isItem="y" originalElement="athleteTeam">
    <athleteTeam_Version>
      <tv:timestamp_TransExtent
        begin="2005-01-01" end="9999-12-31" />
      <athleteTeam
        numAthletes="95">
        <teamName>
          Norway_Army
        </teamName>
        Athletes will take part in various events. The athletes
        participating are listed below
        <rep1:athlete_RepItem
          isItem="y" originalElement="athlete">
          <athlete_Version>
            <tv:timestamp_TransExtent
              begin="2005-01-01" end="9999-12-31" />
            <athlete>
              <athName>
                Kjetil Andre Aamodt
              </athName>
            </athlete>
          </athlete_Version>
        </rep1:athlete_RepItem>
        <rep1:athlete_RepItem
          isItem="y" originalElement="athlete">
          <athlete_Version>
            <tv:timestamp_TransExtent
              begin="2005-01-01" end="9999-12-31" />
            <athlete>
              <athName>
                Andre Agassi
              </athName>
            </athlete>
          </athlete_Version>
        </rep1:athlete_RepItem>
        <rep1:athlete_RepItem
          isItem="y" originalElement="athlete">
          <athlete_Version>
            <tv:timestamp_TransExtent
              begin="2005-01-01" end="9999-12-31" />
            <athlete>
              <athName>
                Trine
                Bakke-Rognmo
              </athName>
              His phone numbers are:
              <phone>
                123-402-0340
              </phone>
              <phone>
                123-402-0000
              </phone>
            </athlete>
          </athlete_Version>
        </rep1:athlete_RepItem>
        <rep1:athlete_RepItem
          isItem="y" originalElement="athlete">
          <athlete_Version>
            <tv:timestamp_TransExtent
              begin="2005-01-01" end="9999-12-31" />
            <athlete>
              <athName>
                Lasse Kjus
              </athName>
            </athlete>
```

