τXSCHEMA:

SUPPORTING TEMPORAL XML DOCUMENTS

By

ERIC PAUL ROEDER

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

JANUARY 2005

Approved by:

Professor Richard T. Snodgrass

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. Users have to resort to ad hoc, non-standard mechanisms to create schemas for time-varying XML documents. This thesis presents a data model and architecture, called τXSchema, for creating a temporal schema from a non-temporal (snapshot) schema, a temporal annotation, and a physical annotation. The annotations specify which portion(s) of an XML document can vary over time, how the document can change, and where timestamps should be placed. The advantage of using annotations to denote the time-varying aspects is that logical and physical data independence for temporal schemas can be achieved while remaining fully compatible with both existing XML Schema documents and the XML Schema recommendation.

This paper presents a system for constructing and validating *temporal* XML documents. 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. This paper describes how to construct a temporal document by "gluing" individual snapshots into an integrated history. We also show how to build a *temporal schema*. A temporal schema guides the construction of a temporal document and is essential to managing, querying, and validating temporal documents. To create a temporal schema, a schema designer annotates a (nontemporal) XML Schema document. The annotations specify which portion(s) of an XML document can vary over time, how those portions can change, and how to glue snapshots.

Authorship

This was a group project consisting of six people: Professor Richard Snodgrass, Faiz Currim, Sabah Currim, and Eric Roeder and Haitao Liu from the University of Arizona and Professor Curtis Dyreson from Washington State University of Arizona. This is a brief description of each member's contribution, focusing on Eric's contribution.

This project was highly collaborative, with all members participating in all major design decisions. All members also participated in weekly conference calls, contributed to discussions, and brought ideas to the group.

Each section was initially written by a group member, and then iterated upon by the rest of the group in subsequent versions of the document. Richard wrote Sections 1, 5.1 and 8. Curtis wrote Section 2, 3 and 5.5. Faiz wrote Sections 4 and 6. Eric wrote Sections 5.2 and 7. Eric and Curtis wrote Sections 5.2.1 and 5.4 together. Parts of Section 7 were taken from previous documents. Sabah wrote the appendices with help from Faiz. Faiz also combined everyone's sections together and edited the document. Haitao wrote the first version of τValidator.

Eric joined the project in the middle. He was responsible for learning XML, XSchema and about the previous version of the project. Eric contributed several designs to the paper. He examined how all types of nodes can change with time. Eric developed syntax for specifying the location of a node within a τXSchema document. He provided feedback that helped other group members with their designs. Eric also added four Java classes to the τValidator tool that is being developed from the paper. These four classes comprise the gluing component of τValidator. Eric made minor modifications to Haitao's validator.

Table of Contents

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 [24].

Several schema languages have been proposed for XML [22]. From among these languages, XML Schema is the most widely used. The syntax and semantics of XML Schema 1.0 are W3C recommendations [35, 36].

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, and incremental queries that involve the changes between two versions.

In this thesis 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 [25, 29, 32]. 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 document 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).

This paper presents a system, called Temporal XML Schema, for constructing and validating temporal documents. Temporal XML Schema extends XML Schema with the ability to define *temporal element types*. A temporal element type denotes that an element can vary over time, describes how to associate temporal elements across snapshots, and provides some temporal constraints that broadly characterize how a temporal element can change over time. An important goal in the development of Temporal XML Schema was to maximally reuse existing XML standards and technology. In Temporal XML Schema, any element type can be turned into a temporal element type by including a single, simple temporal annotation in the type definition. So a Temporal XML Schema document is just a conventional XML Schema document with a few temporal annotations.

The primary contribution of this thesis is to introduce the *τXSchema* (Temporal XML Schema) data model and architecture. τXSchema is a system for constructing schemas for time-varying XML documents[1]. A time-varying 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[2]. The first level is the schema for an individual version, called the *snapshot schema*. The snapshot schema is a conventional XML Schema document. The second level is the *temporal annotations* of the snapshot schema. The temporal annotations identify which elements can vary over time. For those elements, the temporal annotations also effect a temporal semantics to the various integrity constraints (such as uniqueness) specified in the snapshot schema. The third level is the *physical annotations*. The physical annotations describe how the time-varying aspects are represented. The snapshot schema and temporal and physical annotations are collected together in an XML document termed a *temporal bundle*. This document serves as the temporal schema. Similarly, the individual time slices are combined into one XML document, which serves as the XML instance. A temporal validator takes a temporal bundle and a time-varying temporal document and validates both.

Each annotation can be independently changed, so the architecture exhibits *logical* and *physical data independence* [7]. 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 time-varying documents and schemas. There are tools to convert a time-varying 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 time-varying 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*.

Temporal XML Schema provides tools to construct and validate temporal documents. A temporal document is validated by combining a conventional validating parser with a temporal constraint checker. To validate a temporal document, a temporal schema is first converted to a *representational* schema, which is a conventional XML Schema document that describes how the temporal information is represented. The representational schema must be carefully constructed to ensure the *snapshot validation subsumption* of a temporal document, that is, it is

---

[1] We embrace both the document and data centric orientations of XML and will use the terms "document" and "database" interchangeably.

[2] Three-level architectures are a common architecture in both databases [33] and spatio-temporal conceptual modeling [21].

important to guarantee that each snapshot of the temporal document conforms to the original, snapshot schema (without temporal annotations). A conventional validating parser is then used to validate the temporal document against the representational schema.

τXSchema focuses on both *instance versioning* (representing a time-varying XML instance document) and *schema versioning* (representing a time-varying schema document [15, 31]). The schema can describe which aspects of an instance document change over time; this schema can itself be a time-varying document. The temporal bundle, the XML document that serves as the temporal schema, contains this time-varying schema. All three components, (1) the snapshot schema, (2) the temporal annotations, and (3) the physical annotations, can change over time. The temporal validator and associated tools is 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 [26], are gaining popularity. Incorporating intensional XML data is beyond the scope of this document.

While this document concerns temporal XML Schema, we feel that the general approach of separate temporal and physical annotations is applicable to other data models, such as UML [28]. The contribution of this thesis 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 time-varying 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 document first provides an example that illustrates the challenges of instance and schema versioning, respectively, then lists desiderata for comprehensive support for versioning. We separately investigate the implications of snapshot validation subsumption. The tXSchema architecture is fundamental to fulfilling the desiderata. We show how the three schemas and the temporal bundle interact. The next section goes into a detailed examination of the new constructs introduced in the temporal annotations, the physical annotations, and the temporal bundle. The τValidator tool is also described.

2. Motivation

This section discusses whether conventional XML Schema is appropriate and satisfactory for time-varying data. We first present an example that illustrates how a time-varying document differs from a conventional XML document. We then pinpoint some of the limitations of XML Schema. Finally we state the desiderata for schemas for time-varying documents.

2.1. Motivating Example

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 Figure 1 was added to `winter.xml`. Kjetil won a medal; so on 2002-03-01 the fragment was revised to that shown in Figure 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. Figure 3 shows the correct medal information.

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

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

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

Figure 2: Kjetil won a medal, as of 2002-03-01

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

Figure 3: Medal data is corrected on 2002-07-01

A *time-varying document* records a *version history*, which consists of the information in each version, along with timestamps indicating the lifetime of that version. Figure 4 shows a fragment of a time-varying document that captures the history of Kjetil. The fragment is *compact* in the sense that each edit results in only a small, localized change to the document. The history is also *bi-temporal* because both the *valid time* and *transaction time* lifetimes are captured [20]. The *valid time* refers to the time(s) when a particular fact is true in the modeled reality, while the *transaction time* is the time when the information was edited. The two concepts are orthogonal. Time-varying documents can have each kind of time. In Figure 4 the valid- and transaction-time lifetimes of each element are represented with an optional `<rs:timestamp>` sub-element[3]. 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 changes. 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).

[3] The introduced `<rs:timestamp>` element is in the "`rs`" namespace to distinguish it from any `<timestamp>` elements already in the document. This namespace will be discussed in more detail in Section 4.

```
...
<athlete>
  <rs:timestamp ttStart="2002-01-01" ttStop="2002-02-28"
    vtBegin="2002-02-01" vtEnd="2002-02-28"/>
  <athName>Kjetil Andre Aamodt</athName>
  ...
</athlete>
<athlete>
  <rs:timestamp ttStart="2002-03-01" ttStop="now"
    vtBegin="2002-03-01" vtEnd="now"/>
  <athName>Kjetil Andre Aamodt</athName> won a medal in
  <medal mtype="silver">
   <rs:timestamp ttStart="2002-03-01" ttStop="2002-06-30"
     vtAt="2002-03-01"/>
    Men's Combined
  </medal>
  <medal mtype="gold">
    <rs:timestamp ttStart="2002-07-01" ttStop="now" vtAt="2002-03-01"/>
    Men's Combined
  </medal>
...
```

Figure 4: A fragment of a time-varying document

A temporal document records a *version history*, which consists of the information in each version, along with a timestamp indicating the version's lifetime. Figure 4 shows a temporal document that captures the history of Kjetil. The document is largely a list of athlete and medal *items*. An item is an element that *persists* across individual snapshots in a document's history. Each item has an `itemId` attribute that identifies the item. There is one athlete item in the document, and one medal item. Each item is referenced by a time-varying element, which places the item in the context in which it would appear in a snapshot of the document. For example, in Figure 4 the time-varying element `<athlete`$_{\mathit{TimeVarying}}$`>` references the athlete item, which indicates that a version of that item appears within the context of the `<doc>` element for each snapshot in the range of the version's timestamp.

Whenever the item changes, a new *version* of the item is created. A change is defined, roughly, as a difference in an element's content, ignoring changes to content within sub-elements. Hence, the athlete item has two versions. The second version was created on 2002-03-01 when new text content and a `<medal>` element were added to the element. The timestamp for each version indicates the version's lifetime. The end time of the second version is "*now*" indicating that the version is current. The medal item also has two versions, because an attribute value was changed from silver to gold on 2002-07-01.

```
<doc_TimeVarying>
  <doc>
    <athlete_TimeVarying itemRef="1"/>
  </doc>
  <athlete_Item itemId="1">
    <athlete_Version>
      <time start="2002-01-01" end="2002-03-01">
      <athlete>
        <athName>Kjetil Andre Aamodt</athName>
      </athlete>
    </athlete_Version>
    <athlete_Version>
      <time start="2002-03-01" end="now">
      <athlete>
        <athName>Kjetil Andre Aamodt</athName>
        won a medal in
        <medal_TimeVarying itemRef="2"/>.
      </athlete>
    </athlete_Version>
  </athlete_Item>
  <medal_Item itemId="2">
    <medal_Version>
      <time start="2002-03-01" end="2002-07-01">
      <medal mtype="silver">Men's Combined</medal>
    </medal_Version>
    <medal_Version>
      <time start="2002-07-01" end="now">
      <medal mtype="gold">Men's Combined</medal>
    </medal_Version>
  </medal_Item>
</doc_TimeVarying>
```

Figure 5: A time-varying document

The history shown in Figure 4 captures the *transaction time* lifetime of each version [20]. Transaction time is the system time when the information was edited. Temporal documents could also record the *valid time* versions (valid time is real world time) but for simplicity we consider only one kind of time in this paper.

An important feature about a temporal document is that it is *compact* in the sense that each edit results in only a small, localized change to the document (basically, a new version is created within an item). The difference between two versions can also be compactly represented. Figure 6 shows the difference between the second and third versions of the document. The difference is that a new version of the medal item was created. The ability to represent the difference between two versions in isolation from the rest of the document is useful in data streaming and refreshing data from a remote source, since usually the difference is smaller than the entire document, reducing the cost of the refresh.

```
<docTimeVarying>
  <medalItem itemId="2">
    <medalVersion>
      <time start="2002-07-01" end="now">
      <medal mtype="gold">Men's Combined</medal>
    </medalVersion>
  </medalItem>
</docTimeVarying>
```

Figure 6: The difference between two versions

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 [17]. But it changes the nature of validating against a schema. Assume that the file `winOlympic.xsd` contains the *snapshot schema* for `winter.xml`. The snapshot schema is the schema for an individual version. The snapshot schema is a valuable guide for editing and querying individual versions. A fragment of the schema is given in Figure 7. Note that the schema describes the structure of the fragment shown in Figure 1, Figure 2, and Figure 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 Figure 4.

```
<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="1"
        maxOccurs="1"/>
      <element name="phone" type="phoneNumType" minOccurs="0"
        maxOccurs="unbounded"/>
    </sequence>
    <attribute name="age" type="nonNegativeInteger" use="required"/>
  </complexType>
</element>
```

Figure 7: An extract from the winOlympic schema

The snapshot 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, e.g., a Customer Relationship Management database for a large company. Thousands of calls and service interactions may be recorded each day. This would lead to a very large number of versions, making it expensive to instantiate and validate each individually. The number of versions is further increased because there can be both valid time and transaction time versions.

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 will call it the *representational schema*. The representational schema will be related to the underlying snapshot schema (Figure 7), and allows the time-varying document to be validated using a conventional XML Schema validator (though not fully, as discussed in the next section).

Temporal XML Schema takes a different approach to validating a temporal XML document. In Temporal XML Schema, a snapshot schema is annotated with additional information to create a temporal schema. The temporal schema describes, at a logical level, which elements can vary over time, and how those elements can change. Figure 7 shows the temporal schema for the running example. The temporal schema includes temporal annotations for both the athlete and medal element type definitions. The annotations are shaded gray in the figure. Section 4 describes the annotations in detail. We present the temporal schema here to emphasize that Temporal XML Schema is fully-upwards compatible with XML Schema, that is, it extends but does not change XML Schema. In Section 7 we show to use the temporal schema to validate a temporal document.

The schema is also important in constructing, evaluating, and optimizing (temporal) queries. Both the person who formulates a query and the database need to know which elements in the document are temporal elements since additional operations, like temporal slicing, are applicable to temporal elements. Thus the schema language should have some capability of designating temporal elements.

Finally, temporal 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, though 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.

Temporal XML Schema also provides temporal constraints. For instance, it might be important to stipulate that an athlete's name cannot change or that the existence and type of a medal may change over time (for instance if the athlete is disqualified). The first constraint implies that only a single version of the athlete's name should be allowed, while the second constraint implies that a medal can have multiple versions with non-contiguous lifetimes. A schema language for a temporal document needs to have some way of specifying and enforcing such constraints. Section 6 discusses how the temporal constraints are checked.

2.2. Moving Beyond XML Schema

Both the snapshot and representational schemas are needed for a time-varying document. The snapshot schema is useful in queries and updates. For example, a current query applies to the version valid now, a current update modifies the data in the current version, creating a new version, and a timeslice query extracts a previous version. All of these apply to a single version of a time-varying document, a version described by the snapshot schema. The representational schema is essential for validation and representation (storage). Many versions are combined into a single temporal document, described by the representational schema.

