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

Shailesh Joshi

August 20, 2007

TR-89

A TIMECENTER Technical Report

Title	τXSchema - Support for Data- and Schema-Versioned XML Documents
	Copyright © 2007 Shailesh Joshi. All rights reserved.
Author(s)	Shailesh Joshi
Publication History	May 2007, a TIMECENTER Technical Report

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Simonas Šaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Faiz A. Currim, Sabah A. Currim, Bongki Moon, Sudha Ram, Stanley Yao

Individual participants
Yun Ae Ahn, Chungbuk National University, Korea; Michael H. Böhlen, Free University of Bolzano, Italy; Curtis E. Dyreson, Washington State University, USA; Dengfeng Gao, Indiana University South Bend, USA; Fabio Grandi, University of Bologna, Italy; Heidi Gregersen, Aarhus School of Business, Denmark; Vijay Khatri, Indiana University, USA; Nick Kline, Microsoft, USA; Gerhard Knolmayer, University of Bern, Switzerland; Carme Martín, Technical University of Catalonia, Spain; Thomas Myrach, University of Bern, Switzerland; Kwang W. Nam, Chungbuk National University, Korea; Mario A. Nascimento, University of Alberta, Canada; John F. Roddick, Flinders University, Australia; Keun H. Ryu, Chungbuk National University, Korea; Dennis Shasha, New York University, USA; Michael D. Soo, amazon.com, USA; Andreas Steiner, TimeConsult, Switzerland; Paolo Terenziani, University of Torino, Italy; Vassilis Tsotras, University of California, Riverside, USA; Fusheng Wang, Siemens, USA; Jef Wijsen, University of Mons-Hainaut, Belgium; and Carlo Zaniolo, University of California, Los Angeles, USA

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

Any software made available via TIMECENTER *is provided "as is" and without any express or implied warranties, including, without limitation, the implied warranty of merchantability and fitness for a particular purpose.*

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

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

Thesis Statement

By utilizing schema-constant periods and cross-wall validation, it is possible to realize a comprehensive system for representing and validating data- and schema-versioned XML documents, while remaining fully compatible with the XML standards.

Contents

List of Figures

Abstract

The W3C XML Schema recommendation defines the structure and data types for 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. 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.

In this thesis we introduce τXSchema, which is an extension of XML Schema, infrastructure, and a suite of tools to support the creation and validation of time-varying documents, without requiring any changes to XML Schema. The data model and architecture support the creation of a temporal schema from a non-temporal (snapshot) schema, a temporal annotation, and a physical annotation. These annotations specify, respectively, 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. A Temporal Validator (τVALIDATOR) 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 then extend τXSchema to support versioning of the schema itself. When the schema is versioned, the base schema and the temporal and physical annotations can themselves be time-varying documents, each with their own (possibly versioned) schemas. We describe how a temporal data validator can be extended to validate documents in this seeming precarious situation of data that changes over time, while its schema and even its representation are also changing.

1 Introduction

XML is becoming an increasingly popular language for documents and data. XML can be approached from two different orientations: *document-centered* (e.g., HTML) and *data-centered* (e.g., relational and object-oriented databases). Schemas are important for both. 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 (often) consult the schema rather than the data. Finally, a schema can be helpful in query optimization, e.g., in constructing a path index.

Time-varying data naturally arises in both document-centered and data-centered orientations. 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 [29], and incremental queries that involve the changes between two versions.

In this thesis we consider how to accommodate and validate time-varying data within XML Schema. One 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. This approach inherently introduces difficulties with respect to document validation, data independence, tool support, and standardization. The previous group working on TAU Project at the Computer Science Department at the University of Arizona has proposed a novel approach that retains the non-temporal XML Schema for the document, utilizing a series of separate schema documents to achieve data independence, to enable full document validation, and to enable improved tool support, while not requiring any changes to the XML Schema standard.

The system, called Temporal XML Schema (τXSchema), aids in 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. In Temporal XML Schema, any element type can be turned into a temporal element type by including a 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 second type of annotation is the *physical annotation*, which describes how to represent the time-varying aspects of the document. A *temporal bundle*, the XML document that serves as a temporal schema bundles together the non-temporal schema, temporal annotation and physical annotation. Thus τXSchema is consistent and compatible with both XML Schema and the XML data model.

In our thesis research, we refine τXSchema and implement the tools used to construct and validate temporal documents. A temporal document is validated by integrating 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 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.

As mentioned, τXSchema reuses rather than extends XML Schema. τXSchema is consistent and compatible with both XML Schema and the XML data model. In our approach, a Temporal Validator (τVALIDATOR) augments a conventional validator to more comprehensively check the validity constraints of a document, especially temporal constraints that cannot be checked by a conventional XML Schema

validator. We describe a means of validating temporal documents that ensures the desirable property of snapshot validation subsumption. We show how a temporal document can be smaller and faster to validate than the associated XML snapshots.

We then extend τXSchema to support *schema versioning*. When the schema is versioned, the base schema and temporal and physical schemas can themselves be time-varying documents, each with their own (possibly versioned) schemas. In doing so, we leverage both conventional XML Schema and related tools (principally, the conventional validator), as well as τVALIDATOR for data versioning. A challenge with schema versioning is that *anything* can change, and thus must be versioned: the snapshot documents, the base schema, the temporal annotations, the physical annotations, the schema documents included by these documents, even the schemas of these schema components. And, because the physical annotations can change, the concrete representation within a temporal XML document can vary.

With the framework introduced in our research, we will show that we can

- Develop a comprehensive set of tools to support schema and data versioning of XML data or documents,
- Achieve logical data independence by specifying what can change in the temporal annotation,
- Achieve physical data independence by specifying the location of timestamps in the physical annotation,
- Implement a set of tools using just three basic primitives, and
- Achieve code reuse by utilizing most of the code used for data versioning to implement schema versioning.

This thesis document is logically divided into two parts. The initial part concerns instance versioning; the second part extends the approach to support schema versioning. We first provide a motivating example that illustrates the challenges of data and schema versioning. In Part 1 we show how the three schemas (base schema and the two annotations) and the temporal bundle interact to support time varying data; in Part 2 we extend this architecture to incorporate schema versioning. Each part elaborates a theoretical foundation, architecture, and the design of the tools. It is followed by the implementation and testing details concerning both data and schema versioning. We then discuss the support for multiple kinds of time, to be defined in detail later, within this framework. The last section summarizes the contributions of this work.

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 the XML Schema in supporting temporal documents and data. Then we state the desired properties of schemas for time-varying documents. We end with a discussion of some real world applications that would benefit from schema versioning as supported in the τXSchema framework.

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 as 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 Silver medal, as of `2002-03-01`

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

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

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

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

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

Figure 4: Snippet of a Temporal Document

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 also changes the nature of validation 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 5. 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="0"
        maxOccurs="1"/>
    </sequence>
    <attribute name="age" type="nonNegativeInteger" use="required"/>
  </complexType>
</element>
...
```

Figure 5: `winOlympic.xsd`

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 every day. This would lead to a very large number of versions, making it expensive to instantiate and validate each individually. The number of versions could further be increased by the presence of both valid and transaction time.

To validate a time-varying document, a new, different schema is needed. The schema for a time-varying document should take into account the elements (and attributes) and their associated timestamps, specify the kind(s) of time involved, provide hints on how the elements vary over time, and accommodate differences in version and timestamp representation. Since this schema will express how the time-varying information is represented, we call it the *representational schema*. The representational schema will be related to the underlying snapshot schema, and will allow the time-varying document to be validated using a conventional XML Schema validator (though not fully, as discussed in the further sections). The representational schema will also be important in constructing, evaluating, and optimizing temporal queries. Both the person who is formulating a query and the database need to know which elements in the document are temporal elements since additional operations, like temporal slicing, are applicable to the 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, although the existence and/or type of medal may change over time (for instance if the athlete is disqualified). The *valid time* component of this constraint is that only one medal appears in an `<athlete>` element at any point in time. But the *transaction time* component of the constraint is that multiple versions can be present (as the element is modified). A schema language for a temporal document needs to have some way of specifying and enforcing such constraints.

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

Even though the representational and 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 limit 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 [32], 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, only a few applications that are concerned with representational details would need to be changed.

Hence, an improved tool support for representing and validating time-varying information is needed. Creating a time-varying XML document and representational schema for that document is potentially labor-intensive. Currently a user has to manually edit the time-varying document to insert timestamps indicating when versions of XML data are valid (for valid time) or are present in the document (for transaction time). The user also has to modify the 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.

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

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

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

This work has many real-world applications. As an example, the Botanic Garden and Botanical Museum in Berlin-Dahlem (BGBM[1]) maintains a repository of XML Schemas[2] related to index terms, keywords, biodiversity data about specimens and observations, meta-level data about collections, organizations, and networks, and various wrapper and configuration files. Most of these XML schemas have had multiple versions over the last two to three years. The BioCASE Collection Profile is up to version 1.24; the Access

[1] `http://www.bgbm.org`

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

to Biological Collection Data is up to version 2.06.

As another example, the *Pharmacogenetics Knowledge Base* (PharmGKB[3]) "contains genomic, phenotype and clinical information collected from ongoing pharmacogenetic studies." Its schema is up to version 4.0; its evolution is documented.[4] The PHARMGKB XML schema was designed conventionally, not utilizing an architecture that supports schema versioning. As new releases of this schema were developed (for example, on May 12, 2004 Version 4.0, the latest version, was released), all XML documents that were instances of this schema were rendered invalid, with the maintainers responsible for updating their XML documents. The architecture proposed in this thesis retains past data and past schemas, while always allowing the current data and schema to be extracted, for tools that are not schema-versioning aware. This example was discussed further in detail elsewhere [13].

[3] `http://www.pharmgkb.org/`
[4] `http://www.pharmgkb.org/schema/history.html`

3 Background

3.1 XML Schema

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

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

XML Schema, published as a W3C Recommendation in May 2001 [37], is one of the several XML schema languages. It was the first separate schema language for XML to achieve recommendation status by the W3C. An XML schema is a description of a type of XML document, typically expressed in terms of constraints on the structure and content of documents of that type, beyond the basic syntax constraints imposed by XML itself. Thus an XML schema provides a view of the document type at a relatively high level of abstraction. The XML Schema language is also referred to as XML Schema Definition (XSD).

The Document Type Definition (DTD) language [10], which is native to the XML specification, was being used as a schema language before XML Schemas were introduced. XML Schema language was introduced in order to overcome some of the limitations of DTDs like different syntax from that of XML, limited data type capability, and limited data types compatibility with those found in the databases.

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

The process of checking to see if an XML document conforms to a schema is called *validation*, which is separate from XML's core concept of syntactic well-formedness. All XML documents must be well-formed, but it is not required that a document be valid unless the XML parser is "validating", in which case the document is also checked for the conformance with its associated schema. A well formed document obeys the basic rules of XML established for the structural design of a document. Moreover a valid document also respects the rules dictated by its corresponding XML Schema.

The parser, XML's one of the core technologies provides an interface to an XML document, exposing its contents through a well-specified API. At present, two major API specifications define how XML parsers work: SAX [25] and DOM [9]. The DOM specification defines a tree-based approach to navigating an XML document. It processes XML data and creates an object-oriented hierarchical representation of the document that can be navigated at run-time. The tree-based W3C DOM parser creates an internal tree based on the hierarchical structure of the XML data. It can be navigated and manipulated from the software, and it stays in memory until it is released. DOM uses functions that return parent and child nodes, giving programmer full access to the XML data and providing the ability to interrogate and manipulate these nodes.

The SAX specification defines an event-based approach whereby parser scans through XML data, call-

ing handler functions whenever certain parts of the document (e.g., text nodes or processing instructions) are found. In SAX's event-based system, the parser doesn't create any internal representation of the document. Instead, the parser calls handler functions when certain events (defined by the SAX specification) take place. These events may include the start and the end of the document, finding a text node, finding child elements, and hitting a malformed element.

3.2 Temporal Databases

Most applications of database technology are temporal in nature [18]. Some examples include financial applications such as banking and accounting; record-keeping applications such as personnel, and inventory management; scheduling applications such as airline, train, and hotel reservations; and scientific applications such as weather monitoring and forecasting. Applications such as these rely on temporal databases, which record time-referenced data.

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

More specifically the temporal aspects usually include two orthogonal time dimensions: valid time and transaction time. These two kinds together form *bitemporal data* [17].

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

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

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

Consider the following example emphasizing the use of both valid time and transaction time in a database table:
Joe was born on Jan 1^{st}, 2002. His father happily registered his son's birth-date on Jan 2^{nd}, 2002. In the Citizen table, two columns `ValidBegin` and `ValidEnd` would be present to record the date when a citizen is alive. Although the registration was done on Jan 2^{nd}, the database states that the information is valid since Jan 1^{st}. So `ValidBegin` contains Jan 1^{st}. Joe's record is valid while he is alive. So, `ValidEnd` contains an infinity value. To keep a track of the date when the record was inserted into the table two more fields are added to the Citizen table: `TransactionStart` and `TransactionStop`. `TransactionStart` is the time a transaction inserted that data, and `TransactionStop` is the time that a transaction *superseded* that data (or "until changed" if it has not yet been superseded). For this record, the `TransactionStart` would contain Jan 2^{nd} while `TransactionStop` would contain "until changed". What happens if the data entry operator enters Joe's birth date as Jan 1^{st}, 2001 instead of Jan 1^{st}, 2002? When this is realized

e.g., on Jan 10^{th}, 2002, the old transaction started on Jan 2^{nd}, 2002 containing `ValidEnd` date as Jan 1^{st}, 2001 would be terminated and a new record containing correct birth date in `ValidBegin` column would be inserted. The `TransactionStop` column for this record would have a value Jan 10^{th}, 2002.

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

3.3 Schema Versioning

Software systems and especially databases undergo frequent changes following an initial implementation. Lientz has shown that 50% or more of programmer effort arises as a result of system modifications after the implementation [19]. Sjoberg has also shown that the system modifications that cause changes to the structure of the data are relatively frequent [26, 27]. As a result, modifying the database schema is a common but often a troublesome occurrence in database administration.

Schema versioning deals with the need to retain current data and software system functionality in the face of changing structure of the data [24]. It is often not practical to simultaneously replace all the deployments of the old schemas with the new ones. So applications will need to cope with different versions coexisting in the system. Hence, versioning mechanisms in XML Schema should support creation of new versions, and the schema processors should be able to handle the instances defined by different versions. Thus schema versioning should offer a solution to the problem by enabling intelligent handling of any temporal mismatch between data and the data structures.

Schema versioning has been previously researched in the context of temporal databases [23]. But an XML schema is a grammar specification, unlike a (relational) database schema, so new techniques are required to handle schema versioning.

Since XML Schema changes are very common in the industry, there has been some effort to address this issue. Some white papers [14, 7] discuss the need for schema versioning and some common techniques used in the industry to handle it. According to Gabriel, some of the important reasons for changes in the XML schema are as follows [14].

- Extending the scope of a schema.
- Changing constraints.
- Bug-fixing.
- Enabling collaborative development.

The previous literature also discusses some of the common techniques and the best practices currently used to reduce the effects of schema changes on the system maintenance and recommend that schema versioning should be a part of an integrated system evolution plan.

4 Previous Work

Methods to represent temporal data and documents on the web have been actively researched. This research has covered a wide range of issues that include architectures for collecting document versions [11], strategies for storing versions [6], studies on the frequency of data change [6], and temporal query languages [15]. The logical representation of deltas between the versions and the aspects of physical storage policy for storing those versions have been proposed so as to maximize the space utilization [21]. Grandi has created a bibliography of previous work in this area [16].

A logical data model based on XPath for capturing the entire history of an XML document is also proposed [1]. The paper discusses physical representations of the document and proposes two specific representations. Although the paper examines the consistency of a XML document in a limited context, it does not mention modified XML schema for representation and its validation, nor does it consider the general problem of validating against a temporal schcema. The approach does not provide logical and physical data independence. It cannot check temporal constraints as well since there is no notion of temporal constraint.

Version and source control for schemas and schema objects is needed, especially in complex, multi-enterprise development environments. The XML Schema working group at W3C has discussed desirable behaviors for use cases that involve schema versioning in XML [36]. Various techniques to support evolution of XML schemas, where they allow for extensibility in the original design have also been proposed [14]. The emphasis of the paper is to avoid changes to the existing applications by anticipating changes to the schemas and then designing them for evolution. This is typically achieved through a careful use of wildcards, allowing extensions through namespaces, allowing applications to ignore unknown objects, and forcing applications to understand unknown objects when no other option is available. This approach does not address the whole problem, as many schema changes cannot be expressed in their limited notations.

Some version control tools (that are designed for text files) have also been developed for data and schema varying XML documents (e.g., [20]). But, since the tree-structured data has very different semantics as compared to text, these tools are not very effective. Such an approach also lacks the support for any mechanisms for their validation.

Schema versioning has been previously researched in the context of temporal databases [23]. But an XML schema is a grammar specification, unlike a (relational) database schema, so new techniques are required. Although various XML schema languages have been proposed in the literature and in the commercial arena, none model schema changes nor provide versioning. We chose to base our research on XML Schema because it is backed by the W3C and is the most widely-used schema language.

The previous group working on the TAU Project at the Computer Science Department at the University of Arizona has developed a theoretical framework for data versioning in XML documents. The basic architecture of the system along with base schemas for temporal annotation, physical annotation, and the temporal bundle were also created. The initial implementation of the τVALIDATOR, SQUASH and UNSQUASH tools to handle data versioning was also developed.

To summarize, in this research we extend the existing τXSchema system in following ways.

- Reimplement the tools τVALIDATOR, SQUASH and UNSQUASH for XML 1.1 specification with a new design keeping schema versioning in mind.
- Propose a new representation, *non-decomposed representation*, for temporal documents.
- Extend the tools to support schema versioning.

5 Architecture

In this section we describe the overall architecture of τXSchema and illustrate with an example. The design and implementation details of the tools are explained further in Section 7.

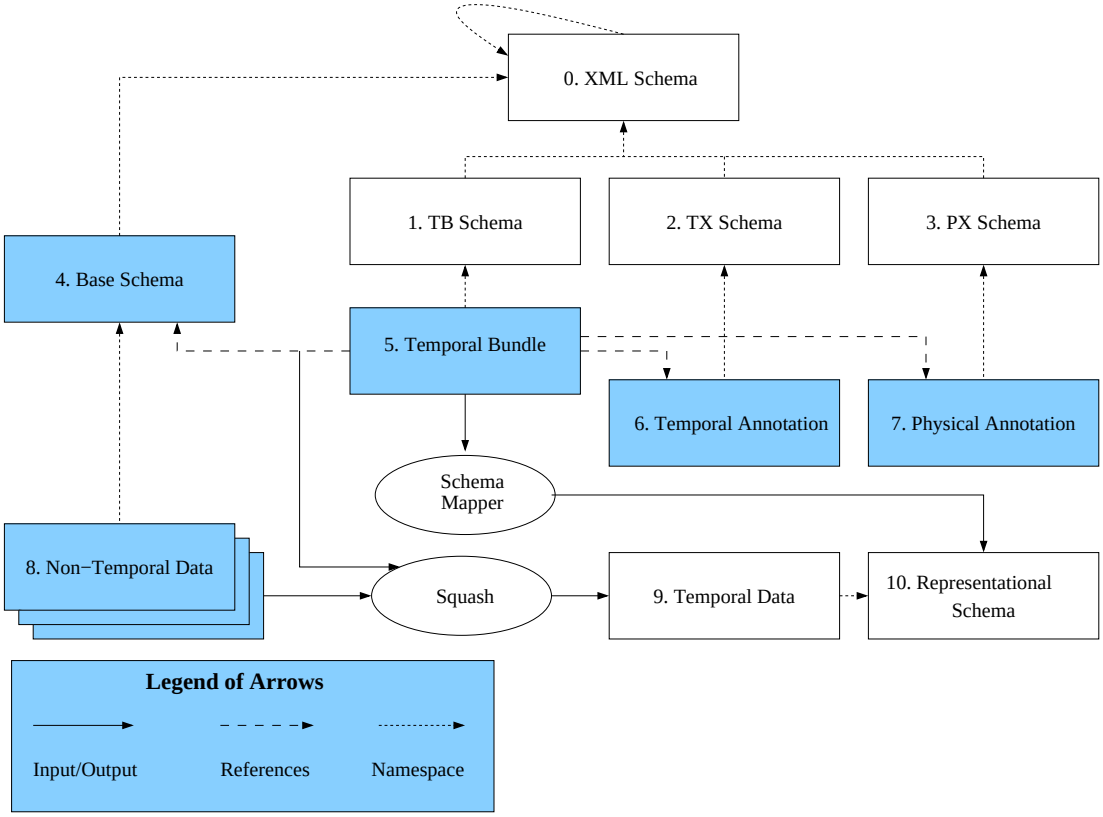


Figure 6: Overall Architecture

Figure 6 illustrates the architecture of τXSchema. Only those components shaded in the figure are specific to an individual time-varying document and need to be supplied by a user. The designer annotates the snapshot schema with temporal annotations (box 6). The temporal annotations together with the snapshot schema form the logical schema.

Figure 7 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. Annotations can be nested, enabling the target to be relative to that of its parent, and inheriting as defaults the kind, content, and existence attribute values specified in the parent. The attribute 'existence' indicates whether the element can be absent at some times and present at others. As an example, the presence of existence="varyingWithGaps" for an `athleteTeam` indicates that a team for a country may be present at some points in time and not at other points in time. The attribute 'content' indicates whether the element's content can change over the time. An elements 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. The schema for the temporal annotations document is given by TXSchema (box 2).

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

Figure 7: Sample WinOlympic Temporal Annotation

The next design step is to create the physical annotations (box 7). 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 8

Physical annotations play two important roles.

- They help to define where the physical timestamps will be placed (versioning level). The location of the timestamps is independent of which components vary over time (as specified by the 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 athleteTeam and medal are time-varying, the user may choose to place the physical timestamp at the country and athlete level. Whenever any element below medal changes, the entire athlete element is repeated.

- The physical annotations also define the type of timestamp (for both valid time and transaction time). A timestamp can be one of two types: *step* or *extent*. An extent timestamp specifies both the start and end instants in the timestamps 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