```
                            </athlete_Version>
                          </rep1:athlete_RepItem>
                        </athleteTeam>
                      </athleteTeam_Version>
                    </rep1:athleteTeam_RepItem>
                  </country>
                </country_Version>
              </rep1:country_RepItem>
            </winOlympic>
          </winOlympic_Version>
        </rep1:winOlympic_RepItem>
      </rep1:tv_root>
    </schemaVersion1>
  </schemaItem>
</rep:sv_root>
```

Part III
Common

In this part, we conclude and discuss future work. Section 21 provides a short summary of all elements and attributes defined in τXSchema.

20 Overall Conclusions and Future Work

In this report we have considered how to accommodate and validate time-varying data within XML Schema. We have presented the constructs of Temporal XML Schema (τXSchema), which is an extension of XML Schema. This report also discusses infrastructure and a suite of tools to support the creation and validation of time-varying documents, without requiring any changes to XML Schema.

In Section 4.2, we introduced ten desiderata for our language and tools. We now revisit these desiderata and evaluate our design against them.

- Simplify the representation of time for the user.

  The SQUASH tool enables a temporal document to be constructed directly from a sequence of conventional documents, by providing only a list of these documents and their timestamps. A conventional schema may optionally be provided; no further (temporal) schemas are required, as the defaults allow all the logical and physical annotations to be omitted.

  The user can later provide more details, such as exactly where the timestamps are to be placed, which representation is to be used, and what portions of the document can vary over time.

- Support a three-level architecture to provide data independence, so that changes in the logical and physical level are isolated.

  Our approach ensures data independence by separating (i) the conventional schema document for the instance document, (ii) information concerning what portion(s) of the instance document can vary over time, and (iii) where timestamps should be placed and precisely how the time-varying aspects should be represented. Since these three aspects are orthogonal, our approach allows each aspect to be changed independently.

  This three-level schema specification approach is exploited in supporting tools; several new, quite useful tools τXMLLINT, SCHEMAMAPPER, SQUASH, UNSQUASH, and RESQUASH are introduced that enable the logical and physical data independence provided by our approach. Additionally, this independence enables existing tools (e.g., the XML Schema validator, XQuery, and DOM) to be used in the implementation of their temporal counterparts.

- Retain full upward compatibly with existing standards and not require any changes to these standards.

  Data and schema versioning are supported in a fashion consistent and upwardly-compatible with XML, XML Schema, and conventional XML validators.

- 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.

  We introduced the following tools for τXMLLINT: SCHEMAMAPPER, SQUASH, UNSQUASH, and RESQUASH and extended them to support schema versioning. The tools comprise in concert around 10K source lines of code including comments. Two new schemas `TSSchema` and `ASchema` comprise more than 500 lines of XML code. The framework contains 50-odd Java interfaces and classes.

- Support both valid time and transaction time.

  Both kinds of time are fully supported in τXSchema.

- Accommodate a variety of physical representations for time-varying data.

- Accommodate different kinds of time, such as indeterminate times, unknown times, the current time, and times at a variety of temporal granularities.

- Support instance versioning.

  τXSchema provides an efficient way to define time-varying element types; specifically, an element type that can vary over time, describes how to associate time-varying elements across snapshots, and provides some temporal constraints that broadly characterize how a time-varying element can change over time.

- 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.

  τXSchema fully supports schema versioning, including (time-varying) schemas that include or reference other (time-varying) schemas. In doing so, we leveraged both conventional XML Schema and related tools (principally, the conventional validator), as well as τXMLLINT for data versioning.

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

τXSchema and τXMLLINT can be further enhanced to provide a better system and more features.

- Future work includes extending the τXSchema model to fulfill the issues not addressed during the initial implementation. One of the goals in the desiderata that were not fulfilled is of supporting temporal granularity and indeterminacy. However, it is easy to augment the τXSchema model timestamps with concepts of granularity and incorporate additional physical representations for valid and transaction time that account for indeterminacy, since these extensions would require additions to the TSSchema, ASchema and the generated Representational Schema (Figure 20, boxes 1, 2 and 9), but no changes to the user designed schema documents (boxes 3-6). These augmentations would maintain upward compatibility with previous versions of τXSchema and be transparent to the user.

- Another broad area of work is optimization and efficiency. Although we do talk about the space-efficiency of the tools described in Section 9, we haven't given much attention to their performance. New representations can be proposed, incorporated and evaluated to improve the space-efficiency of the temporal document. We have seen that the DOM API could prove to be a memory bottleneck for huge documents. So instead of parsing the complete document at once, other options need to be evaluated.

  One option is to validate the document in parts, bringing only one item at a time in the memory. This could be achieved by replacing the immediate descendant item elements by their dummy equivalents and then validating the item for its sequenced and non-sequenced constraints. This would result in less memory utilization since only a part of the document is being kept in the memory. As few changes would be required to manage the items one at a time, a major part of the existing algorithm for τXMLLINT could be reused. Here, if a DOM-based parser is used, the whole document needs to be parsed at least once, even if we are validating one item at a time. This could be avoided by using an event-based SAX parser and building an in-memory tree of only the required elements in order to perform those aspects of the validation that are synchronized with the parsing. This approach would require complex memory management and parsing of the document multiple times, but memory use would be greatly reduced.

As described earlier, all the tools are based on the elementary functions pushUp, pushDown and coalesce. If we can modify them to use a SAX parser instead of a document-object-model, we can easily convert all the tools to use a SAX parser. We think that, converting pushDown to use a SAX parser woule be easier; the timestamps could be pushed down easily as the document is being parsed from start to end. After initial thought it appears that, pushUp would need building of an in-memory tree, pushing the timestamps up and then serializing the tree. This could also be achieved by building the tree in parts resulting in more complexity. coalesce would also need to build a tree in memory. But instead of building a complete tree at once, it can build a subtree for each item at a time and then coalesce it.

- Future work also includes enabling the legacy applications or the data inconsistent with a subsequently changed schema, by exploiting information about the evolving schema that is already captured in the temporal schema.

- Current implementation of tools does not support all described features of τXSchema completely. These features need to be implemented to provide completeness to the tools. The unimplemented features, the anticipated changes and the estimated efforts required to implement them are listed below. The estimated effort does not include becoming familiarized with the architecture and the source code.

  - Support for the '*Step*' representation of timestamp: Some changes to the classes Item and RepItem would be needed to support the '*Step*' representation. Some changes would also be needed to the algorithms implemented in class `Primitives`. 15–20 hours of work is anticipated.
  - Support for the generic validation of non-sequenced constraints: Currently, the validation for each non-sequenced constraints is implemented using a separated function inside `Item` class. To provide a framework for the generic support of non-sequenced constraints, a 'Visitor' pattern could be used. In that case, the validator for each non-sequenced constraint will be implemented in a separate class and a reference to an Item element will be passed to it. The addition of a new constraint could be made easier by some properties file; this will eliminate any changes to the `Item` class for addition/modification of constraints. 15–20 hours of work is anticipated.
  - Support for the schemaPath expressions containing 'wildcards' characters and shortcut representation: This will change the way targets are being evaluated. Changes to the classes `SchemaPathEvaluator`, `Item` and `ItemIdentifier` would be needed. Around 30–40 hours of work is anticipated.
  - Support for the item-identifiers specified in terms of existing items or schema keys, and targets containing 'wildcard' characters: Some changes to the classes `Item` and `ItemIdentifier` would be needed. Some changes to the functions from class `Primitive` may also be needed since the procedure for coalescing may change. 20–30 hours of work is anticipated for this change.
  - Support for nested time-varying schemas: We anticipate, this would result in a considerable change to all the tools. A couple of weeks of work may be needed to support this feature.
  - Support for RESQUASHing of a temporal document using a new temporal annotation: The changes needed for this functionality are mentioned in the Section 9.6. 4–5 hours of work should be sufficient for this change.