Unfortunately the XML Schema validator is *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. For instance, if the schema asserts that an element is mandatory (`minOccurs=1`) in the context of another element, there is no way to ensure that the element is in every snapshot since the element's timestamp may indicate that it has a shorter lifetime than its parent (resulting in times during which the element is not there, violating this integrity constraint); XML Schema provides no mechanism for reasoning about the timestamps.

Even though the representational and snapshot schemas are closely related, there are no existing techniques to automatically derive a representational schema from a snapshot 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 whether to timestamp with periods or with *temporal elements* [16], which are sets of non-overlapping periods and which elements should be time-varying. By adopting a tiered approach, where the snapshot XML Schema, temporal 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 snapshot 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, few applications that are unconcerned with representational details would need to be changed.

Finally, 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 snapshot 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.

2.3. Desiderata

In augmenting XML Schema to accommodate time-varying data, we had several goals in mind. At a minimum, the new approach would exhibit the following desirable features.

- 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.

3. Theoretical Framework

This section sketches the process of constructing a schema for a time-varying document from a snapshot 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 (deeply) nested collection of elements, with each element potentially having (text) content and attributes.

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[4]. The timestamp records (part of) the lifetime of an element[5]. 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[6].

[4] The general case is that a timestamp is a collection of periods from multiple temporal dimensions (a multidimensional temporal element).

[5] Physically, there are myriad ways to represent a timestamp. It could be represented as an `<rs:timestamp>` subelement in the content of the timestamped element as is done in the fragment in Figure 4. Or it could be a set of additional attributes in the timestamped element, or it could even be a `<rs:version>` element that wraps the timestamped element.

[6] Note that the lifetime captures only when an element appears in the context of the enclosing elements. The same element can appear in other contexts (enclosed by different elements) but clearly it has a different lifetime in those contexts.

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

$$snp(t, D^T) = D$$

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 *snapshot schema*. Let $vldt(S,D)$ represent 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

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

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

$$\begin{tikzcd} D^T \arrow[r, "vldt^T(S^T{,}D^T)"] \arrow[d, "snp(t{,}\ D^T)"'] & v \arrow[d, "v \Rightarrow w"] \\ D \arrow[r, "vldt(S{,}D)"'] & w \end{tikzcd}$$

Figure 8: Snapshot validation subsumption

Details of the process for constructing a schema for a time-varying document that conforms to the snapshot validation subsumption property from a snapshot schema are available in a technical report by the authors [12].

4. Architecture

In this section we describe the overall architecture of τXSchema and illustrate with an example. Further discussion as to the design decisions related to the τXSchema components is in Section 5. The associated software tools are discussed in Section 7.

The architecture of τXSchema is illustrated in Figure 9. This figure is central to our approach, so we describe it in detail and illustrate it with the example. We note that although the architecture has many components, only those components shaded gray 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 snapshot schema, temporal annotations, and physical annotations being the crucial specifications to be created by the designer.

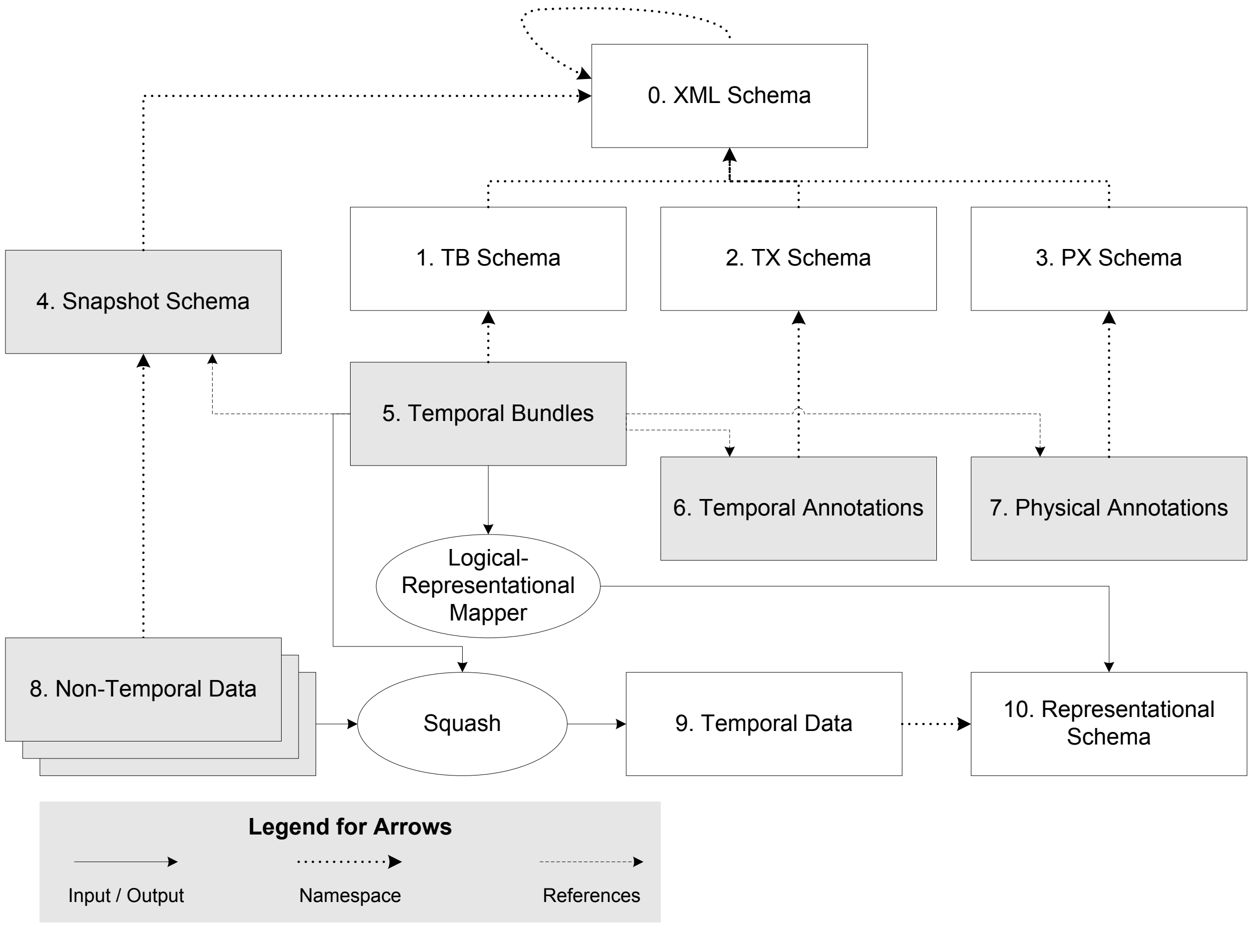

Figure 9: Architecture of τXSchema

The designer annotates the snapshot schema with *temporal annotations* (box 6). The temporal annotations together with the snapshot schema form the *logical* schema.

Figure 10 provides an extract of the temporal annotations on the winOlympic schema. The temporal annotations specify a variety of characteristics such as whether an element or attribute varies over valid time or transaction time, whether its lifetime is described as a continuous state or a single event, whether the 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.

Elements that are not described as time-varying are static and must have the same content and existence across every XML document in box 8. 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 temporal annotations. The schema for the temporal annotations document is given by TXSchema (box 2).

```
<temporalAnnotations
  xmlns="http://www.cs.arizona.edu/tau/tauXSchema/TXSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.cs.arizona.edu/tau/tauXSchema/TXSchema.xsd">
<snapshotSchema schemaLocation="http://www.cs.arizona.edu/
   tau/tauXSchema/examples/schemas/winOlympic.xsd"/>
...
  <validTime target="/winOlympic/…/athlete" kind="state" contentVarying="true">
     <validTime target="@age"/>
     <validTime target="athName"/>
     <validTime target="medal" kind="event"/>
     <validTime target="phone" existenceVarying="true"/>
  </validTime>
...
  <transactionTime target="/winOlympic"/>
  <transactionTime target="/winOlympic/…/athlete/@age"/>
  <transactionTime target="/winOlympic/…/athlete/athName"/>
...
</temporalAnnotations>
```

Figure 10: Sample temporal annotations

The next design step is to create the *physical annotations* (box 7). In general, the physical annotations specify the timestamp representation options chosen by the user. An excerpt of the physical annotations for the winOlympic schema is given in

Figure 11. Physical annotations may also be nested, inheriting the specified attributes from their parent; these values can be overridden in the child element.

```
<physicalAnnotations xmlns= "http://www.cs.arizona.edu/tau/tauXSchema/PXSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.cs.arizona.edu/tau/tauXSchema/PXSchema.xsd">

  <temporalAnnotations schemaLocation="http://www.cs.arizona.edu/
    tau/tauXSchema/examples/schemas/winOlympicTemporal.xml"/>
...
  <stampPosition target="/winOlympic" transactionTimeStampType="step" />
  <stampPosition target="/winOlympic/.../athlete" validTimeStampType="extent">
     <stampPosition target="@age" validTimeStampType="step"
        transactionTimeStampType="step"/>
     <stampPosition target="athName" transactionTimeStampType="step"/>
     <stampPosition target="medal" validTimeStampType="none" />
     <stampPosition target="phone" transactionTimeStampType="extent"/>
  </stampPosition>
...
</physicalAnnotations>
```

Figure 11: Sample physical annotations

Physical annotations play two important roles.

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

The schema for the physical annotations document is PXSchema (box 3). 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.

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

The *temporal bundles* (box 5) tie the schema, temporal annotations and physical annotations together. Each bundle in this document contains sub-elements that associate a specific snapshot schema with temporal and physical annotations, along with the time span during which the association was in effect. The schema for the temporal bundles document is TBSchema (box 1).

At this point, the designer is finished. She has written one conventional XML schema (box 4), specified two sets of annotations (boxes 6 and 7), and provided the linking information via the bundle document (box 5). We provide boxes 1, 2, and 3; XML Schema (box 0) is of course provided by W3C.

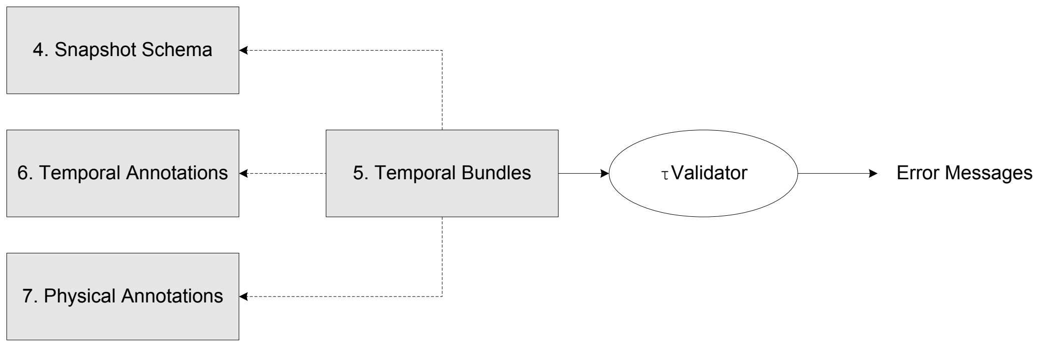

Figure 12: τValidator: Checking the schemas

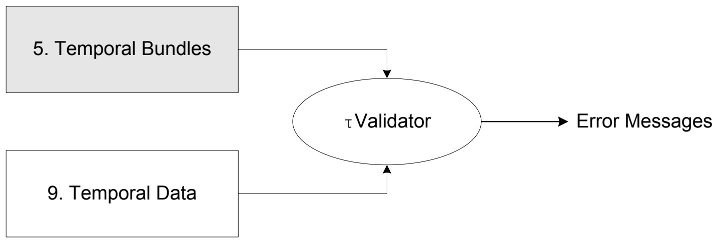

Figure 13: τValidator: Checking the instance

Let's now turn our attention to the tools that operate on these various specifications. The temporal bundles document (box 5) is passed through the τ`Validator` (see Figure 12) which checks to ensure that the temporal and physical annotations are consistent with the snapshot schema. The τ`Validator` utilizes the conventional validator for many of its checks. For instance, it validates the temporal annotations against the TXSchema. But it also checks that the temporal annotations are not inconsistent. Similarly, the physical annotations document is passed through the τ`Validator` to ensure consistency of the physical annotations.

The temporal constraint checker then evaluates the temporal constraints expressed in the schema (oval 9). Finally, the temporal validator reports whether the temporal document was valid or invalid.

Once the annotations are found to be consistent, the *Logical to Representational Mapper* (software oval, Figure 9) generates the *representational schema* (box 10) from the original snapshot schema and the temporal and physical annotations. The representational schema (mentioned in Section 2 as "`rs:`") is needed to serve as the schema for a time-varying document/data (box 9). The time-varying data can be created in four ways: 1) automatically from the non-temporal data (box 8) using τXSchema's `squash` tool (described in Section **Error! Reference source not found.**) 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. Using a conventional XML Schema validating parser is not sufficient due to the limitations of XML Schema in checking temporal constraints. For example, a regular XML Schema validating parser has no way of checking something as basic as "the valid time boundaries of a parent element must encompass those of its child". These types of checks are implemented in the τ`Validator`. So the second step is to pass the temporal data to τ`Validator` as shown in Figure 13. A temporal XML data file (box 9) is essentially a timestamped representation of a sequence of non-temporal XML data files (box 8). The namespace is set to its associated XML Schema document (i.e. representational schema). The timestamps are based on the characteristics defined in the temporal and physical annotations (boxes 6 and 7). The τ`Validator`, by checking the temporal data, effectively checks the non-temporal constraints specified by the snapshot schema simultaneously on all the instances of the non-temporal data (box 8), as well as the constraints *between* snapshots, which cannot be expressed in a snapshot schema.

To reiterate, the conventional approach has the user start with a representational schema (box 10); our proposed approach is to have the user design a snapshot schema and two annotations, with the representational schema automatically generated.

5. Design Goals and Design Decisions

This section summarizes the design goals and decisions of τXSchema. The design goals are explained first to motivate the rest of the section. We then explain SchemaPath as an extension of XPath and show how τXSchema handles all types of XPath nodes. Gluing is then defined and explained. Finally, the canonical representation is described.

5.1. Design Goals

The underlying design goal was to have the existence and content dimensions orthogonal to each other. By making the two dimensions orthogonal, any combination of them between a defining element and its child is allowed. The design does allow all combinations, including a defining element that specifies content as constant with a child that specifies existence as varying.

We also indented for a defining element to specify time-varying independently from its parent. To satisfy this goal, an element's content is designated as only its loose text, attributes and child elements. If the content also included all of an element's descendants, then once a defining element specified content as constant, none of its descendants could specify content as varying, undermining this independence.

Another design decision was to limit the scope of a data element's existence to only within the scope of its parent's existence. If a data element is not present in the instance document, none of its child elements will be present either. This rule defines what happens when a defining element specifies existence as constant and its parent specifies existence as varying.