```
<?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/country">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>
...
  <stamp target="/winOlympic/country/athleteTeam/athlete">
    <stampKind timeDimension="transactionTime" stampBounds="step"/>
  </stamp>
...
</physicalAnnotations>
```

Figure 8: Sample WinOlympic Physical Annotation

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 the focus of physical annotation is on capturing relevant aspects of physical representations, not on the specific representations themselves, the design of which is itself challenging and is described in detail in Section 8

The temporal and physical annotations are orthogonal and serve two separate goals. A user can change where the timestamps are located, independently of specifying the temporal characteristics of that particular element. Thus two documents with the same logical information will look very different if we change the location of the physical timestamp in physical annotation. The temporal bundles (box 5) tie the base 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.

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). The boxes 1, 2, and 3 are provided by us while, XML Schema (box 0) is provided by W3C. Thus new time-varying schemas can be quickly and easily developed and deployed.

6 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.

6.1 Snapshot Validation Subsumption

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

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

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

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 $vldt(S, D)$ represents the validation status of document D with respect to schema S. The status is true if the document is valid but false otherwise. Validation also applies to time-varying documents, e.g., $vldt^T(S^T, D^T)$ is the validation status of D^T with respect to a representational schema, S^T, using a temporal validator.

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

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

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 Figure 9.

6.2 SchemaPath

SchemaPath is a language for locating element definitions 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 the schema itself. The data model is created as follows. Each element and the 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 of that node.

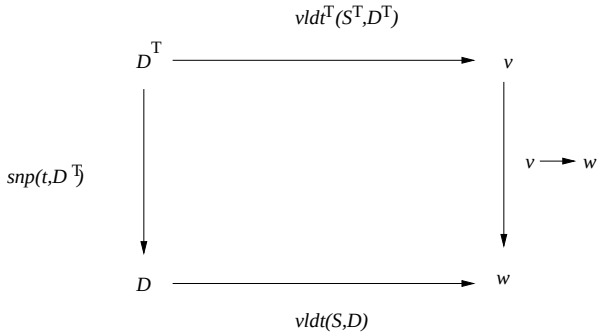


Figure 9: Snapshot Validation Subsumption

SchemaPath expressions, like XPath expressions, are composed of a number of steps. Each step consists of an axis (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 descendant axis, or the "any element wild-card" axis that explores non-neighbors of the context node in the graph). 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 "`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. '`*`' and '`|`' constructs are not supported in the current implementation.

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`'.

6.3 Content and Existence Variance

The data stored in XML documents may change over time. It is useful to be able to validate the way data can change. The 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. The content of an item includes the entire sub-tree rooted at a node. Each branch in the sub-tree terminates at the first item on the branch, or at a leaf (text value, attribute, empty element). Some nodes, especially those containing loose text, will change their content. Some nodes will exist in one version of an XML instance document but will not be present in another version. Other nodes will have both their content and existence change over time.

An item definition specifies how a data node may vary in its content and its existence. Let's first consider how an item specifies existence. There are three possible alternatives. The first is "varying with gaps", which means that each of its corresponding data nodes may be present in some versions of the XML instance document and absent in others. A second, more restrictive form is "varying without gaps." The data

node is not required to always be present. When it is present there may not be any gaps in its existence. The third value is "constant". Then the corresponding data node is either always present or never present. Again the existence-constant can have many different semantics. We have identified three of them and provide support for the first two in our implementation.

- Existence is constant over all time (exists in every instant in lifetime of universe).
- Existence is constant over document lifetime (document lifetime may have gaps).
- Existence is constant over immediate ancestors items' lifetimes.

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 nodes content may change over time when the corresponding item specifies content as constant. The restrictions are different for each of the type of content (e.g., elements, attributes and loose text). The detailed explanation of the restrictions can be found in [30].

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

6.4 Items

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

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

Definition [XPath evaluation] Let $Eval(n, E)$ denote the result of evaluating an XPath expression E from a context node n. Given a list of XPath expressions, $L = (E1, , Ek)$, then $Eval(n, L) = (Eval(n, E1), , Eval(n, Ek))$.

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

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

An item identifier has a target and at least one field, an itemref or a keyref. A target is a 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). Each field expression specifies either an attribute or an element. If an attribute is indicated, then the item identifier uses the attribute's value. If an element is indicated, then the item identifier uses the element's loose text. The current implementation supports only fields.

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

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

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

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

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

A temporal association relates elements that are adjacent in time and that belong to the same item. For instance, the athlete element in 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).

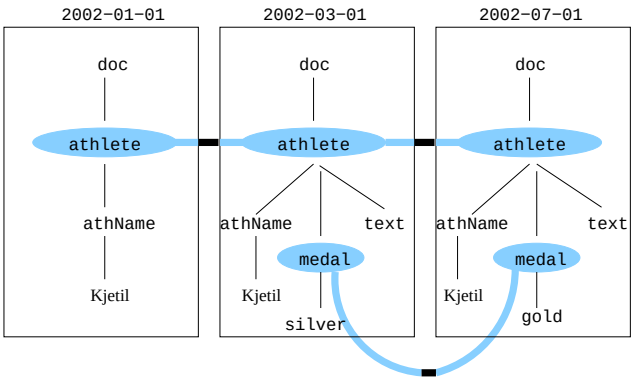


Figure 10: Items and Versions

6.5 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 [17]. DOM equivalence is used to determine versions of an item, as follows.

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

A version's lifetime is extended when the element from the next snapshot (or a future snapshot) is DOM equivalent (the lifetime can have gaps or holes, although having a gap may violate a schema constraint as described in section 6.3). A new version is created when a temporal association is not DOM equivalent.

Figure 10 depicts the items and versions in the example. An abstract representation of the DOM for each snapshot of the document is shown. The items in the sequence of snapshots are connected within each shaded region. There is one athlete item and one medal item. The athlete item has two versions; the transition between versions is shown as a black stripe between the regions.

6.6 Extending Temporal XML Schema Constraints

In this section we briefly 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 are restricted to a specific snapshot document. These constraints need to be augmented for τXSchema.

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. Technical Report document [33] further discusses the sequenced and non-sequenced temporal annotations to the XML schema constraints in detail.

7 Tools and Algorithms

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

The tools are open-source and beta versions are available [33]. The tools have been implemented in Java using the DOM API [9]. The DOM API was chosen over SAX API due to its ability to create an object-oriented hierarchical representation of the XML document that can be navigated and manipulated at the run-time. The primitives explained below use this ability of the DOM API to easily manipulate the document-tree.

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

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

7.1 Implementation Primitives

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

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

7.1.1 The pushUp Function

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

Consider the snapshot schema in Figure 11 and corresponding temporal annotation (Figure 12) and physical annotation (Figure 13). Figures 14–17 depict step by step working of the pushUp function when applied to a temporal document having timestamps at the temporal elements.

The first tree representation in Figure 14 represents the original document before applying the pushUp function. The timestamps are present at element <B>, which is temporal in nature (i.e., present in the

temporal annotation). The pushUp function moves the timestamp to element <A>, which is present in the physical annotation. It results in the three copies of element <A> corresponding to the three versions of item B. Elements <A>, <C> and <D> are non-temporal in nature. Thus their contents are the same and hence are duplicated in all the three versions.

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

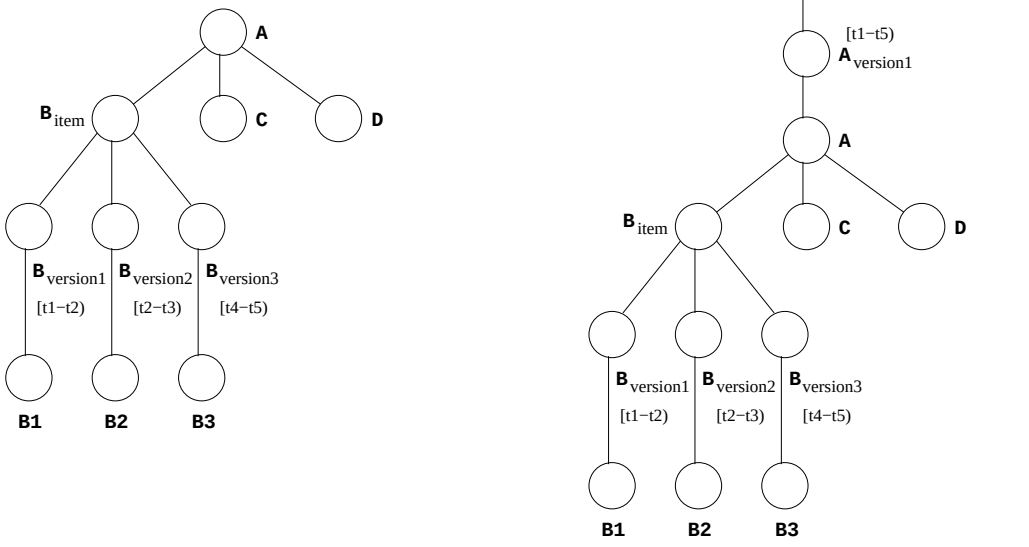
Figure 11: Snapshot Schema

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

Figure 12: Temporal Annotation

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

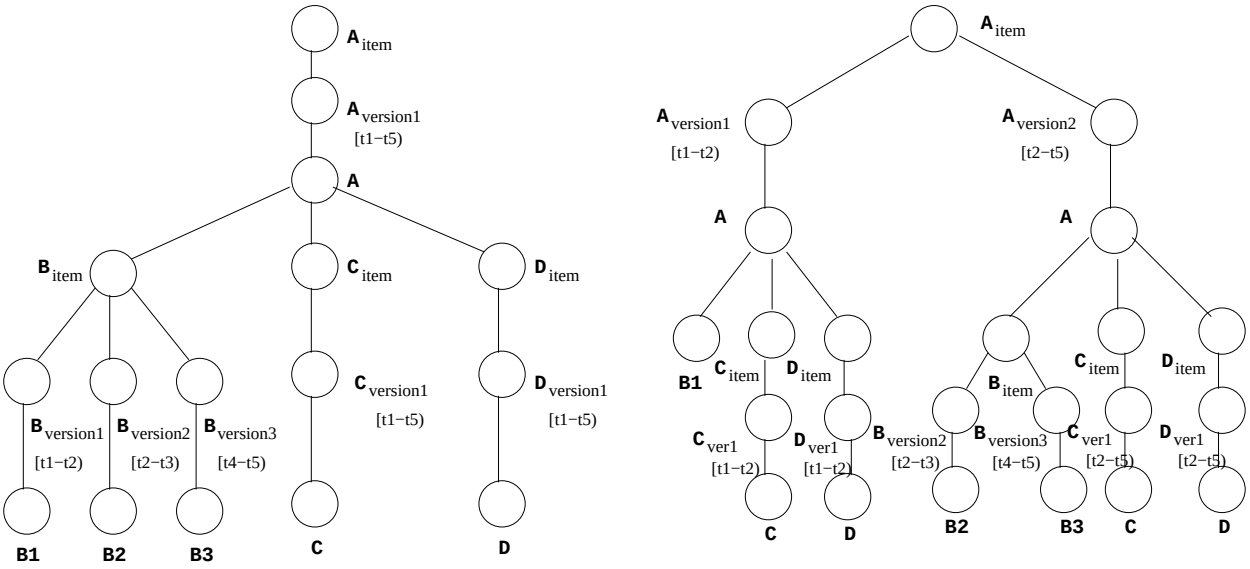
Figure 13: Physical Annotation



Original Document

Before call pushUp $(A_{item}$, *physicalAnnotation*)

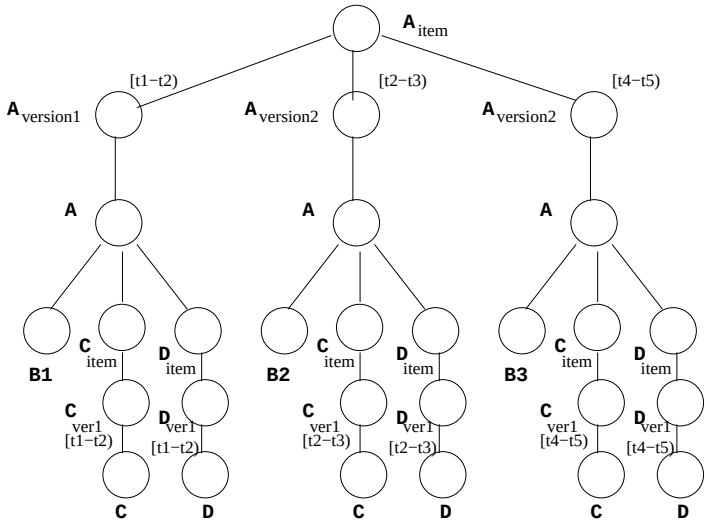
Figure 14: Example of pushUp



Before call to function splitChildVersions $(A_{item}$, *physicalAnnotation*)

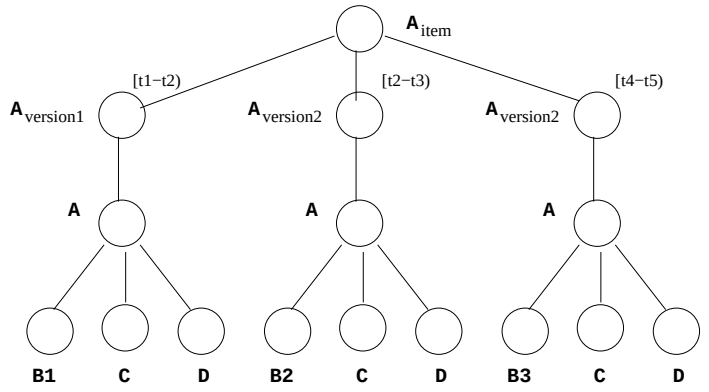
After first iteration of third **for** loop in function splitChildVersions

Figure 15: Example of pushUp: Continued



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

Figure 16: Example of pushUp: Continued



Final Result

Figure 17: Example of pushUp: Continued

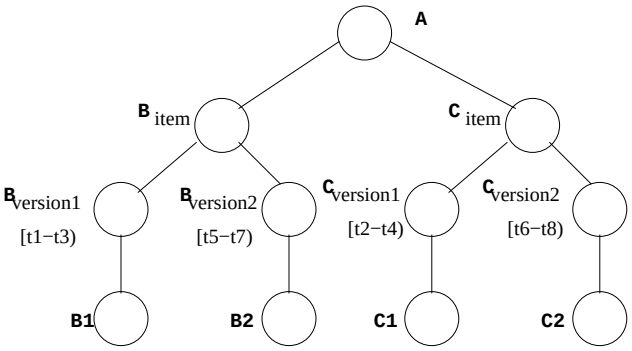
The pushUp function is used in SQUASH and RESQUASH tools. These tools first construct the temporal document with the timestamps located at the temporal elements. The timestamps are then moved up in the hierarchy to the elements present in the physical annotation.

The recursive algorithm for pushUp is given in Figure 19. The function accepts an item representation of an XML element as one of its parameters. The algorithm is called on the root item in the temporal XML document. If the root element is not an item, it is converted into an item using createItem function before pushUp is called. The pushUp function recurses until it reaches the bottom of the XML tree. At that point, it moves timestamps up in the hierarchy by using the function splitChildVersions. The nested **for** loop in the function splitChildVersions may multiply the existing versions of the item by splitting them depending upon its versions's overlap with its child items' versions. The child items from the versions of the parent item are replaced by the child items' versions removing the child items not present in the physical annotation. The timestamp is thus pushed one level up in the hierarchy, closer to the elements present in the physical annotation.

Other helper functions used in the algorithm are as follows.

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

Figure 18 shows a slightly more complicated case, where two temporal elements are siblings of each other. In this case, movement of timestamps *Up* in the hierarchy could result in the multiplication of the total number of versions depending upon the time overlap of individual versions from the sibling items. In this case, two versions of <B> and two versions of <C> give six versions of <A> after the application of pushUp function.




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

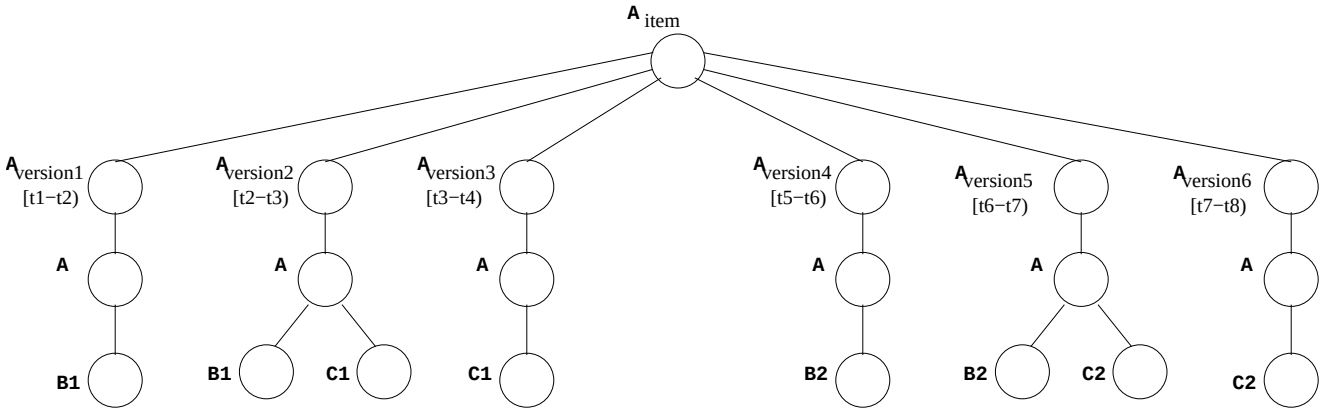


Figure 18: Example of pushUp

Figure 19: Algorithm: pushUp

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

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

7.1.2 The pushDown Function

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

Consider the example in Figures 14–17. According to the physical annotation in Figure 13, the tree-structured representation of the temporal document is given in Figure 17. Although <B> is a temporal element, timestamp is present at the element <A> higher up in the hierarchy. This results in the duplication of elements <A>, <C> and <D>. When pushDown function is applied to the above document, the timestamps are moved down the hierarchy, the redundancy is eliminated and the final document looks as shown in the first tree of Figure 14. At this point, the user might be wondering, what if the elements <C> and <D> are not the same in three different versions of <A> in the given temporal document. This would not happen, since the elements <C> and <D> are not defined to be time-varying in the temporal annotation; so they better be the same. If they are different, the algorithm would report this as an error.

The recursive algorithm for the pushDown function is given in Figure 21. The algorithm is called on the root element in the temporal XML document. If the root element is not an item, it is first converted to an item element using function createItem function. The algorithm moves the timestamps down the hierarchy one level at a time. If an item is not a time-varying element and if it has multiple versions (e.g. element <A> of Figure 16), it is converted into a single version by using the mergeVersions function. The function groups corresponding child elements having the same item-identifier from its different versions into the same child item. The child element from the first version is then replaced by its corresponding child item XML element. After merging, since the parent item has only single version, the item is replaced by its single version.

Other helper functions used in the algorithm are as follows.

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

Figures 23, 24 and 25 depict the stepwise working of function pushDown. For the given tree, element <D> is temporal in nature but the timestamp is present at the element <A> which is two levels up in the hierarchy. In the first step, the timestamp is moved to element <B>, while in the next step, the timestamps are moved to element <D>, which is actually a time-varying element.

7.1.3 The coalesce Function

As explained in Section 6, elements in two snapshots of a temporal XML document can be temporally-associated. If the elements are DOM-equivalent and the snapshot periods are contiguous, those two elements could be replaced by a single element with the time period extending from the start time of the first element to the stop time of the last element. This process is termed *coalescing* and is an integral part of SQUASH to compact the document.

After the snapshots are glued and the items are formed, coalesce is called for each item. The algorithm for coalesce is given in Figure 22. The algorithm compares the time-periods of the two contiguous versions. If they meet, and if the contents of the two versions are the same (i.e., if they are DOM-Equivalent as

Figure 20: Algorithm: pushDown

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

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

Figure 21: Algorithm: mergeVersions

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

Figure 22: Algorithm: coalesce

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

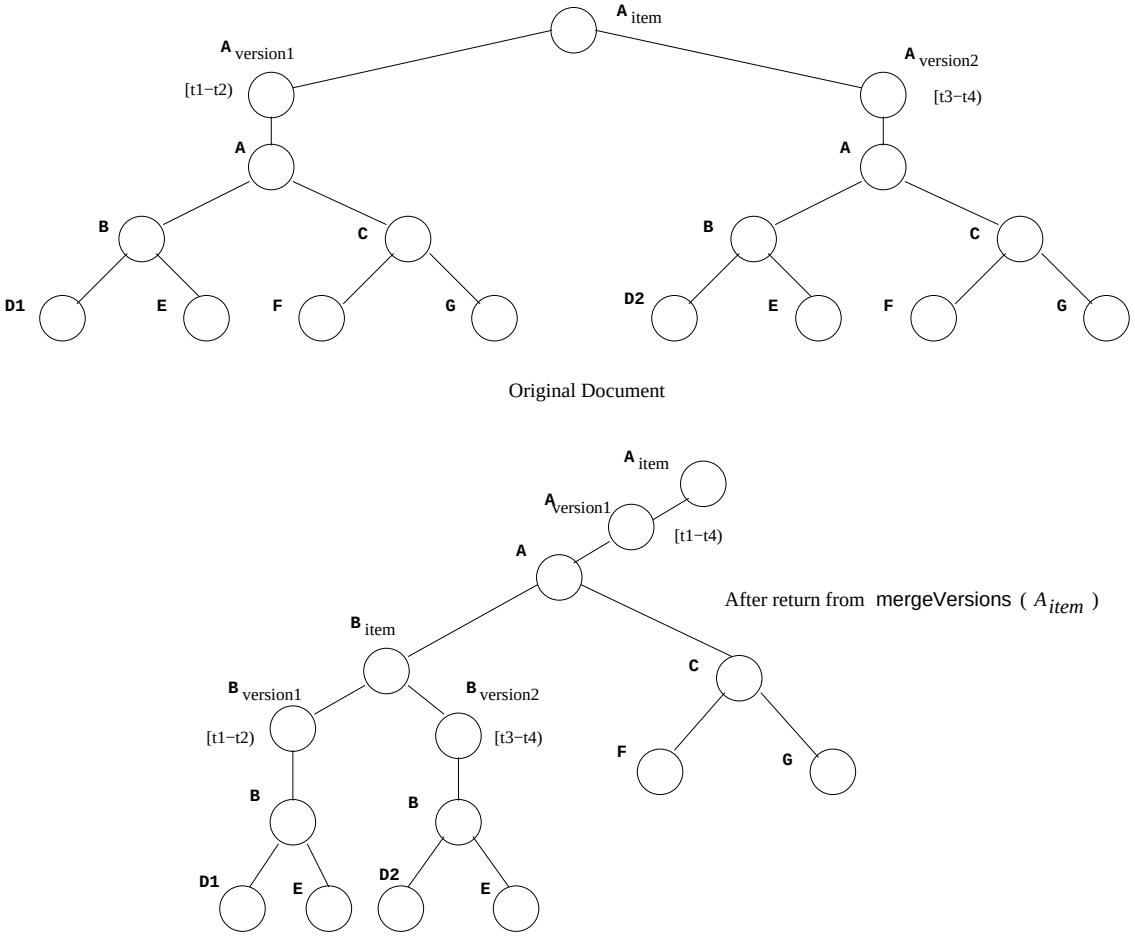


Figure 23: Example of pushDown

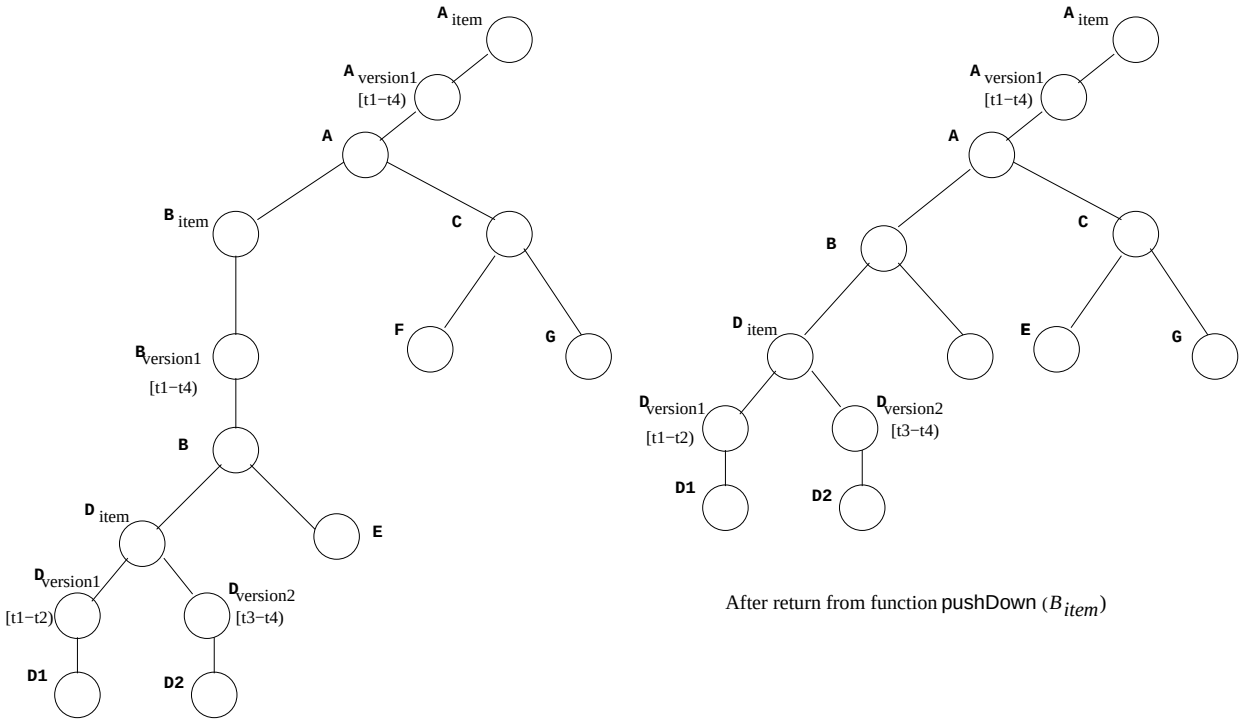


Figure 24: Example of pushDown: Continued

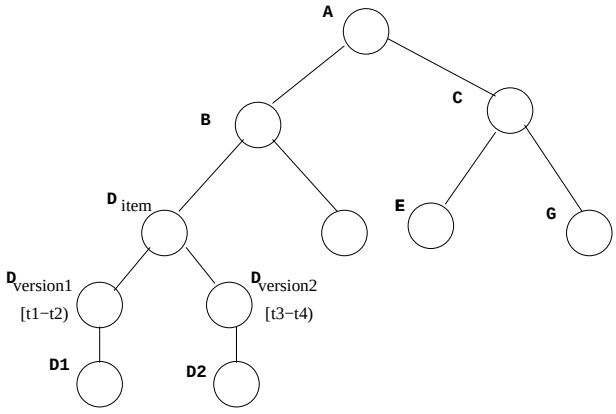


Figure 25: Example of pushDown: Continued

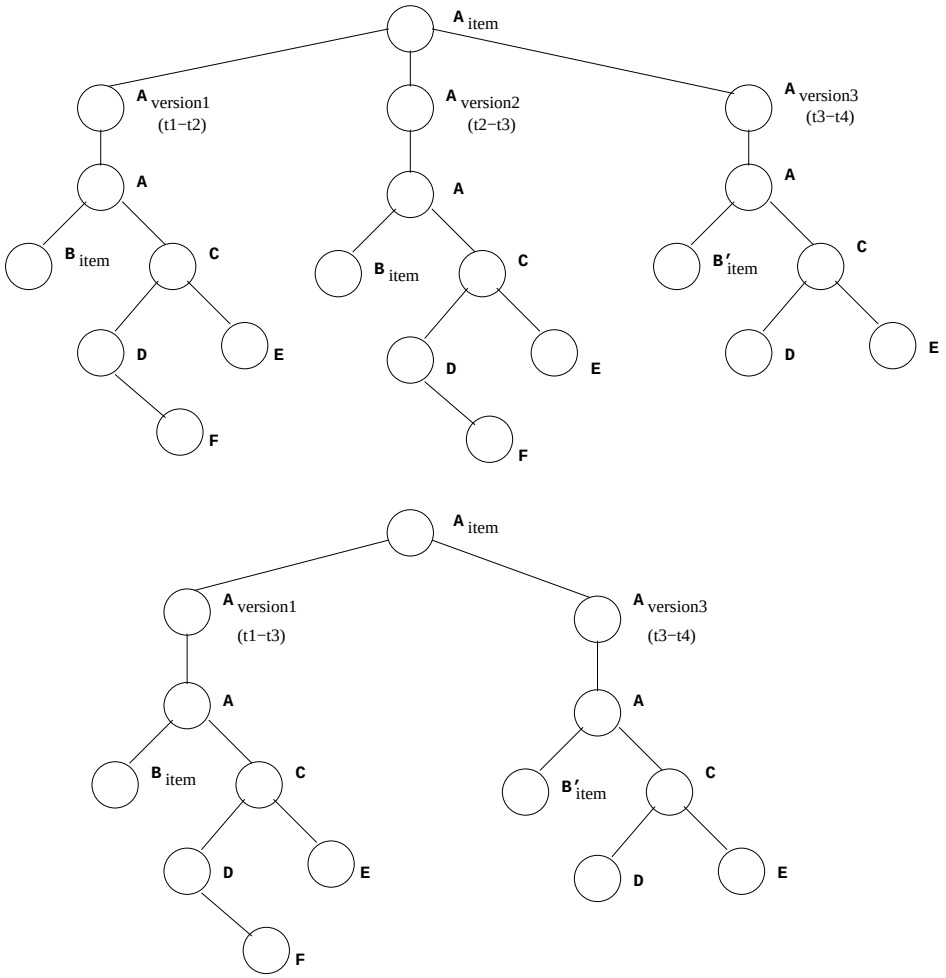


Figure 26: Example of coalesce

explained in Section 6.5), the stop time of the first version is then extended to the stop time of the second version.

Figure 26 shows the process of applying coalescing on Item A. In the tree-representation of the document, versions `A1 [t1-t2)` and `A2 [t2-t3)` are contiguous. They are also DOM-Equivalent (Section 6.5). Thus the two versions are replaced by a single version with time period `(t1-t3)`. After merging `A1` and `A2`, although the resulting version is contiguous with the next version `A3 [t3-t4)`, they are not merged, as they are not DOM-Equivalent. Thus, in the resulting document, there remain two versions `A1` and `A2`.

7.2 SCHEMA MAPPER

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

Once the annotations are found to be consistent, the logical-to-representational mapper (software oval of Figure 6) generates the representational schema (box 10) from the original snapshot schema and the temporal and physical annotations. The representational schema 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:

- Automatically from the non-temporal data using SQUASH tool.
- Automatically from the data stored in a database, i.e., as the result of a "temporal" query or view.
- Automatically from a third-party tool, or
- Manually.

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

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

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

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

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

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

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

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

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

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

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

OR

$t \Rightarrow$ <validTime begin="..." 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 *SItem*. For example, assume that the schema has a production of the following form:

$X \Rightarrow$ <*X*> β *S* γ </*X*>

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

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

The algorithm for SCHEMA MAPPER is shown in Figure 27. The algorithm uses the same procedure explained in the above paragraphs to create the representational schema from the snapshot schema. The helper function isConsistent checks whether the physical annotation is consistent with the given snapshot schema. As part of consistency, it checks whether all the targets in the physical annotation are present in the snapshot schema.

Figure 27: Algorithm: SCHEMA MAPPER

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

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

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

add following definition to the *snapshotSchema*