- In this work, only conceptual support for the bitemporal elements is defined. The tools need to be extended to support bitemporal elements.

- τXSchema should be integrated with a schema-aware XML-based editor like XMLSpy [88]. Schema-aware editors generate easy-to-use templates for updating each type of element defined in a schema. But they do not track changes to either the schema or the data. Enabling versioning for both will support unlimited undo/redo, improve change tracking, and aid in cooperative editing. Another direction of future work is to add versioning to XUpdate [89]. XUpdate is a language for specifying changes to the XML document.

- τXSchema can also be extended to support generic aspects [29]. In that approach, we generalized τXSchema to represent any generic aspect instead of just timestamps.

- We plan to extend our approach to also accommodate intensional XML data [58] which refer to programs that generate data. Some of these programs may be evaluated (a process termed materialization), with the results replacing the programs in the document. There are several interesting time-varying aspects of intensional XML data: (i) the programs themselves may change over time, (ii) even if the programs are static, the results of program evaluations may change over time, as external data the programs access changes, and (iii) even if the programs and the external data are static, different versions of the program evaluators (e.g., Java compiler) may be present, may generate different results due to incompatibilities between versions. It is challenging to manage this combination of schema and instance versioning over time.

- Currently there is no separation of elements or attributes based on the relative frequency of update. In the situation that some elements (for example) vary at a significantly different rate than other elements, it may prove more efficient to split the schema up into pieces such that elements with similar "rates of change" are together [56, 62, 71]. This would avoid redundant repetition of elements that do not change as frequently. Related to optimization, there is the issue of optimizing the use of time-varying loose text. For instance it may be desirable to capture order among different loose text pieces within an element (e.g., different pieces may be used to describe a particular sub-element and may therefore vary with a frequency strongly correlated to the sub-element's temporal characteristics). We want to incorporate recently proposed representations (e.g., [7, 13, 19, 22] into our physical annotations. Finally, the efficiency of the tools mentioned in Sections 7 and 13 can be improved. For example, it would be interesting to investigate whether incremental validation approaches [5, 9, 63] are applicable in the temporal schema validator.

- In Section 6 we discussed temporal augmentations to XML Schema constraints. For non-sequenced uniqueness constraints, we do not currently support the specification of a constraint that applies solely between nodes. For example, given the constraint on employee email addresses in Listing 24, if we wished to refine it to say: "the same employee could have a repetition of an email address over time, but two different people were not allowed to have the same address over time", we need to extend our work to support it. We leave for the future a detailed discussion of and specification of the syntax and semantics for such unique constraints that apply solely between nodes.

- Another extension for constraints (Section 6) is to consider the constraints under temporal indeterminate times. So for instance, suppose we don't know when exactly an employee is employed. We have some time that we know he is employed (e.g., 2005–2009), but some fuzziness on each end of that employment (exactly which month and day). Then the evaluation of each constraint can be done with respect to what is definite and what is possible. For example if we have a sequenced constraint that each employee's email has to be unique, if two employees have the same e-mail but the time at which they co-exist is indeterminate then the constraint may possibly be maintained, rather than definitely violated (the user would chose the validation semantics).

The three-level schema specification approach introduced in this work by τXSchema, the infrastructure, and a suite of tools provide a system for creation and validation of data-versioned XML documents, without requiring any changes to the XML Schema specification. By clever use of schema-constant periods and cross-wall validation, schema versioning is also integrated in the framework with the support for time-varying documents in a fashion consistent and upwardly-compatible with XML, XML Schema, and conventional XML validators. This work has shown that by utilizing schema-constant periods and cross-wall validation, it is possible to realize a comprehensive system for representing and validating data- and schema-versioned XML documents, while remaining fully compatible with the XML standards.

21 τXSchema Reference

21.1 Conventions

The following are conventions used in this section.

- Indented text is used to specify a sub-element.
- "Datatype" refers to the "base datatype" which may be restricted or extended via datatype definitions. The restrictions are specified in the "Notes" column.
- The column for "[min:max]" is used both for elements (`minOccurs`, `maxOccurs`) and attributes (`optional`, `required`). For example, optional is denoted using [0:1], while required is denoted with a [1:1]. The value "U" is used to denote "unbounded" for `maxOccurs`.

21.2 TSSchema

- Filename: `TSSchema.xsd`
- Purpose: Defines temporal schemas used to associate schemas and annotations
- Target Namespace: `http://www.cs.arizona.edu/tau/tauXSchema/TSSchema`
- Root Element: `temporalSchema`
- Details:

Table 10 sub-elements of `temporalSchema`

Table 11 sub-elements of multiple elements

Table 12 sub-elements of `itemIdentifierCorrespondence`

21.3 ASchema

- Filename: `ASchema.xsd`
- Purpose: Schema for Logical and Physical Annotations
- Target Namespace: `http://www.cs.arizona.edu/tau/tauXSchema/ASchema`
- Root Element: `annotationSet`
- Details:

Table 13 sub-elements of `annotationSet`

Table 14 sub-elements of `logical`

Tables 15 and 16 sub-elements of `item`

Table 17 sub-elements of `itemIdentifier`

Table 18 sub-elements of `validTime`

Table 19 sub-elements of `attribute`

Table 20 sub-elements of `defaultTimeFormat`

Table 21 sub-elements of `nonSeqUnique`

Table 22 sub-elements of `nonSeqKey`

Table 23 sub-elements of `uniqueNullRestricted`

Table 24 sub-elements of `nonSeqKeyref`

Table 25 sub-elements of `cardConstraint`

Table 26 sub-elements of `transitionConstraint`

Table 27 sub-elements of `physical`

Table 28 sub-elements of `stamp`

Table 29 sub-elements of `orderBy`

21.4 TDSchema

- Filename: `TDSchema.xsd`
- Purpose: Defines Temporal Documents
- Target Namespace: `http://www.cs.arizona.edu/tau/tauXSchema/TDSchema`
- Root Element: temporalDocument
- Details:

Table 30 sub-elements of `temporalDocument`

21.5 MDSchema

- Filename: `MDSchema.xsd`
- Purpose: Defines mapping pairs to associate old an new item identifier values
- Target Namespace: `http://www.cs.arizona.edu/tau/tauXSchema/MDSchema`
- Root Element: mappings
- Details:

Table 31 sub-elements of `mappings`

Table 32 sub-element of `oldValue` and `newValue`

Element	Notes	[min:max]
`conventionalSchema`	specifies the conventional schema(s)	[1:1]
`sliceSequence`	see details of sub-element in Table 11	[0:1]
`include`	see details of sub-element in Table 11	[0:1]
`annotationSet`	specifies the annotation schema(s)	[1:1]
`sliceSequence`	see details of sub-element in Table 11	[0:1]
`include`	see details of sub-element in Table 11	[0:1]

Table 10: TSSchema: Sub-elements of `temporalSchema`

Table 11: SliceSequence: Sub-elements of multiple elements

Element	Attribute	Notes	datatype	[min:max]
include		includes another document into current document		[0:1]
	schemaLocation	the URI of the document to include	xs:string	[1:1]
sliceSequence		specifies a sequence of slices		[0:1]
slice		details about a single slice		[0:U]
	location	the URI of the slice document	xs:string	[1:1]
	begin	the begin date for the slice	xs:date	[1:1]
	end	the end date for the slice	xs:date	[0:1]
itemIdentifierCorrespondence		specifies how to bridge item identifiers between instance documents		[0:U]

Table 12: TSSchema: Sub-elements of `itemIdentifierCorrespondence`

Element	Attribute	Notes	datatype	[min:max]
	`oldRef`	references the identifier in the old logical annotation	`xs:string`	[1:1]
	`newRef`	references the identifier in the succeeding logical annotation	`xs:string`	[0:1]
	`mappingType`	mapping type, one of `useBoth`, `useOld`, `useNew`, `replace`	`xs:string`	[0:1]
	`mappingLocation`	location of mapping file containing old and new values that correspond (schema for this file is provided in a separate Appendix)	`xs:anyURI`	[0:1]

Element	Notes	[min:max]
`logical`	contains all logical annotations see details of sub-elements in Table 14	[0:1]
`physical`	contains all physical annotations see details of sub-elements in Table 27	[0:1]

Table 13: ASchema: sub-elements of `annotationSet`

Table 14: ASchema: Sub-elements of `logical`

Element	Attribute	Notes	datatype	[min:max]
`include`		contains the location / URI of one or more (possibly time-varying) logical or physical annotation files.		[0:U]
	`annotationLocation`	URI for location	`xs:anyURI`	[1:1]
`defaultTimeFormat`		default time format used in the document (details in Table 20 below)		[0:1]
`item` [subelements in Table 15]		defines a time varying item		[0:U]
	`target`	location of the element being annotated	`xs:anyURI`	[1:1]

Table 15: ASchema: Sub-elements of `item`

Element	Attribute	Notes	datatype	[min:max]
`validTime` [subelements in Table 18]		information on the valid time annotations for the element		[0:1]
	`kind`	the time kind, either `state` or `event`	`xs:string`	[1:1]
	`content`	if the content of an element changes over time (if leaf element), or its loose text / order of sub-elements change (if not leaf element), either `constant` or `varying`	`xs:string`	[0:1]
	`existence`	if the element itself can exist / not-exist over time, one of: `constant`, `varyingWithGaps`, `varyingWithoutGaps`	`xs:string`	[0:1]
`transactionTime`		describes if the element varies in transaction time		[0:1]