We chose to keep τXSchema consistent with the XSchema standard. Comments and processing instructions are not validated by XSchema. Hence in τXSchema, comments and processing instructions are not considered part of an elements content. This way, they can always specify content and existence as varying, and do not need to be validated by τXSchema.

Another goal was to have better validation without overcomplicating the language. Tighter restrictions lead to better validation. A useful restriction that we allow is "varying without gaps". Suppose a data element has no gaps in its existence, but does not exist for the entire life of its parent. The corresponding defining element could specify existence as varying, but "varying without gaps" provides a tighter restriction.

Items can be specified by itemrefs, fields and keyrefs. XML Schema already has validation rules for keyrefs. SchemaPath was purposely designed to be very similar to the XPath used for keyrefs. The few differences are explained. Some extra restrictions to SchemaPath based on what it using it.

A key in the snapshot schema is specified by XPath. An item in the temporal annotations is specified with SchemaPath. The item has a target and one or more fields. The key has a selector and one or more fields. τValidator should validate both using the same rules. The conventional validator already has rules for validating selector and field XPath expressions.

5.2. Design Decisions Related to Content and Existence Varying

The data stored in XML documents may change over time. It is useful to be able to validate the way data can change. The XSchema standard provides a way to validate XML documents, but does not define how an XML document is allowed to change with time. To meet this need, τXSchema was created as an extension of the XML standard that validates time-varying XML documents.

The two ways that a node in an XML document can vary with time are (1) in its content or (2) in its existence. 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.

τXSchema introduces the concept of "items." An *item* is a collection of XML elements that represent the same real-world entity. Items are in the temporal bundle and elements are in the temporal document. Since both the temporal bundle and temporal document are written in XML, both contain elements. It is useful to introduce the term "data element" to distinguish elements in the two types of documents. A *data element* is an element in the temporal document. The temporal document can contain all types of XML nodes. It is therefore useful to use the term *data node*, which is a node of any type in the temporal document.

The temporal bundle contains all the item definitions. An item 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.

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 (eg. elements, attributes and loose text) and are explained below.

When an item specifies content or existence as varying, the corresponding data node may vary with time, but is not required to. First we examine a language for specifying the elements that compose an item. Then we explore how the time-varying aspects of a defining element affect how the corresponding data node and its content are allowed to vary with time. XML has seven types of nodes: root, element, attribute, comment, processing instruction, loose text and namespace. Each is examined in turn in 5.2.2 through 5.2.7.

5.2.1. *SchemaPath*

SchemaPath is a language for locating elements in a snapshot schema. Physical and temporal annotations annotate element definitions in the snapshot schema. Each annotation has a "target" attribute that designates the location of an element in the schema. The value of the target attribute is a SchemaPath expression. SchemaPath is very similar to XPath, but has a different data model and a reduced functionality. XPath's data model is tree-like structure that is created by parsing an instance of a schema, i.e., an XML document, but SchemaPath's data model is a graph that is created by parsing an instance of a schema. The data model is created as follows. Each element and attribute definition is a node in the graph. A "child" edge is added from a node to each node that represents a possible sub-element of the node. There is also a special "attribute" edge from a node to each attribute for that node.

SchemaPath expressions, like XPath expressions, are composed of a number of *steps*. Each step consists of an *axis* and a *node test* (again, like XPath, with the exception that predicates are not supported). SchemaPath supports only three axes: `parent`, `child` and `attribute` (unlike XPath which supports many; in particular there is no `descendent` axis, or the "any element wild-card" axis that explores non-neighbors of the context node in the graph). The only legal node test is "`name(`*X*`)`" which tests whether the node has the name *X*. The abbreviated syntax '`.`' may be used to specify the current context node.

SchemaPath does allow two wildcards, the '`*`' to select all elements and the '`|`' union operation. Schemas can be recursive. Using '`*`' and '`|`' in combination provides a way to specify elements in a recursive schema that is more specific than the XPath "`//`" wildcard. So, the expression "`C | */*/C | */*/*/*/C`" could specify the same elements as "`//C`". The '`*`' may not be in the final step or be the entire expression. The union operation is only be allowed if the final labels match.

SchemaPath expressions are evaluated exactly like XPath expressions. Each step is evaluated with respect to a context node. For instance the expression "`/child::name(winolympics)`" locates the `winolympics` child relative to the schema root. SchemaPath has an abbreviated syntax similar to XPath, so the above expression can be succinctly composed as "`/winolympics`". As another example, the expression "`attribute::name(age)`", which locates the age attribute of the current node, can be abbreviated as "`@age`". Other SchemaPath examples can be found in the annotations documents in Figure 10 and Figure 11.

5.2.2. Comment and Processing instruction Nodes

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. XSchema does not validate comments and processing instructions. Hence τXSchema does not validate them either. 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 `timeVaryingDocument` that wraps the entire temporal document. Any comments or processing instructions that appear outside of the roots of individual snapshots will be wrapped by `timeVaryingDocument`.

5.2.3. Loose Text Nodes

Loose text is the simplest type of content. In XML, loose text must always be contained in an element. The existence of loose text is determined by the existence of its enclosing data element. The loose text can not exist when the enclosing data element does not exist. When an item specifies content as varying, the loose text in the corresponding data element may change to any permitted value. The loose text may also go away in one snapshot and return in a later one. For a item that specifies content as constant, the loose text in the corresponding data element must remain constant over time.

A data element can have loose text distributed between its child elements. If the distribution of the loose text changes, then the content is considered to have changed. The content is different even if the concatenated value is the same. Our semantics are to keep the text at each level as separate content. A data node's content is only its own loose text, not the loose text of its children.

5.2.4. Element Nodes

The most interesting type of node is a data element. 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, as specified in the design goals of 5.1. 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.

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 affect on the child. However, 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:

1. Any item so designated must exist in every document snapshot.
2. 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.
3. The third option is like the second, but the child cannot switch parents.

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 nodes A, B and C. Nodes A, B and C are designated as items in the temporal annotation for all three tables and all use attribute n as their item identifier.

The examples vary the values of existence for Bob and Bob's parent to highlight the differences between the three possibilities. The A node, IBM, is constant within every example. Table 1 shows snapshots when the B node is constant. Table 2 has B nodes 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 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 parent. There are cells where it does not matter if Bob exists or not. 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 parents 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.

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**

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="p1"> <C n="Bob"> </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"

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.

5.2.5. Root Node

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.

5.2.6. Attributes Nodes

The next type of node is an attribute. Attributes can vary over time but can not 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 XSchema constraints. The second is by specifying how the enclosing data element may vary with time.

Current XSchema lets attributes 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 can not be both fixed and required/prohibited. The only two things that can't be specified with conventional XSchema are existence as "varying with gaps" and both existence as constant and content as constant.

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.

5.2.7. Namespace Nodes

A temporal document may use element types that are defined in schemas from multiple namespaces. Namespace nodes indicate the namespaces that are used. A schema specifies the namespaces that can be used. The namespace for each data element must be specified in the schema, which is part of the temporal bundle. Thus, changing a namespace in the temporal document also requires a change to the temporal bundle. A namespace node can only change if the schema is allowed to change. When the temporal bundle is not allowed to change, a namespace node must specify both content as constant and existence as constant.

5.3. Support for Multiple Snapshot Schema Documents

There are two possible ways to support documents that reference multiple snapshot schemas via namespaces. The first is to annotate each snapshot schema using a temporal bundle element and allow the temporal document to reference multiple temporal bundle elements instead of snapshot schemas. This would involve creating multiple sets of annotations – one for each schema referenced by the document.

The second is to annotate all the snapshot schemas referenced by the document using a single temporal bundle element, and allow the temporal document to reference this bundle element instead of the snapshot schemas. To disambiguate between targets in different namespaces, the annotations would use SchemaPath expressions.

We chose the first approach since it is a top down approach. It does not assume that the schema documents know anything about the instance documents. The second approach would involve a bottom up approach, since the annotations would need to know about all the namespaces used by the instance document.

5.4. Gluing

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 means that the element is the "same" in both snapshots. We will sometimes refer to the process of associating a pair of elements as *gluing* the elements. When a pair of elements is glued, an *item* is created. An item is an element that evolves over time through various versions. An Item can be composed of any number of elements. Several elements that compose the same item may exist in the same snapshot document.

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 identify and *temporally-associate* elements in different snapshots of a temporal XML document. A temporal association between the elements means that the element is the "same" in both snapshots. We will sometimes refer to the process of associating a pair of elements as *gluing* the elements. When a pair of elements is glued, an *item* is created. An item is an element that evolves over time through various *versions*.

5.4.1. Items

Only temporal elements (that is, elements of types that have a temporal annotation) are candidates for gluing. Determining which pairs should be glued depends on two factors: the *type* of the element, and the *item identifier* for the element's type.

The type of an element is the element's definition in the schema. Only elements of the same type can be glued (when the type changes, for instance when a schema evolves, a different, related process called *bridging* is used to create a temporal-association, which is a topic of future work). In this section, the type of an element will be denoted as $\mathcal{T}$.

An item identifier serves to semantically identify elements of a particular type. The identifier is a list of XPath expressions (much like a key in XML Schema) so we first define what it means to evaluate an XPath expression.

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

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

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

A schema designer specifies item identifiers. As an example, a designer might specify the following item identifiers for the temporal elements in Figure 4.

- `<athlete>` → [`athName/*`]
- `<medal>` → [`../athName/*, ./*`]

The item identifier for an `<athlete>` is the name of the athlete, while the item identifier for a `<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.

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 $\mathcal{T}$ in the i^{th} snapshot of a temporal document. Let y be an element of type $\mathcal{T}$ in the j^{th} snapshot of the document. Finally let L be the item identifier for elements of type $\mathcal{T}$. 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 $\mathcal{T}$ 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 Figure 1 is temporally associated to the athlete element in Figure 2 but not the athlete element in Figure 3 (though the athlete element in Figure 2 is temporally related to the one in Figure 3).

5.4.2. Versions

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

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

- Their parents are the same item or their parents are non-temporal elements.
- They have the same number of children.
- For each child that is a temporal element, the child is the same item as the corresponding child of the other (in a lexical ordering of the children).
- For each child that is something other than a temporal element the child is the same *value* as the corresponding child of the other (in a lexical ordering of the children).
- They have the same set of attributes (an attribute is a name, value pair).

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

Definition [Version] Let x be an item of type $\mathcal{T}$ in a temporal document, with a lifetime that ends at time t. Let y be an element of type $\mathcal{T}$ 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. A new version is created when a temporal association is not DOM equivalent.

Figure 14 depicts an example of items and versions. 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 region shaded gray. There is one athlete item and one medal item. The athlete item has two versions; the transition between versions is shown as a wide diagonal pattern in the area shaded gray.

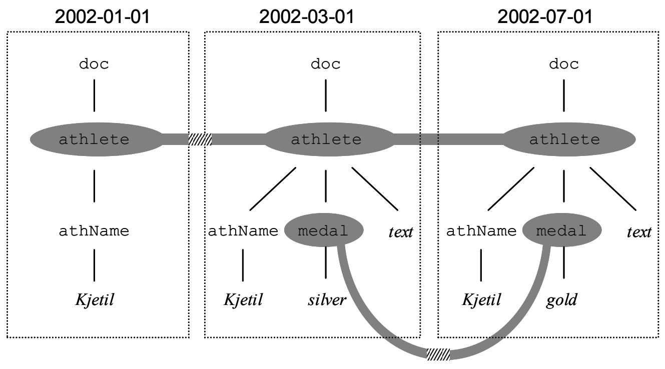

Figure 14: Items and versions in the example

Any pair of elements is a candidate for gluing. Determining which pairs should be glued depends on two factors: the *type* of the elements, and the type's *item identifier*. The type of an element is the element's definition in the schema. Only temporal elements (that is, elements that have a temporal annotation) are candidates for gluing. Only elements of the same type can be glued. Two elements of the same type are glued if their item identifiers evaluate to the same value, creating an item. An item can be subsequently glued to an element in another snapshot, using the same criteria outlined above. Note that only elements with item identifiers are glued. Elements with no item identifier are assumed to be a different instance in each snapshot and may therefore always vary in existence. Elements are the only nodes that may be specified as items.

When a pair of elements is glued, the elements might or might not be *DOM equivalent*. DOM equivalence in a temporal XML context is akin to value equivalence in a temporal relational database context. A pair of elements is DOM equivalent if the pair meets the following conditions.

- The parent of one is the same item as the parent of the other.
- They have the same number of children.
- Each child of one is the same item as the corresponding child of the other (in a lexical ordering of the children).
- They have the same number of attributes. Each attribute of one element is an attribute of the other element, with exactly the same value.

DOM equivalence is important in determining versions of an item. A new version is created when elements in successive snapshots that belong to the same item are not DOM equivalent. 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 tauXSchema constraint).

Item identifiers, play a similar role to keys, or really, temporal keys in that they are used for identification. Unlike keys however, item identifiers are not constraints, rather they are helpful tools in the complex process of computing versions. An item identifier has a target and at least one field, itemref or keyref. A *target* is an SchemaPath expression that specifies an element's location in the snapshots. A *field, itemref* and a *keyref* each specify part of an item identifier. A field contains a *path*, a SchemaPath 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 SchemaPath expressions. The item identifier may consist of any combination of field(s), itemref(s) and keyref(s).

The SchemaPath context node for a target is the root node. The SchemaPath context node for each field is the node designated by the target. The abbreviated syntax '`.`' can be used to specify the current context node. Thus `B/C/D` and "`./B/C/D`" are equivalent. The leading "`./`" is unnecessary, but may be used.

A target and field are each specified with a slightly different subset of SchemaPath. A field may not use '`|`' since a field specifies only one attribute or element. A target may not use the attribute axis. The parent axis is only allowed in fields.

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 element does not have to be a leaf node. Loose text from the element's child(ren) is not part of the item identifier. A field may specify an element either above or below the step level of the current context node. All fields of an item identifier must be either an element that is a direct descendant or ancestor of the target's context node or an attribute with a parent element that is such an element.

When using an itemref or keyref, the referenced target/selector must always be the same path or a more restrictive path than the item's target. This guarantees that all the fields specified in the referenced item/key exist in the elements that compose the item. Suppose an item has a target of "`*/*/C`". It could use an item/key with a target/selector of "`*/B`", `A/B` or "`D/B | A/B`". It could also use a key with selector of `A/B/C`, "`*/B/C`" or "`*/*/C`". The detailed example in section 7.3.4 illustrates all legal SchemaPath expressions.

The target must specify disjoint sets. In other words, an element is a constituent of at most one item. Each target must be unique, but having all unique targets is not sufficient to guarantee disjoint sets. When the final labels of two targets match, the two sets of elements may overlap. For example, "`./J`" and `J` are different strings, but they specify the same elements. A target that uses the '`*`' wildcard is still subject to the uniqueness requirement.