```
<xs:element name="timeVaryingRoot">
    <xs:complexType>
        <xs:element ref="currentRoot" />
    </xs:complexType>
</xs:element>
```

return modified *snapshotSchema*
else
display error

7.3 TEMPORAL VALIDATOR

Figure 28 provides the validation procedure used by τVALIDATOR . The temporal bundle document (box 5 of Figure 6) is passed through the τVALIDATOR which first checks to ensure that the temporal and physical annotations are consistent with the snapshot schema and with each other. Once the annotations are found to be consistent, the logical-to-representational Mapper (SCHEMA MAPPER) generates the representational schema (box 10) from the original snapshot schema and the temporal and physical annotations. The representational schema is needed to serve as the schema for a time-varying document and is used to validate the temporal document using conventional validator.

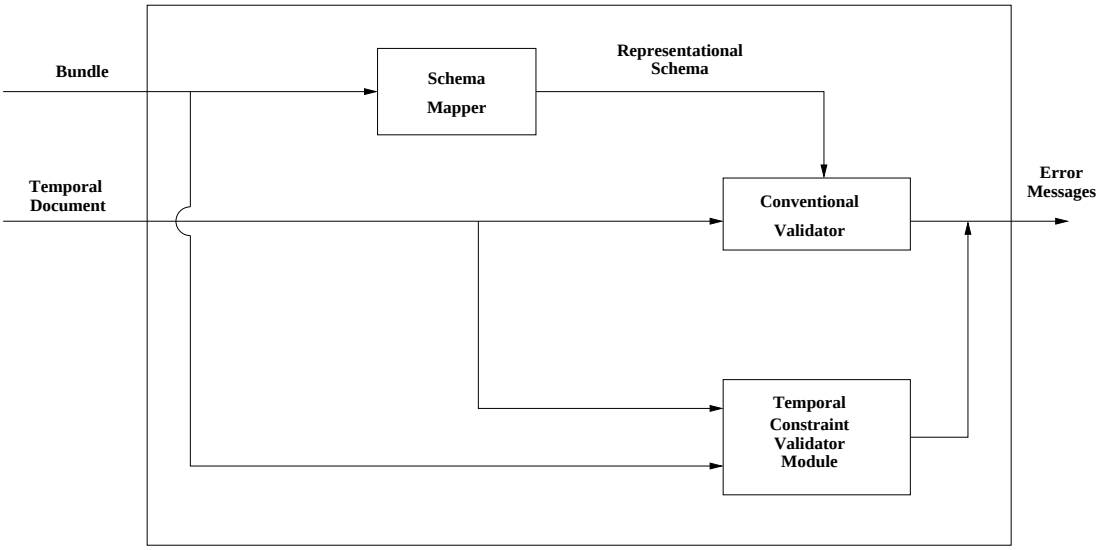


Figure 28: Validating a document with Time-Varying Data

Once the representational schema is ready, a conventional validator is used to parse and validate the time-varying data. The τVALIDATOR utilizes the conventional validator for many of its checks. For instance, it validates the temporal annotations against the TXSchema and physical annotation against the PXSchema. But using a conventional XML Schema validating parser is not sufficient due to the limitations of XML Schema in checking temporal constraints. So the second step is to pass the temporal data to *Temporal Constraint Validator Module*. The module, by checking the temporal data, effectively checks the non-temporal constraints specified by the snapshot schema simultaneously on all the instances of the non-temporal data (box 8), as well as the (non-sequenced temporal) constraints between snapshots, which cannot be expressed in a snapshot schema.

Figures 29 and 30 depict the two tasks performed by the τVALIDATOR. (i) validating the consistency of a temporal schema and (ii) validating the instance of a temporal document against the temporal schema.

The time-varying data is validated against the representational schema in two stages. First, a conventional XML Schema validating parser is used to parse and validate the time-varying data since the representational schema is an XML Schema document that satisfies the snapshot validation subsumption property. But 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 *Temporal Constraint Validator Module* of τVALIDATOR. So the second step is to pass the temporal data to τVALIDATOR as shown in Figure 28. A temporal XML data file (box 9 of Figure 6) is essentially a timestamped representation of a sequence

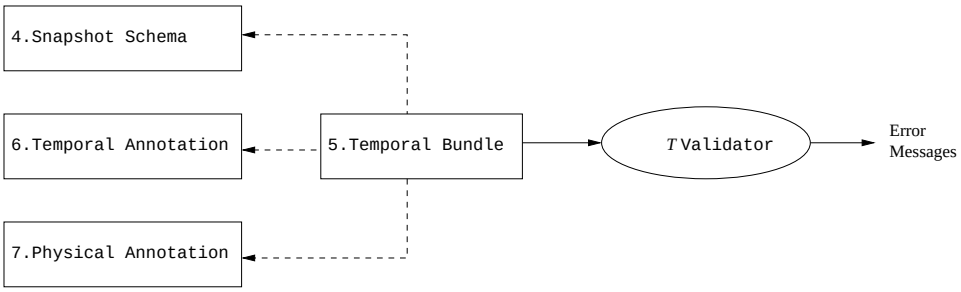


Figure 29: τVALIDATOR – Checking the Schema

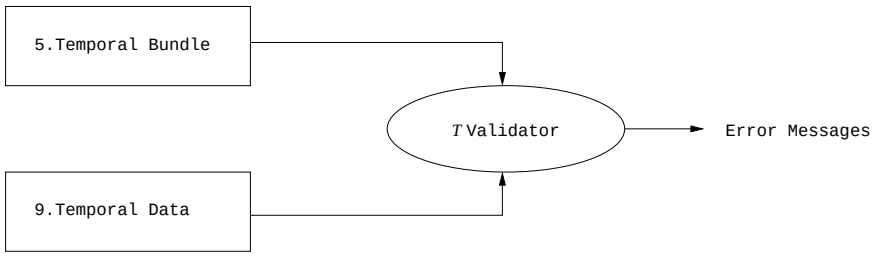


Figure 30: τVALIDATOR – Checking the Instance

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).

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

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

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

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

Existence Constant: The element cannot disappear and reappear again.

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

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

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

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

The algorithm for τVALIDATOR is given in Figure 31. The algorithm uses a hash-map to maintain a mapping between item-identifier and the corresponding item. After checking the consistency of the schemas, the function creates a representational schema using the SCHEMA MAPPER. The given temporal document is parsed against this schema using the conventional validator. The for loop creates the items by gluing together the elements with the same item-identifier. Each item is then validated for sequenced and non-sequenced constraints explained in Section 6.6.

7.4 SQUASH

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

The algorithm for SQUASH tool is given in Figure 32. It cleverly reuses pushUp, pushDown and coalesce primitives to create a compressed document from a set of snapshot documents as per the given temporal schema.

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

7.5 UNSQUASH

The UNSQUASH utility performs the opposite operation of SQUASH. It takes a temporal XML document, a temporal bundle and generates multiple non-temporal XML documents. It also provides the functionality of extracting a particular snapshot from the given temporal document using UNSQUASH utility. The algorithm for UNSQUASH is given in Figure 33.

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

7.6 RESQUASH

The RESQUASH utility takes the temporal XML data and the two physical annotated schemas (the original schema and the target one) and converts the temporal XML document based on the target physical annotated schema. The algorithm for RESQUASH is given in Figure 34.

The algorithm first checks for the consistency of the temporal annotation and the source and target physical annotations with the snapshot schema. It then performs the operation pushDown on the given temporal document. The given temporal document has the representation as per the *srcPhysicalAnnotation*. The

Figure 31: Algorithm: τVALIDATOR

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

Figure 32: Algorithm: SQUASH

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

Figure 33: Algorithm: UNSQUASH

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

pushDown function moves all the timestamps to the actual time-varying elements as per the *temporalAnnotation*. The function pushUp is then called with the *targetPhysicalAnnotation* as its parameter, which then moves the timestamps up in the hierarchy to the elements mentioned in the new physical annotation.

Figure 34: Algorithm: RESQUASH

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

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

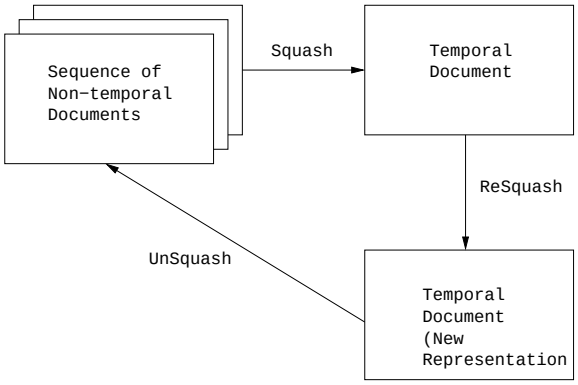


Figure 35: Squash/UnSquash/ReSquash Commutativity Diagram

SQUASH, UNSQUASH and RESQUASH tools retain snapshot reducibility [3] in that the commutativity diagram in Figure 35 is maintained. Specifically, if we take a particular sequence of static XML documents, each associated with a time slice, and squash them into a temporal XML document, then resquash that into a separate temporal XML document, with a different physical schema, and then unsquash it again, we will get exactly the same sequence of static XML documents. This of course assumes that the static documents corresponding to the non-temporal schema provided and that the temporal XML documents are valid instances of the schema produced by the logical-to-representational mapper.

8 Representations

In this section we present 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 annotation. τVALIDATOR and other utilities are designed to generate and recognize multiple representations of temporal document. Among the many physical representations, 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 considered the following criteria to select a candidate.

- 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.
- 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 temporal or non-temporal queries in XPath/XQuery/XSLT/DOM that can be expressed or evaluated.
- 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 representation of a versions lifetime must be unique. Two lifetimes that are the "same" time (i.e., two or more copies of the version of an item present at two different places in a temporal document) must have the "same" representation.

After applying all the above criteria, two representations were considered for the initial implementation. The final choice between them was a trade-off between processing complexity and the document compression.

Decomposed Representation In this representation, every item is present as a child of the top level `<timeVaryingRoot>` element and is given a unique `ID`. All possible occurrences of the corresponding item in the document are replaced by the elements referencing corresponding `ID`. XML datatype `IDREF` is used for referencing.

The decomposed representation works best when the timestamps are present at the time-varying elements. In this cases it is more space-efficient since duplication of items if any is avoided by using its referencing capability to the fullest.

Non-decomposed Representation In this representation, an item can occur at any level in the XML tree hierarchy. Every occurrence of the actual time-varying element from the snapshot document is replaced by its corresponding item. Multiple XML elements corresponding to the *same* item (i.e., item with the same item-identifier) may exist at multiple places in the document. This happens if the same element is being referenced at multiple locations in the original snapshot document. In that case, even though the item-identifier for all the item elements is the same, each may contain different sets of versions depending upon the time-period of enclosing items. All sets of versions must be consistent with each other and when combined, denote a single item with non-overlapping versions. The grammar for this representation is explained in detail in Section 7.2.

This representation is better from the processing complexity point of view and hence is easier to implement. It gives the same space-efficiency in most cases as the decomposed representation.

Both above representations have 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 sub element is inserted,
  - a sub element is deleted,
  - a sub element changes position, or
  - the text content changes.

  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.

We decided to support the non-decomposed representation since it is easier to implement and has the same space-efficiency as the decomposed representation in most cases. In this representation, 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 reasoning behind this design decision is given at the end of Section 11 when we discuss *transaction-time* and *valid-time splitting*.

9 Schema Versioning

Much of the power of a database management system stems from the presence of a schema that describes the structure of the database. When the data is versioned, a schema helps even more, because it expresses the commonality among the different versions, as well as indicating which parts of the data can change, and how. The schema is the solid ground upon which the data structures can stand. When the schema itself is versioned, there is no solid ground. How schema versioning is supported makes the difference between a fluid motion between versions and awkward struggling against quicksand.

One challenge is that in this potential quicksand, anything can change, and thus must be versioned: the snapshot documents, the base schema, the temporal annotations, the physical annotations, the schema documents included by these documents, even the schemas of these schema components. And, because the physical annotations can change, the concrete representation within a temporal XML document can also vary.

We now extend τXSchema to also support schema versioning. In doing so, we leverage both conventional XML Schema and related tools (principally, the validator), as well as the τVALIDATOR for data versioning.

In this section, we first explain the extensions to τXSchema architecture followed by the theoretical foundation for schema-versioning.

Before that, we introduce a key idea first appeared in a paper on temporal aggregation [31], that we will call here, *schema-constant periods*. It is possible, even with versioned schemas having themselves versioned schemas, to identify contiguous periods of time when there are no schema changes, anywhere. These are termed as schema-constant periods. These periods are non-overlapping and continuous; between the periods are schema change *walls*. Now, during these periods the data may be (and probably is) versioned, but at least we have a fixed base schema and fixed temporal annotations, each of which has a fixed schema. And since the physical annotations are fixed, the representation is also fixed, so it is possible to read and interpret the temporal document during that schema-constant period, and even to validate that portion of the document. So a general temporal document can be viewed as a sequence of data-varying documents, each over a single schema-constant period. Since we can validate each schema-constant period, given the approaches elaborated on earlier, all we have to do is validate across schema changes.

While schema versioning has been considered in the context of valid time [8], doing so is quite complex and in our opinion not worth this complexity. Thus in τXSchema schemas vary and are versioned only over transaction time.

9.1 Architecture and Example

We now generalize the architecture explained in Section 5 to support versioned schemas. Consider the Winter Olympic example explained in Section 2. We extend this example for schema versioning. All the files mentioned in this example are available in the examples directory in the distribution [33]. The example is also present in the Appendix B.

Consider the snapshot schema in Figure 5. This schema was initially designed on `2002-01-01`. The schema has been reproduced again in Figure 36 for convenience. This snapshot schema undergoes a series of changes. The corresponding temporal and physical annotations are given in Figures 37 and 38.

Now, assume that the designers decide to add a new element `<phone>` to the schema before beginning of Torino, Italy Olympics in 2006. The changes to the schema are done on `2005-01-01`. The modified schema is given in Figure 39. In the new schema, a new element `<phone>` for a phone number of an athlete is added as a child of `<athlete>` element. The existence of multiple versions of the base schema implies that box 4 of Figure 6 is actually a *sequence* of base schemas. Not only does the base schema changes over time, but the schemas included by it if any, could also vary over time. Similarly, the temporal annotations

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

Figure 36: `winOlympic.ver1.xsd`