`frequency`		frequency of change for the annotated item	`xs:string`	[0:1]
`itemIdentifier` [subelements in Table 17]		item identifier definitions, required for all time-varying elements; if not defined defaults to the contents of the element (i.e., `//text`)		[0:1]
	`name`	unique (across the current logical annotation file) name of the item-identifier to allow it to be referenced by other identifier definitions	`xs:string`	[0:1]
	`timeDimension`	time dimension applicable, one of `validTime`, `transactionTime`, `bitemporal`; default is `validTime`	`xs:string`	[0:1]

Table 16: ASchema: Sub-elements of `item`, cont.

`attribute` [subelements in Table 19]	`name`	defines a time varying attribute name of the attribute being annotated	`xs:string`	[0:U] [1:1]
`nonSeqUnique` [details in Table 21]		defines a non-sequenced Unique constraint		[0:U]
`nonSeqKey` [details in Table 22]		defines a non-sequenced Key constraint		[0:U]
`uniqueNullRestricted` [details in Table 23]		defines a non-sequenced Unique constraint with Null value restrictions		[0:U]
`nonSeqKeyref` [details in Table 24]		defines a non-sequenced referential integrity Constraint		[0:U]
`cardConstraint` [details in Table 25]		defines a non-sequenced Cardinality constraint		[0:U]
`transitionConstraint` [details in Table 26]		defines a non-sequenced transition constraint		[0:U]

Table 17: ASchema: Sub-elements of `itemIdentifier`

Element	Attribute	Notes	datatype	[min:max]
`keyref`		information on the referenced keys		[0:U]
	`refName`	the name of the referenced key	`xs:string`	[1:1]
	`refType`	whether the keyref is to a conventional key, an item identifier	`xs:string`	[0:1]
`field`		*information on the location of the key elements and/or attributes*		[0:U]
	`path`	simplified XPath expression to specify the element / attribute picked as part of the key	`xs:string`	[0:1]

Table 18: ASchema: Sub-elements of `validTime` element with `item`

Element	Attribute	Notes	datatype	[min:max]
`contentVaryingApplicability`		captures the periods over which the content can vary (inapplicable if content is `constant`). The max occurrence is set to unbounded to allow for temporal elements.		[0:U]
	`begin`	the earliest time the content of the element or attribute may vary	`xs:string`	[0:1]
	`end`	the latest time the content of the element or attribute may vary	`xs:string`	[0:1]
`maximalExistence`		captures the periods over which the existence can vary (for elements: inapplicable if existence is `constant`; can only take on a single period if it is `varyingWithoutGaps`; for attributes: inapplicable if attribute is required)		[0:U]
	`begin`	restriction on the earliest time of existence for an element or attribute	`xs:string`	[0:1]
	`end`	restriction on the latest time of existence for an element or attribute	`xs:string`	[0:1]
`frequency`		frequency of change for the annotated attribute	`xs:string`	[0:1]

Table 19: ASchema: Sub-elements of `attribute`

`validTime` [subelements similar to those in Table 18 with no `maximal-Existence`]		information on the valid time annotations for the attribute		[0:1]
	`kind`	the time kind, either `state` or `event`	`xs:string`	[1:1]
	`content`	if the content of an attribute changes over time, either `varying` or `constant`	`xs:string`	[1:1]
`transactionTime`		describes if the attribute varies in transaction time		[0:1]
`frequency`		frequency of change for the annotated attribute	`xs:string`	[0:1]

Table 20: ASchema: Sub-elements of `defaultTimeFormat`

Element	Attribute	Notes	datatype	[min:max]
`defaultTimeFormat`		*default time format used*		[0:1]
	`plugin`	plugin used: `tauZaman`, `unix`, `XMLSchema` / etc.	`xs:string`	[0:1]
	`granularity`	granularity of the time format (if `XMLSchema` is the plugin, it refers to the datatype)	`xs:string`	[0:1]
	`calendar`	calendric system used, e.g., `Gregorian`	`xs:string`	[0:1]
	`properties`	date format properties	`xs:string`	[0:1]
	`valueSchema`	value schema used for the date	`xs:anyURI`	[0:1]

Table 21: ASchema: Attributes and sub-elements for nonSeqUnique

Element	Attribute	Notes	datatype	[min:max]
	name	The name of the constraint	xs:string	[0:1]
	conventionalIdentifier	The referenced conventional identifier	xs:string	[0:1]
	dimension	validTime, transactionTime, or bitemporal (default: validTime	xs:string	[0:1]
	evaluationWindow	Time window over which the constraint should be checked (default: lifetime of document)	xs:string	[0:1]
	slideSize	Size of the slide for successive evaluation windows (default: granularity of constrained data type); Only used in conjunction with evaluationWindow	xs:string	[0:1]
applicability		When the constraint is applicable (default: lifetime of document)		[0:1]
(begin, end)		Temporal element to specify applicability; we use a series of intervals (indicated by begin and end)	xs:date, xs:date	[0:U]
selector		For the definition of a new constraint.		[0:1]
field		For the definition of a new constraint.		[0:U]

Element	Attribute	Notes	datatype	[min:max]
	name	The name of the constraint	xs:string	[0:1]
	conventionalIdentifier	The referenced conventional identifier	xs:string	[0:1]
	dimension	validTime, transactionTime, or bitemporal (default: validTime	xs:string	[0:1]
	evaluationWindow	Time window over which the constraint should be checked (default: lifetime of document)	xs:string	[0:1]
	slideSize	Size of the slide for successive evaluation windows (default: granularity of constrained data type); Only used in conjunction with evaluationWindow	xs:string	[0:1]
applicability		When the constraint is applicable (default: lifetime of document); if a value is specified (e.g., *lifetime*)—the begin, end sub-elements should be empty		[0:1]
(begin, end)		Temporal element to specify applicability, with a series of intervals	xs:date, xs:date	[0:U]
selector		For the definition of a new constraint. It is similar to the selector sub-element in the uniqueConstraint definition		[0:1]
field		For the definition of a new constraint. It is similar to the field sub-element in the uniqueConstraint definition		[0:U]

Table 22: ASchema: Attributes and sub-elements for nonSeqKey

Table 23: ASchema: Attributes and sub-elements for `uniqueNullRestricted`

Element	Attribute	Notes	datatype	[min:max]
	`name`	The name of the constraint	`xs:string`	[0:1]
	`conventionalIdentifier`	The referenced conventional identifier	`xs:string`	[0:1]
	`nullCountMin`	The number of null values allowable (either this attribute or `nullCountMax` should have a value)	`xs:nonNegativeInteger`	[0:1]
	`nullCountMax`	The number of null values allowable (used only within `uniqueNullRestricted`)	`xs:nonNegativeInteger`	[0:1]
	`dimension`	`validTime`, `transactionTime`, or `bitemporal` (default: `validTime`)	`xs:string`	[0:1]
	`evaluationWindow`	Time window over which the constraint should be checked (default: lifetime of document)	`xs:string`	[0:1]
	`slideSize`	Size of the slide for successive evaluation windows (default: granularity of constrained data type); Only used in conjunction with `evaluationWindow`	`xs:string`	[0:1]
`applicability`		When the constraint is applicable (default: lifetime of document)		[0:1]
(`begin`, `end`)		Temporal element to specify applicability, with a series of intervals	`xs:date`, `xs:date`	[0:U]
`selector`		For the definition of a new constraint. It is similar to the `selector` sub-element in the `uniqueConstraint` definition		[0:1]
`field`		For the definition of a new constraint. It is similar to the `field` sub-element in the `uniqueConstraint` definition		[0:U]

Table 24: ASchema: Attributes and sub-elements for `nonSeqKeyref`

Element	Attribute	Notes	datatype	[min:max]
	`name`	The name of the constraint	`xs:string`	[0:1]
	`refer`	The referenced identifier or referential integrity constraint	`xs:string`	[0:1]
`applicability`		When the constraint is applicable (default: the lifetime of the document)		[0:U]
`selector`		Used in the definition of a new constraint		[0:1]
`field`		Used in the definition of a new constraint		[0:U]

Table 25: ASchema: Attributes and sub-elements for `cardConstraint`

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

Table 26: ASchema: Attributes and sub-elements for `transitionConstraint`

Element	Attribute	Notes	datatype	[min:max]
	`name`	The name of the constraint	`xs:string`	[0:1]
	`dimension`	Either `validTime` or `transactionTime` (default: `validTime`)	`xs:string`	[0:1]
`selector`		Role and definition is similar to the `selector` sub-element in the conventional XML Schema constraint definitions (e.g., for `keyref` constraints)		[1:1]
`field`		Similar to the `field` sub-element in the conventional XML Schema constraint definitions. Allowing for multiple `field` elements lets us constraint combinations of entities.		[1:U]
`valuePair`		Sub-element listing possible pairs for discrete changes		[0:U]
`old`, `new`		Sub-elements of `valuePair`		
`valueEvolution`		Sub-element specifying direction of continuous changes		[0:1]
`applicability`		When the constraint is applicable (default: lifetime of document)		[0:1]

Table 27: ASchema: Sub-elements of `physical`

Element	Attribute	Notes	datatype	[min:max]
`include`		contains the location or URL of one or more (possibly time-varying) physical annotation files.		[0:U]
	`annotationLocation`	location or URL of (possibly many) physical annotation files being included	`xs:anyURI`	[1:1]
`defaultTimeFormat` [sub-elements in Table 20]		default time format used in the document		[0:1]
`stamp` [subelements in Table 28]	`target`	path of the element or attribute being annotated	`xs:string`	[0:1]
	`dataInclusion`	specifies sub-element representation, one of `expandedEntity`, `referencedEntity`, `expandedVersion`, `referencedVersion`	`xs:string`	[0:1]

Table 28: ASchema: Sub-elements of `stamp`

Element	Attribute	Notes	datatype	[min:max]
`stampKind`		contains the stamp time dimension and representation of bounds		[1:1]
	`timeDimension`	time dimension applicable, one of `validTime`, `transactionTime`, `bitemporal`	`xs:string`	[0:1]
	`stampBounds`	either `step` or `extent`	`xs:string`	[0:1]
`defaultTimeFormat` [sub-elements in Table 20]		format for the timestamp	`xs:string`	[0:1]
`orderBy` [sub-elements in Table 29		ordering instructions for elements in temporal data; order the multiple instances of this element by: time or specified target	`xs:string`	[0:1]

Table 29: ASchema: sub-elements of `orderBy`

Element	Attribute	Notes	datatype	[min:max]
`field`		ordering field		[0:1]
`target`		path of element or attribute to order by	`xs:string`	[0:1]