An item identifier must always use at least one of its constituent element's own attributes or loose text. This may be either a field or keyref. It may also be specified in terms of an ancestor or descendant by an itemref or keyref. It may have more than one keyref or itemref. A keyref may be at any step level. An itemref must be at a different step level since no two items can have the same target. Note that an item identifier will contain at most one keyref or itemref at each step level. An item can not be both an `A/B/C` and an `A/B/D`. If it does have more than one reference at the same level, one is redundant. If one reference is `A/B/C` then a second reference of "`*/*/C`" is redundant.

There are a few additional points about keyrefs. A key and item are in separate documents. Thus, when using a keyref, the key must be used in an all or nothing fashion. Allowing keyref to snapshot keys is a good design. Keys are useful in XML documents and will be used regularly. The use of a keyref in an item identifier makes the annotation more understandable, and also allows DBA to change in only one place. Items and keys have a different scope. The scope of an item is the entire document. The scope of a key is its enclosing element within each snapshot.

Since a key's scope is not the whole document, there can be multiple elements within the same snapshot that are part of the same item.

Using itemrefs does not cause circular references. The process of gluing connects elements in the temporal document and all fields resolve to a value. If two items are specified in terms of each other, each instance of their constituent elements will have a value for each of its fields.

5.5. Canonical Representation

In this section we develop a *canonical* (*physical*) *representation* for temporal documents. A single temporal document has many possible physical representations. The choice of a representation is dictated by a temporal document's physical annotations. Among the many physical annotations, some are more conducive than others in representing the *meaning* of a document. The canonical representation is intended to make manifest in the physical representation the temporal semantics of a document. One use of the representation is to compare whether two documents are the same (semantically). If both documents are rendered in the canonical representation they can be physically compared to determine whether they are the same.

Coming up with a canonical representation turns out to be quite involved. We adopted the following desiderata to winnow the candidates.

- Size is unimportant—the temporal document(s) may grow or shrink in size, with respect to the sequence of snapshot documents.
- Versions of a time-varying element must be explicitly represented. The representation must capture the version history by representing both the time-varying element and its versions. The meaning of "version" does not have to be represented, nor do non-temporal elements have to have versions, rather this desiderata is only that there is some representation of each time-varying element and each distinct version.
- A version could have an *N*-dimensional temporal lifetime. In general there could be many temporal dimensions, with one or two being the norm. The versioning could occur in any of the dimensions. A version lifetime is an N-dimensional temporal element, that is, a set of regions in the *N*-dimensional temporal space described by the dimensions.
- The "tree structure" of a document should be retained when possible. The value of retaining the tree structure is that XML parsers, query languages, schema validators, etc. have a better chance of working.
- Whitespace, attributes, text, comments, processing instructions, and sub-elements should be explicitly captured within each version. It should be possible to exactly reconstruct any desired snapshot document (with the exception of information that is discarded by an XML parser such as the ordering of the attributes, whitespace within an element, and empty content tags).
- The representation should not adversely impact the range of queries. This desideratum is somewhat nebulous and fuzzy. The general idea is that the representation should not limit the kind of temporal or non-temporal queries in XPath/XQuery/XSLT/DOM that can be expressed or evaluated. Included in the temporal queries are sequenced queries, and queries that ask for the next/previous

version. Included in the non-temporal queries are queries that navigate within a single snapshot or compare values gathered from multiple snapshots.

- Every copy of a time-varying element and version must have the same information and lifetime (there are no partial versions). If a time-varying element is represented in multiple locations in a temporal document, the element's version history must be the same for every copy.
- The lexical order of versions is important. The order is by transaction-time first, and within transaction-time by valid time, and within valid-time by other time dimensions.
- The representation of a version's lifetime must be unique. Two lifetimes that are the "same" time must have the "same" representation.

The proposed canonical representation has the following features.

- Only elements are time-varying and can have versions. The immediate content, that is 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 if/when any of the following happen.
    - any attribute value changes,
    - an attribute is deleted,
    - an attribute is inserted,
    - the element namespace changes,
    - a subelement is inserted,
    - a subelement is deleted,
    - a subelement changes position,
    - the text content changes, or
    - a comment/processing instruction is inserted/deleted/changes position in the element's content.

The above conditions capture the idea that a version is any change to the element from the previous state of the element. The change must be observable through DOM: only changes observable through DOM create a new version.

- 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.
- The timestamp that represents the version's lifetime is a *N*-dimensional temporal element. It may include now, until changed, and/or indeterminate times. A user-supplied function, `canonicalTS`, produces the canonical representation of the timestamp itself. For example, `canonicalTS` could produce a list of coordinates in N-dimensional space starting with the minimal point in the following dimension order: transaction time, then valid time other dimensions.

- A `canonical:` namespace is used to identify generic components in the canonical representation (note that there are none at present).
- A `timeVary:` namespace is used to identify time-varying components in the canonical representation.
- An `ident:` namespace is used to identify ID and IDREF components in the canonical representation.
- A `timeVary:`*X* represents a time-varying <*X*> element. The successive versions of the element are enclosed within `<canonical:`*X*`-version>` elements within the `<timeVary:`*X*`>` element.
- Schema changes to an element are accommodated by pre-pending the schema version number to the element, e.g., `<timeVary:1-company>` represents the `<company>` element in the first version of the schema.
- Namespace are accommodated by pre-pending the namespace to the element, e.g., `<a:company>` in version 1 of the schema would be captured in the `<timeVary:a-1-company>` element in the canonical representation.
- The representation does not state nor capture how the versioning is done (i.e., were adjacent value-equivalent snapshots coalesced and called a version?).
- No references to versions are introduced. All versions are explicitly represented in full.
- The representation is not the most compact possible.
- ID and IDREFs are placed into an `ident:` namespace that stipulates that both are of type integer (effectively turning off the validation check that only one element can have the given ID in the document).
- The timestamp format is Unix (Note: for clarity in the example below we have the timestamp as Monday, Tuesday, etc., but in a real canonical document it would be Unix).

5.6. Support for Multiple Representations of Time

Stepwise constant is a particularly space-efficient representation for valid and transaction timestamps. It assumes that the end time of one timestamp is the beginning for another. However, as may be expected, one cannot use this in the case where there exist “holes” in the timestamp duration, or overlaps. Extent time stamping does not have this limitation. An example using extent timestamping is given in Figure 15. In comparison, the same information using stepwise constant timestamping is given in Figure 16. Stepwise constant timestamping results in a slightly smaller representation.

```
<phone>
   <rs:timestamp vtbegin="2002-02-19"
   vtend="2002-02-20" ttStart="2002-01-16"
   ttStop="2002-02-15" id="2"
   xsi:type="rs:vtExtentTtExtent"/>
    <value>520-621-1111</value>
</phone>
<phone>
   <rs:timestamp vtbegin="2002-02-20"
   vtend="forever" ttStart="2002-01-16"
   ttStop="2002-02-15" id="2"
   xsi:type="rs:vtExtentTtExtent" />
   <value>520-621-1211</value>
</phone>
```

Figure 15: An example of extent timestamping

```
<phone>
  <rs:timestamp vtbegin="2002-02-19"
  ttStart="2002-01-16" id="2"
  xsi:type="rs:vtStepTtStep"/>
  <value>520-621-1111</value>
</phone>
<phone>
  <rs:timestamp vtbegin="2002-02-20"
  ttStart="2002-01-16" id="2"
  xsi:type="rs:vtStepTtStep />
  <value>520-621-1211</value>
</phone>
```

Figure 16: An example of stepwise constant timestamping

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.

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

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

6.1. XML Schema Constraints

The representational schema is created by mapping the constraints from the non-temporal schema and physical annotation. The goal is to have a mapping that can take maximum use of the regular validator. The regular validator can check most of the XSchema constraints, but not all. This section explains the mapping process, lists the constraints that can be checked by the regular validator and explains how `τValidator` validates the remaining constraints.

The way the mapping works is as follows. For each timestamp, the mapper adds a wrapper directly above the data element corresponding to the timestamp. The following example shows how the insertion occurs. The timestamp is at B. All elements have the default of minOccurs=1 and maxOccurs=1.

```
<A>
  <B>
    <C />
  </B>
</A>
```

Monday.xml

```
<A>
  <B>
    <C />
  </B>
</A>
```

Tuesday.xml

```
<A>
  <Bversion>
    <B>
      <C />
    </B>
  </Bversion>
  <Bversion>
    <B>
      <C />
    </B>
  </Bversion>
</A>
```

Temporal Document

Figure 17: Simple Mapping

Each nontemporal document has a different version of B. Each version of B contains exactly one C. In the temporal document, A contains one of each version of B. Each version of B contains on C element.

There can be timestamps at multiple levels in a document. Also, minOccurs and maxOccurs at each level are not necessarily always 1. The method of adding an enclosing version element to each timestamped element extends to these cases. In the following example, both B and D have timestamps. All elements have a minOccurs=0 and maxOccurs=2.

```
<A>
  <B>
    <C>
      <D>
        <E />
      </D>
    </C>
  </B>
  <B>
    <C>
    </C>
  </B>
</A>
```

Monday.xml

```
<A>
  <B>
    <C>
      <D>
        <E />
        <E />
      </D>
    </C>
    <C>
      <D>
        <E>
      </D>
      <D>
        <E />
      </D>
    </C>
  </B>
</A>
```

Tuesday.xml

```
<A>
  <Bversion>
    <B>
      <C>
        <Dversion>
          <D>
            <E />
          </D>
        </Dversion>
      </C>
    </B>
    <B>
      <C>
      </C>
    </B>
  </Bversion>
  <Bversion>
    <B>
      <C>
        <Dversion>
          <D>
            <E />
            <E />
          </D>
        </Dversion>
      </C>
      <C>
        <Dversion>
          <D>
            <E />
          </D>
          <D>
            <E />
          </D>
        </Dversion>
      </C>
    </B>
  </Bversion>
</A>
```

Temporal Document

Figure 18: Multiple Timestamp Mapping

In Monday.xml of Figure 18, `A` contains two `B` elements. The first `Bversion` element in the temporal document contains the two `B` elements for Monday. The version elements mark the beginning and end of each version of a timestamped element. In Tuesday.xml of Figure 18, `A` contains one `B` element. The second `Bversion` contains Tuesday's `B` element.

A schema defines the types and structure of an XML document. Validation of types is unaffected by the mapping. For example, a string is a string at every point in time. The following XSchema constraints can be checked by the regular validator.

- standard datatypes such as string
- derived types
- simple types
- complex types
- element substitution
- simple content
- complex content
- `any` element
- union
- loose text
- attributes

The representational schema must also validate the structure correctly. When there are multiple versions of an element, each must have a valid structure. The regular validator can check the following constraints.

- `sequence`
- `all`
- `choice`
- list

The final XSchema constraint is the cardinality constraint of `minOccurs` and `maxOccurs`. `τValidator` must check this constraint. It does so by starting with the earliest timestamp and scanning forward in time. `τValidator` keeps track of the net number of elements. Each time an element's lifetime begins, the count increases by one. Similarly, when an element's lifetime ends, the count decreases by one. The default value for minOccurs and maxOccurs is 1. The cardinality constraint of all elements must be checked.

There is a special case of the cardinality constraint that can be checked by the regular validator. The regular validator checks `minOccurs` and `maxOccurs` constraints relative to the scope of their parent data element. The regular validator can validate cardinality for all elements that are below all timestamps.

6.1.1. Identity Constraints

Identity 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 `<xs:key>`[7] or `<xs:unique>` 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). 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 definition for an XML Schema identity constraint is in Figure 19.

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

Figure 19: Sample Identity Constraint Definition

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

It may seem that the functionality of `key` and `unique` constraints are not very different from the XML 1.0 `ID` definitions (and the equivalent `ID` simple type in XML Schema). The functionality is however different in a number of ways. We highlight the differences in next, and show how XML Schema `key` and `unique` constraints have a number of advantages. The importance of this distinction will also be apparent later when we discuss temporal extensions to the constraints (section 6.2.1).

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.

[7] Note: This assumes the namespace for XML Schema is set to `xs`.

- 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 an order should always be for a valid product follows.

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

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 limit there being from zero to four website URLs for suppliers, the `minOccurs` of `<sURL>` is set to `0`, and the `maxOccurs` to `4`.

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

An example of cardinality definitions in XML Schema follows.

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

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.

```
<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>
```

6.2. Temporal Augmentations to the XML Schema Constraints

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

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

Given a snapshot XML Schema constraint, we define the corresponding temporal semantics in τXSchema in terms of a *sequenced constraint*. For example, a snapshot (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 every point in time*."

Non-sequenced constraints are not defined based on snapshot XML Schema equivalents. An example of a non-sequenced (cardinality) constraint is: "There should be no more than ten website URLs for each supplier *in any year*."

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

We now proceed to discuss temporal enhancements to each of the XML Schema constraints. 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

Snapshot 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 **Error! Reference source not found.** 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 versions of these constraints, but do not consider non-sequenced extensions to `ID` types. If an element or attribute in an XML Schema document is said to have a type of `ID`, then that only translates to a sequenced constraint.

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

A time-invariant restriction specifies that the value of the given snapshot `unique` / `key` constraint should not change over time. Without this restriction, snapshot `unique` / `key` constraints simply say that the values must not have duplicates. However, this does not preclude the values from changing as long as the new value does not appear elsewhere in the snapshot XML document. For example, given the key definition for product in Figure 19, the following snippet reflects a perfectly legal change from state **A** to state **B** for the `productNo` attribute of the first `<product>` element:

```
...
<product productNo="500">
 <name>15" LCD, Model 350</name>
 <qtyOnHand>25</qtyOnHand>
</product>
<product productNo="501">
 <name>17" LCD, Model 370</name>
 <qtyOnHand>10</qtyOnHand>
</product>
...
```

A

```
...
<product productNo="599">
 <name>15" LCD, Model 350</name>
 <qtyOnHand>25</qtyOnHand>
</product>
<product productNo="501">
 <name>17" LCD, Model 370</name>
 <qtyOnHand>10</qtyOnHand>
</product>
...
```

B

Figure 20: Extracts from Snapshot XML Documents

However, if the `productNo` is declared as time invariant, then no change can be made to its value. A new `productNo` value indicates an instance of a new product.

As seen in the previous example, snapshot identity constraints do not necessarily imply a *non-sequenced identity constraint* (i.e., a value that can uniquely identify the particular element or attribute across time). Thus, the same `productNo` (a snapshot key) can be changed between snapshots (as long as it remains unique) and re-used for another product. Additionally, we may end up with two or more products with the same product number over time. Therefore we avoid terming the non-sequenced constraints as "temporal identity constraints," and instead use the terms *non-sequenced unique* or *non-sequenced key* constraints as applicable (with the optional qualifier of *time-invariant*). When writing the non-sequenced restrictions in the temporal annotations, we use a newly introduced element `<uniqueConstraint>` (since a key constraint is a special kind of unique constraint).