```
<?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/country/athleteTeam/athlete">
    <transactionTime content="varying" existence="constant" />
    <itemIdentifier name="athID" timeDimension="transactionTime">
      <field path="@athID"/>
    </itemIdentifier>
  </item>
...
  <item target="/winOlympic/country/athleteTeam/athlete/medal">
    <transactionTime content="varying" existence="constant" />
    <itemIdentifier name="medalId1" timeDimension="bitemporal">
      <field path="./text"/>
      <field path="../@athID"/>
    </itemIdentifier>
  </item>
...
</temporalAnnotations>
```

Figure 37: `winolympic_temporal_annotation.ver1.xml`

```
<?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/country/athleteTeam/athlete">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>
...
  <stamp target="/winOlympic/country/athleteTeam/athlete/medal">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>
</physicalAnnotations>
...
```

Figure 38: winolympic_physical_annotation.ver1.xml

(box 6) and those annotations included by them and the physical annotations (box 7) and those annotations included by them all can vary over time, resulting in multiple versions.

As an example, on 2005-01-01, the designers also decide to make <phone> time-varying and hence the temporal annotation also undergoes a change. The modified temporal annotation is represented in Figure 40. A new item corresponding to the element <phone> is added to the temporal annotation. For this example, lets assume that the original physical annotation on 2002-01-01 remains as it is. Thus, even for the new schema, the timestamps will be represented at the elements <athlete> and <medal>.

This versioning is handled by timestamping the <schemaAnnotation> element in the temporal bundle. To each such element is added a <tTime> element that specifies when that annotation element became applicable. The sample temporal bundle document is given in Figure 41. The bundle contains two <schemaAnnotation> elements. They refer to the two versions of snapshot schemas and their corresponding temporal annotations. The first <schemaAnnotation> element contains <itemIdentifierCorresponde element as its child, which will be explained in detail in Section 9.2.1.

One approach to handle this schema versioning is to have a different document (file) for each version, similar to what is shown in box 8. While this approach is allowed, τXSchema also permits temporal schemas, in place of multiple versions of conventional schemas. As one possibility, the sequence of snapshot schemas could be squashed together to produce a single temporal document tv_snapshot.xml, which would then be referenced by multiple schema annotation elements. Similarly, the SQUASH utility could be used to generate temporal schemas for the schemas included by the main snapshot schema.

This rather involved state of affairs, with time-varying documents and time-varying schemas, is illustrated with a T Diagram in Figure 42. In this notation, first described almost forty years ago [4], the input of a translator is given on the left arm of the "T" (for example, for SCHEMAMAPPER in the upper right-hand-side of the figure, the input is the logical schema document, bundle.xml), the name of the translator is given at the base of the "T" (here, "SchemaMapper"), and the output of the translator is given on the right arm of the "T" (here, a representational schema, rep.xml). The name of these diagrams was to the best of our knowledge given by McKeeman, Horning, and Wortman in their classic compiler book [22].

We extend these diagrams to allow multiple inputs, which unfortunately complicates them somewhat. As shown in Figure 42, SQUASH takes both a bundle and a sequence of snapshot documents and produces a

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

Figure 39: winOlympic.ver2.xsd

```
<?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/country/athleteTeam/athlete">
    <transactionTime content="varying" existence="constant" />
    <itemIdentifier name="athNumber" timeDimension="transactionTime">
      <field path="@athNumber"/>
    </itemIdentifier>
  </item>
...
  <item target="/winOlympic/country/athleteTeam/athlete/phone">
    <transactionTime/>
    <itemIdentifier name="phoneId" timeDimension="transactionTime">
      <field path="./countryCode"/>
      <field path="./phoneNumber"/>
    </itemIdentifier>
  </item>
...
</temporalAnnotations>
```

Figure 40: winolympic_temporal_annotation.ver2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<temporalBundle xmlns="http://www.cs.arizona.edu/tau/tauXSchema/TBSchema"
                    xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <format plugin="XMLSchema" granularity="date"/>
    <bundleSequence defaultTemporalAnnotation="defaultTA.xml"
                    defaultPhysicalAnnotation="defaultPA.xml">
        <schemaAnnotation snapshotSchema="winOlympic.ver1.xsd"
         temporalAnnotation="winolympic_temporal_annotation.ver1.xml"
         physicalAnnotation="winolympic_physical_annotation.ver1.xml">
            <tTime>2002-01-01</tTime>
            <itemIdentifierCorrespondence oldRef="athID" newRef="athNumber"
                                          mapping="useBoth">
        </schemaAnnotation>
        <schemaAnnotation snapshotSchema="winOlympic.ver2.xsd"
         temporalAnnotation="winolympic_temporal_annotation.ver2.xml"
         physicalAnnotation="winolympic_physical_annotation.ver1.xml">
            <tTime>2005-01-01</tTime>
        </schemaAnnotation>
    </bundleSequence>
</temporalBundle>
```

Figure 41: `winOlympic_temporal_bundle.xml`

temporal document, and UNSQUASH does just the opposite (this is illustrated for the temporal annotations, which are SQUASHed into a single `tv_temp_anno.xml` document, then UNSQUAHed back into their constituent time slices.

In this figure we show a bundle (`bundle.xml`, right in the middle of the figure, with the arrows pointing left) referencing two temporal schemas, one of the base schema and one of the physical annotations; the bundle also references several temporal annotation documents. Note that the base schema for the base schema (!) is XSchema, which has as its base schema `XSchema.xsd`.

τVALIDATOR treats each URI it encounters as the specification of a temporal timeslice operation to select the appropriate version. The timeslice is as of the time of the document or context that contains the URI. If the URI represents a temporal document, the τVALIDATOR calls UNSQUASH, passing it (a)the corresponding bundle, (b) the temporal document, and (c)a timestamp. It would do so as well, for all the schemas included by that schema if any. The underlying semantics ensures that at any point in time, there is a single base schema, a single temporal annotation, and a single physical annotation. The τVALIDATOR recursively calls UNSQUASH so that at any point in time, there is a single schema in effect.

The snippet of a sample temporal document generated by the SQUASH utility is shown in Figure 43. The document uses two representational schemas http://www.cs.arizona.edu/tau/RepSchema0 and http://www.cs.arizona.edu/tau/RepSchema0 for schema constant periods `[2002-01-01, 2005-01-01)` and `[2005-01-01, 9999-12-31)`, respectively. The data versioned temporal documents for those schema-constant periods are embedded inside `<schemaVersion0>` and `<schemaVersion1>` elements.

What would the representational schema look like for this temporal document? We could see that schema directly by running SCHEMAMAPPER on the bundle. The SCHEMAMAPPER for the schema versioned documents generates `<schemaVersion`*i*`>` element within `<schemaItem>` for every change of the base schema or the physical annotation. The representational schema is given in the Figure 44. The final representational schema for the schema-versioned temporal document is a sequence of `<schemaVersion`*i*`>` elements corresponding to different schema-constant periods.

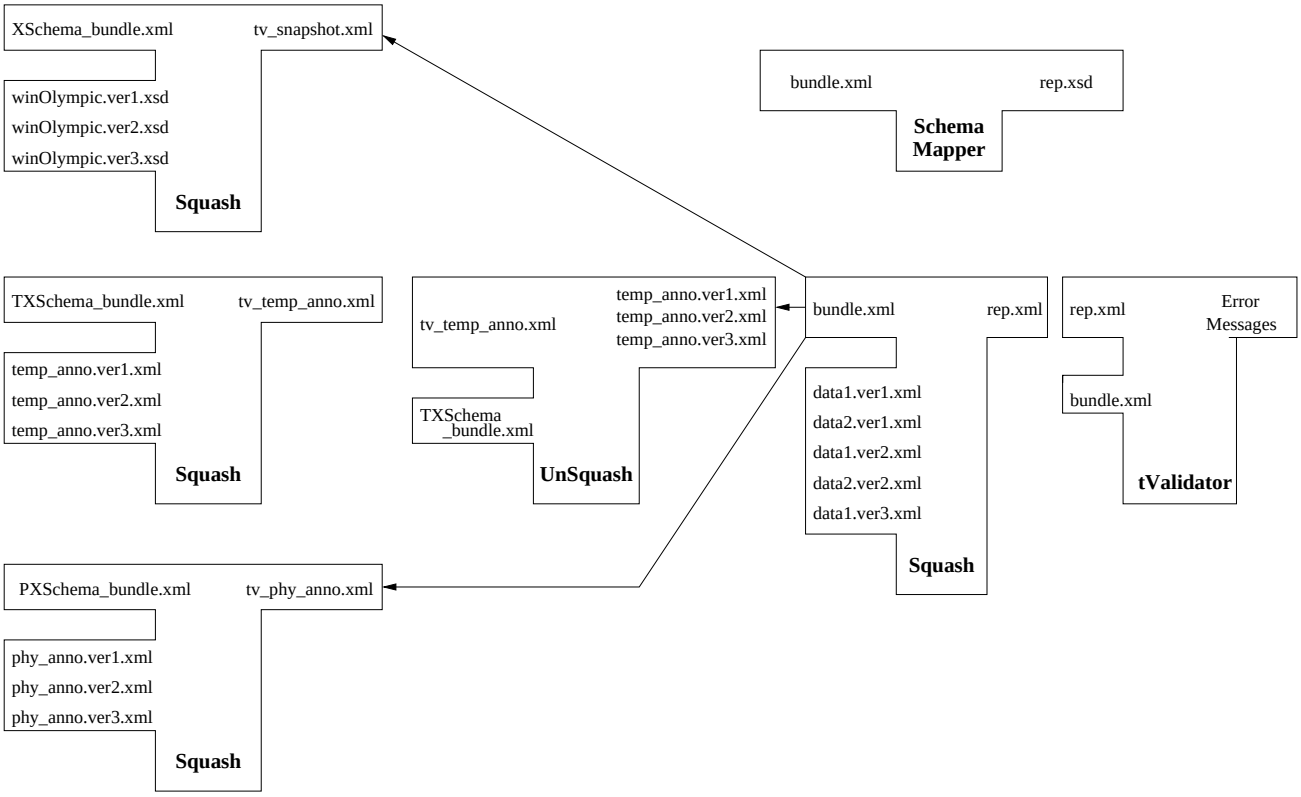


Figure 42: T Diagram of Validation

```
<?xml version="1.0" encoding="UTF-8" ?>
<rep:sv_root
    xmlns:rep="http://www.cs.arizona.edu/tau/RepSchema"
    bundle="winolympic_bundle.xml"
    xmlns:rep0="http://www.cs.arizona.edu/tau/RepSchema0"
    xmlns:rep1="http://www.cs.arizona.edu/tau/RepSchema1"
    xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.cs.arizona.edu/tau/RepSchema
                        winolympic_rep1.xsd">
    <schemaItem>
        <schemaVersion0>
            <tv:timestamp_TransExtent begin="2002-01-01" end="2005-01-01" />
            <rep0:tv_root>
                <rep0:winOlympic_RepItem originalElement="winOlympic">
                ...

                ...
                </rep0:winOlympic_RepItem>
            </rep0:tv_root>
        </schemaVersion0>

        <schemaVersion1>
            <tv:timestamp_TransExtent begin="2005-01-01" end="9999-12-31" />
            <rep1:tv_root>
                <rep1:winOlympic_RepItem originalElement="winOlympic">
                ...

                ...
                </rep1:winOlympic_RepItem>
            </rep0:tv_root>
        </schemaVersion1>
    </schemaItem>
</rep:sv_root>
```