`time`	`dimension`	time dimension to order by, either `validTime` or `transactionTime`	`xs:string`	[0:1] [0:1]

Table 30: TDSchema: Sub-elements of `temporalDocument`

Element	Attribute	Notes	datatype	[min:max]
`sliceSequence`		see details of sub-element in Table 11		[0:1]
`include`		see details of sub-element in Table 11		[0:1]

Table 31: MDSchema: Sub-elements of `mappings`

Element	Attribute	Notes	datatype	[min:max]
`pair`		*information about the mapping pair to link old and new item identifiers*		[1:U]
`oldValue`		specifies value of the item identifier in old data (sub-element description in Table 32)		[1:1]
`newValue`		specifies value of the item identifier in old data (sub-element description in Table 32)		[1:1]

Table 32: MDSchema: Sub-element of oldValue and newValue

Element	Attribute	Notes	datatype	[min:max]
field		contains the data value for the old / new item identifier	xs:anyType	[1:U]

Acknowledgements

We thank Lingeshwaran Palaniappan for the development of the initial version of the logical to representational mapper and the temporal data validator. NSF grants IIS-0100436, IIS-0415101, IIS-0515101, IIS-0639106, IIS-0803229, and EIA-0080123 and grants from the Boeing Corporation and Microsoft provided partial support for this work.

References

[1] Serge Abiteboul, Angela Bonifati, Gregory Cobena, Ioana Manolescu, and Tova Milo. Dynamic XML documents with distribution and replication. In *ACM SIGMOD International Conference on Management of Data*, pages 527–538, San Diego, CA, 2003.

[2] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A data model for temporal XML documents. In *DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 334–344, London, UK, 2000. Springer-Verlag.

[3] John Bair, Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Notions of upward compatibility of temporal query languages. *Business Informatics (Wirtschafts Informatik)*, 39(1):25–34, 1997.

[4] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems*, 29(4):710–751, 2004.

[5] Denilson Barbosa, Alberto Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. Efficient incremental validation of XML documents. In Meral Ozsoyoglu and Stan Zdonik, editors, *20th International Conference on Data Engineering*, Boston, MA, 2004. IEEE Computer Society.

[6] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML Schema definitions from XML data. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 998–1009. VLDB Endowment, 2007.

[7] Dorian Birsan, Harm Sluiman, and Stacey-Anne Fernz. Xml diff and merge tool, 1999.

[8] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Temporal statement modifiers. *ACM Transactions on Database Systems*, 25(4):407–456, 2000.

[9] Beatrice Bouchou and Mirian Halfeld-Ferrari. Updates and incremental validation of XML documents. In Georg Lausen and Dan Suciu, editors, *9th International Workshop on Data Base Programming Languages*, Potsdam, Germany, 2003. Springer.

[10] Zouhaier Brahmia and Rafik Bouaziz. Schema versioning in multi-temporal XML databases. In *ICIS 08: Seventh IEEE/ACIS International Conference on Computer and Information Science, 2008*, pages 158–164. IEEE Computer Society, 2008.

[11] Harvey Bratman. A alternate form of the "uncol diagram". *Communications of the ACM*, 4(3):142, 1961.

[12] Peter Buneman, Susan Davidson, Weifei Fan, Carmem Hara, and WangChiew Tan. Keys for XML. *Computer Networks*, 39(5):473–487, 2002.

[13] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan. Archiving scientific data. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *ACM SIGMOD International Conference on Management of Data*, pages 1–12, Madison, WI, 2002. ACM.

[14] Thomas Burns, Elizabeth N. Fong, David Jefferson, Richard Knox, Leo Mark, Christopher Reedy, Louis Reich, Nick Roussopoulos, and Walter Truszkowski. Reference model for dbms standardization, database architecture framework task group of the ansi/x3/sparc database system study group. *SIGMOD Record*, 15(1):19–58, 1986.

[15] Marcela Campo and Alejandro Vaisman. Consistency of temporal XML documents. In *XSym 2006: Database and XML Technologies. 4th International XML Database Symposium, Proceedings*, Lecture Notes in Computer Science Vol. 4156, pages 31–45, Seoul, South Korea, 2006. Springer-Verlag. 9144700, temporal XML document, temporal data representation, historical information tracking, document state recovery, temporal XML abstract model, temporal constraint, document validation, temporal XML consistency.

[16] XML Schema Versioning Use Cases. Framework for discussion of versioning, 2006. URL http://www.w3.org/XML/2005/xsd-versioning-use-cases, Viewed January 15th, 2007.

[17] Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. Schema versioning for multitemporal relational databases. *Information Systems*, 22(5):249–290, 1997.

[18] Sudarshan S. Chawathe, Serge Abiteboul, and Jennifer Widom. Representing and querying changes in semistructured data. In *14th International Conference on Data Engineering*, pages 4–13, Orlando, FL, USA, 1998. IEEE Computer Society.

[19] Shu Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient schemes for managing multiversionXML documents. *The VLDB Journal*, 11(4):332–353, 2002.

[20] Junghoo Cho and Hector Garcia-Molina. Estimating frequency of change. *ACM Transactions on Internet Technology*, 3(3):256–290, 2003.

[21] James Clifford, Curtis Dyreson, Tomás Isakowitz, Christian S. Jensen, and Richard Thomas Snodgrass. On the semantics of "now" in databases. *ACM Transactions on Database Systems*, 22(2):171–214, 1997.

[22] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting changes in XML documents. In *18th International Conference on Data Engineering*, pages 41–52, San Jose, California, 2002. IEEE Computer Society.

[23] Roger L. Costello and Melissa Utzinger. Impact of XML schema versioning on system design, 2007. URL http://www.xfront.com/SchemaVersioning.html, Viewed February 7th, 2007.

[24] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: the prism workbench. In *Very Large Data Base (VLDB)*, 2008.

[25] Faiz Currim, Sabah Currim, Curtis E. Dyreson, and Richard T. Snodgrass. A tale of two schemas: Creating a temporal XML schema from a snapshot schema with τxschema. In *9th International Conference on Extending Database Technology*, pages 559–560, Heraklion-Crete, Greece, 2004. Springer Berlin / Heidelberg.

[26] Faiz Currim and Sudha Ram. Conceptually modeling windows and bounds for space and time in database constraints. *Commun. ACM*, 51(11):125–129, 2008.

[27] Curtis Dyreson. Towards a temporal world-wide web: A transaction time web server. In *12th Australasian Database Conference*, volume 23, pages 169–175, Gold Coast, Australia, 2001.

[28] Curtis Dyreson, Richard T. Snodgrass, Faiz Currim, and Sabah Currim. Schema-mediated exchange of temporal XML data. In *ER 2006: Proceedings of the 25th International Conference on Conceptual Modeling*, Lecture Notes in Computer Science, Vol. 4215, pages 212–227, Tucson, AZ, USA, 2006. Springer-Verlag. 9496307, schema-mediated temporal XML data exchange, Web servers, temporal data collection.

[29] Curtis Dyreson, Richard T. Snodgrass, Faiz Currim, Sabah Currim, and Shailesh Joshi. Weaving temporal and reliability aspects into a schema tapestry. *Data and Knowledge Engineering*, 63(3):752–773, 2007.

[30] Curtis E. Dyreson, Michael Böhlen, and Christian S. Jensen. Capturing and querying multiple aspects of semistructured data. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *25th International Conference on Very Large Data Bases*, pages 290–301, Edinburgh, Scotland, UK, 1999. Morgan Kaufmann.

[31] Curtis E. Dyreson, Hui ling Lin, and Yingxia Wang. Managing versions of web documents in a transaction-time web server. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 422–432, New York, NY, USA, 2004. ACM.

[32] Massimo Franceschet, Angelo Montanari, and Donatella Gubiani. Modeling and validating spatio-temporal conceptual schemas in XML schema. In *18th International Conference on Database and Expert Systems Applications*, pages 25–9, Regensburg, Germany, 2007. IEEE. 9876499, spatio-temporal conceptual schema validation, W3C XML schema language, Java library.

[33] Enrico Franconi, Fabio Grandi, and Federica Mandreoli. Schema evolution and versioning: A logical and computational characterisation. In Herman Balsters, Bert de Brock, and Stefan Conrad, editors, *9th International Workshop on Foundations of Models and Languages for Data and Objects, FoMLaDO/DEMM 2000*, Database Schema Evolution and Meta-Modeling, pages 85–99, Dagstuhl, Germany, 2000. Springer.

[34] Jim Gabriel. How to version schemas. In *XML-Conference and Exhibition, Washington DC, November, 2004*, 2004. URL http://www.idealliance.org/proceedings/xml04/papers/74/howToVersionSchemas.html, Viewed February 7th, 2007.

[35] E Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[36] Dengfeng Gao and Richard T. Snodgrass. Syntax, semantics, and evaluation in the τxquery temporal XML query language. Technical Report Technical Report TR-72, TimeCenter, February 2003.

[37] Dengfeng Gao and Richard T. Snodgrass. Temporal slicing in the evaluation of XML queries. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 632–643. VLDB Endowment, 2003.

[38] Manolis Gergatsoulis and Yannis Stavrakas. Representing changes in XML documents using dimensions. In *Database and XML Technologies*, volume 2824 of *Lecture Notes in Computer Science*, pages 208–222. Springer-Verlag Berlin, Berlin, 2003. ISI Document Delivery No.: BY03G Heidelberg Platz 3, D-14197 Berlin, Germany.

[39] Ana Isabel González-Tablas, L. M. Salas, Benjamín Ramos, and Arturo Ribagorda. Providing personalization and automation to spatial-temporal stamping services. In *Proceedings - International Workshop on Database and Expert Systems Applications, DEXA*, volume 2006, pages 219–225, Copenhagen, Denmark, 2006. Institute of Electrical and Electronics Engineers Inc., New York, NY 10016-5997, United States. Compilation and indexing terms, Copyright 2008 Elsevier Inc. 064010145615, Stamping services, Spatial-temporal stamping, Information model, Personalization.

[40] Fabio Grandi. A bibliography on temporal and evolution aspects in the world wide web. Technical Report Technical Report TR-75, TimeCenter, September 2003.

[41] Fabio Grandi and Federica Mandreoli. The valid web: its time to go. Technical Report Technical Report TR-46, TimeCenter, October 1999.