A non-sequenced `unique` constraint indicates that the unique value should not be re-used at a later time. We further make a distinction between the default semantics for `unique` and the semantics of `uniqueNullRestricted`. Since the XML Schema definition of unique allows a NULL value at each point in time, the default semantics for `unique` allows for multiple NULL values across time (one in each snapshot). A non-sequenced `uniqueNullRestricted` constraint restricts the appearance of the number of NULL values by allowing the user to specify a finite number (one or more) across time; the default number being one. Setting the number of nulls allowed across time to 0 is equivalent to specifying a non-sequenced `key` constraint.

A non-sequenced `key` constraint, as might be expected, disallows duplicate values across time for the key field combinations. NULL values are not permitted in any of the key fields at any time.

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

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

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

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

`type`: The type is one of `key`, `unique` or `uniqueNullRestricted`, the semantics of which have been discussed previously. By default, the type is the same as that of the snapshot constraint (i.e., `key` or `unique`). In the event a new constraint is defined, the usage of this attribute (`type`) is required. Further, an existing snapshot `unique` constraint can be restricted to be a non-sequenced `key` (i.e., no null values allowed), "for a given period of applicability". The converse is not meaningful since coercing a snapshot `key` constraint to `unique` in a non-sequenced context would violate the `key` constraint during some snapshot state. Snapshot `unique` constraints can also have a type of `uniqueNullRestricted`.

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

`content`: Specifies time-invariance and denotes if the value of the `unique/key` can vary over time. To maintain consistency with the rest of the temporal annotations, this attribute can take on one of two values: `constant` or `varying`. If not specified, the default value is assumed to allow `varying` over time (i.e., time invariance must be explicitly specified). Logically, the temporal annotations corresponding to each of the elements and attributes contained in the `key` / `unique` constraints should imply they do not vary over time. If such a restriction is not made in the temporal annotations document, a warning should be issued (not an error—since the checking of the constraint will automatically handle this).

`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. This allows uniqueness to be specified for an interval, e.g., year. This is useful when, for example, a particular key value should not be re-used for a period of a year. The value then must be "unique over any period of a year". By default the evaluation window is the lifetime of the time varying XML document. The specification of intervals for the time window depends on the datatypes defined by the plugin (see section 0) adopted in the temporal annotations. The plugin determines the format in which dates and times are written. Assuming the plugin uses XML Schema to specify the datatypes for time intervals, we can extend it by creating a new type that is a union of the plugin type and 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.

`applicability`[8]: The applicability of a constraint specifies when it was valid. Thus a key constraint may be enforced between 2002 and 2005. 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`: These two sub-elements have a usage identical to their snapshot XML Schema counterparts are needed to specify a new constraint (i.e., those that were not defined as identifiers in the snapshot schema). If a new constraint is defined, the `snapshotIdentifier` attribute should not be used. A new constraint can be either defined as either a `key` or a `unique` constraint.

[8] Applicability applies to the valid time dimension. Transaction time applicability would be when a constraint exists in the schema document.

Term	Definition	Cardinality
`name`	The name of the constraint	optional
`snapshotIdentifier`	The referenced snapshot identifier	optional
`type`	`key`, `unique` or `uniqueNullRestricted` (default: the snapshot constraint type)	optional
`nullCount`	The number of null values allowable (used only with `uniqueNullRestricted`)	optional
`content`	`constant` or `varying` (default: `varying`)	optional
`dimension`	`validTime`, `transactionTime`, or `bitemporal` (default: `validTime`)	optional
`evaluationWindow`	Time window over which the constraint should be checked (default: lifetime of document)	optional
`applicability (begin, end)`	When the constraint is applicable (default: lifetime of document)	[0:U]
`selector`	For the definition of a new constraint. It is similar to the `selector` sub-element in the conventional XML Schema definition	[0:1]
`field`	For the definition of a new constraint. It is similar to the `field` sub-element in the conventional XML Schema definition	[0:1]

Table 4: Attributes and sub-elements for unique`Constraint`

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

1. *Supplier numbers are keys but may be re-used over time—however the reuse should not occur for at least one year after discontinuation. Supplier numbers are also allowed to change as long as no other supplier has that number. Part numbers on the other hand may not be re-used later, but may change. Order numbers are time-invariant in valid time. The constraints are applicable between 2000 and 2004.*

```
<uniqueConstraint type="key" name="idSupplierNo"
     snapshotIdentifier="supplier_key" dimension="validTime"
     evaluationWindow="year">
   <applicability begin="2000-01-01" end="2004-12-31" />
</uniqueConstraint>
<uniqueConstraint type="key" name="idPartNo" snapshotIdentifier="part_key"
     dimension="validTime" evaluationWindow="lifetime">
   <applicability begin="2000-01-01" end="2004-12-31" />
</uniqueConstraint>
<uniqueConstraint type="key" name="idOrderNo" snapshotIdentifier="order_key"
     content="constant" dimension="validTime" evaluationWindow="lifetime">
   <applicability begin="2000-01-01" end="2004-12-31" />
</uniqueConstraint>
```

2. *A time invariant key (in both valid and transaction time) is to be defined for products—its RFID number. The constraint is applicable from 2004 onwards.*

```
<uniqueConstraint idType="key" name="product_RFID" dimension="bitemporal"
     content="constant" evaluationWindow="lifetime">
   <applicability begin="2004-01-01" />
   <selector xpath="product"/>
   <field xpath="@RFID"/>
</uniqueConstraint>
```

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

```
<uniqueConstraint idType="unique" name="employee_email"
     evaluationWindow="two-years" />
```

6.2.2. Referential Integrity Constraints

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

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

One might think that there may be a limitation for referential integrity constraints between *state* data and *event* data. However, 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 temporal annotations. We now proceed to discuss the different attributes and sub-elements for the `nonSeqKeyref` (summarized in Table 5; 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 an identifier (either snapshot or non-sequenced) that the non-sequenced referential integrity constraint refers to. The usage is identical to its XML Schema `keyref` definition counterpart.

`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 snapshot XML Schema counterparts are needed to specify a new constraint (i.e., those that were not defined as referential integrity constraints in the snapshot schema).

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

Term	Definition	Cardinality
`name`	The name of the constraint	optional
`refer`	The referenced identifier	optional
`applicability`	When the constraint is applicable (default: 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	[0:1]

Table 5: Attributes and sub-elements for `nonSeqKeyref`

4. *A non-sequenced referential integrity constraint is to be defined for the product number in orders. It should reference a valid part number. The corresponding unique constraint is defined on parts as* `idPartNo`*. The constraint is applicable from 1990-2004.*

```
<nonSeqKeyref name="orders_productNo" refer="idPartNo">
   <applicability begin="1990-01-01" end="2004-12-31" />
</nonSeqKeyref >
```

6.2.3. Cardinality Constraints

As discussed in section 6.1.3, the cardinality of elements in snapshot 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. For example, a non-sequenced constraint can be used to place a limit of one hundred orders from a supplier in any given month. We can also extend the functionality of snapshot cardinality constraints by allowing for user-defined aggregation levels. For example, a restriction can be placed on the total number of orders placed by the company, in addition to the restriction on the number of orders from any given supplier. We believe this functionality is useful and allow its specification at a sequenced level even though XML Schema does not provide for this in snapshot documents.

We represent temporal cardinality constraints using a `cardConstraint` element in the temporal annotations document. We now proceed to discuss the different attributes and sub-elements for the `cardConstraint` (summarized in Table 6; the sub-elements are indented).

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

`target`: A schema path expression to select the element or attribute being constrained.

`restriction`: Cardinality constraints can restrict the `existence`, `content`, `distinctContent`, or `distinctExistence` of elements and attributes. `Existence` refers to the actual number of sub-elements that can appear over time, and is analogous to the snapshot `minOccurs` and `maxOccurs` for sequenced constraints. Restricting the `content`, limits the min/max number of "content values" that an element or attribute can have over a specific period of time. It is related to the datatype of the item (which specifies the maximum number of possible values over all time). The difference between `content` and `distinctContent` is akin to the SQL notion of distinct. 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 quarter, it can potentially have seven values. However the number of *distinct* values that `status` can have is five or fewer. The difference between `existence` and `distinctExistence` is similar, in that duplicate sub-elements are not counted for `distinctExistence`. Duplication is determined using by referencing an applicable uniqueness constraint (which in terms specifies the fields to be evaluated).

`uniqueRef`: Used along with `distinctExistence` 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 / maximum 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. The specification of intervals in the time window depends on the plugin type adopted in the temporal annotations.

`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. Sequenced constraints can only be placed on entities (or attributes) that are defined as states (i.e., not events).

`aggregationLevel`: 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.

`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.

Term	Definition	Cardinality
`name`	Name of the constraint	optional
`target`	Schema path target	required
`restriction`	One of: `existence`, `content`, `distinctContent`, `distinctExistence`	required
`uniqueRef`	Reference to a uniqueness constraint—only used with `distinctExistence`	optional
`dimension`	`validTime`, `transactionTime` (default: `validTime`)	optional
`evaluationWindow`	Time window over which the constraint should be checked (default: lifetime of document)	optional
`sequenced`	If it is a sequenced constraint (default: `false`)	optional
`aggregationLevel`	The level at which the aggregation is performed (default: parent level)	optional
`min`	`minOccurs` equivalent (default: `0`)	optional
`max`	`maxOccurs` equivalent (default: `unbounded`)	optional
`applicability`	When the constraint is applicable (default: lifetime of the document)	[0:U]

Table 6: Attributes and sub-elements for `cardConstraint`

5. *There should be no more than fifty active suppliers (i.e., in the "database") in any year. This constraint is true between 2002 and 2004. [`Existence` constraint]*

```
<cardConstraint name="supplierCardYear" target="company/supplier"
    restriction="existence" dimension="validTime" evaluationWindow="year"
    max="50">
   <applicability begin="2002-01-01" end="2004-12-31" />
</cardConstraint>
```

6. *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 2003-2005. [`Content` & `distinctContent` constraints; different evaluation window sizes used]*

```
<cardConstraint name="prodNameMonth" target="product/@productNo"
    restriction="content" dimension="validTime" evaluationWindow="month"
    min="1" max="1">
   <applicability begin="2003-01-01" end="2005-12-31" />
</cardConstraint>
```

```
<cardConstraint name="prodNameYear" target="product/@productNo"
     restriction="distinctContent" dimension="validTime"
     evaluationWindow="year" min="1" max="3">
   <applicability begin="2003-01-01" end="2005-12-31" />
</cardConstraint>
```

7. *No supplier should be given more than one hundred orders a 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 covered by minOccurs/maxOccurs). [Different aggregation levels]*

```
<cardConstraint name="supOrders" target="company/supplier/order"
     restriction="existence" dimension="validTime" evaluationWindow="month"
     max="100"/>
<cardConstraint name="supParts" target="company/supplier/order/product"
     restriction="existence" dimension="validTime" evaluationWindow="month"
     aggregationLevel="company/supplier" max="500" />
```

8. *There should be a maximum of 250 potential suppliers for the company across all products. We assume there exists a unique constraint on the potential supplier's `supplierNo` attribute. This constraint is to be enforced during 2004. [Sequenced constraint; use of `distinctExistence`]*

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

```
<cardConstraint name="potential_suppliers_seq"
     target="company/product/potential_supplier"
     restriction="distinctExistence" uniqueRef="potential_supplierNo"
     dimension="validTime" sequenced="true" aggregationLevel="company"
     max="250">
   <applicability begin="2004-01-01" end="2004-12-31" />
</cardConstraint>
```

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 snapshot 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 snapshot level.

We now proceed to discuss the different attributes and sub-elements for the `transitionConstraint` (summarized in Table 7; the sub-elements are indented).

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

`target`: A schema path expression to select the element or attribute being constrained.

`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.

`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 snapshot `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 snapshot `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 / 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 2004 and June 2004".

`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.

Term	Definition	Cardinality
`name`	Name of the constraint	optional
`target`	Schema path target	required
`dimension`	`validTime` or `transactionTime` (default: `validTime`)	
`valuePair`	Sub-element listing possible pairs for discrete changes	
`old, new`	Sub-elements of `valuePair`	
`valueEvolution`	Sub-element listing direction of continuous changes	
`applicability`	When the constraint is applicable (default: lifetime of document)	optional

Table 7: Attributes and sub-elements for `transitionConstraint`

9. *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 2003 and 2005.*

```
<transitionConstraint name="supplierRating" target="supplierRatingType"
   dimension="validTime">
   <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="2003-01-01" end="2005-12-31"/>
</transitionConstraint>
```

10. *Employee Salaries should not go down, but may increase between 2003 and 2004. However, a salary freeze is in place between March and June 2004.*

```
<transitionConstraint name="employeeSalary1" target="employeeSalaryType"
   dimension="validTime">
   <valueEvolution direction=">=" />
   <applicability begin="2003-01-01" end="2004-12-31"/>
</transitionConstraint>

<transitionConstraint name="employeeSalary2" target="employeeSalaryType"
   dimension="validTime">
   <valueEvolution direction="=" />
   <applicability begin="2004-03-01" end="2004-06-30"/>
</transitionConstraint>
```

7. Tools

Our three-level schema specification approach enables a suite of tools operating both on the schemas and the data they describe. The tools are open-source and beta versions are available[9]. The tools were implemented in Java using the DOM API [34]. This section gives an overview of the suite of tools. First, the logical to representational mapper is introduced. This tool takes as input the non-temporal schema, temporal and physical annotations and automatically generates the representational schema. Next, temporal schema validation is explained. Before the data can be validated, the snapshot schema and the temporal and physical annotations must all be consistent. Temporal data validation is a several step process. A major part of this process is gluing elements to form items. Other tools in the suite are for squishing, unsquishing and resquishing. The section concludes with a brief discussion of the performance of the tools.

7.1. Logical to Representational Mapper

Once the annotations are found to be consistent, the *Logical to Representational Mapper* (software oval, Figure 9) generates the *representational schema* (box 10) from the original snapshot schema and the temporal and physical annotations. The representational schema (mentioned in Section 2 as "`rs:`") is needed to serve as the schema for a time-varying document/data (box 9). The time-varying data can be created in four ways: 1) automatically from the non-temporal data (box 8) using τXSchema's `squash` tool (described in our technical report [9]) 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 logical to representational `mapper` (software oval of Figure 9) takes as input the non-temporal schema, temporal and physical annotations and automatically generates the representational schema. Briefly, every time-varying element is given a timestamp for valid time and/or transaction time as appropriate. Each time-varying attribute is represented by a sub-element that contains both the timestamp and the attribute name and value. Non-temporal elements and attributes are translated as is. We present a snippet (Figure 21) that illustrates the key concepts in the translation (the reader is referred to Figure 7 for an extract of the non-temporal schema,

Figure 10 for a fragment of the temporal annotations, and

Figure 11 for an extract of the physical annotations).