Figure 43: `tv_winOlympic.xml`

```
<?xml version="1.0" encoding="UTF8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="unqualified"
    targetNamespace="http://www.cs.arizona.edu/tau/RepSchema"
    xmlns="http://www.cs.arizona.edu/tau/RepSchema"
    xmlns:rep0="http://www.cs.arizona.edu/tau/RepSchema0"
    xmlns:rep1="http://www.cs.arizona.edu/tau/RepSchema1"
    xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema">
  <xsd:import namespace="http://www.cs.arizona.edu/tau/TVSchema"
                          schemaLocation="TVSchema.xsd" />
  <xsd:import namespace="http://www.cs.arizona.edu/tau/RepSchema0"
                          schemaLocation="rep0.xsd" />
  <xsd:import namespace="http://www.cs.arizona.edu/tau/RepSchema1"
                          schemaLocation="rep1.xsd" />
  <xsd:element name="sv_root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="schemaItem">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element maxOccurs="1" minOccurs="1" name="schemaVersion0">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element maxOccurs="1" minOccurs="1"
                                 ref="tv:timestamp_TransExtent" />
                    <xsd:element maxOccurs="1" minOccurs="1"
                                 ref="rep0:tv_root" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element maxOccurs="1" minOccurs="1" name="schemaVersion1">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element maxOccurs="1" minOccurs="1"
                                 ref="tv:timestamp_TransExtent" />
                    <xsd:element maxOccurs="1" minOccurs="1"
                                 ref="rep1:tv_root" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="bundle" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 44: winOlympic_rep_schema.xsd

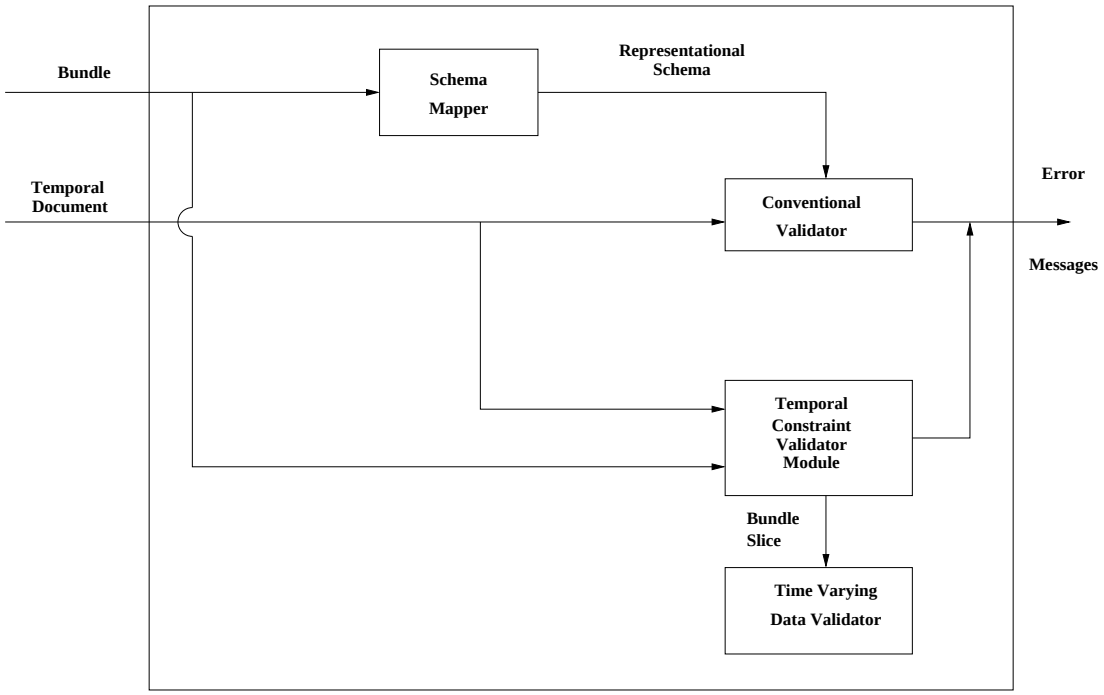


Figure 45: Validating a Document with Time-Varying Schema

Figure 45 shows the method for validating a document with time-varying schema. To validate such a document, τVALIDATOR applies the conventional validator to the document, using the representational schema produced by SCHEMAMAPPER. It then determines the times when the schema changes, thus determining schema-constant periods. For each such period, the time-varying data checker is invoked to check the temporal integrity constraints over the time-varying data, with the single base schema, temporal annotation, and physical annotation. Then the *temporal constraint checker* glues across the schema change walls and performs the temporal checks across these walls.

9.2 Theoretical Framework

The above arrangement works very well. However, there are four remaining aspects that do not show up with time-varying data, but rather are unique to versioned schemas: (1) an evolving definition of keys, (2) accommodating gaps in lifetimes, (3) the semantics of mixed data and schema changes, and (4) checking non-sequenced constraints across schema changes. We examine each in turn in this section.

9.2.1 Accommodating Evolving Keys

When documents vary over time, it is important to identify which elements in successive snapshots are in actuality the same item, varying over time. We refer to the process of associating elements that persist across various snapshots as *gluing* the elements. SQUASH must do this gluing; the time-varying data checker within τVALIDATOR must also on occasion glue elements.

When a pair of elements is glued, an *item* is created. An item is an element that evolves over time through various versions. Determining which elements should be glued depends on two factors: the *type* of the element, and the *item identifier* for that element's type. The item identifiers and gluing of elements to form items is already explained in detail in Section 6.4.

When a schema-change wall is encountered, items across the wall need to be associated. This process is called as *cross-wall gluing*, or *bridging*. Figure 46 depicts the concepts of gluing and bridging.

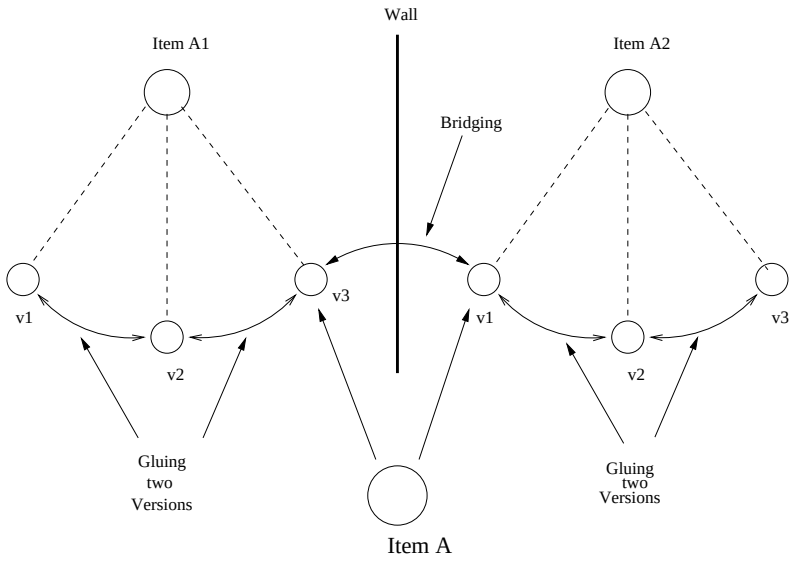


Figure 46: Gluing and Bridging

In this figure, individual elements in individual versions of an XML document are depicted as small circles in the center of the figure. Here we see six elements, three of which are determined to be versions of the same item (A1) and three of which are determined to be versions of another item (A2). The wall indicates that the schema was changed between the third and fourth version of the document.

Gluing uses the item identifier to associate the first three elements with an item and likewise the next three elements. Bridging determines that the element that is version 3 of item A1 and the element that is version 1 of item A2 are actually versions of the same item, item A. So in fact item A has *six* versions, the three elements before the schema change and the three elements after the schema change. Gluing and bridging occur in different stages within the validator; both conspire to realize an item across schema

changes, which is the first step in checking the temporal constraints associated with that item's definition in the schema.

What is relevant for our purposes here is that item identifiers specified in the temporal annotations, are usually the (snapshot) key of the element type [5] given in the base schema, and are used by τVALIDATOR to extract the items from the temporal document and then check the temporal constraints on those items.

What if either the snapshot key (specified in the base schema) upon which an item identifier is defined, or if the item identifier itself (specified in the temporal annotation) changes? This is a particularly insidious kind of quicksand. Even worse is when the underlying element type of an item changes. As an example, if the `<athlete>` element in the `winolympic.ver1.xsd` is replaced by `<player>` in the future versions, an item that was a particular `<athlete>` element before the schema change could be associated with a particular `<player>` element in the snapshot document associated with the later schema.

Our solution is to put in the `<schemaAnnotation>` element, which signals a change in some aspect of the schema, an `<itemIdentifierCorrespondence>` element, specifying how old item identifiers are to be mapped to new item identifiers. This element has four attributes: `oldRef`, a string naming an item that appears in the old schema, `newRef`, a string naming an item that appears in the new schema, `mappingType`, an XML Schema enumeration, and optionally a `mappingLocation`, which is a URI. We have defined four mutually exclusive mapping types.

- `useNew`: The new identifier must also be present in the old element.
- `useOld`: The old identifier must also be present in the new element.
- `useBoth`: An attribute's name is changed, but its value isn't.
- `replace`: Use an externally-defined mapping.

This could be best described with an example. Say that in 2002 the item identifier is the `athID` attribute of the `<athlete>` element. In January 2005, this attribute is renamed `athNumber`; we specify a mapping type of `useBoth`. In March 2005, the item identifier is changed to the `athName` element, with a mapping type of `useNew`. (This attribute has been around since 2002, but it wasn't used as a key until January 2005.) Assume that, in June 2005 we add a new attribute, `athKey`, and specify that as the item identifier, with a mapping type of `useOld`. Finally, in July 2005, just before the beginning of the games, we replace the `<athlete>` element with a `<player>` element, with a `playerID` attribute as the item identifier and a mapping type of `replace`.

The gluing of elements into items is then done the following way. Before 2005, the `athID` is used for gluing. When the schema change occurs sometime in January 2005, we glue across the schema change by matching the `athID` value of the element before the schema change with the `athNumber` value after the change: these (integer) values must match for the two elements to be glued. In March 2005, we glue across the schema change by matching up old elements and new elements that have the same (string) value for their `athName` element, the new item identifier. The only difference is that before the schema change, that element was present but wasn't being used as a key. In a consistent fashion, in June we also glue using the `athName` element, which was the *old* item identifier.

July is the most complex. We need to glue an `<athlete>` element with an item identifier of `athKey` with a `<player>` element with an item identifier of `playerID`. For this, we use the `MappingLocation` attribute in the bundle to access a mapping table that provides a list of pairs, each with an `athKey` and a `playerID` value.

This list of pairs is termed a *replace mapping list*. As it is instance-based, containing as it does a list of key *values*, the replace mapping list should only be used as a last resort. Its role is to allow bridging for all cases in which the other three mapping types, which have no need for storing instance information in the schema, are not appropriate.

Of course, the mapping location document can also be time-varying; τVALIDATOR extracts the relevant timeslice with UNSQUASH.

9.2.2 Accommodating Gaps

Bridging is more involved when there are *gaps* in the lifetime of an item. Gaps make the process of finding the correspondence between the items from consecutive schema-constant periods more difficult. If there are gaps in the lifetime of an item, bridging becomes even more complex.

Figure 47 shows three cases that may arise while bridging the items from consecutive schema-constant periods. It shows the data and schema changes along the transaction-time dimension, from left to right. The schema-change walls are shown as bold vertical lines. The horizontal lines depicts the evolution of a particular item (in this case, three separate items). The bridging process is shown by the jumpers over schema change walls. An absence of a line indicates when the item did not exist in the database. The first item existed during the entire transaction time period depicted in this figure. There is a single gap in the existence time of the second item: it ceased to exist sometime during P_1 but reappeared in P_2. The third item had a much longer gap, reappearing only in P_3.

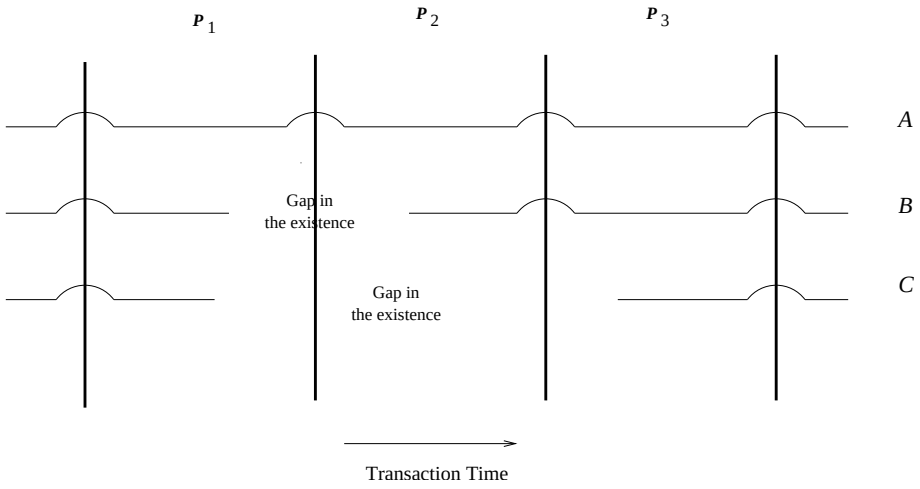


Figure 47: Presence of Gaps

We now now examine each item in turn.

1. The item A (the first horizontal line) is present throughout schema-constant periods P_1 and P_2. Thus the last snapshot of P_1 and the first snapshot of P_2 contains versions of item A.

2. The item B (the second horizontal line) disappeared for some time in P_1 and reappeared about halfway through in P_2. Thus the last snapshot of P_1 and first snapshot of P_2 will not contain versions of item B.

3. An item could also disappear for one or more schema-constant periods and then reappear again. Item C (the bottom horizontal line) was present for initial part of P_1. It then disappeared over entire period P_2 and again appeared in the later half of P_3.

For the first case, no extra work is needed as the items can be bridged directly using one of the above four methods.

But, to handle cases 2 and 3, the following two approaches were considered.

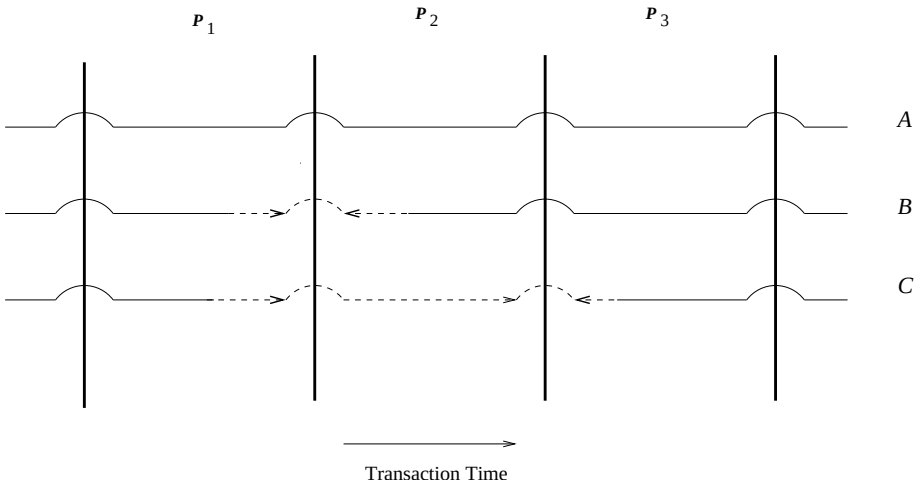


Figure 48: Cross Wall Gluing: Option 1

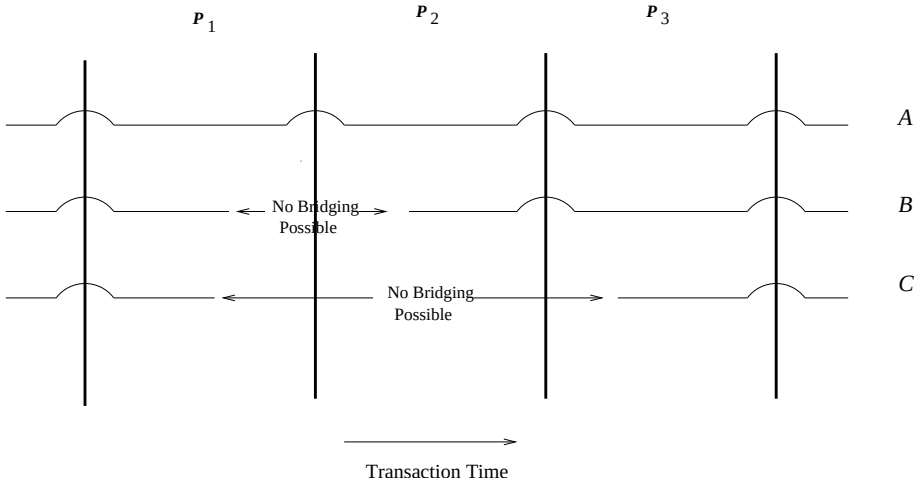


Figure 49: Cross Wall Gluing: Option 2

- Associate the pieces of an item across a schema change wall by virtually extending period of versions of the item. As an example, in Figure 48, bridging the two pieces of item B involves virtually extending period of item B's last version until the end of P_2 as if it were present during the last snapshot; and virtually extending its first version's period until the start of P_3; and then performing the bridging using one of the above four methods. Similarly, for the item corresponding to the third line, bridging involves virtually extending the period of the item C's last version from P_1 over multiple schema-constant periods followed by bridging using one of above methods. So P_1's version is extended to the wall, then bridged to a virtual element over all of P_2, then bridged to the extended element in P_3.
- The second option is not to extend the "item" across a schema change wall if it does not exist. So the item matching semantics, e.g., "useNew" matches only those items that exist immediately before the wall with those that exist immediately after the wall. As an example, in Figure 49, bridging the two pieces of items B and C having gaps in their existence across the schema change walls is not possible.

We decided to take the second approach, since we couldn't really "know" a priori if an item that reappears is the same item or a different one from the earlier one.

9.2.3 Semantics for mixed data and schema changes

A data change in XML documents can co-exist with schema changes within a single transaction, and hence can occur at exactly the same (transaction commit) time. With schema changes coming into picture, we also need to consider other factors like name and relative path changes for item identifier fields and other elements that constitute the content of an item, complicating the process of bridging and hence validation.

We considered three ways to handle this situation.

1. Not allow any data change in a transaction containing schema changes. This is the most stringent option and makes the user's job more difficult, forcing him to split the task into multiple transactions. This may not be always feasible from real world point of view. Consider a situation where an element is modified to have a new 'required attribute', data change is mandatory in this case and hence cannot be separated from schema change. It could be argued that this is achievable with addition of a new 'optional' attribute, followed by required data changes and then making the attribute required. But it requires more work from the user's side.

2. Allow schema changes to coexist with data changes, except for schema changes to item identifier fields. This will eliminate the need of replace mapping list and the bridging could always be done using one of the three options '`useNew`', '`useOld`', or '`useBoth`'.

3. Allow data changes to coexist with schema changes within a transaction without any restrictions.

We decided to go with the third approach, as it is the most general. A schema change for an element can consist of changes to its structure or its attributes or to the element definitions nested within it. Thus, given two schemas, it becomes very difficult to find the difference between the schemas and to validate the versions. So, we decided not to validate versions of an item across schema change walls if a schema change is detected for it.

9.2.4 Non-Sequenced Constraints

A constraint is *non-sequenced* if it is applied to a temporal item as a whole (including the lifetime of the data entity) rather than to individual time slices. They are defined in the temporal annotation as an extension of snapshot XML Schema constraints. An example of a non-sequenced (cardinality) constraint is: "An item cannot change more than three times in a year.". This type of constraint cannot be validated using the conventional validator and thus needs to be validated using the 'Temporal Constraint Checker' module of τVALIDATOR.

As mentioned earlier, schemas vary only over transaction time. Hence, non-sequenced constraint validation is easier in valid time, as schema changes cannot occur.

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

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

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

Consider a situation shown in Figure 50. It maintains the same conventions as Figure 48. Changes to an item are shown by X's. A new non-sequenced constraint is introduced during third schema-constant period P_3 stating that "An item cannot change more than three times in a year." But the item has already undergone four changes during previous schema-constant periods P_1 and P_2.

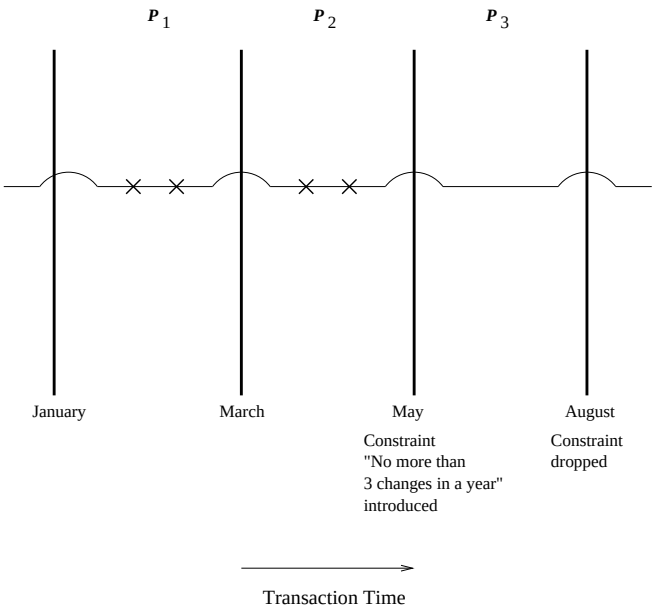


Figure 50: Non-Sequenced Constraints

According to first alternative listed above, the constraint is not violated as long as the item does not change more than three times in the third schema-constant period. Until there are four changes made after the schema change, the constraint is not considered to be violated.

According to the second alternative semantics, there is immediately a violation of the constraint, due to activity during the previous two schema-constant periods.