[42] Fabio Grandi and Federica Mandreoli. The valid web: An XML/XSL infrastructure for temporal management of web documents. In *ADVIS '00: Proceedings of the First International Conference on Advances in Information Systems*, pages 294–303, London, UK, 2000. Springer-Verlag.

[43] Bo Huang, Shanzhen Yi, and Weng Tat Chan. Spatio-temporal information integration in xml. *Future Generation Computer Systems*, 20(7):1157–1170, 2004. ISI Document Delivery No.: 861BM.

[44] Mizuho Iwaihara, Somchai Chatvichienchai, Chutiporn Anutariya, and Vilas Wuwongse. Relevancy based access control of versioned XML documents. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 85–94, New York, NY, USA, 2005. ACM.

[45] Christian S. Jensen and Curtis E. Dyreson. A consensus glossary of temporal database concepts, 1998.

[46] Christian S. Jensen and Richard T. Snodgrass. Temporal data management. Technical Report Technical Report TR-17, TimeCenter, June 1997.

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

[48] Vijay Khatri, Sudha Ram, and Richard T. Snodgrass. Augmenting a conceptual model with geospatiotemporal annotations. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1324–1338, 2004.

[49] Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87, 2000.

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

[51] Federica Mandreoli, Riccardo Martoglia, and Enrico Ronchetti. Supporting temporal slicing in XML databases. In *EDBT 2006: 10th International Conference on Extending Database Technology. Proceedings*, (Lecture Notes in Computer Science Vol.3896), pages 295–312, Munich, Germany, 2006. Springer-Verlag. 8923096.

[52] Amélie Marian. Detecting changes in XML documents. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 41, Washington, DC, USA, 2002. IEEE Computer Society.

[53] Amélie Marian, Serge Abiteboul, Gregory Cobena, and Laurent Mignet. Change-centric management of versions in an XML warehouse. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 581–590, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[54] Jason McHugh and Jennifer Widom. Query optimization for XML. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *25th International Conference on Very Large Databases*, pages 315–326, Edinburgh, Scotland, UK, 1999. Morgan Kaufmann.

[55] William M. McKeeman, James J. Horning, and David B. Wortman. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, NJ, 1970.

[56] L. Edwin McKenzie and Richard T. Snodgrass. An evaluation of relational algebras incorporating the time dimension in databases. *ACM Computing Surveys*, 23(4):501–543, December 1991.

[57] Alberto O. Mendelzon, Flavio Rizzolo, and Alejandro Vaisman. Indexing temporal XML documents. In *In Proceedings of the 30th International Conference on Very Large Databases*, pages 216–227, 2004.

[58] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Fred Dang Ngoc. Exchanging intensional XML data. In *ACM SIGMOD International Conference on Management of Data*, pages 289–300, San Diego, CA, 2003.

[59] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mehai Preda. Monitoring XML data on the web. In Timos Sellis, editor, *ACM SIGMOD International Conference on Management of Data*, pages 437–448, Santa Barbara, CA, 2001.

[60] Kjetil Nørvåg. Algorithms for temporal query operators in XML databases. In *EDBT Workshops*, pages 169–183, 2002.

[61] OMG. Unified modeling language (UML), v2.2, February 2009.

[62] Gultekin Ozsoyoglu and Richard T. Snodgrass. Temporal and real-time databases:a survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

[63] Yannis Papakonstantinou and Victor Vianu. Incremental validation of XML documents. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *9th International Conference on Database Theory*, pages 47–63, Siena, Italy, 2003. Springer.

[64] SAX project. Sax project, official website, 2007. URL http://www.saxproject.org, Viewed March 26, 2007.

[65] TAU Project. τxschema, computer science department at the university of arizona, 2007. URL http://www.cs.arizona.edu/projects/tau/txschema/index.htm, Viewed March 26, 2007.

[66] Mukund Raghavachari and Oded Shmueli. Efficient revalidation of XML documents. *IEEE Transactions on Knowledge and Data Engineering*, 19(4):554–567, 2007. 1041-4347.

[67] Flavio Rizzolo and Alejandro A. Vaisman. Temporal XML: modeling, indexing, and query processing. *The VLDB Journal The International Journal on Very Large Data Bases*, 17(5):1179–1212, 2008.

[68] John F. Roddick. Schema evolution in database systems: an annotated bibliography. *SIGMOD Rec.*, 21(4):35–40, 1992.

[69] John F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[70] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987.

[71] Richard T. Snodgrass. Temporal object oriented databases: A critical comparison. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*, pages 386–408. Addison-Wesley/ACM Press, 1995.

[72] Richard T. Snodgrass. *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[73] Richard T. Snodgrass and Ilsoo Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, 1986.

[74] Richard Thomas Snodgrass, S. Gomez, and E. McKenzie. Aggregates in the temporal query language tquel. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):826–842, 1993.

[75] Na Tang, Yong Tang, and MiaoMiao Cai. Bitemporal extension and mapping of XML data model. In *Proceedings of the 2007 11th International Conference on Computer Supported Cooperative Work in Design*, pages 757–61, Melbourne, Vic., Australia, 2007. IEEE. 9720117.

[76] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass, editors. *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[77] W3C. XML path language (XPath), version 1.0, w3c recommendation, november 1999, 1999. URL http://www.w3.org/TR/xpath, Viewed February 5, 2008.

[78] W3C. XML schema part 2: Datatypes, May 02 2001.

[79] W3C. XQuery 1.0: An XML query language, W3C working draft 16 august 2002, August 16 2002.

[80] W3C. XML schema part 1: Structures second edition, W3C recommendation, october 2004, October 2004. URL http://www.w3.org/TR/xquery, Viewed February 5, 2008.

[81] W3C. XML schema, second edition, W3C recommendation, 2004. URL http://www.w3.org/XML/Schema.html, Viewed March 25, 2009.

[82] W3C. Document object model, 2007. http://www.w3.org/DOM, Viewed March 26, 2007.

[83] W3C. Document type definition (DTD) language, 2007. URL http://www.w3.org/TR/REC-xml/dt-doctype, Viewed March 25, 2007.

[84] W3C. Extensible Markup Language (XML) 1.0, August 2006. http://www.w3.org/TR/REC-xml, Viewed August 25, 2008.

[85] Fusheng Wang and Carlo Zaniolo. Temporal queries and version management in XML-based document archives. *Data and Knowledge Engineering*, 65(2):304–324, 2008. Compilation and indexing terms, Copyright 2008 Elsevier Inc.081411182348 0169-023X.

[86] Raymond K. Wong and Nicole Lam. Managing and querying multi-version XML data with update logging. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 74–81, New York, NY, USA, 2002. ACM.

[87] Vilas Wuwongse, Masatoshi Yoshikawa, and Toshiyuki Amagasa. Temporal versioning of XML documents. In *7th International Conference on Asian Digital Libraries, ICADL 2004. Proceedings (Lecture Notes in Computer Science Vol.3334)*, Digital Libraries: International Collaboration and Cross-Fertilization, pages 419–28, Shanghai, China, 2004. Springer-Verlag. 8411139.

[88] XMLSpy. XML editor for modeling, editing, transforming, & debugging XML technologies., 2007. URL http://www.altova.com/products/xmlspy/xml_editor.html, Viewed April 18, 2007.

[89] XUpdate. XML update language, working draft 2000-09-14, 2000. URL http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html, Viewed April 18, 2007.

[90] Lucie Xyleme. Xyleme: A dynamic warehouse for XML data of the web. In *IDEAS '01: Proceedings of the International Database Engineering & Applications Symposium*, pages 3–7, Washington, DC, USA, 2001. IEEE Computer Society.

[91] Cong Yu and Lucian Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1006–1017. VLDB Endowment, 2005.

A Base Schemas

A.1 TSSchema: Schema for Temporal Schema

Listing 174: TSSchema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema  targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema"
            xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            version="December 5, 2008">

  <xs:include schemaLocation="./SliceSequence.xsd"/>

  <xs:element name="temporalSchema">
    <xs:complexType>
      <xs:sequence>

        <xs:element name="conventionalSchema" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:choice>
              <xs:element name="sliceSequence" type="ts:sliceSequenceType"/>
              <xs:element name="include"       type="ts:includeType"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>

        <xs:element name="annotationSet" minOccurs="0" maxOccurs="1">
          <xs:complexType>
            <xs:choice>
              <xs:element name="sliceSequence" type="ts:sliceSequenceType"/>
              <xs:element name="include"       type="ts:includeType"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>

      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A.2 ASchema: Schema for Annotation Schema

Listing 175: ASchema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema  targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/ASchema"
            xmlns:a="http://www.cs.arizona.edu/tau/tauXSchema/ASchema"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            version="December 5, 2008">


  <xs:element name="annotationSet">
    <xs:complexType>
      <xs:all>

        <xs:element name="logical"  type="a:logicalType"  minOccurs="0" maxOccurs="1" />
        <xs:element name="physical" type="a:physicalType" minOccurs="0" maxOccurs="1"/>

      </xs:all>
    </xs:complexType>
  </xs:element>
```

```
19
20
21  <xs:complexType name="logicalType">
22      <xs:sequence>
23        <xs:element name="include" minOccurs="0" maxOccurs="unbounded">
24          <xs:complexType>
25            <xs:attribute name="annotationLocation" type="xs:anyURI"/>
26          </xs:complexType>
27        </xs:element>
28        <xs:element name="defaultTimeFormat" minOccurs="0">
29          <xs:complexType>
30            <xs:sequence>
31              <xs:element name="format" minOccurs="0">
32                <xs:complexType>
33                  <xs:attribute name="plugin"      type="xs:string" use="optional"/>