The temporal annotations specify that `<athlete>` is content-varying and its kind is `state`. The physical annotations indicate that the timestamp is also defined at the level of the `<athlete>` element and that the `validTimeStampType` (timestamp for valid time) specified is `extent`. The representational schema captures this information by inserting an `<rs:timestamp>` element inside the schema definition for `<athlete>`.

By default, when non-leaf elements are specified as temporal, they are assumed to be `existenceVarying`, i.e., the existence of the element can change, but the content cannot change. Leaf elements are assumed to be `contentVarying` by default.

[9] `http://www.cs.arizona.edu/tau/txschema/` and `http://www.cs.arizona.edu/tau/tdom/`

existenceVarying cannot be derived from minOccurs and maxOccurs in the non-temporal schema. An element can be defined as existenceVarying even if the minOccurs is 1 (and maxOccurs=1). For example, it is possible that an athlete has exactly one phone but the athlete may change phones during the Olympics. Even though this example situation is existence-varying, it would not violate the minOccurs=1 condition. contentVarying is orthogonal to existenceVarying and it indicates whether the content of the element or attribute can change. For non-leaf elements, we have used this to mean time-varying loose text can be present and sub-elements may be existence-varying.

```
...
<element name="athlete">
  <complexType mixed="true">
    <sequence>
      <element ref="rs:timestamp"/>
      <element ref="rs:timeVaryingAttribute"
        minOccurs="1" maxOccurs="unbounded"/>
      ...
      <element name="birthPlace" type="xs:string"
        minOccurs="1" maxOccurs="1"/>
      <element ref="phone" minOccurs="0"
        maxOccurs="unbounded"/>
      ...
    </sequence>
  </complexType>
</element>
<element name="phone">
  <complexType>
    <sequence>
      <element ref="rs:timestamp"/>
      <element name="value" type="phoneNumType"/>
    </sequence>
  </complexType>
```

Figure 21: An extract from the representational schema

The mapper tool was implemented in Java using the DOM API [11]. Such a transformation assumes that the three schemas had first been validated, as we now describe.

An XML Schema specification defines the type 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.

$S \rightarrow$ <s>α</s>

In the above production, α describes the content 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 a snapshot schema for each temporal element. Several new productions are added to the schema for each temporal element type; 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'll illustrate the process by describing what is done for a single, representative temporal element type, *S*.

The first step is to add a production to indicate that the element type *S* is time varying. The time-varying production has following form:

$S_{TimeVarying} \rightarrow$ <$s_{TimeVarying}$ itemRef="m"/>

where <s$_{TimeVarying}$> denotes a temporal element of type *S* and `itemRef` is a reference to an item of type *S*. Next a production is added to define the *S* item type.

$S_{Item} \rightarrow$ <s$_{Item}$ itemId="*n*"> S_{Verson}+ </s$_{Item}$>

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:

$S_{Version} \rightarrow$ <s$_{Version}$> τ *S* </s$_{Version}$>

where τ is the timestamp's schema 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 imported into the temporal document's schema. Without loss of generality we will assume that each timestamp has the following form.

$\tau \rightarrow$ `<time start="…" end="…"/>`

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

$X \rightarrow$ <x> $\beta\, S\, \gamma$ </x>

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

$X \rightarrow$ <x> $\beta\, S_{TimeVarying}\, \gamma$ </x>

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 augment the document root with an additional production to indicate that the versions and items can be separated from the rest of the document as follows. Let the document root be an element of type *R*. Then the new root becomes the following production.

$R_{TimeVarying} \rightarrow$ <doc$_{TimeVarying}$>
R? X_{Item}*
</doc$_{TimeVarying}$>

where X_{Item} is a list of item types. The production for X_{Item} is given below, where each S^{i}_{Item} is one of *k* item types.

$X_{Item} \rightarrow S^{1}_{Item} \mid \dots \mid S^{k}_{Item}$

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; `idRef`s are similarly rewritten. Finally, schema keys are rewritten to include `itemId`s and version start and end times, creating a temporal key.

Let's go through the construction process with an example. Assume that the productions in the schema for the example fragment in Figure 3 are given below.

$R \rightarrow$ `<doc>` $A+$ `</doc>`
$A \rightarrow$ `<athlete>` $N\ [\ M \mid text\ ]^*$ `</athlete>`
$N \rightarrow$ `<athName>` $text$ `</athName>`
$M \rightarrow$ `<medal type="`$type$`">` $text$ `</medal>`

Next, assume that the `<athlete>` and `<medal>` element types are temporally annotated. The schema would be transformed as follows. First, productions would be added for the time-varying elements.

$A_{TimeVarying} \rightarrow$ `<athlete`$_{TimeVarying}$ `itemRef="`m`"/>`
$M_{TimeVarying} \rightarrow$ `<medal`$_{TimeVarying}$ `itemRef="`m`"/>`

Next, productions are added for the items of temporal elements.

$A_{Item} \rightarrow$ `<athlete`$_{Item}$ `itemId="`n`">` $A_{Version}+$
`</athlete`$_{Item}$`>`

$M_{Item} \rightarrow$ `<medal`$_{Item}$ `itemId="`n`">` $M_{Version}+$
`</medal`$_{Item}$`>`

Productions are then added for each type of version, and for the timestamp(s) within each version.

$A_{Version} \rightarrow$ `<athlete`$_{Version}$`>` $\tau\ A$ `</athlete`$_{Version}$`>`
$M_{Version} \rightarrow$ `<medal`$_{Version}$`>` $\tau\ M$ `</medal`$_{Version}$`>`
$\tau \rightarrow$ `<time start="…" end="…"/>`

Next, the root is modified to include the new productions.

$R_{TimeVarying} \rightarrow$ `<doc`$_{TimeVarying}$`>`
$R?\ [A_{Item} \mid M_{Item}]^*$
`</doc`$_{TimeVarying}$`>`

It is important to note that the production for the temporal document root allows a temporal document to be just a list of items. This enables a temporal document to be incrementally validated, which is important in data streaming applications.

Finally, DTD ids and keys are redefined to remove uniqueness constraints since an individual id or key value could appear in more than one version.

As a concrete example, let's consider the temporal schema in Figure 7. When the temporal schema is converted to a representational schema, the element type definitions listed in Figure 22 are added (and the temporal annotations are removed). Figure 22 is incomplete however, it does not list the definition of the timestamp element type, which we assume is specified elsewhere and imported into the temporal schema, nor the production for the document root, since the fragment in Figure 7 did not define the root (omitted to save space).

```
<element name="athlete$_{timeVarying}$">
  <complexType mixed="false">
    <attribute name="itemRef" type="idRef"/>
  </complexType>
</element>
<element name="athlete$_{item}$">
  <complexType mixed="false">
    <attribute name="itemId" type="id"/>
    <sequence>
      <element name="athlete$_{version}$">
        <complexType mixed="false">
          <sequence>
            <element ref="timestamp"/>
            <element ref="athlete"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
<element name="medal$_{timeVarying}$">
  <complexType mixed="false">
    <attribute name="itemRef" type="idRef"/>
 </complexType>
</element>
<element name="medal$_{item}$">
  <complexType mixed="false">
    <attribute name="itemId" type="id"/>
    <sequence>
      <element name="medal$_{version}$">
        <complexType mixed="false">
          <sequence>
            <element ref="timestamp"/>
            <element ref="medal"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
```

Figure 22: An extract from the temporal winOlympic schema

7.2. Temporal Schema Validation

The logical and physical temporal annotations (**Figure 9**, boxes 6 and 7) for a non-temporal XML Schema (**Figure 9**, box 5) are XML documents and hence can be validated as such. However, a validating XML parser cannot perform all of the necessary checks to ensure that the annotations are correctly specified. For example it cannot check that elements that have a `minOccurs` of 0 do not use a step-wise constant timestamp representation (i.e. a compact representation that assumes continuous existence, and where only the begin/start time of a timestamp is specified and the end/stop time of a timestamp is assumed to be the same as the begin/start point of the succeeding timestamp). This motivates the need for a special validator for the temporal and physical annotations. We implemented a tool, called `τValidator`, to check the annotations. First, `τValidator` validates the temporal and physical annotations against the TXSchema and PXSchema, respectively. Then it performs additional tests to ensure that the snapshot schema and the temporal and physical annotations are all consistent.

`τValidator` can be run in two different modes. The first validates the schema. The second mode validates the data. Two modes are available to provide the user with more flexibility. τValidator could be used as a stand alone application or as part of a managed environment.

7.3. Temporal Data Validation

τValidator also validates time-varying data. A temporal data validator must ensure that every snapshot of the time-varying document conforms to the snapshot schema. It does this, in part, by using an existing XML Schema validating parser to validate the temporal document against the representational schema. τValidator then performs two additional kinds of checks: representational checks and checks to compensate for differences in the semantics of temporal and non-temporal constraints. For instance it needs to check that there are no gaps in the lifetimes of versions for elements that have `minOccurs=1` in the representational schema.

An existing XML Schema validator is used to validate as much as possible. This reduces the complexity of τValidator and avoids unnecessary duplication of functionality. An existing validator can not check constraints for `minOccurs` and `maxOccurs`. τValidator uses a scanning algorithm to validate these constraints. The number of elements at the first timestamp are counted. Each time an element's lifetime begins, the count is incremented. Each time an element's lifetime ends, the count is decremented. If the count exceeds `maxOccurs` or is less than `minOccurs`, τValidator gives an error. This sets an error flag. Once count is within the bounds again, the error flag is cleared. A second error is issued if the count goes outside the bounds after the error flag has been cleared.

τValidator must check the constraints related to items. The following sections describe how to glue and validate items.

7.3.1. Gluing Component

The gluing component is the part of τValidator responsible for gluing item together. The first step in validation is to use the conventional validator to validate the temporal bundle and temporal document. If this is successful, then elements are glued together to form items. Finally, the items are validated. The input to the gluing component is always valid XML, but the gluing component still needs to perform validation. τValidator does not continue past gluing until all gluing errors are fixed. This section explains the validation performed by the gluing component, gives detail on how items are glued and overviews the architecture of the gluing component.

The gluing component needs to validate the temporal document. Every item's target and field(s) must be specified with a valid SchemaPath expressions. All itemrefs and keyrefs must reference a valid item/key. The gluing component checks all targets with matching final labels. If one is the subset of the other τValidator issues an error. The check is done by working back recursivly from the final label. This is a schema level check and is an error even if no instance violates the condition.

The SchemaPath used by τValidator was designed to be very similar to the XPath used by the conventional validator. There are, however, a few differences between the SchemaPath expressions allowed by τValidator and the XPath expressions allowed by the conventional validator. The conventional validator does not allow the parent axis in a key. τValidator allows the parent axis when specifying items. This difference has to remain since we can't make the conventional validator less restrictive. τValidator has a more restrictive usage for '`*`' and '`|`'

than the conventional validator. A key in the snapshot schema must conform to τValidator's stricter rules.

The conventional validator allows special symbols such as '`*`' and '`@`' to be used in element names. Since any element name can potentially be used in a SchemaPath expression to specify an item, τValidator places additional restrictions on the use of all symbols that are special symbols in XPath. No symbol may be used in an element or attribute name in such a way that it could be interpreted with a different meaning when used in a SchemaPath expression. For example, no element name may begin with '`@`'. The list of errors below enumerates all such errors. A snapshot schema that is validated by the conventional validator my still contain τValidator errors.

7.3.2. Possible Errors

The gluing component takes as input the temporal bundle and the temporal document. These are required for gluing and validating the items. As many errors as possible are checked directly from the temporal annotations as this is easiest. The following errors can be determined from only the temporal annotations.

- a target or field uses the '`*`' wildcard as the final label or entire expression
- a target or field uses the "`//`" wildcard
- a target or field uses "`()`" for a function or node test other than `name()`
- a target or field uses "`[]`" for a predicate
- a target is identical to another target
- a target specifies a subset of another target
- a target uses the abbreviated syntax "`..`" to specify the parent axis
- a target uses "`::`" to specify an axis other than `child`
- a target uses the '`|`' union operation and the final labels don't match
- a target uses '`@`' to specify an attribute
- a field uses "`::`" to specify an axis other than `parent`, `child` or `attribute`
- a field uses the '`|`' union operation
- a keyref references an item that does not exist
- two item have identical names

The following errors are checked by referencing the snapshot schema.

- a field violates the direct ancestor/dependant requirement
- a field element has `minOccurs = 0`
- a field attribute is not specified as `required`
- a keyref references a key that does not exist

The following errors occur in the snapshot schema. These are all allowed by the conventional validator but are not allowed by τValidator.

- a snapshot schema key uses '`*`' as the final label or entire expression in either the selector or any field
- a snapshot schema key uses the '`|`' in the selector and the final labels don't match
- a snapshot schema key uses the '`|`' in a field
- an element or attribute name contains '`/`', '`|`', "`[]`", "`()`" or "`::`"
- an element name is '`*`', '`.`' or "`..`"
- an element name is or begins with '`@`'

The gluing component issues the following warnings.

- an item has more than one reference at the same step level
- a field duplicates a field in a referenced item or key

The gluing component does not care about the order that items are listed in the temporal annotations. Items may be reordered in any way without causing an error or warning message.

7.3.3. Creating Items

Once the gluing component is finished with validation, it creates all the items and their item identifiers. Both the schema and the value of each field, itemref and keyref are needed to determine if two elements should be glued. A field my contain the '`.`' to indicate that the current context node's text is part of the item identifier. When a keyref or itemref is used, τValidator substitutes the fields of the referenced key/item into the item identifier. The context node for an item and the referenced key/item do not have to be the same. When the referenced key/item is above the step level of the item, τValidator must prepend one or more "`../`" to the font of each field as it does the substitution. When the referenced key/item is below the step level of the item, tauValidator prepends a path to each field. For example, suppose an item's target was `A/B/C` and used a keyref to a key with a selector of `A/B/C/D` and field of "`@n`". τValidator would substitute a field of "`D/@n`" into the item identifier.

τValidator concatenates all of the fields together. It creates one string that is the schema for all fields and a second string that is the value of all fields. Element and attribute names can not contain the '`|`' symbol so it is used to separate each field string in the concatenated string. The fields are concatenated in the order specified in the item identifier.

τValidator has a data structure to hold all 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 string must be equal. Even the amount and location of white space in a field element's loose text must be identical.

The gluing component is part of τValidator. It creates a data structure that stores all the items. The first step is to parse both the temporal bundle and the temporal document. The gluing component uses the item identifiers from the temporal annotations to create the items as it parses the snapshots. Each item has its own data structure that stores a pointer to each of its instances in the snapshots. The gluing component determines which elements are constituents of which items. For each element that is a constituent of an item, the gluing component determines whether to create a new item or to glue this element to an existing item.