We decided on the first alternative: to apply a non-sequenced constraint only within the schema-constant period in which it is defined. Thus the non-sequenced constraints are "turned off" on any schema change. So for instance a constraint that says that the content must be constant is checked only up to the schema wall, and then checked within the new schema starting from the wall. In effect the schema change deletes all the old constraints and then adds them back as new constraints.

10 Implementation

10.1 Technology

τVALIDATOR and other tools have been written in Java and have been developed using Java 2, Standard Edition, v1.4. They use W3C specifications APIs for parsing the XML documents, building DOM trees and processing XPath expressions. 'W3C DOM API' is used for parsing the XML documents. 'XML Path Language (XPath) Specification Version 1.0' is used for processing XPath expressions. Third party implementations of these APIs from Apache Software Foundation available as part of Apache XML project [2] were used. The details of these implementations are given below.

- XERCES, a part of the Apache XML project is a family of software packages for parsing and manipulating XML documents. Xerces provides both XML parsing and generation. Xerces provides the implementation for the W3C DOM API. The implementation is available under 'Apache Software License' and is available freely [35].
- XALAN, a popular open source software library from the Apache Software Foundation, is used as an implementation of XPath API. It implements the XSLT XML transformation language and the XPath XML query language. The implementation is available under 'Apache Software License' and is available freely [34].

10.2 Class Diagram

The class diagrams for the tool implementation are given in the Figures 51–53. The classes are divided into three packages.

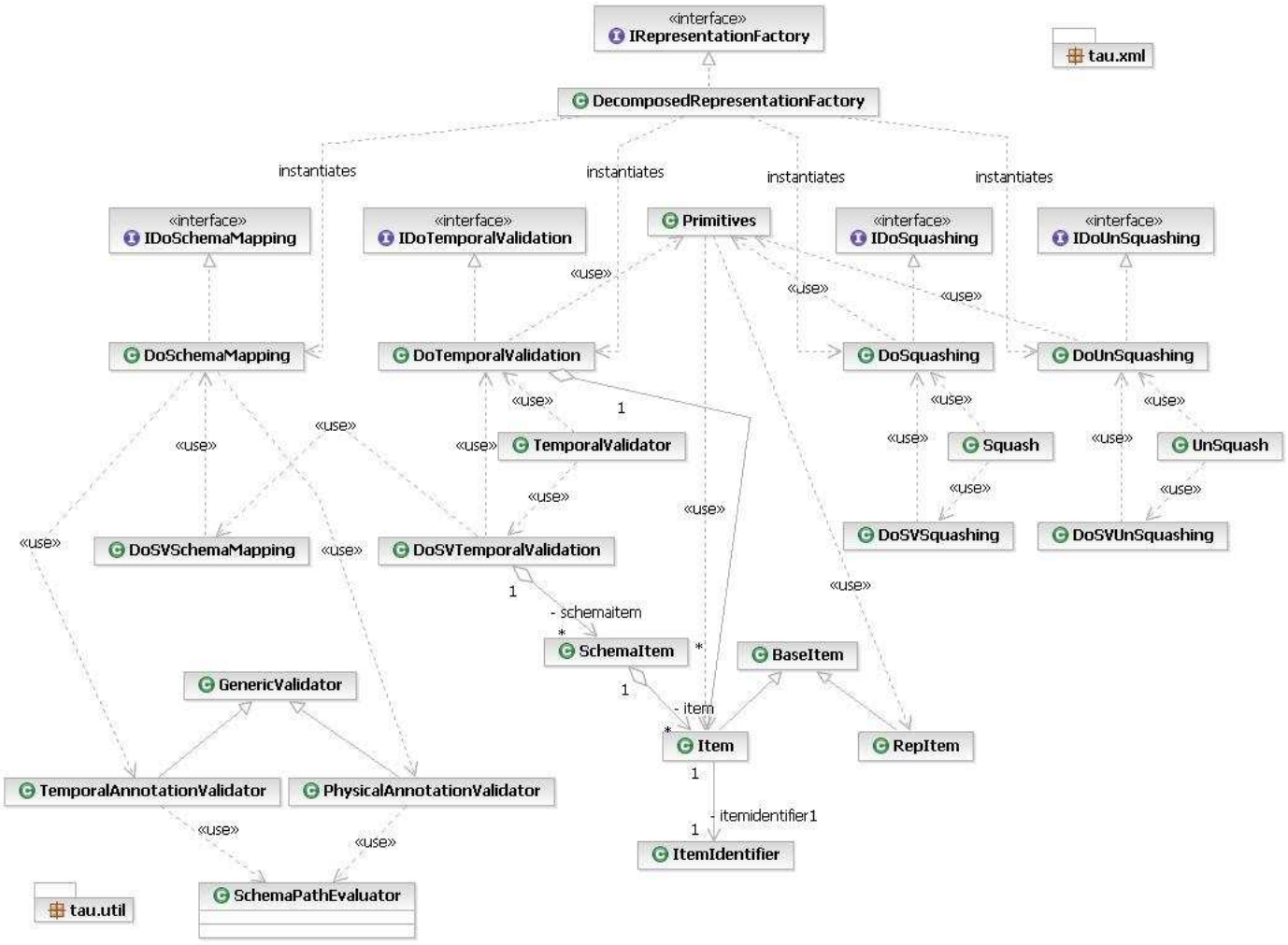


Figure 51: Overview class diagram for the tools

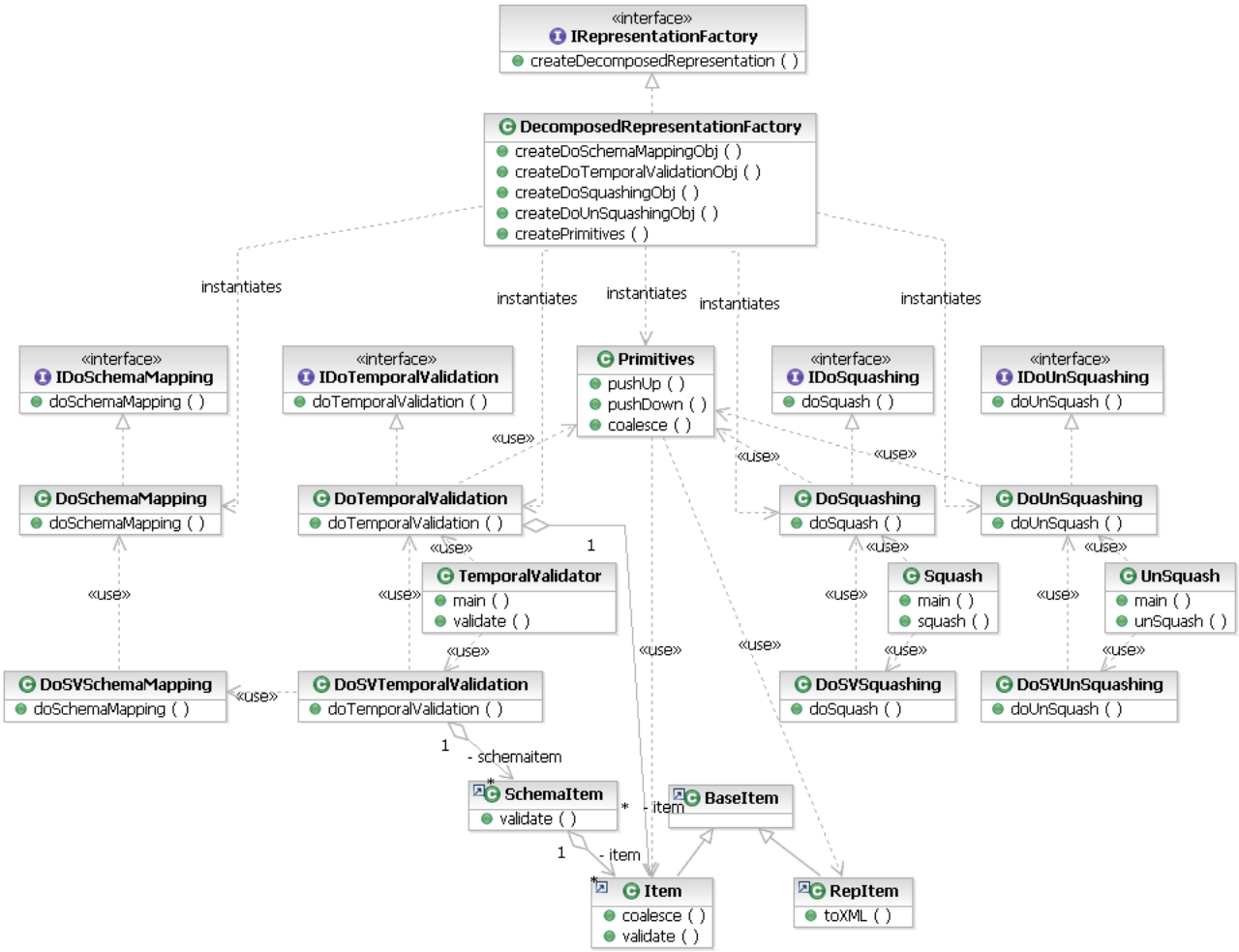


Figure 52: Detailed class diagram for `tau.xml`

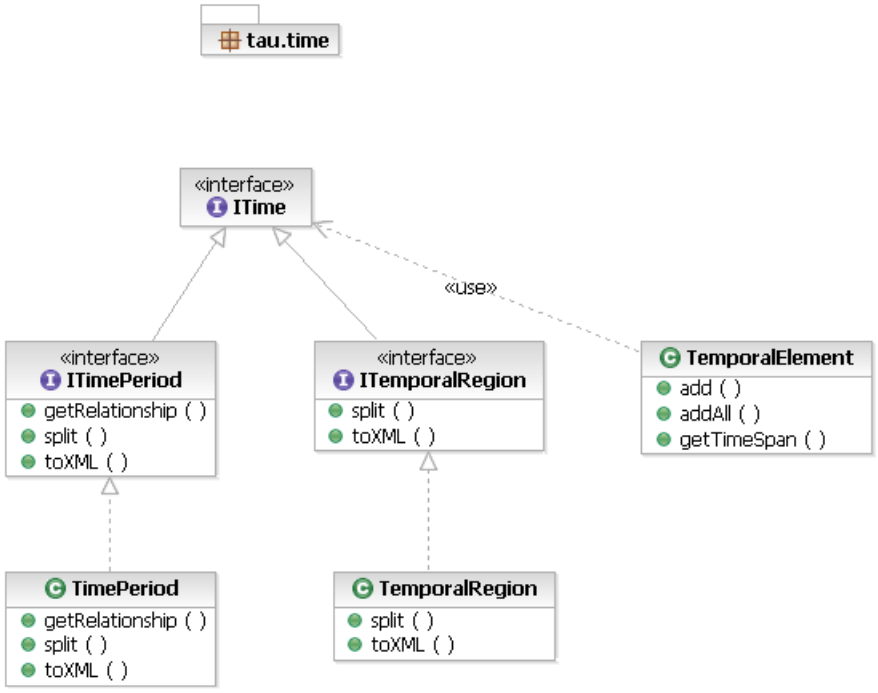


Figure 53: Detailed class diagram for `tau.time`

tau.xml This package contains interfaces and classes corresponding to tools τVALIDATOR, SCHEMAMAPPER, SQUASH, and UNSQUASH. The details of the important classes used for data versioning are given below.

- `Item`: Provides an abstraction for a logical item. It contains methods for manipulating versions, their coalescing validation.
- `RepItem`: Provides an abstraction for actual representation item element in the XML document. It provides methods for conversion of an XML element to/from a logical item. Both these classes extend from the base class `BaseItem`, which provides common functionality.
- `ItemIdentifier`: Provides an abstraction for item-identifier.
- `Primitives`: Provides implementation for primitives explained in Section 7.1.
- `TemporalAnnotationValidator` and `PhysicalAnnotationValidator`: Provide checks for the consistency of temporal and physical annotations with the snapshot schema.
- `DoSchemaMapping`, `DoSquashing`, `DoUnSquashing`, `DoTemporalValidation`: Provide the implementation for the algorithms explained in Section 7. Each of the classes implement corresponding interfaces preceding their name by '`I`'. As an example, `DoSquashing` implements `IDoSquashing`.

The extended tools for schema versioning use these classes internally to manipulate schema-versionsed XML documents. The classes used for schema-versioning are `DoSVDataSquashing`, `DoSVDataUnSquashing`, `DoSVTemporalValidation`, and `DoSVSchemaMapping`, where '`SV`' stands for 'schema-versioned'. The implementation of these classes first identify schema-constant-periods and call corresponding data-versioning classes on individual schema-constant-periods.

The classes `Squash`, `UnSquash` and `TempralValidator` provide commond-line tools for the end-user. These classes accept temporal bundle and configuration files as command line parameters and internally invoke schema-versioning or data-versioning tools depending upon whether the schema is versioned or not.

tau.time This package contains the implementation of classes to handle time. It provides implementation for both `TimePeriod` (used for single time dimension) and `TemporalRegion` (used for bitemporal elements).

tau.util This package contains utility classes. `SchemaPathEvaluator` provides abstraction for evaluating schemapath expressions explained in Section 6.2. Given a target and reference element, the function checks for the consistency of the target according to the snapshot schema. This functionality is used by both `TemporalAnnotationValidator`, `PhysicalAnnotationValidator` and `ItemIdentifier`.

As explained, the class for every tool implements its corresponding interface. Thus, it is easily possible to accommodate a new implementation of these tools for a new representations without necessitating many changes to the overall picture. Use of 'Abstract Factory' design pattern makes the integration and selection of the new representation seamless by addition of just a few lines of codes to the `RepresentationFactory` class.

To add a new representation, we need to add new classes implementing the new representation for each tool. Each class needs to implement the corresponding interface mentioned earlier. Once these classes are

added, a small addition of code is needed to the `RepresentationFactory` class. Then, any representation can be easily selected by providing corresponding parameter to the `RepresentationFactory` class.

11 Support for Bitemporal Data

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

For illustration, we consider a modified example from the chapter 10 of the book **Developing Time-Oriented Database Applications in SQL** [28].

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

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

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

Let us assume that, the information about the property is represented in XML using the schema given in Figure 54. For simplicity, only `property_number` and `address` attributes of the `Property` are considered. Property is associated with a owner by the `owner_name` attribute of the `<property>` element. To simplify the things a little, we assume that the owner is uniquely represented by the `owner_name` attribute.

Corresponding temporal and physical annotations are given in Figures 55 and 56. As can be seen from the temporal annotation, the `<property>` element is content varying both in transaction-time and valid-time.

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

Figure 54: `property.xsd`

To illustrate the process of gluing in two dimensions, we consider the history, over both valid time and transaction time, of a flat in Aalborg, at Skovvej 30 for the month of January 1998. All its transactions are

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

Figure 55: `property_temporal_annotation.xsd`

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

Figure 56: `property_physical_annotation.xsd`

listed below in the chronological order of transaction-time. The corresponding bitemporal-time diagrams and snippets of snapshot XML documents are also given for understanding.

Assume that, initially, the mortgage for the flat was being handled by some other company. So, although Nykredit maintained the property information, no information about the owner is stored in the database. We also assume that the flat exists in Nykredit's database from January 1. The snippet of the snapshot document corresponding to this period is shown in Figure 58.

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

- Valid Time [01-01, Forever)

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

Figure 57: Mortgage being handled by other company. No customer

On January 10, this flat was purchased by Eva Nielsen. We record this information at a current valid-time(`01-10`), current transaction-time (`01-10`). The snippets of the snapshot documents corresponding to this transaction period starting on `01-10` are shown in Figure 58.

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

The interplay between valid time and transaction time can be confusing, so it is useful to have a visualization of the information content of a bitemporal table. Figure 59 shows the *bitemporal time diagram*, or simply *time diagram*, corresponding to the above insertion.

In this figure, the horizontal axis tracks transaction time and the vertical axis tracks the valid time. Information about the owners associated with the property are depicted as two-dimensional polygonal regions in the diagram. Arrows extending rightward denote "until changed" in transaction time; arrows extending

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

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

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

- Valid Time [01-10, Forever)

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

Figure 58: Eva purchased the flat on January 10

upward denote "forever" in valid time. Here we have but one region, associated with Eva Nielsen, that starts at time 10 (January 10) in transaction time and extends to "until changed," and begins also at time 10 in valid time and extends to "forever." The arrow pointing upward extends to the largest valid time value ("forever"); the arrow pointing to the right extends to "now," that is, it advances day by day to the right (a transaction time in the future is meaningless).

On January 15 Peter Olsen buys this flat; this legal transaction transfers ownership from Eva to him. Figure 60 illustrates how this update impacts the time diagram. The valid-time extent of this modification is always "now" to "forever," so from time 15 on, the property is owned by Peter; at the rest of the time, from time 10 to 15, the property was owned by Eva. Both regions extend to the right to "until changed." This time diagram captures two facts: Eva owning the flat and Peter owning the flat, each associated with a bitemporal region.
The snippets of the snapshot documents corresponding to this transaction are shown in Figure 61.

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

Figure 62 shows the resulting time diagram. If we now request the valid-time history as best known, we will learn that Eva owned the flat from January 10 to January 15, and Peter owned the flat from January 15 to January 20. All prior states are retained. We can still time travel back to January 18 and request the valid-time history, which will state that on that day we thought that Peter still owned the flat.
The snippets of the snapshot documents corresponding to this transaction are shown in Figure 63.

On January 23, we find out that Eva had purchased the flat not on January 10, but on January 3, a week earlier. So we insert those additional days, to obtain the time diagram shown in Figure 64. Corresponding snippets of the snapshot documents are given in Figure 66

We learn on January 26 that Eva bought the flat not on January 10, as initially thought, nor on January 3, as later corrected, but on January 5. We specify a period of applicability of January 3 through 5, with the result shown in the time diagram in Figure 65. Corresponding snapshot snippets are given in Figure 67

Finally, we learn on January 28 that Peter bought the flat on January 12, not on January 15 as previously thought. This change requires a period of applicability of January 12 through 15, setting the `owner_name` to Peter, which results in the time diagram in Figure 68. Effectively, the ownership must be transferred from

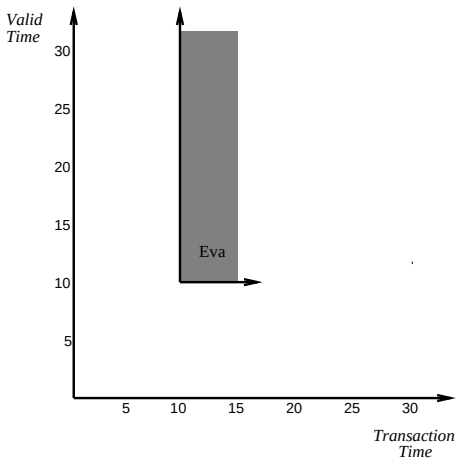


Figure 59: A bitemporal time diagram corresponding to Eva purchasing the flat, performed on January 10

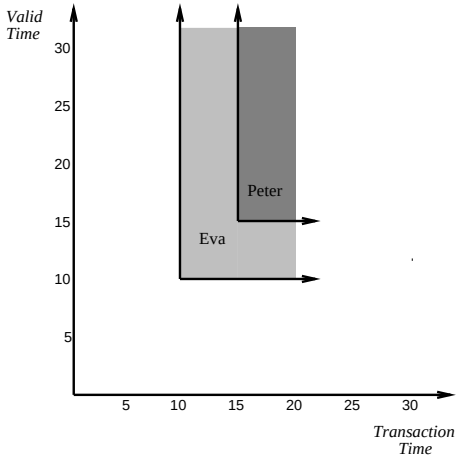


Figure 60: Peter buys the flat, performed on January 15

Eva to Peter for those three days, resulting in the snapshot documents given in Figure 69.

Gluing elements in two dimensions involves gluing them along one dimension (e.g., valid-time) followed by their gluing along the other dimension (e.g., transaction-time). The last timing diagram on January 28 in Figure 68 could be divided into 7 time-periods along the transaction time dimension as shown in Figure 70, i.e., `[01-01 - 01-10)`, `[01-10 - 01-15)`, `[01-15 - 01-20)`, `[01-20 - 01-23)`, `[01-23 - 01-26)`, `[01-26 - 01-28)`, `[01-28 - UC)`.

All above snapshot documents are first squashed along valid-time dimension as explained soon to give seven temporal documents corresponding to each of the above periods. The sample sample representation of these documents corresponding to periods `[01-10, 01-15)`, `[01-20, 01-23)`, `[01-26, 01-28)` are given below in Figure 71, Figure 72 and Figure 73 respectively. These documents are temporal documents themselves.

Other representations are also possible for these documents. As an example, The document in Figure 72 could also be represented as shown in Figure 74. In this representation, multiple DOM-equivalent versions of the `<property>` are merged into a single version and their time periods are represented as a single

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

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

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

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

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

- Valid Time [01-15, Forever)

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

Figure 61: Peter buys the flat, performed on January 15

temporal element, i.e., a set of periods.

These temporal documents then act as snapshot documents while performing squashing along transaction-time dimension. When squashed along transaction-time dimension, they give the final temporal document shown in Figures 76, 77 and 78.

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

While representing these regions in the XML document, they could be split with the vertical lines (termed as *transaction-time splitting* shown in Figure 75) or horizontal lines(termed as *valid-time splitting*). Due to the semantics of transaction time, regions are often split with vertical lines in the timing diagram.

The temporal document in Figures 76–78 uses the first approach, since it minimizes the representation of the document.

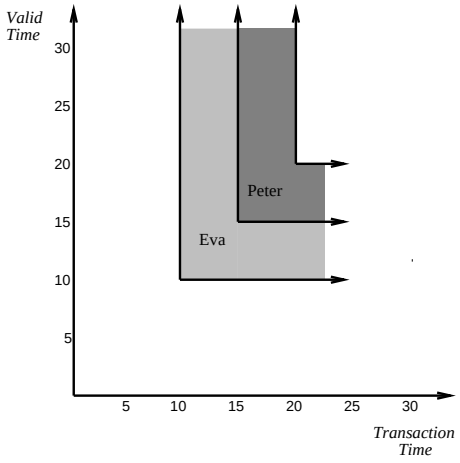


Figure 62: Peter sells the flat, performed on January 20

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

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

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

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

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

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

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

- Valid Time [01-20, Forever)

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

Figure 63: Peter sells the flat, performed on January 20

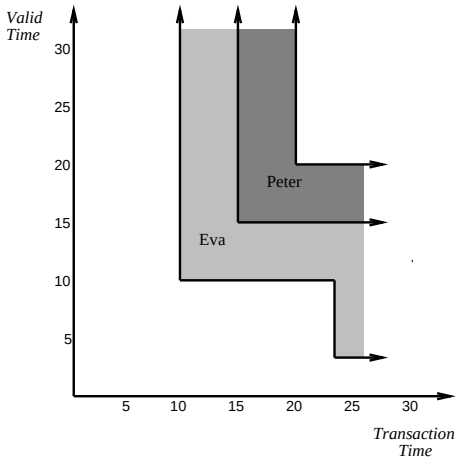


Figure 64: Discovered on January 23: Eva actually purchased the flat on January 3

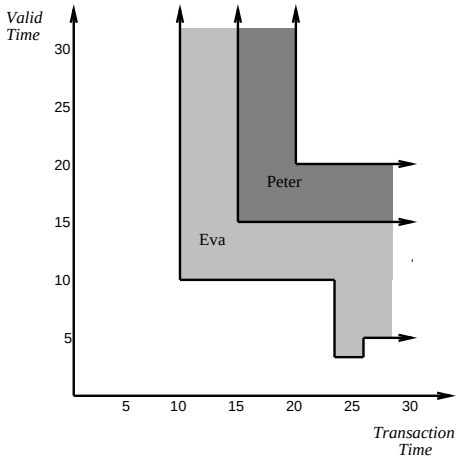


Figure 65: Discovered on January 26: Eva actually purchased the flat on January 5

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

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

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

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

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

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

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

- Valid Time [01-20, Forever)

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

Figure 66: Discovered on January 23: Eva actually purchased the flat on January 3

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

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

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

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

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

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

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

- Valid Time [01-20, Forever)

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

Figure 67: Discovered on January 26: Eva actually purchased the flat on January 5

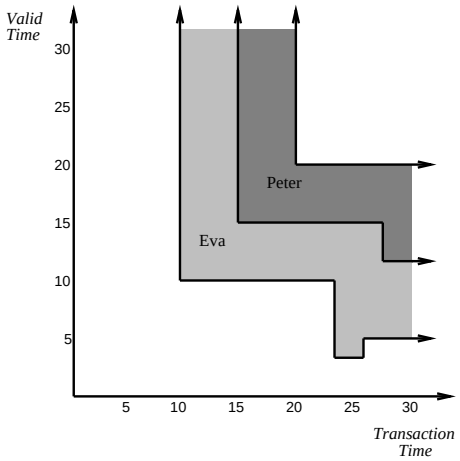


Figure 68: January 28: Peter actually purchased the flat on January 12

Transaction Time [01-28, UC)

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

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

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

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

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

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

- Valid Time [01-20, Forever)

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

Figure 69: January 28: Peter actually purchased the flat on January 12

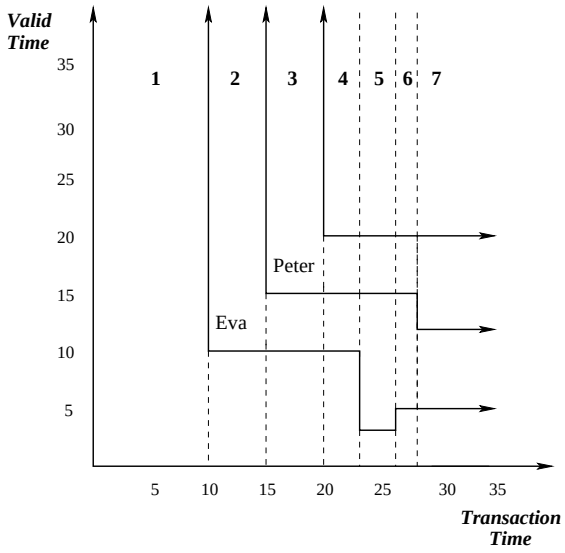


Figure 70: Transaction Time Regions