34                  <xs:attribute name="granularity" type="xs:string" use="optional"/>
35                  <xs:attribute name="calendar"    type="xs:string" use="optional"/>
36                  <xs:attribute name="properties"  type="xs:string" use="optional"/>
37                  <xs:attribute name="valueSchema" type="xs:anyURI" use="optional"/>
38                </xs:complexType>
39              </xs:element>
40            </xs:sequence>
41          </xs:complexType>
42        </xs:element>
43        <xs:element name="item" minOccurs="0" maxOccurs="unbounded">
44          <xs:complexType>
45            <xs:sequence>
46              <xs:element name="validTime" minOccurs="0">
47                <xs:complexType>
48                  <xs:sequence>
49                    <xs:element name="contentVaryingApplicability"
50                                minOccurs="0" maxOccurs="unbounded">
51                      <xs:complexType>
52                        <xs:attribute name="begin" type="xs:string" use="optional"/>
53                        <xs:attribute name="end"   type="xs:string" use="optional"/>
54                      </xs:complexType>
55                    </xs:element>
56                    <xs:element name="maximalExistence" minOccurs="0">
57                      <xs:complexType>
58                        <xs:attribute name="begin" type="xs:string" use="optional"/>
59                        <xs:attribute name="end"   type="xs:string" use="optional"/>
60                      </xs:complexType>
61                    </xs:element>
62                      <xs:element name="frequency" type="xs:string" minOccurs="0"/>
63                  </xs:sequence>
64                    <xs:attribute name="kind"
65                                  type="a:kindType"      use="optional"/>
66                    <xs:attribute name="content"
67                                  type="a:contentType"   use="optional"/>
68                    <xs:attribute name="existence"
69                                  type="a:existenceType" use="optional"/>
70                </xs:complexType>
71              </xs:element>
72              <xs:element name="transactionTime" minOccurs="0">
73                <xs:complexType>
74                  <xs:sequence>
75                    <xs:element name="frequency" type="xs:string" minOccurs="0"/>
76                  </xs:sequence>
77                  <xs:attribute name="kind"      type="a:kindType"      use="optional"/>
78                  <xs:attribute name="content"   type="a:contentType"   use="optional"/>
79                  <xs:attribute name="existence" type="a:existenceType" use="optional"/>
80                </xs:complexType>
81              </xs:element>
82              <xs:element name="itemIdentifier" minOccurs="0">
83                <xs:complexType>
84                  <xs:sequence>
85                    <xs:element name="keyref" minOccurs="0" maxOccurs="unbounded">
86                      <xs:complexType>
```

```
                        <xs:attribute name="refName"
                                      type="xs:string"      use="required"/>
                        <xs:attribute name="refType"
                                      type="a:keyrefTypeII" use="optional"/>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="field" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="path" type="xs:string" use="required"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name"
                                type="xs:string"           use="optional"/>
                  <xs:attribute name="timeDimension"
                                type="a:timeDimensionType" use="optional"/>
                </xs:complexType>
              </xs:element>
              <xs:element name="attribute" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="validTime" minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="contentVaryingApplicability"
                                      minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                              <xs:attribute name="begin"
                                            type="xs:string" use="optional"/>
                              <xs:attribute name="end"
                                            type="xs:string" use="optional"/>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="frequency" type="xs:string" minOccurs="0"/>
                        </xs:sequence>
                        <xs:attribute name="kind"
                                      type="a:kindType"    use="required"/>
                        <xs:attribute name="content"
                                      type="a:contentType" use="optional"/>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="transactionTime" minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="frequency" type="xs:string" minOccurs="0"/>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="optional"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="target" type="xs:anyURI" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>

  <!--  Simple Types used by the logical annotations above -->
   <xs:simpleType name="kindType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="state"/>
      <xs:enumeration value="event"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="keyrefTypeII">
    <xs:restriction base="xs:string">
```

```
      <xs:enumeration value="snapshot"/>
      <xs:enumeration value="itemIdentifier"/>
    </xs:restriction>
    <!-- II in "keyrefTypeII" stands for ItemIdentifier -->
  </xs:simpleType>
  <xs:simpleType name="contentType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="constant"/>
      <xs:enumeration value="varying"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="existenceType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="constant"/>
      <xs:enumeration value="varyingWithGaps"/>
      <xs:enumeration value="varyingWithoutGaps"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="timeDimensionType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="validTime"/>
      <xs:enumeration value="transactionTime"/>
      <xs:enumeration value="bitemporal"/>
    </xs:restriction>
  </xs:simpleType>


  <xs:complexType name="physicalType">
      <xs:sequence>
        <xs:element name="include" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="annotationLocation" type="xs:anyURI"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="defaultTimeFormat" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="format" minOccurs="0">
                <xs:complexType>
                  <xs:attribute name="plugin"
                                type="xs:string" use="optional"/>
                  <xs:attribute name="granularity"
                                type="xs:string" use="optional"/>
                  <xs:attribute name="calendar"
                                type="xs:string" use="optional"/>
                  <xs:attribute name="properties"
                                type="xs:string" use="optional"/>
                  <xs:attribute name="valueSchema"
                                type="xs:string" use="optional"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="stamp" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="stampKind">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="format" minOccurs="0">
                      <xs:complexType>
                        <xs:attribute name="plugin"
                                      type="xs:string" use="optional"/>
                        <xs:attribute name="granularity"
                                      type="xs:string" use="optional"/>
                        <xs:attribute name="calendar"
                                      type="xs:string" use="optional"/>
```

```
                        <xs:attribute name="properties"
                                      type="xs:string" use="optional"/>
                        <xs:attribute name="valueSchema"
                                      type="xs:string" use="optional"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="timeDimension"
                                type="a:timeDimensionType" use="optional"/>
                  <xs:attribute name="stampBounds"
                                type="a:stampType"         use="optional"/>
                </xs:complexType>
              </xs:element>
              <xs:element name="orderBy" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="field" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:choice>
                          <xs:element name="target" type="xs:string"/>
                          <xs:element name="time">
                            <xs:complexType>
                              <xs:attribute name="dimension" type="a:timeDimensionType"/>
                            </xs:complexType>
                          </xs:element>
                        </xs:choice>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="target"
                          type="xs:string"           use="required"/>
            <xs:attribute name="dataInclusion"
                          type="a:dataInclusionType" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>


  <xs:simpleType name="stampType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="step"/>
      <xs:enumeration value="extent"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="dataInclusionType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="expandedEntity"/>
      <xs:enumeration value="referencedEntity"/>
      <xs:enumeration value="expandedVersion"/>
      <xs:enumeration value="referencedVersion"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

A.3 SliceSequenceSchema: Schema for Slice Sequences

Listing 176: `SliceSequence.xsd`

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

    <xs:complexType name="sliceSequenceType">
        <xs:sequence>
           <xs:element name="slice" minOccurs="1" maxOccurs="unbounded">
             <xs:complexType>
               <xs:attribute name="location" type="xs:string" use="required"/>
               <xs:attribute name="begin"    type="xs:date"   use="required"/>
               <xs:attribute name="end"      type="xs:date"   use="optional"/>
             </xs:complexType>
           </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="includeType">
       <xs:attribute name="schemaLocation" type="xs:string" use="required"/>
    </xs:complexType>
</xs:schema>
```