7.3.4. Gluing Example

Let's now examine how the gluing component works with an example. Figure 21 is the temporal annotation for the snapshots that are in Table 8. The items designated in the temporal annotations are glued together. Not all elements are constituents of items. Elements of type `C` are not constituents of an item and are not glued. Note that the example temporal annotations are not complete. The only parts of the temporal annotations that are relevant to gluing are the item target and identifier field(s), itemref(s) and keyref(s). Only these are shown in the example temporal annotation to save space and to avoid unnecessary detail. The example illustrates six important points; each is discussed in turn.

```
<item target="." name="compItem">
  <field path="@n" />
</item>
<item target="B" name="projItem">
  <field path="@n" />
</item>
<item target="D">
  <keyref refName="deptKey" />
</item>
<item target="E" name="ItemE">
  <field path="@n" />
</item>
<item target="*/F">
  <field path="attribute::a" />
</item>
<item target="B/child::G">
  <itemref refName="projItem" />
  <field path="attribute::a" />
</item>
<item target="H | */H">
  <field path="@n" />
</item>
<item target="./J">
  <field path="@n" />
</item>
<item target="D/J">
  <field path="@n" />
  <field path=".." />
</item>
<item target="C/J">
  <field path="@n" />
</item>
<item target="B/K">
  <keyref refName="kKey" />
  <field path="@n" />
  <itemref refName="projItem" />
</item>
<item target="E/L">
  <itemref refName="compItem" />
  <itemref refName="ItemE" />
  <field path="@n" />
</item>
```

Figure 23: Temporal Annotations

```
<xs:key name="deptKey">
  <xs:selector xpath="D" />
  <xs:field xpath="@n" />
</xs:key>
<xs:key name="kKey">
  <xs:selector xpath="B/K" />
  <xs:field xpath="." />
</xs:key>
```

Figure 24: Snapshot Schema

XML Snapshots			
Monday	*Tuesday*	*Wednesday*	*Thursday*
`<A n="IBM"> [1]`	`<A n="IBM"> [1]`	`<A n="IBM"> [1]`	`<A n="IBM"> [1]`
`<B n="proj1"> [2]`	`<B n="proj1"> [2]`	`<B n="proj1"> [2]`	`<B n="proj1"> [2]`
`<F a="1" /> [8]`	`<F a="2" /> [9]`	`<F a="1" />[8]`	`<F a="2" /> [9]`
`<G a="1" /> [20]`	`<G a="2" /> [21]`	`<G a="1" /> [20]`	`<G a="2" /> [21]`
`<K n="1"> [23]`	`<K n="2"> [24]`	`<K n="1"> [23]`	`<K n="2"> [25]`
`foo`	`foo`	`foo`	`bar`
`</K>`	`</K>`	`</K>`	`</K>`
`</B>`	`</B>`	`</B>`	`</B>`
`<B n="proj2"> [3]`	`<B n="proj2"> [3]`	`<B n="proj3"> [11]`	`<B n="proj3"> [11]`
`<F a="1" /> [8]`	`<F a="2" /> [9]`	`<F a="2" /> [9]`	`<F a="1" /> [8]`
`<G a="1" /> [10]`	`<G a="2" /> [12]`	`<G a="2" /> [13]`	`<G a="1" /> [14]`
`<K n="2"> [26]`	`<K n="2"> [26]`	`<K n="1"> [27]`	`<K n="1"> [28]`
`bar`	`bar`	`foo`	`bar`
`</K>`	`</K>`	`</K>`	`</K>`
`</B>`	`</B>`	`</B>`	`</B>`
`<C b="1">`	`<C b="1">`	`<C b="1">`	
`<H n="1" /> [15]`		`<H n="1" /> [15]`	
`<J n="1" /> [16]`		`<J n="1" /> [16]`	
`</C>`	`</C>`	`</C>`	
`<D n="1"> [6]`	`<D n="1"> [6]`	`<D n="1"> [6]`	`<D n="1"> [6]`
`foo`		`foo`	`baz`
`<H n="1" /> [15]`		`<H n="1" /> [15]`	`<H n="1" /> [15]`
`<J n="1" /> [17]`		`<J n="1" /> [17]`	`<J n="1" /> [22]`
`</D>`	`</D>`	`</D>`	`</D>`
`<D n="2"> [7]`	`<D n="2"> [7]`	`<D n="2"> [7]`	`<D n="2"> [7]`
`bar`		`bar`	`baz`
`<H n="1" /> [15]`		`<H n="1" /> [15]`	`<H n="1" /> [15]`
`<J n="1" /> [18]`		`<J n="1" /> [18]`	`<J n="1" /> [22]`
`</D>`	`</D>`	`</D>`	`</D>`
`<H n="1" /> [15]`		`<H n="1" /> [15]`	
`<J n="1" /> [19]`		`<J n="1" /> [19]`	
`<E n="proj1"> [4]`	`<E n="proj2"> [5]`	`<E n="proj1"> [4]`	`<E n="proj2"> [5]`
`<F a="1" /> [8]`	`<F a="2" /> [9]`	`<F a="2" /> [9]`	`<F a="1" /> [8]`
`<G a="1" />`	`<G a="2" />`	`<G a="1" />`	`<G a="2" />`
`<L n="1" /> [29]`	`<L n="1" /> [29]`	`<L n="2" /> [30]`	`<L n="3" /> [31]`
`</E>`	`</E>`	`</E>`	`</E>`
`</A>`	`</A>`	`</A>`	`</A>`

Table 8: **Snapshots**

The first point is that gaps in an item's existence should not affect gluing of the item. Elements of the same item glue together even if they are not in consecutive snapshots. As long as the item identifiers match, the elements will be glued together. The `B` and `E` elements illustrate this point. The `B` elements are constituents of items 2, 3 and 11 while the `E` elements form items 4 and 5. Items 4 and 5 are composed of elements in non-consecutive snapshots.

The second point is that the way the item identifier field is specified affects how items are glued. Elements of type `H` and `J` illustrate this. Identical `H` and `J` elements exist side-by-side in the snapshots. The items are specified differently in the temporal annotations and glue very differently. All `H` elements are constituents of item 15.

There are three types of `J` items. All three are specified as disjoint sets. Note that all have an attribute `n=1`. The first item is specified with "`./J`". Only the `J` elements that are children of the root `A` are associated with this item. This SchemaPath expression could also be specified as `J`. These elements compose item 19.

The second is specified with a target of `D/J`. It has a two part identifier: "`@n`" and "`..`". In other words, both the `J` element's `n` attribute plus the loose text of its parent `D` element must match for the `J` elements to be glued. Items 17 and 18 are straightforward. The interesting case is item 22. These do glue because they match on both fields. The two parent `D` elements are constituents of different items. This does not affect the gluing of the `J` elements.

The third uses the target `C/J`. The `C` element is allowed in the target SchemaPath expression because otherwise there is no way to designate these `J` elements as items. Any element that is in a direct path from the root to the node may be in the target. It does not matter if an element is a constituent of an item or not. These elements compose item 16.

This leads to the third point. Elements that are children of elements that do not compose an item can still be glued. Not all elements in a snapshot are constituents on an item. Elements of type `C` are an example of this kind of item. All the `H` elements have `n=1`. They all glue together including the elements that are children of `C` elements. Similarly, the `J` elements of item 16 glue together.

Only elements that are constituents of items are allowed identify another item. The third `J` item could not have "`../@b`" as a field. The `C` element is assumed to be a different instance in each snapshot so this makes no sense for it to be part of an identifier. Putting "`../@b`" in a field for the third `J` would result in an error message.

The fourth point is that the root is not required to be an item. Just as with other elements, the root must be specified as an item to be glued. Other elements are still glued even if the root is not specified as an item. Elements can be glued even if their parent elements are not constituents of an item. The `A` node is designated as an item in this example. The `A` root nodes glue together to form item 1.

The fifth point is that keyrefs and itemrefs may be used in any combination with fields and each other. The `D` items are a simple example of using a keyref to specify an item identifier. The item is specified with a keyref to the snapshot schema key "`deptKey`". The elements glue as if the item was specified with a field of "`@n`". These are items 6 and 7.

The `K` items are specified with a field, a keyref and an itemref. There are many `K` elements in the snapshots. To glue together, two `K` elements must have the same loose text, `n` attribute and the same "`B/@n`" attribute. The `K` elements compose items 23 – 28.

The `L` item illustrates using several itemrefs. Each `L` item is specified by both its parent `E` item and grandparent `A` item. In addition, the `L` element's `n` attribute is also part of the identifier. An item identifier must include either the element's own loose text or one of its attributes. The `L` elements compose items 29 – 31.

The final point is that many elements can compose an item when a target contains the '`*`' wildcard. The `F` items are specified with a '`*`' wildcard. Identical `G` items occur in the same place in snapshots but glue very differently. The `F` elements compose items 8 and 9. The `G` elements compose items 10, 12 – 14, 20 and 21. Not all `G` elements are constituents of an item. The `A/E/G` elements are not.

7.3.5. Implementation Details

τValidator was written in Java and developed with the Eclipse Platform. It uses the W3C DOM API for processing the temporal bundle and temporal document. The gluing component is part of τValidator. τValidator uses the gluing component to create a data structure of all items. τValidator uses this data structure to validate the content-varying and existence-varying constraints of the items.

The following four classes were written for the gluing component. Java class names are capitalized so they may be easily distinguished from other words with the same name.

- The `Identifier` class is the representation of item identifiers. An instance of this class stores all the information necessary to identify an item: the target and all field(s), itemref(s) and keyref(s). The `Identifier` stores three `String`s. The first is the target. The second is the concatenation of the schema for all field(s), itemref(s) and keyref(s). The third is the concatenation of the value for all field(s), itemref(s) and keyref(s). The class provides its own implementation of the `equals` method so that item identifiers can be compared.
- Each instance of the `Item` class corresponds to one item. An `Item` stores its `Identifier` and a data structure of references to its element `Node`s in the snapshot `Document`. The class also provides a method for gluing two `Item`s together.
- `ItemSet` is a data structure that contains instances of `Item`s. An item may be composed of multiple elements. All of them belong to the same instance of the `Item` class. The `ItemSet` contains one instance of each such item; their order does not matter. There is only one instance of this class in τValidator and it contains all the items from the snapshots.
- `AnnoGluer` is the main class for the gluing component. It parses the temporal annotations and snapshots and creates the `ItemSet`. The `AnnoGluer` glues items when the schema does not change. In the future, the `AnnoGluer` will be extended to handle schema versioning.

8. Conclusion

τXSchema is an extension to XSchema. It adds the ability to specify how items in XML documents are allowed to change over time. The design of τXSchema is compatible with existing standards. τXSchema also has the potential to be expanded. These possibilities are explained in the section on future work.

8.1. Summary of Work

In this document we presented the τXSchema model and notation to annotate XML Schemas to support temporal information. τXSchema provides an efficient way to annotate temporal elements and attributes. Our design conforms to W3C XML Schema definition and is built on top of XML Schema. Our approach ensures data independence by separating (i) the snapshot 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. A small change to the physical annotations (or temporal annotations) can effect a large change in the representational schema and the associated XML file.

This separation of concerns may be exploited in supporting tools; several new, quite useful tools are introduced that require the logical and physical data independence provided by our approach. Additionally, this independence enables existing tools (the XML Schema validator was discussed) to be used in the implementation of their temporal counterparts.

8.2. Future Work

This thesis addresses time-varying XML documents. The current design allows XML documents to change, but the schema must remain static. It is useful to allow the schema to change too, termed *schema versioning*. This is more complex and represents an interesting area for future work.

With schema versioning, there can be periods when the schema remains static and the snapshot documents change. Between schema changes τXSchema works as described in this thesis. It would be highly desirable to extend τXSchema to work across schema changes.

When the schema changes, there is no restriction on the existence of items. An item can exist for several versions of the schema. The item identifier can change when the schema changes. There needs to be a way to glue items across schema changes. One idea is to extend the temporal bundle. We propose to introduce another document that describes how the items on one side of the schema change correspond to the items on the other side. However, that discussion is beyond the scope of this document.

τValidator is being developed to match the design presented in this thesis. The current version of τValidator requires two changes to meet the current design. The original τValidator was written before the idea of a Temporal Bundle. The original τValidator needs to add this functionality. The gluing component also needs to be integrated with τValidator.

τValidator will need to be extended to match the future design. The existing τValidator will be used as a component of the extended validator. The extension will need to add functionality to glue items across schema changes. The current τValidator will still work between schema changes.

9. References

1. Abiteboul, S., Bonifati, A., Cobena, G., Manolescu, I. and Milo, T., Dynamic XML Documents with Distribution and Replication. in *ACM SIGMOD International Conference on Management of Data*, (San Diego, CA, 2003), 527-538.
2. Amagasa, T., Yoshikawa, M. and Uemura, S., A Data Model for Temporal XML Documents. in *11th International Workshop on Database and Expert Systems Applications*, (London, England, 2000), Springer, Berlin, New York, 334-344.
3. Barbosa, D., Mendelzon, A., Libkin, L., Mignet, L. and Arenas, M., Efficient Incremental Validation of XML Documents. in *20th International Conference on Data Engineering*, (Boston, MA, 2004), IEEE Computer Society.
4. Birsan, D., Sluiman, H. and Fernz, S.-A. XML Diff and Merge Tool, IBM alphaWorks, 1999.
5. Bouchou, B. and Halfeld-Ferrari, M., Updates and Incremental Validation of XML Documents. in *9th International Workshop on Data Base Programming Languages*, (Potsdam, Germany, 2003), Springer.
6. Buneman, P., Khanna, S., Tajima, K. and Tan, W.C., Archiving scientific data. in *ACM SIGMOD International Conference on Management of Data*, (Madison, WI, 2002), ACM, 1-12.
7. Burns, T., Fong, E.N., Jefferson, D., Knox, R., Mark, L., Reedy, C., Reich, L., Roussopoulos, N. and Truszkowski, W. 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.
8. Chawathe, S., Abiteboul, S. and Widom, J., Representing and Querying Changes in Semistructured Data. in *14th International Conference on Data Engineering*, (Orlando, FL, USA, 1998), IEEE Computer Society, 4-13.
9. Chien, S., Tsotras, V. and Zaniolo, C. Efficient schemes for managing multiversion XML documents. *VLDB Journal*, *11* (4). 332-353.
10. Cho, J. and Garcia-Molina, H., The Evolution of the Web and Implications for an Incremental Crawler. in *26th International Conference on Very Large Data Bases*, (Cairo, Egypt, 2000), Morgan Kaufmann, 200-209.
11. Cobena, G., Abiteboul, S. and Marian, A., Detecting Changes in XML Documents. in *18th International Conference on Data Engineering*, (San Jose, California, 2002), IEEE Computer Society, 41-52.
12. Currim, F., Currim, S., Snodgrass, R.T. and Dyreson, C.E. τXSchema: Managing Temporal XML Schemas, TimeCenter, 2003.
13. Dyreson, C., Towards a Temporal World-Wide Web: A Transaction Time Web Server. in *12th Australasian Database Conference*, (Gold Coast, Australia, 2001), 169-175.
14. Dyreson, C.E., Bohlen, M. and Jensen, C.S., Capturing and Querying Multiple Aspects of Semistructured Data. in *25th International Conference on Very Large Data Bases*, (Edinburgh, Scotland, UK, 1999), Morgan Kaufmann, 290-301.
15. Franconi, E., Grandi, F. and Mandreoli, F., Schema Evolution and Versioning: A Logical and Computational Characterisation. in *9th International Workshop on Foundations of Models and Languages for Data and Objects, FoMLaDO/DEMM 2000*, (Dagstuhl, Germany, 2000), Springer, 85-99.
16. Gadia, S. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transactions on Database Systems*, *13* (4). 418-448.