```
...
<property_RepItem>
  <property_Version>
    <timestamp_ValidExtent begin="1998-01-01" end="1998-01-10" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-10" end="9999-12-31" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>
</property_RepItem>
...
```

Figure 71: Transaction Time `[01-10, 01-15)`

```
...
<property_RepItem>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-01" end="1998-01-10" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-10" end="1998-01-15" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-15" end="1998-01-20" />
    <property property_number="7797" owner_name="Peter">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-20" end="9999-12-31" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

</property_RepItem>
...
```

Figure 72: Transaction Time `[01-20, 01-23)`

```
...
<property_RepItem>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-01" end="1998-01-05" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-05" end="1998-01-12" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-12" end="1998-01-20" />
    <property property_number="7797" owner_name="Peter">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-20" end="9999-12-31" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

</property_RepItem>
...
```

Figure 73: Transaction Time `[01-26, 01-28)`

```
...
<property_RepItem>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-01" end="1998-01-10" />
    <timestamp_ValidExtent begin="1998-01-20" end="9999-12-31" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-10" end="1998-01-15" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_ValidExtent begin="1998-01-15" end="1998-01-20" />
    <property property_number="7797" owner_name="Peter">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

</property_RepItem>
...
```

Figure 74: Transaction Time `[01-20, 01-23)`

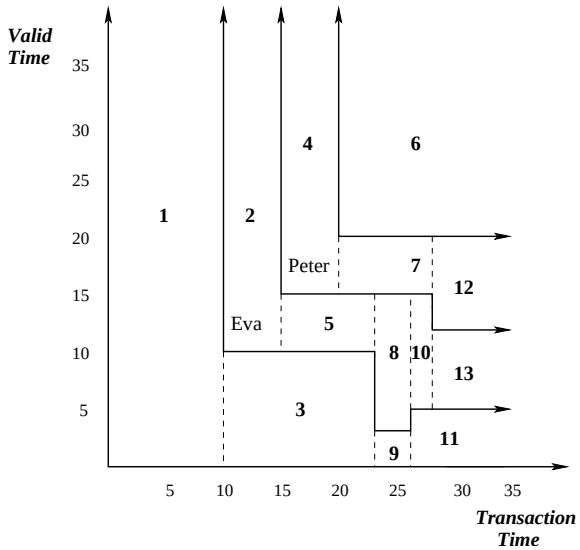


Figure 75: Transaction-time splitting of regions

```
...
<property_RepItem>

  <property_Version>
    <timestamp_TransExtent start="1998-01-01" stop="1998-01-10" />
    <timestamp_ValidExtent begin="1998-01-01" end="9999-12-31" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="1998-01-10" stop="1998-01-15" />
    <timestamp_ValidExtent begin="1998-01-10" end="9999-12-31" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="1998-01-10" stop="1998-01-23" />
    <timestamp_ValidExtent begin="1998-01-01" end="1998-01-10" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="1998-01-15" stop="1998-01-20" />
    <timestamp_ValidExtent begin="1998-01-15" end="9999-12-31" />
    <property property_number="7797" owner_name="Peter">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="1998-01-15" stop="1998-01-23" />
    <timestamp_ValidExtent begin="1998-01-10" end="1998-01-15" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>
```

Figure 76: Temporal Document along both valid-time and transaction-time

```
<property_Version>
  <timestamp_TransExtent start="1998-01-20" stop="9999-12-31" />
  <timestamp_ValidExtent begin="1998-01-20" end="9999-12-31" />
  <property property_number="7797">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>

<property_Version>
  <timestamp_TransExtent start="1998-01-20" stop="1998-01-28" />
  <timestamp_ValidExtent begin="1998-01-15" end="1998-01-20" />
  <property property_number="7797" owner_name="Peter">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>

<property_Version>
  <timestamp_TransExtent start="1998-01-23" stop="1998-01-26" />
  <timestamp_ValidExtent begin="1998-01-03" end="1998-01-15" />
  <property property_number="7797" owner_name="Eva">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>

<property_Version>
  <timestamp_TransExtent start="1998-01-23" stop="1998-01-26" />
  <timestamp_ValidExtent begin="1998-01-01" end="1998-01-03" />
  <property property_number="7797">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>

<property_Version>
  <timestamp_TransExtent start="1998-01-26" stop="1998-01-28" />
  <timestamp_ValidExtent begin="1998-01-05" end="1998-01-15" />
  <property property_number="7797" owner_name="Eva">
    <address> Skovvej 30, Alborg </address>
  </property>
</property_Version>
```

Figure 77: Temporal Document along both valid-time and transaction-time. **Continued**

```
  <property_Version>
    <timestamp_TransExtent start="1998-01-26" stop="9999-12-31" />
    <timestamp_ValidExtent begin="1998-01-01" end="1998-01-05" />
    <property property_number="7797">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="1998-01-28" stop="9999-12-31" />
    <timestamp_ValidExtent begin="1998-01-12" end="1998-01-20" />
    <property property_number="7797" owner_name="Peter">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

  <property_Version>
    <timestamp_TransExtent start="1998-01-28" stop="9999-12-31" />
    <timestamp_ValidExtent begin="1998-01-05" end="1998-01-12" />
    <property property_number="7797" owner_name="Eva">
      <address> Skovvej 30, Alborg </address>
    </property>
  </property_Version>

</property_RepItem>
...
```

Figure 78: Temporal Document along both valid-time and transaction-time. **Continued**

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

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

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

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

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

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

12 Evaluation and Conclusion

In this thesis we have considered how to accommodate and validate time-varying data within XML Schema. We have presented Temporal XML Schema (τXSchema), which is an extension of XML Schema, infrastructure, and a suite of tools to support the creation and validation of time-varying documents, without requiring any changes to XML Schema. τXSchema provides an efficient way to define temporal element types; specifically, an element type that 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. 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.

This three-level schema specification approach is exploited in supporting tools; several new, quite useful tools τVALIDATOR, SCHEMAMAPPER, SQUASH, UNSQUASH, and RESQUASH are introduced that require the logical and physical data independence provided by our approach. Additionally, this independence enables existing tools (e.g., the XML Schema validator, XQuery, and DOM) to be used in the implementation of their temporal counterparts.

We have then extended τXSchema to support schema versionoing. We showed how schema versioning can be integrated with support for time-varying documents in a fashion consistent and upwardly-compatible with XML, XML Schema, and conventional XML validators. Schema versioning in its full generality is supported, including (time-varying) schemas that include or reference other (time-varying) schemas. In doing so, we leveraged both conventional XML Schema and related tools (principally, the conventional validator), as well as τVALIDATOR for data versioning.

To summarize, in this work we introduced tools τVALIDATOR, SCHEMAMAPPER, SQUASH, UNSQUASH, and RESQUASH and extended them to support schema versioning. The tools comprise the code of somewhat more than 8000 lines including comments. Five new schemas `TBSchema`, `TXSchema`, `PXSchema`, `TVSchema`, and `ConfigSchema` are introduced and comprise around 400 lines of XML code. The framework contains total 44 interfaces and classes.

τXSchema and τVALIDATOR can be further enhanced to provide a better system and more features.

- Future work includes extending the τXSchema model to fulfill the issues not addressed during the initial implementation. Indeterminacy and granularity are two significant and related issues, and should be fully supported by τXSchema. We anticipate that providing this support would require additions to the TVSchema / TXSchema / PXSchema, but no changes to the user-designed schemas would be needed. These augmentations would maintain upward compatibility with previous versions of τXSchema and be transparent to the user.

- Another broad area of work is optimization and efficiency. Although we do talk about the space-efficiency of the tools described in Section 7, we haven't given much attention to their performance. New representations can be proposed, incorporated and evaluated to improve the space-efficiency of the temporal document. We anticipate that the DOM API could prove to be a memory bottleneck for huge documents. So instead of parsing the complete document at once, other options need to be evaluated.

  One option is to validate the document in parts, bringing only one item at a time in the memory. This could be achieved by replacing the immediate descendant item elements by their dummy equivalents

and then validating the item for its sequenced and non-sequenced constraints. This would result in less memory utilization since only a part of the document is being kept in the memory. As few changes would be required to manage the items one at a time, a major part of the existing algorithm for τVALIDATOR could be reused. Here, if a DOM-based parser is used, the whole document needs to be parsed at least once, even if we are validating one item at a time. This could be avoided by using an event-based SAX parser and building an in-memory tree of only the required elements in order to perform those aspects of the validation that are synchronized with the parsing. This approach would require complex memory management and parsing of the document multiple times, but memory use would be greatly reduced.

As described earlier, all the tools are based on the elementary functions pushUp, pushDown and coalesce. If we can modify them to use a SAX parser instead of a document-object-model, we can easily convert all the tools to use a SAX parser. We think that, converting pushDown to use a SAX parser woule be easier; the timestamps could be pushed down easily as the document is being parsed from start to end. After initial thought it appears that, pushUp would need building of an in-memory tree, pushing the timestamps up and then serializing the tree. This could also be achieved by building the tree in parts resulting in more complexity. coalesce would also need to build a tree in memory. But instead of building a complete tree at once, it can build a subtree for each item at a time and then coalesce it.

- Although, the existing representation is easy to implement and space efficient in the average case, it may become very space inefficient in certain cases. Certain new representations such as 'diff' or 'zip' could be added to tools to increase the space efficiency of the temporal documents. The support for these new representations could be built on top of the existing tools by first creating the temporal documents in the decomposed or non-decomposed representations and then converting them to 'diff' or 'zip' format for efficient storage on the disk.

- Future work also includes enabling the legacy applications or the data inconsistent with a subsequently changed schema, by exploiting information about the evolving schema that is already captured in the temporal schema.

- Current implementation of tools does not support all described features of τXSchema completely. These features need to be implemented to provide completeness to the tools. The unimplemented features, the anticipated changes and the estimated efforts required to implement them are listed below. The estimated effort does not include becoming familiarized with the architecture and the source code.

  - Support for the '*Step*' representation of timestamp: Some changes to the classes Item and RepItem would be needed to support the '*Step*' representation. Some changes would also be needed to the algorithms implemented in class `Primitives`. 15–20 hours of work is anticipated.
  - Support for the generic validation of non-sequenced constraints: Currently, the validation for each non-sequenced constraints is implemented using a separated function inside `Item` class. To provide a framework for the generic support of non-sequenced constraints, a 'Visitor' pattern could be used. In that case, the validator for each non-sequenced constraint will be implemented in a separate class and a reference to an Item element will be passed to it. The addition of a new constraint could be made easier by some properties file; this will eliminate any changes to the `Item` class for addition/modification of constraints. 15–20 hours of work is anticipated.
  - Support for the schemaPath expressions containing 'wildcards' characters and shortcut representation: This will change the way targets are being evaluated. Changes to the classes

`SchemaPathEvaluator`, `Item` and `ItemIdentifier` would be needed. Around 30–40 hours of work is anticipated.

  - Support for the item-identifiers specified in terms of existing items or schema keys, and targets containing 'wildcard' characters: Some changes to the classes `Item` and `ItemIdentifier` would be needed. Some changes to the functions from class `Primitive` may also be needed since the procedure for coalescing may change. 20–30 hours of work is anticipated for this change.
  - Support for nested time-varying schemas: We anticipate, this would result in a considerable change to all the tools. A couple of weeks of work may be needed to support this feature.
  - Support for RESQUASHing of a temporal document using a new temporal annotation: The changes needed for this functionality are mentioned in the Section 7.6. 4–5 hours of work should be sufficient for this change.

- In this work, only conceptual support for the bitemporal elements is defined. The tools need to be extended to support bitemporal elements.

In the future, τXSchema should be integrated with a schema-aware XML-based editor like XMLSpy [38]. Schema-aware editors generate easy-to-use templates for updating each type of element defined in a schema. But they do not track changes to either the schema or the data. Enabling versioning for both will support unlimited undo/redo, improve change tracking, and aid in cooperative editing. Another direction of future work is to add versioning to XUpdate [39]. XUpdate is a language for specifying changes to the XML document.

τXSchema can also be extended to support generic aspects [12]. In that approach, we generalized τXSchema to represent any generic aspect instead of just timestamps.

A three-level schema specification approach introduced in this work by τXSchema, the infrastructure, and a suite of tools provide a system for creation and validation of data-versioned XML documents, without requiring any changes to the XML Schema specification. By clever use of schema-constant periods and cross-wall validation, schema versioning is also integrated in the framework with the support for time-varying documents in a fashion consistent and upwardly-compatible with XML, XML Schema, and conventional XML validators. The design conforms to W3C XML Schema definition and is built on top of XML Schema. Thus, this research has shown that by utilizing schema-constant periods and cross-wall validation, it is possible to realize a comprehensive system for representing and validating data- and schema-versioned XML documents, while remaining fully compatible with the XML standards.

References

[1] T. Amagasa, M. Yoshikawa, and S. Uemura, "A Data Model for Temporal XML Documents," in *Proceedings of Database and Expert Systems Applications, 11th International Conference, DEXA 2000*, pages 334–344, London, UK, September 2000.

[2] Apache XML Project, Official website, URL http://xml.apache.org, Viewed April 12, 2007.

[3] M. H. Bohlen, C. S. Jensen and R. T. Snodgrass, "Temporal Statement Modifiers," in *ACM Transactions on Database Systems* 25(4): 407-456, December 2000.

[4] H. Bratman, "An Alternate Form of the "UNCOL Diagram," in *Communications of the ACM (CACM)* 4(3):142, 1961.

[5] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan, "Keys for XML," in *Computer Networks* 39(5): 473-487, 2002.

[6] S. Chien, V. Tsotras, and C. Zaniolo, "Efficient schemes for managing multiversionXML documents," *Very Large Data Bases Journal*, 11(4): 332–353.

[7] L. Costello and M. Utzinger, "Impact of XML Schema Versioning on System Design" URL http://www.xfront.com/SchemaVersioning.html, Viewed February 7th, 2007.

[8] C. De Castro, F. Grandi, and M. R. Scalas, "Schema Versioning for Multitemporal Relational Databases," in *Information Systems* 22(5): 249-290, 1997.

[9] Document Object Model, W3C. URL http://www.w3.org/DOM, Viewed March 26, 2007.

[10] Document Type Definition (DTD) language. URL http://www.w3.org/TR/REC-xml/dt-doctype, Viewed March 25, 2007.

[11] C. Dyreson, H. L. Lin, and Y. Wang, "Managing Versions of Web Documents in a Transaction-time Web Server," in *Proceedings of World Wide Web*, New York, NY, pp. 422–432, 2004.

[12] C. Dyreson, R. T. Snodgrass, F. Currim, S. Currim, and S. P. Joshi, "Weaving Temporal and Reliability Aspects into a Schema Tapestry," in *Data & Knowledge Engineering.*

[13] C. Dyreson, R. T. Snodgrass, F. Currim, S. Currim, and S. P. Joshi, "Validating Quicksand: Schema Versioning in τXSchema," in *22nd IEEE International Conference on Data Engineering Workshops*, 2006.

[14] J. Gabriel, "How to Version Schemas," in *Proceedings of XML 2004-Conference and Exhibition*, Washington DC, November, 2004. URL http://www.idealliance.org/proceedings/xml04/papers/74/howToVersionSchemas.html, Viewed February 7th, 2007.

[15] D. Gao and R. T. Snodgrass, "Temporal Slicing in the Evaluation of XML Queries," in *Proceedings of Very Large Data Bases (VLDB)*, pp. 632–643, 2003.

[16] F. Grandi, "A Bibliography on Temporal and Evolution Aspects in the World Wide Web," *TimeCenter* TR-75, 2003.

[17] C. S. Jensen and C. E. Dyreson (Editors), "The Consensus Glossary of Temporal Database Concepts," February 1998 Version.

[18] C. S. Jensen and R. T. Snodgrass, “Temporal Database Management,” *TimeCenter* TR-17, 1997.

[19] B. P. Lientz, “Issues in software maintenance,” in *ACM Comput. Surv* 15(3):271–278, 1983.

[20] A. Marian, “Detecting Changes in XML Documents,” in *Proceedings of the 18th International Conference on Data Engineering*, pp. 41–53, 2002

[21] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet, “Change-Centric Management of Versions in an XML Warehouse,” in *Proceedings of Very Large Data Bases (VLDB)*, Rome, Italy, pp. 581–590, 2001.

[22] W. M. McKeeman, J. J. Horning, and D. B. Wortman, **A Compiler Generator**, Prentice-Hall, Englewood Cliffs, NJ, 1970.

[23] J. F. Roddick, “Schema Evolution in Database Systems—An Annotated Bibliography,” *SIGMOD Record* 21(4): 35–40, December, 1992.

[24] J. F. Roddick, “A Survey of Schema Versioning Issues for Database Systems,” in *Information and Software Technology* 37(7):383-393, 1995.

[25] SAX project, Official website. URL http://www.saxproject.org, Viewed March 26, 2007.

[26] D. Sjoberg, “Measuring schema evolution”, Technical Report FIDE/92/36, Department of Computer Science, University of Glasgow, 1992.

[27] D. Sjoberg, Quantifying schema evolution, in *Inf. Softw. Technol.* 35(1):35-44, 1993.

[28] R. T. Snodgrass, **Developing Time-Oriented Database Applications in SQL**, Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[29] R. T. Snodgrass, “The Temporal Query Language TQuel,” in *ACM Transactions on Database Systems* 12(2):247–298, June 1987.

[30] R. T. Snodgrass, C. Dyreson, F. Currim, S. Currim, and S. P. Joshi, “τXSchema: Support for Data and Schema Versioned XML Documents,” *TimeCenter* TR, 2007.

[31] R. T. Snodgrass, S. Gomez, and E. McKenzie, “Aggregates in the Temporal Query Language TQuel,” in *IEEE Transactions on Knowledge and Data Engineering* 5(5):826–842, October, 1993.

[32] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, **Temporal Databases: Theory, Design, and Implementation**, Benjamin/Cummings Publiihing Company, 1993.

[33] TAU Project, τXSchema, Computer Science Department at the University of Arizona. URL http://www.cs.arizona.edu/projects/tau/txschema/index.htm, Viewed March 26, 2007.

[34] XALAN, Official website of Xalan-Java Version 2.7.0, URL http://xalan.apache.org, Viewed April 12, 2007.

[35] XERCES, Official website of Apache Xerces Project Version 1.4.4, URL http://xerces.apache.org/xerces-j, Viewed April 12, 2007.

[36] XML Schema Versioning Use Cases “Framework for discussion of versioning” URL http://www.w3.org/XML/2005/xsd-versioning-use-cases, Viewed January 15th, 2006.

[37] XML Schema, W3C Recommendation, May 2001. URL http://www.w3.org/XML/Schema, Viewed March 25, 2007.

[38] XMLSpy, “XML editor for modeling, editing, transforming, & debugging XML technologies.” URL http://www.altova.com/products/xmlspy/xml_editor.html, Viewed April 18, 2007.

[39] XUpdate, XML Update Language. URL http://xmldb-org.sourceforge.net/xupdate, Viewed April 18, 2007.

A Base Schemas

A.1 TBSchema: Schema for Temporal Bundle

```
<?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" minOccurs="0"/>
                    <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>
```

A.2 TXSchema: Schema for Temporal Annotation

```
<?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="optional"/>
                    <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:attribute name="kind" type="ts:kindType" use="optional"/>
              <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="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>
```

A.3 PXSchema: Schema for Physical Annotation

```
<?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">
  <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: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>
```

A.4 TVSchema: Schema for Timestamp Representations

```
<xsd:schema targetNamespace="http://www.cs.arizona.edu/tau/TVSchema"
            xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified">
  <xsd:element name="timestamp_TransStep">
    <xsd:complexType>
      <xsd:attribute name="begin" type="xsd:date" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="timestamp_TransExtent">
    <xsd:complexType>
      <xsd:attribute name="begin" type="xsd:date" />
      <xsd:attribute name="end" type="xsd:date" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="timestamp_ValidStep">
    <xsd:complexType>
      <xsd:attribute name="begin" type="xsd:date" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="timestamp_ValidExtent">
    <xsd:complexType>
      <xsd:attribute name="begin" type="xsd:date" />
      <xsd:attribute name="end" type="xsd:date" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

A.5 ConfigSchema: Schema for Configuration Document

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/ConfigSchema"
           xmlns:cs="http://www.cs.arizona.edu/tau/tauXSchema/ConfigSchema"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified" attributeFormDefault="unqualified" >
  <xs:element name="config">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="snapshot" minOccurs="1" maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="beginDate" type="xs:string"/>
            <xs:attribute name="endDate" type="xs:string"/>
            <xs:attribute name="file" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="bundle" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

B Schema-Versioning Example

B.1 Snapshot Schemas

B.1.1 Snapshot Schema on `2002-01-01`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xsd:element name="winOlympic">
    <xsd:annotation>
      <xsd:documentation>
        Schema for recording non temporal country information
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element ref="country" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="country">
    <xsd:complexType mixed="false">
      <xsd:sequence>
        <xsd:element ref="athleteTeam"/>
      </xsd:sequence>
      <xsd:attribute name="countryName" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athleteTeam">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element name="teamName" minOccurs="1" maxOccurs="1" type="xsd:string"/>
        <xsd:element ref="athlete" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="numAthletes" type="xsd:positiveInteger" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athlete">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element name="athName" type="xsd:string"/>
        <xsd:element name="phone" type="phoneNumType" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:simpleType name="phoneNumType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="12"/>
      <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

B.1.2 Snapshot Schema on `2005-01-01`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xsd:element name="winOlympic">
    <xsd:annotation>
      <xsd:documentation>
        Schema for recording non temporal country information
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType mixed="true">
```

```
      <xsd:sequence>
      <!--numEvents added on Wednesday-->
        <xsd:element name="numEvents" type="xsd:nonNegativeInteger"/>
        <xsd:element ref="country" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="country">
    <xsd:complexType mixed="false">
      <xsd:sequence>
        <xsd:element ref="athleteTeam"/>
      </xsd:sequence>
      <xsd:attribute name="countryName" type="xsd:string" use="required"/>
      <xsd:attribute name="countryLead" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athleteTeam">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element name="teamName" minOccurs="1" maxOccurs="1" type="xsd:string"/>
        <xsd:element ref="athlete" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="numAthletes" type="xsd:positiveInteger" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athlete">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element name="athName" type="xsd:string"/>
        <xsd:element name="phone" type="phoneNumType" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:simpleType name="phoneNumType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="12"/>
      <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

B.2 Temporal Annotations

B.2.1 Temporal Annotation on `2002-01-01`

```
<?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 content="varying" existence="constant"/>
    <itemIdentifier name="olympicId1" timeDimension="transactionTime">
      <field path="./text"/>
    </itemIdentifier>
  </item>

  <item target="/winOlympic/country">
  <transactionTime content="varying" existence="varyingWithGaps"/>
    <itemIdentifier name="countryId1" timeDimension="transactionTime">
      <field path="./@countryName"/>
    </itemIdentifier>
  </item>
```

```
  <item target="/winOlympic/country/athleteTeam">
  <transactionTime content="varying" existence="varyingWithGaps"/>
    <itemIdentifier name="teamName" timeDimension="transactionTime">
      <field path="./teamName/text"/>
    </itemIdentifier>
  </item>

  <item target="/winOlympic/country/athleteTeam/athlete">
  <transactionTime content="varying" existence="varyingWithGaps"/>
    <itemIdentifier name="atheleteId1" timeDimension="transactionTime">
      <field path="./athName/text"/>
    </itemIdentifier>
  </item>
</temporalAnnotations>
```