A.4 TDSchema: Schema for Temporal Document

Listing 177: `TDSchema.xsd`

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema  targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/TDSchema"
            xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TDSchema"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            version="December 5, 2008">

  <xs:include schemaLocation="./SliceSequence.xsd"/>

  <xs:element name="temporalDocument">
    <xs:complexType>
      <xs:sequence>

        <xs:element name="temporalSchemaSet" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="temporalSchema" minOccurs="1" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="location" type="xs:string"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="sliceSequence" type="td:sliceSequenceType"/>

      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

B Evaluation Tools

B.1 Slice Generator

Listing 178: Slice Generator script

```
#!/usr/bin/perl -w

#####################################################################

#

# Author:  Stephen W. Thomas

# Date:    Fall 2008

# Purpose: To generate a large number of XML slices. The

#          output of this script is a set of sliceXX.xml

#          files along with config.xml file used in the

#          tools.

#

#####################################################################


if ($#ARGV < 3){

   print "Usage: $0 amountChange docSize changeKind numSlices\n";

   exit 1;

}


$amountChange = $ARGV[0];

$docSize      = $ARGV[1];

$changeKind   = $ARGV[2];

$numSlices    = $ARGV[3];


# Set begin date

$lastDate       = "2008-01-01";


my $numParts    = $docSize;


# Probability that any given element is changed

my $pNewVersion = $amountChange * $changeKind;


# Probability that a new item (element) is created
```

```
62
63 my $pNewItem    = $amountChange * (1.0-$changeKind);
64
65
66
67 #Initialize values
68
69 $totalPossible = $numParts * (10*$numSlices);
70
71 my @partQuant;
72
73 for ($j=0; $j < $totalPossible; ++$j){
74
75     $partQuant[$j] = $j;
76
77 }
78
79
80
81 # Print header of config document
82
83 open(CONFIG, ">config.xml");
84
85 print CONFIG "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n";
86
87 print CONFIG "<config bundle=\"./test_bundle.xml\"
88
89              xmlns=\"http://www.cs.arizona.edu/tau/tauXSchema/ConfigSchema\">\n";
90
91
92
93 # Print each slice to disk, and add the name of the slice to the config file
94
95 for ($i = 0; $i <= $numSlices; ++$i){
96
97     printLargeFile($i);
98
99     addSnapshotToConfig($i);
100
101 }
102
103 print CONFIG "</config>\n";
104
105
106
107
108
109 # Adds an entry into the config file
110
111 sub addSnapshotToConfig{
112
113     $outfile    = "slice$i.xml";
114
115     $date1      = $lastDate;
116
117     $lastDate   = incrementDate($date1);
118
119     print CONFIG
120
121       "  <snapshot file=\"$outfile\" beginDate=\"$date1\" endDate=\"$lastDate\"/>\n";
122
123 }
124
125
126
127
128
129 # Increments a simple date format "YYYY-MM-DD" with wrappint
```

```
sub incrementDate{
   $firstDate = shift(@_);
   $incMonth = 0;
   $incYear  = 0;


   @ar = split("-", $firstDate);
   $firstYear  = $ar[0];
   $firstMonth = $ar[1];
   $firstDay   = $ar[2];


   $newDay = $firstDay + 1;
   if ($newDay > 28){
     $newDay   = 1;
     $incMonth = 1;
   }
   $newMonth = $firstMonth + $incMonth;
   if ($newMonth > 12){
     $newMonth   = 1;
     $incYear = 1;
   }
   $newYear = $firstYear + $incYear;


   return sprintf "%4d-%02d-%02d",  $newYear,$newMonth,$newDay;
}



################################################################################
# In each additional slice, we may add more elements to each section.
# We may also change the content of existing elements.
################################################################################
sub printLargeFile{


    # Should we increase the number of <part> elements?
    if (rand() <= $pNewItem){
```

```
198
199       $numParts = $numParts + rand(10);
200
201     }
202
203
204
205     $outfile = "slice$i.xml";
206
207     open(OUT, ">$outfile");
208
209     print OUT "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n";
210
211     print OUT "<root>\n";
212
213
214
215     print OUT "<parts>\n";
216
217
218
219     # Print out every <part> element that we have.
220
221     for ($j=0; $j < $numParts; ++$j){
222
223
224
225         # Should we create a new "version" of this element?
226
227         # A new version is created by just changing the value
228
229         # of the <quantity> subelement.
230
231         if (rand() <= $pNewVersion){
232
233             $partQuant[$j]++;
234
235         }
236
237
238
239         print OUT "  <part>\n";
240
241         print OUT "    <name>part_$j</name>\n";
242
243         print OUT "    <id>$j</id>\n";
244
245         print OUT "    <quantity>$partQuant[$j]</quantity>\n";
246
247         print OUT "  </part>\n";
248
249     }
250
251     print OUT "</parts>\n";
252
253
254
255     print OUT "</root>\n";
256
257 }
```

B.2 Scenario Tester

Listing 179: AllRuns script

```
#!/usr/bin/perl
######################################################################
#
# Author:  Stephen W. Thomas
# Date:    Fall 2008
# Purpose: To execute a large set of runs and output results
#          to stdout.
#
######################################################################

use Time::HiRes qw( gettimeofday tv_interval );

my $N = 30;

# Define the variables and their values
my @amountChanges = (0, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64);
my @docSizes      = (1000, 4000, 16000, 64000);
my @degrees       = (0, .25, .5, .75, 1);
my @numSlices     = (100, 1000, 5000, 10000);

my ($t1, $t2, $squashTime, $squashSize, $valTime, $unsquashTime);

# Full factorial design space. Run all cases.
foreach my $amountChange (@amountChanges) {
  foreach my $docSize (@docSizes) {
    foreach my $degree (@degrees) {
      foreach my $numSlice (@numSlices) {

        $squashTime   = 0;
        $squashSize   = 0;
        $valTime      = 0;
        $unsquashTime = 0;

        for (my $i=0; $i<$N; ++$i){

            #print "Producing test case $amountChange $docSize $degree $numSlice \n";
            system("generator.pl $amountChange $docSize $degree $numSlice");


            # Run Squash
            $t1 = [gettimeofday];
            system("squash config.xml");
            $squashTime += tv_interval($t1);
            $squashSize += `ls -l squashed.xml | awk '{ print\$5 }'`;


            # Run tXMLLint
            $t1 = [gettimeofday];
            system("txmllint config.xml squashed.xml > /dev/null");
            $valTime += tv_interval($t1);

            # Run UnSquash
            $t1 = [gettimeofday];
            system("unsquash config.xml squashed.xml");
            $unsquashTime += tv_interval($t1);

            system("rm slice*");
            system("rm squashed.xml");

        }

        # Get averages
        $squashTime   = $squashTime   / $N;
        $squashSize   = $squashSize   / $N;
```

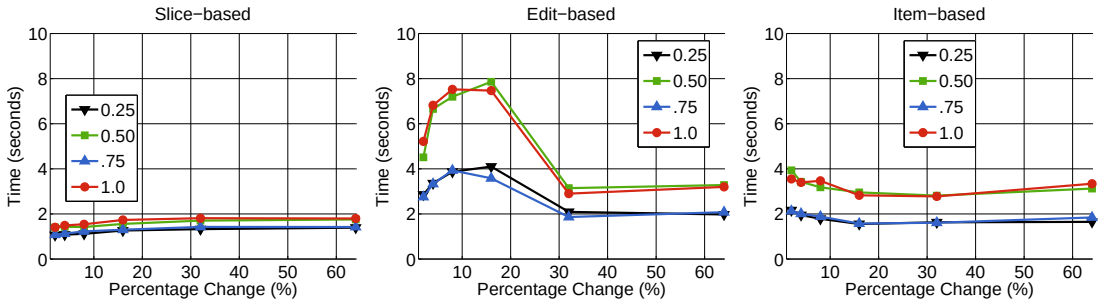
```
        $valTime      = $valTime      / $N;
        $unsquashTime = $unsquashTime / $N;

        # Output results
        printf "%12.5f %12.5f %12.5f %12.5f %12.5f %12.5f %12.5f %12.5f\n",
            $amountChange, $docSize,  $degree, $numSlice,
            $squashTime, $squashSize,
            $valTime, $unsquashTime;
          }
        }
  }
}
```

C Initial Sensitivity to Parameters

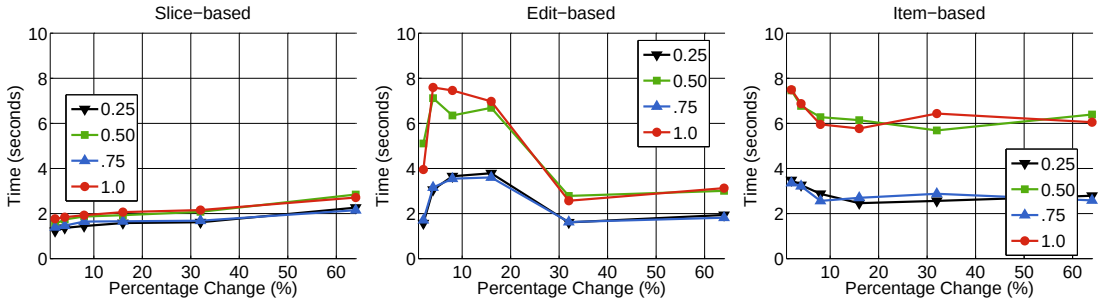
This appendix provides details on the results of the initial experiments executed. The goal was to determine whether or not the dependent variables of interest were sensitive to the amount of change and type of change between each slice. Figure 71 shows the results of SQUASH in these scenarios. In this figure, the slice-based and item-based representation schemes show almost no difference in performance between both percentage change increases (x-axis) or type of change increased (lines). The edit-based scheme shows some small variation, but no general trend is evident and the absolute amount of change is small.

We see that for each representation type, as the percentage of change increases, the time required to squash the document does not increase significantly. Figure 72 shows the results of a larger scenario, but the trends exhibited by the results are similar. . Again, the slice-based and item-based representation schemes show almost no difference in performance between both percentage change increases (x-axis) or type of change (lines). The edit-based scheme shows some small variation, but no general trend is evident and the absolute amount of change is small.



(a) Slice-based (b) Edit-based (c) Item-based

Figure 71: Time required to squash 10 slices, each with about 10 elements



(a) Slice-based (b) Edit-based (c) Item-based

Figure 72: Time required to squash 100 slices, each with about 200 elements (20 slices with 20 elements in the case of the item-based scheme)

Both UNSQUASH and τXMLLINT show similar trends. In effort to reduce to number of experiments run, we conclude that the type and frequency of change is not a large factor in representation performance, and thus we can fix these parameters for the remainder of the experiments.