17. Gao, D. and Snodgrass, R.T., Temporal Slicing in the Evaluation of XML Queries. in *29th International Conference on Very Large Databases*, (Berlin, Germany, 2003), Morgan Kaufmann, 632-643.
18. Grandi, F. An Annotated Bibliography on Temporal and Evolution Aspects in the WorldWideWeb, TimeCenter, 2003.
19. Grandi, F. and Mandreoli, F. The Valid Web: its time to Go…, TimeCenter, 1999.
20. Jensen, C.S. and Dyreson, C.E. A Consensus Glossary of Temporal Database Concepts. Etzion, O., Jajodia, S. and Sripada, S. eds. *Temporal Databases: Research and Practice*, Springer-Verlag, 1998, 367-405.
21. Jensen, C.S. and Snodgrass, R.T. Semantics of Time-Varying Information. *Information Systems*, *21* (4). 311-352.
22. Khatri, V., Ram, S. and Snodgrass, R.T. Augmenting a Conceptual Model with Spatio-Temporal Annotations. *IEEE Transactions on Knowledge and Data Engineering*.
23. Lee, D. and Chu, W. Comparative Analysis of Six XML Schema Languages. *SIGMOD Record*, *29* (3). 76-87.
24. Marian, A., Abiteboul, S., Cobena, G. and Mignet:, L., Change-Centric Management of Versions in an XML Warehouse. in *Very Large Data Bases Conference*, (Roma, Italy, 2001), Morgan Kaufmann, 581-590.
25. McHugh, J. and Widom, J., Query Optimization for XML. in *25th International Conference on Very Large Databases*, (Edinburgh, Scotland, UK, 1999), Morgan Kaufmann, 315-326.
26. McKenzie, E. and Snodgrass, R.T. An Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, *23* (4). 501-543.
27. Milo, T., Abiteboul, S., Amann, B., Benjelloun, O. and Ngoc, F.D., Exchanging Intensional XML Data. in *ACM SIGMOD International Conference on Management of Data*, (San Diego, CA, 2003), 289-300.
28. Nguyen, B., Abiteboul, S., Cobena, G. and Preda, M., Monitoring XML Data on the Web. in *ACM SIGMOD International Conference on Management of Data*, (Santa Barbara, CA, 2001), 437-448.
29. OMG. Unified Modeling Language (UML), v1.5, 2003.
30. Ozsoyoglu, G. and Snodgrass, R.T. Temporal and Real-Time Databases:A Survey. *IEEE Transactions on Knowledge and Data Engineering*, *7* (4). 513-532.
31. Papakonstantinou, Y. and Vianu, V., Incremental Validation of XML Documents. in *9th International Conference on Database Theory*, (Siena, Italy, 2003), Springer, 47-63.
32. Roddick, J.F. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, *37* (7). 383-393.
33. Snodgrass, R.T. Temporal Object-Oriented Databases: A Critical Comparison. in Kim, W. ed. *Modern Database Systems: The Object Model, Interoperability and Beyond*, Addison-Wesley/ACM Press, 1995, 386-408.
34. Snodgrass, R.T. The Temporal Query Language TQuel. *ACM Transactions on Database Systems (TODS)*, *12* (2). 247-298.
35. Steel, T.B., Jr., Chairman Interim Report: ANSI/X3/SPARC Study Group on Data Base Management Systems 75-02-08. *FDT-Bulletin of ACM SIGMOD*, *7* (2). 1-140.
36. W3C. Document Object Model (DOM) Level 2 HTML Specification Version 1.0. Hors, A.L. ed., W3C, 2002.
37. W3C. XML Schema Part 1: Structures. Mendelsohn, N. ed., W3C, 2001.

38. W3C. XML Schema Part 2: Datatypes. Malhotra, A. ed., W3C, 2001.
39. Xyleme, L. A dynamic warehouse for XML Data of the Web. *IEEE Data Engineering Bulletin*, *24* (2). 40-47.

Appendices

The appendicies give the complete TBSchema, TXSchema, and PXSchema (see Figure 9). Next are the complete documents for theWinOlympic and Company examples used in the body of the thesis. Both examples show the Temporal Annotations, Physical Annotations, schema and snapshot documents. The WinOlympic example also includes a Temporal Bundle. The Company example also includes the temporal document.

TBSchema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/TBSchema"
    xmlns:tb="http://www.cs.arizona.edu/tau/tauXSchema/TBSchema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    attributeFormDefault="unqualified" version="May 5, 2004">
  <xs:element name="temporalBundle">
    <xs:annotation>
      <xs:documentation>XML Schema file for temporal bundle file. currently mainly discusses
    identifier evolution</xs:documentation>
    </xs:annotation>
    <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:element name="bundleSequence" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="schemaAnnotation" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="tTime" type="xs:string"/>
                    <xs:element name="itemIdentifierCorrespondence" minOccurs="0"
    maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="oldRef" type="xs:string"/>
                        <xs:attribute name="newRef" type="xs:string"/>
                        <xs:attribute name="mappingType" type="tb:mappingType"/>
                        <xs:attribute name="mappingLocation" type="xs:anyURI"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="snapshotSchema" type="xs:anyURI" use="required"/>
                  <xs:attribute name="temporalAnnotation" type="xs:anyURI" use="optional"/>
                  <xs:attribute name="physicalAnnotation" type="xs:anyURI" use="optional"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="defaultTemporalAnnotation" type="xs:string" use="optional"/>
            <xs:attribute name="defaultPhysicalAnnotation" type="xs:string" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="defaultTemporalAnnotation" type="xs:string" use="optional"/>
      <xs:attribute name="defaultPhysicalAnnotation" type="xs:string" use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:annotation>
    <xs:documentation>
      Datatype definitions for temporal bundle file follow
    </xs:documentation>
  </xs:annotation>
  <xs:simpleType name="mappingType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="useBoth"/>
      <xs:enumeration value="useOld"/>
      <xs:enumeration value="useNew"/>
      <xs:enumeration value="replace"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

TXSchema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/TXSchema"
    xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TXSchema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    attributeFormDefault="unqualified" >
  <xs:element name="temporalAnnotations">
    <xs:annotation>
      <xs:documentation>
      XML Schema file for temporal annotations file
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <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="default" 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:anyURI" use="optional"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="item" 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="maximalExistence" minOccurs="0">
                      <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="ts:kindType" use="required"/>
<xs:attribute name="content" type="ts:contentType" use="optional"/>
<xs:attribute name="existence" type="ts:existenceType" 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:element name="itemIdentifier" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
<xs:element name="keyref" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
<xs:attribute name="refName" type="xs:string" use="required"/>
<xs:attribute name="refType" type="ts: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="ts: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="ts:kindType" use="required"/>
<xs:attribute name="content" type="ts: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>
  </xs:element>
  <xs:annotation>
    <xs:documentation>
    Datatype definitions for temporal annotations file follow
    </xs:documentation>
  </xs:annotation>
  <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:schema>
```

PXSchema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/PXSchema"
    xmlns:ps="http://www.cs.arizona.edu/tau/tauXSchema/PXSchema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    attributeFormDefault="unqualified" version="April 1, 2004">
  <xs:element name="physicalAnnotations">
    <xs:annotation>
      <xs:documentation>XML Schema file describing the physical annotations XML
    file</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <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="default" 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="ps:timeDimensionType" use="optional"/>
                  <xs:attribute name="stampBounds" type="ps: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="ps: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="ps:dataInclusionType" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <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: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:annotation>
    <xs:documentation>Note: "referenced-entity" should not be used in conjunction with
    "contained" timeBoundary</xs:documentation>
  </xs:annotation>
</xs:schema>
```

WinOlympic Example

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:annotation>
      <xs:documentation>
        Schema for recording non temporal country information
      </xs:documentation>
    </xs:annotation>
    <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>
```

Schema Annotations

Temporal Bundle

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<temporalBundle xmlns="http://www.cs.arizona.edu/tau/tauXSchema/TBSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.cs.arizona.edu/tau/tauXSchema/TBSchema TBSchema.xsd">
    <format plugin="XMLSchema" granularity="date"/>
    <bundleSequence defaultTemporalAnnotation="defaultTA.xml"
    defaultPhysicalAnnotation="defaultPA.xml">
        <schemaAnnotation snapshotSchema="winOlympic.xsd"
    temporalAnnotation="olympicTemporalAnnotations.xml"
    physicalAnnotation="olympicPhysicalAnnotations.xml">
          <tTime>2002-01-01</tTime>
        </schemaAnnotation>
    </bundleSequence>
</temporalBundle>
```

Temporal Annotations

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

  <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>
```

```
</temporalAnnotations>
```

Physical Annotations

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

Snapshot Documents

Data on January 1, 2002

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../schemas/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 phone numbers are:
            <phone>123-402-0340</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
                Lasse Kjus
            </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

Data on March 1, 2002

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../schemas/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>
              won a medal in
              <medal mtype="silver">Men's Combined</medal>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His phone numbers are:
        <phone>123-402-0430</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

Data on July 1, 2002

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../schemas/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>
              won a medal in
              <medal mtype="gold">Men's Combined</medal>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His phone numbers are:
        <phone>123-402-0430</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

Company Example

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:annotation>
      <xs:documentation>
        Schema for recording company information
      </xs:documentation>
    </xs:annotation>
    <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>
```

Schema Annotations

Temporal Annotations

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

Physical Annotations

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

Snapshot Documents

Data on March 29, 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>
```

Data on March 30, 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>
```

Data on March 31, 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>AAB</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>
```

Temporal Document

```
<?xml version="1.0" encoding="UTF-8"?>
<timeVaryingRoot xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema"
    xsi:noNamespaceSchemaLocation="../repCompany.xsd">
  <tv:entity>
    <!-- Version 1 of company -->
    <tv:version versionID="1" begin="2004-03-29">
      <company>
        <tv:entity>
          <tv:version begin="2004-03-29" end="now">
            <companyData>
              <companyName>IBM</companyName>
              <cURL>http://www.ibm.com</cURL>
            </companyData>
          </tv:version>
        </tv:entity>
        <tv:entity>
          <!-- Version 1 of supplier 1 (Seagate) -->
          <tv:version begin="2004-03-29" end="now">
            <supplier supplierNo="1" supplierName="Seagate">
              <sURL>http://seagate.com</sURL>
              <sRating>AAA</sRating>
              <tv:entity>
                <tv:version begin="2004-03-29" end="now">
                  <order orderType="normal">
                    <orderNo>1</orderNo>
                    <oProductNo>2</oProductNo>
                    <oQty>50</oQty>
                  </order>
                </tv:version>
              </tv:entity>
              <tv:entity>
                <tv:version begin="2004-03-30" end="now">
                  <order orderType="rush">
                    <orderNo>2</orderNo>
                    <oProductNo>1</oProductNo>
                    <oQty>100</oQty>
                  </order>
                </tv:version>
              </tv:entity>
              <tv:entity>
                <tv:version begin="2004-03-31" end="now">
                  <order orderType="normal">
                    <orderNo>3</orderNo>
                    <oProductNo>2</oProductNo>
                    <oQty>25</oQty>
                  </order>
                </tv:version>
              </tv:entity>
            </supplier>
          </tv:version>
        </tv:entity>
        <tv:entity>
          <!-- Version 1 of supplier 2 (Wistron) -->
          <tv:version begin="2004-03-29" end="now">
            <supplier supplierNo="2" supplierName="Wistron Corporation">
              <sURL>http://www.wistron.com</sURL>
              <sRating>AA</sRating>
              <tv:entity>
                <tv:version begin="2004-03-30" end="now">
                  <order orderType="normal">
                    <orderNo>1</orderNo>
                    <oProductNo>2</oProductNo>
                    <oQty>10</oQty>
                  </order>
                </tv:version>
              </tv:entity>
            </supplier>
          </tv:version>
        </tv:entity>
        <tv:entity>
          <!-- Version 1 of supplier 3 (Small supplier 1) -->
          <tv:version begin="2004-03-29" end="now">
            <supplier supplierNo="3" supplierName="small_supplier_1"/>
          </tv:version>
        </tv:entity>
        <tv:entity>
          <!-- Version 1 of product 1 (7200rpm hard disk) -->
          <tv:version begin="2004-03-29">
            <product productNo="1" productName="hard disk 73 GB 7200rpm">
              <priceinDollars>100</priceinDollars>
              <qtyOnHand>100</qtyOnHand>
            </product>
          </tv:version>
          <!-- Version 2 of product 1 (7200rpm hard disk) -->
          <tv:version begin="2004-03-30">
            <product productNo="1" productName="hard disk 73 GB 7200rpm">
              <priceinDollars>100</priceinDollars>
              <qtyOnHand>40</qtyOnHand>
            </product>
          </tv:version>
          <!-- Version 3 of product 1 (7200rpm hard disk) -->
          <tv:version begin="2004-03-31">
            <product productNo="1" productName="hard disk 73 GB 7200rpm">
              <priceinDollars>105</priceinDollars>
              <qtyOnHand>120</qtyOnHand>
            </product>
```

```
          </tv:version>
        </tv:entity>
        <tv:entity>
          <!-- Version 1 of product 2 (SCSI 10000rpm hard disk) -->
          <tv:version begin="2004-03-29">
            <product productNo="2" productName="SCSI hard disk 147 GB 10000rpm">
              <priceinDollars>150</priceinDollars>
              <qtyOnHand>100</qtyOnHand>
            </product>
          </tv:version>
          <!-- Version 2 of product 2 (SCSI 10000rpm hard disk) -->
          <tv:version begin="2004-03-30">
            <product productNo="2" productName="SCSI hard disk 147 GB 10000rpm">
              <priceinDollars>125</priceinDollars>
              <qtyOnHand>80</qtyOnHand>
            </product>
          </tv:version>
          <!-- Version 3 of product 2 (SCSI 10000rpm hard disk) -->
          <tv:version begin="2004-03-31">
            <product productNo="2" productName="SCSI hard disk 147 GB 10000rpm">
              <priceinDollars>125</priceinDollars>
              <qtyOnHand>70</qtyOnHand>
            </product>
          </tv:version>
        </tv:entity>
      </company>
    </tv:version>
  </tv:entity>
</timeVaryingRoot>
```