B.2.2 Temporal Annotation on `2005-01-01`

```
<?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 content="varying" existence="constant"/>
    <itemIdentifier name="olympicId1" timeDimension="transactionTime">
      <field path="./text"/>
    </itemIdentifier>
  </item>

  <item target="/winOlympic/country">
  <transactionTime content="varying" existence="varyingWithGaps"/>
    <itemIdentifier name="countryId1" timeDimension="transactionTime">
      <field path="./@countryName"/>
      <field path="./@countryLead"/>
    </itemIdentifier>
  </item>

  <item target="/winOlympic/country/athleteTeam">
  <transactionTime content="varying" existence="varyingWithGaps"/>
    <itemIdentifier name="teamName" timeDimension="transactionTime">
      <field path="./teamName/text"/>
    </itemIdentifier>
  </item>

  <item target="/winOlympic/country/athleteTeam/athlete">
  <transactionTime content="varying" existence="varyingWithGaps"/>
    <itemIdentifier name="atheleteId1" timeDimension="transactionTime">
      <field path="./athName/text"/>
    </itemIdentifier>
  </item>
</temporalAnnotations>
```

B.3 Physical Annotations

B.3.1 Physical Annotation on `2002-01-01`

```
<?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">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>

  <stamp target="/winOlympic/country">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>

  <stamp target="/winOlympic/country/athleteTeam">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>

  <stamp target="/winOlympic/country/athleteTeam/athlete">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>
</physicalAnnotations>
```

B.3.2 Physical Annotation on `2005-01-01`

```
<?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">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>

  <stamp target="/winOlympic/country">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>

  <stamp target="/winOlympic/country/athleteTeam">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>

  <stamp target="/winOlympic/country/athleteTeam/athlete">
    <stampKind timeDimension="transactionTime" stampBounds="extent"/>
  </stamp>
</physicalAnnotations>
```

B.4 Snapshot Documents

B.4.1 Snapshot Document on `2002-01-01`

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="winOlympic.ver1.xsd">
  There are
  events in the Olympics.
  <country countryName="Norway">
    <athleteTeam numAthletes="95">
      <teamName>Norway_Army</teamName>
      Athletes will take part in various events. The athletes participating are listed below
      <athlete>
        <athName>
            Kjetil Andre Aamodt
        </athName>
      </athlete>
      <athlete>
        <athName>
          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>
```

B.4.2 Snapshot Document on `2003-01-01`

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="winOlympic.ver1.xsd">
  There are
  events in the Olympics.
  <country countryName="Norway">
    <athleteTeam numAthletes="95">
      <teamName>Norway_Army</teamName>
      Athletes will take part in various events. The athletes participating are listed below
      <athlete>
        <athName>
            Kjetil Andre Aamodt
        </athName>
      </athlete>
      <athlete>
        <athName>
            Andre Agassi
        </athName>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His phone numbers are:
        <phone>123-402-0340</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

B.4.3 Snapshot Document on `2005-01-01`

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="winOlympic.ver2.xsd">
  There are
  <numEvents>11</numEvents>
  events in the Olympics.
  <country countryName="Norway" countryLead="Andre Agassi">
    <athleteTeam numAthletes="95">
      <teamName>Norway_Army</teamName>
      Athletes will take part in various events. The athletes participating are listed below
      <athlete>
        <athName>
```

```
            Kjetil Andre Aamodt
        </athName>
      </athlete>
      <athlete>
        <athName>
            Andre Agassi
        </athName>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His phone numbers are:
        <phone>123-402-0340</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

B.4.4 Snapshot Document on `2006-01-01`

```
<?xml version="1.0" encoding="UTF-8"?>
<winOlympic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="winOlympic.ver2.xsd">
  There are
  <numEvents>11</numEvents>
  events in the Olympics.
  <country countryName="Norway" countryLead="Andre Agassi">
    <athleteTeam numAthletes="95">
      <teamName>Norway_Army</teamName>
      Athletes will take part in various events. The athletes participating are listed below
      <athlete>
        <athName>
            Kjetil Andre Aamodt
        </athName>
      </athlete>
      <athlete>
        <athName>
            Andre Agassi
        </athName>
      </athlete>
      <athlete>
        <athName>
          Trine Bakke-Rognmo
        </athName>
        His phone numbers are:
        <phone>123-402-0340</phone>
        <phone>123-402-0000</phone>
      </athlete>
      <athlete>
        <athName>
          Lasse Kjus
        </athName>
      </athlete>
    </athleteTeam>
  </country>
</winOlympic>
```

B.5 Temporal Bundle

```
<?xml version="1.0" encoding="UTF-8"?>
<temporalBundle xmlns="http://www.cs.arizona.edu/tau/tauXSchema/TBSchema"
          xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema"
          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.ver1.xsd"
                temporalAnnotation="winolympic_temp_anno.ver1.xml"
                physicalAnnotation="winolympic_phy_anno.ver1.xml">
      <tTime>2002-01-01</tTime>
    </schemaAnnotation>
    <schemaAnnotation snapshotSchema="winOlympic.ver2.xsd"
                temporalAnnotation="winolympic_temp_anno.ver2.xml"
                physicalAnnotation="winolympic_phy_anno.ver2.xml">
      <tTime>2005-01-01</tTime>
    </schemaAnnotation>
  </bundleSequence>
</temporalBundle>
```

B.6 Representational Schema

B.6.1 Representational Schema for [2002-01-01, 2005-01-01)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema attributeFormDefault="unqualified"
  elementFormDefault="unqualified"
  targetNamespace="http://www.cs.arizona.edu/tau/RepSchema0"
  xmlns="http://www.cs.arizona.edu/tau/RepSchema0"
  xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsd:import namespace="http://www.cs.arizona.edu/tau/TVSchema" schemaLocation="TVSchema.xsd" />
  <xsd:simpleType name="phoneNumType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="12" />
      <xsd:pattern value="\d{3}-\d{3}-\d{4}" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="tv_root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="winOlympic_RepItem" />
      </xsd:sequence>
      <xsd:attribute name="begin" type="xsd:date" />
      <xsd:attribute name="end" type="xsd:date" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athleteTeam_RepItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="athleteTeam_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="athleteTeam">
                <xsd:complexType mixed="true">
                  <xsd:sequence>
                    <xsd:element maxOccurs="1"
                      minOccurs="1" name="teamName" type="xsd:string" />
                    <xsd:element
                      maxOccurs="unbounded" ref="athlete_RepItem" />
                  </xsd:sequence>
                  <xsd:attribute name="numAthletes"
```

```
                type="xsd:positiveInteger" use="optional" />
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="isItem" type="xsd:string" />
  <xsd:attribute name="originalElement" type="xsd:string" />
 </xsd:complexType>
</xsd:element>
<xsd:element name="country_RepItem">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element maxOccurs="unbounded" minOccurs="1"
    name="country_Version">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element ref="tv:timestamp_TransExtent" />
      <xsd:element name="country">
       <xsd:complexType mixed="false">
        <xsd:sequence>
         <xsd:element
          ref="athleteTeam_RepItem" />
        </xsd:sequence>
        <xsd:attribute name="countryName"
         type="xsd:string" use="required" />
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="isItem" type="xsd:string" />
  <xsd:attribute name="originalElement" type="xsd:string" />
 </xsd:complexType>
</xsd:element>
<xsd:element name="winOlympic_RepItem">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element maxOccurs="unbounded" minOccurs="1"
    name="winOlympic_Version">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element ref="tv:timestamp_TransExtent" />
      <xsd:element name="winOlympic">
       <xsd:annotation>
        <xsd:documentation>
         Schema for recording non
         temporal country information
        </xsd:documentation>
       </xsd:annotation>
       <xsd:complexType mixed="true">
        <xsd:sequence>
         <xsd:element
          maxOccurs="unbounded" minOccurs="0" ref="country_RepItem" />
        </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="isItem" type="xsd:string" />
  <xsd:attribute name="originalElement" type="xsd:string" />
 </xsd:complexType>
</xsd:element>
<xsd:element name="athlete_RepItem">
```

```
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="athlete_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="athlete">
                <xsd:complexType mixed="true">
                  <xsd:sequence>
                    <xsd:element name="athName"
                      type="xsd:string" />
                    <xsd:element
                      maxOccurs="unbounded" minOccurs="0" name="phone"
                      type="phoneNumType" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="isItem" type="xsd:string" />
      <xsd:attribute name="originalElement" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

B.6.2 Representational Schema for [2002-01-01, 2005-01-01)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema attributeFormDefault="unqualified"
  elementFormDefault="unqualified"
  targetNamespace="http://www.cs.arizona.edu/tau/RepSchema1"
  xmlns="http://www.cs.arizona.edu/tau/RepSchema1"
  xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsd:import namespace="http://www.cs.arizona.edu/tau/TVSchema" schemaLocation="TVSchema.xsd" />
  <xsd:simpleType name="phoneNumType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="12" />
      <xsd:pattern value="\d{3}-\d{3}-\d{4}" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="tv_root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="winOlympic_RepItem" />
      </xsd:sequence>
      <xsd:attribute name="begin" type="xsd:date" />
      <xsd:attribute name="end" type="xsd:date" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athleteTeam_RepItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="athleteTeam_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="athleteTeam">
                <xsd:complexType mixed="true">
                  <xsd:sequence>
                    <xsd:element maxOccurs="1"
                      minOccurs="1" name="teamName" type="xsd:string" />
```

```
                <xsd:element
                  maxOccurs="unbounded" ref="athlete_RepItem" />
              </xsd:sequence>
              <xsd:attribute name="numAthletes"
                type="xsd:positiveInteger" use="optional" />
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="isItem" type="xsd:string" />
  <xsd:attribute name="originalElement" type="xsd:string" />
 </xsd:complexType>
</xsd:element>
<xsd:element name="country_RepItem">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element maxOccurs="unbounded" minOccurs="1"
    name="country_Version">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element ref="tv:timestamp_TransExtent" />
      <xsd:element name="country">
       <xsd:complexType mixed="false">
        <xsd:sequence>
         <xsd:element
          ref="athleteTeam_RepItem" />
        </xsd:sequence>
        <xsd:attribute name="countryName"
         type="xsd:string" use="required" />
        <xsd:attribute name="countryLead"
         type="xsd:string" use="required" />
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="isItem" type="xsd:string" />
  <xsd:attribute name="originalElement" type="xsd:string" />
 </xsd:complexType>
</xsd:element>
<xsd:element name="winOlympic_RepItem">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element maxOccurs="unbounded" minOccurs="1"
    name="winOlympic_Version">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element ref="tv:timestamp_TransExtent" />
      <xsd:element name="winOlympic">
       <xsd:annotation>
        <xsd:documentation>
         Schema for recording non
         temporal country information
        </xsd:documentation>
       </xsd:annotation>
       <xsd:complexType mixed="true">
        <xsd:sequence>
         <!--numEvents added on Wednesday-->
         <xsd:element name="numEvents"
          type="xsd:nonNegativeInteger" />
         <xsd:element
          maxOccurs="unbounded" minOccurs="0" ref="country_RepItem" />
        </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
```

```
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="isItem" type="xsd:string" />
      <xsd:attribute name="originalElement" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="athlete_RepItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1"
          name="athlete_Version">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="tv:timestamp_TransExtent" />
              <xsd:element name="athlete">
                <xsd:complexType mixed="true">
                  <xsd:sequence>
                    <xsd:element name="athName"
                      type="xsd:string" />
                    <xsd:element
                      maxOccurs="unbounded" minOccurs="0" name="phone"
                      type="phoneNumType" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="isItem" type="xsd:string" />
      <xsd:attribute name="originalElement" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

B.6.3 Final Representational Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="unqualified"
  targetNamespace="http://www.cs.arizona.edu/tau/RepSchema"
  xmlns="http://www.cs.arizona.edu/tau/RepSchema"
  xmlns:rep0="http://www.cs.arizona.edu/tau/RepSchema0"
  xmlns:rep1="http://www.cs.arizona.edu/tau/RepSchema1"
  xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsd:import namespace="http://www.cs.arizona.edu/tau/TVSchema" schemaLocation="TVSchema.xsd" />
  <xsd:import namespace="http://www.cs.arizona.edu/tau/RepSchema0" schemaLocation="rep0.xsd" />
  <xsd:import namespace="http://www.cs.arizona.edu/tau/RepSchema1" schemaLocation="rep1.xsd" />
  <xsd:element name="sv_root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="schemaItem">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element maxOccurs="1" minOccurs="1"
                name="schemaVersion0">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element maxOccurs="1"
                      minOccurs="1" ref="tv:timestamp_TransExtent" />
                    <xsd:element maxOccurs="1"
                      minOccurs="1" ref="rep0:tv_root" />
                  </xsd:sequence>
                </xsd:complexType>
```

```
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="1"
              name="schemaVersion1">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element maxOccurs="1"
                    minOccurs="1" ref="tv:timestamp_TransExtent" />
                  <xsd:element maxOccurs="1"
                    minOccurs="1" ref="rep1:tv_root" />
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="bundle" type="xsd:string" />
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

B.7 Temporal Document

```
<?xml version="1.0" encoding="UTF-8"?>
<rep:sv_root xmlns:rep="http://www.cs.arizona.edu/tau/RepSchema"
  bundle="winolympic_bundle.xml"
  xmlns:rep0="http://www.cs.arizona.edu/tau/RepSchema0"
  xmlns:rep1="http://www.cs.arizona.edu/tau/RepSchema1"
  xmlns:tv="http://www.cs.arizona.edu/tau/TVSchema">
  <schemaItem>
    <schemaVersion0>
      <tv:timestamp_TransExtent begin="2002-01-01"
        end="2005-01-01" />
      <rep0:tv_root
        xmlns:rep0="http://www.cs.arizona.edu/tau/RepSchema0"
        begin="2002-01-01" end="2005-01-01">
        <rep0:winOlympic_RepItem isItem="y"
          originalElement="winOlympic">
          <winOlympic_Version>
            <tv:timestamp_TransExtent begin="2002-01-01"
              end="2005-01-01" />
            <winOlympic>
              There are events in the Olympics.
              <rep0:country_RepItem isItem="y"
                originalElement="country">
                <country_Version>
                  <tv:timestamp_TransExtent
                    begin="2002-01-01" end="2005-01-01" />
                  <country countryName="Norway">
                    <rep0:athleteTeam_RepItem
                      isItem="y" originalElement="athleteTeam">
                      <athleteTeam_Version>
                        <tv:timestamp_TransExtent
                          begin="2002-01-01" end="2003-01-01" />
                        <athleteTeam
                          numAthletes="95">
                          <teamName>
                            Norway_Army
                          </teamName>
                          Athletes will take part in various events. The athletes participating are listed below
                          <rep0:athlete_RepItem
                            isItem="y" originalElement="athlete">
                            <athlete_Version>
                              <tv:timestamp_TransExtent
                                begin="2002-01-01" end="2003-01-01" />
                              <athlete>
                                <athName>
```

```
            Kjetil Andre Aamodt
          </athName>
        </athlete>
      </athlete_Version>
    </rep0:athlete_RepItem>
    <rep0:athlete_RepItem
      isItem="y" originalElement="athlete">
      <athlete_Version>
        <tv:timestamp_TransExtent
          begin="2002-01-01" end="2003-01-01" />
        <athlete>
          <athName>
            Trine
            Bakke-Rognmo
          </athName>
          His phone numbers are:
          <phone>
            123-402-0340
          </phone>
          <phone>
            123-402-0000
          </phone>
        </athlete>
      </athlete_Version>
    </rep0:athlete_RepItem>
    <rep0:athlete_RepItem
      isItem="y" originalElement="athlete">
      <athlete_Version>
        <tv:timestamp_TransExtent
          begin="2002-01-01" end="2003-01-01" />
        <athlete>
          <athName>
            Lasse Kjus
          </athName>
        </athlete>
      </athlete_Version>
    </rep0:athlete_RepItem>
  </athleteTeam>
</athleteTeam_Version>
<athleteTeam_Version>
  <tv:timestamp_TransExtent
    begin="2003-01-01" end="2005-01-01" />
  <athleteTeam
    numAthletes="95">
    <teamName>
      Norway_Army
    </teamName>
    Athletes will take part in various events. The athletes participating are listed below
    <rep0:athlete_RepItem
      isItem="y" originalElement="athlete">
      <athlete_Version>
        <tv:timestamp_TransExtent
          begin="2003-01-01" end="2005-01-01" />
        <athlete>
          <athName>
            Kjetil Andre Aamodt
          </athName>
        </athlete>
      </athlete_Version>
    </rep0:athlete_RepItem>
    <rep0:athlete_RepItem
      isItem="y" originalElement="athlete">
      <athlete_Version>
        <tv:timestamp_TransExtent
          begin="2003-01-01" end="2005-01-01" />
        <athlete>
          <athName>
            Andre Agassi
```

```
                        </athName>
                      </athlete>
                    </athlete_Version>
                  </rep0:athlete_RepItem>
                  <rep0:athlete_RepItem
                    isItem="y" originalElement="athlete">
                    <athlete_Version>
                      <tv:timestamp_TransExtent
                        begin="2003-01-01" end="2005-01-01" />
                      <athlete>
                        <athName>
                          Trine
                          Bakke-Rognmo
                        </athName>
                        His phone numbers are:
                        <phone>
                          123-402-0340
                        </phone>
                        <phone>
                          123-402-0000
                        </phone>
                      </athlete>
                    </athlete_Version>
                  </rep0:athlete_RepItem>
                  <rep0:athlete_RepItem
                    isItem="y" originalElement="athlete">
                    <athlete_Version>
                      <tv:timestamp_TransExtent
                        begin="2003-01-01" end="2005-01-01" />
                      <athlete>
                        <athName>
                          Lasse Kjus
                        </athName>
                      </athlete>
                    </athlete_Version>
                  </rep0:athlete_RepItem>
                </athleteTeam>
              </athleteTeam_Version>
            </rep0:athleteTeam_RepItem>
          </country>
        </country_Version>
      </rep0:country_RepItem>
    </winOlympic>
  </winOlympic_Version>
</rep0:winOlympic_RepItem>
</rep0:tv_root>
</schemaVersion0>
<schemaVersion1>
  <tv:timestamp_TransExtent begin="2005-01-01"
    end="9999-12-31" />
  <rep1:tv_root
    xmlns:rep1="http://www.cs.arizona.edu/tau/RepSchema1"
    begin="2005-01-01" end="9999-12-31">
    <rep1:winOlympic_RepItem isItem="y"
      originalElement="winOlympic">
      <winOlympic_Version>
        <tv:timestamp_TransExtent begin="2005-01-01"
          end="9999-12-31" />
        <winOlympic>
          There are
          <numEvents>11</numEvents>
          events in the Olympics.
          <rep1:country_RepItem isItem="y"
            originalElement="country">
            <country_Version>
              <tv:timestamp_TransExtent
                begin="2005-01-01" end="9999-12-31" />
              <country countryLead="Andre Agassi"
```

```
countryName="Norway">
<rep1:athleteTeam_RepItem
  isItem="y" originalElement="athleteTeam">
  <athleteTeam_Version>
    <tv:timestamp_TransExtent
      begin="2005-01-01" end="9999-12-31" />
    <athleteTeam
      numAthletes="95">
      <teamName>
        Norway_Army
      </teamName>
      Athletes will take part in various events. The athletes participating are listed below
      <rep1:athlete_RepItem
        isItem="y" originalElement="athlete">
        <athlete_Version>
          <tv:timestamp_TransExtent
            begin="2005-01-01" end="9999-12-31" />
          <athlete>
            <athName>
              Kjetil Andre Aamodt
            </athName>
          </athlete>
        </athlete_Version>
      </rep1:athlete_RepItem>
      <rep1:athlete_RepItem
        isItem="y" originalElement="athlete">
        <athlete_Version>
          <tv:timestamp_TransExtent
            begin="2005-01-01" end="9999-12-31" />
          <athlete>
            <athName>
              Andre Agassi
            </athName>
          </athlete>
        </athlete_Version>
      </rep1:athlete_RepItem>
      <rep1:athlete_RepItem
        isItem="y" originalElement="athlete">
        <athlete_Version>
          <tv:timestamp_TransExtent
            begin="2005-01-01" end="9999-12-31" />
          <athlete>
            <athName>
              Trine
              Bakke-Rognmo
            </athName>
            His phone numbers are:
            <phone>
              123-402-0340
            </phone>
            <phone>
              123-402-0000
            </phone>
          </athlete>
        </athlete_Version>
      </rep1:athlete_RepItem>
      <rep1:athlete_RepItem
        isItem="y" originalElement="athlete">
        <athlete_Version>
          <tv:timestamp_TransExtent
            begin="2005-01-01" end="9999-12-31" />
          <athlete>
            <athName>
              Lasse Kjus
            </athName>
          </athlete>
        </athlete_Version>
      </rep1:athlete_RepItem>
```

```
                    </athleteTeam>
                  </athleteTeam_Version>
                </rep1:athleteTeam_RepItem>
              </country>
            </country_Version>
          </rep1:country_RepItem>
        </winOlympic>
      </winOlympic_Version>
    </rep1:winOlympic_RepItem>
  </rep1:tv_root>
 </schemaVersion1>
</schemaItem>
</rep:sv_root>
```