MASTER'S THESIS

Implementation and Evaluation of Temporal Representations in XML

Stephen W. THOMAS
sthomas@cs.arizona.edu

Department of Computer Science, University of Arizona, Tucson, AZ 85721

March 6, 2009

Abstract

The design space for representing temporal data in XML has thus far received limited empirical attention. As a result, designers do not fully understand the design space and trade-offs between various representational approaches. This thesis presents an initial characterization of that design space and provides a qualitative and quantitative analysis of the extreme ends of the spectrum, as well as an expressive language for describing temporal data. We extend an existing language, τXSchema, to implement three classes of representations and show that the edit-based scheme provides the best performance in terms of creation time and representation size, although the item-based and sliced-based schemes can be validated more quickly. We also provide an analysis of where temporal constraint functionality should be implemented and show that, with a few exceptions, temporal constraint functionality must be implemented within the tools and cannot be implemented in the representational schema. These results provide insight into the overall design space for temporal representations that will be useful to researchers, tool implementers, and users of τXSchema.

Thesis Statement

By evaluating disparate representations through language design, tool enhancement, and empirical analysis, we will provide an initial characterization of the design space of XML representations of time-varying documents and their schemas.

ACKNOWLEDGEMENTS

I would like to give very special thanks to my thesis advisor, Dr. Richard T. Snodgrass, for his excellent guidance, great ideas, and hard work during this thesis journey. I would also like to express sincere gratitude to the other members of my thesis committee, Dr. Curtis Dyerson and Dr. Bongki Moon; without their ideas and feedback I would be lost.

Finally, none of this would have been possible without my very special lady, Tandy.

Contents

List of Figures

List of Tables

Listings

1 Introduction

XML is becoming an increasingly popular language for documents and data. XML can be approached from two quite separate orientations: a document-centered orientation (e.g., HTML) and a data-centered orientation (e.g., relational and object-oriented databases). Time-varying data naturally arises in both document-centered and data-centered orientations. Consider the following wide range of scenarios. In a university, students take various courses in different semesters. At a company, job positions and salaries change. At a warehouse, inventories evolve as deliveries are made and goods are shipped. In a hospital, drug treatment regimes are adjusted. And finally at a bank, account balances are in flux. In each scenario, querying the current state is important, e.g., how much is in my account right now, but it also often useful to know how the data has changed over time, e.g., when has my account been below $200.

A temporal document records the evolution of a document over time, i.e., all of the versions of the document. Capturing a document's evolution is vital to supporting time travel queries that delve into a past version, and incremental queries that involve the changes between two versions. A key issue is how to represent this time-based evolution: what is the *temporal representation*? Understanding this issue is fundamental to all temporal systems and is the focus of this research. We are concerned with the time and space tradeoffs between various representation approaches, the language used to create the representation, and the placement of functionality when enforcing temporal constraints.

XML Schema is a language for describing XML documents, typically expressed in terms of constraints on the structure and content of documents beyond the basic syntax constraints imposed by XML itself [43]. Thus an XML schema provides a view of the document type at a relatively high level of abstraction.

τXSchema extends XML Schema with the ability to define temporal element types [19]. 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.

1.1 Previous Work on τXSchema

The group working on the TAU Project at the Computer Science Department at the University of Arizona had previously developed a theoretical framework for data and schema versioning in XML documents. The basic architecture of the system, along with base schemas for logical annotations, physical annotations, and temporal schemas, all existed at the start of this thesis research. Initial implementations of τXMLLINT, SQUASH and UNSQUASH tools to handle data versioning had been developed. The tools supported data versioning and partially supported temporal validation.

1.2 Research Goals

In this research, we will use and extend τXSchema in order to determine how to represent time-varying data within XML in the most time and space efficient ways and how to validate temporal documents in the most time efficient way. Various approaches have already been taken by many researchers for the relational and object-oriented models [24, 27, 34]; much less work has been done with time-varying XML documents. More work is needed to fully understand the tradeoffs and options in the temporal representation, and what advantages exist for each tradeoff.

This research provides three main contributions. First, we extend the language and organization of τXSchema to allow a rich capability of describing temporal data. Second, we provide a qualitative analysis of the options for functionality placement in temporal systems. Finally, we characterize the design space for the different classes of representation types and provide a performance evaluation of each.

This research also makes two practical contributions: we add schema versioning to the representation and implementation of the tools, and we provide project support by standardizing the code organization, providing software engineering patterns to the Java classes, providing configuration management, and creating a wiki for developer and user collaboration.

1.3 Organization of the Thesis

The thesis is organized as follows. In Section 2 we provide background information on XML, XML Schema, τXSchema, and we summarize previous work. Section 3 describes the design goals and decisions we used when redesigning the τXSchema language and document organization. Section 4 extends this design to include schema versioning. In Section 5 we provide a qualitative analysis of the functionality placement of temporal constraints. Section 6 introduces and describes the design space of temporal representations. In Section 7 we provide an overview of the tools and the changes required for this research. In Section 8 we provide an empirical evaluation each of the representation classes. Section 9 concludes.

2 Background

In this section we briefly review concepts related to XML and their schemas and provide details on τXSchema. We then outline and summarize previous work in this area.

2.1 XML

The extensible markup language (XML) has recently emerged as a new standard for information representation and exchange on the Internet [43]. It has gained popularity for representing many classes of data, including structured documents, heterogeneous and semi-structured records, data from scientific experiments and simulations, and 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.

2.2 XML Schema

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 can have 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 [40], 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 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 [40]. 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. 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.

Several open-source tools exist that allow for the verification and validation of XML documents against XML Schemas. One of note is XMLLINT, which is part of the `Libxml2` [38] toolkit developed by MIT for the Gnome project. XMLLINT is a command line tool that implements a set of XML-related W3C standards and provides a wide range of capabilities, including validating XML documents against XML Schemas, transforming XML documents into other formats, compressing documents, and debugging.

2.3 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. This is referred to as *data versioning*.

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.

2.4 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 [21]. Sjoberg has also shown that the system modifications that cause changes to the structure of the data are relatively frequent [32, 33]. 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 [31]. 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 [30]. 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 [8, 12] 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 [12].

- 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.

2.5 τXSchema

Researchers have proposed a language called τXSchema (Temporal XML Schema) [19] to enable the construction and validation of temporal documents. τXSchema 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* (individual versions of a document), and provides some temporal constraints that broadly characterize how a temporal element can change over time. An important goal in the development of τXSchema was to maximally reuse existing XML standards and technology. In τXSchema, any element type can be turned into a temporal element type by including a single, simple *logical annotation* (stating 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) in the type definition and the structure of the temporal document is controlled by one or more *physical annotations* (characteristics that specify the timestamp options for the representation, such as where the timestamps are placed and their kind (e.g., valid time or transaction time) and the kind of representation). So a τXSchema document is just a conventional XML Schema document with a few logical and physical annotations.

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

τXSchema focuses on both *instance versioning* (representing a time-varying XML instance document) and *schema versioning* (representing a time-varying schema document [10, 9]). The schema can describe which aspects of an instance document change over time; this schema can itself be a time-varying document. The temporal schema contains this time-varying schema. All three components, (1) the conventional schema, (2) the logical annotations, and (3) the physical annotations, can change over time.

τXSchema is supported with a set of closely related tools:

- SQUASH. This tool takes as input a sequence of XML documents, a temporal annotation, and a physical annotation and generates a temporal XML document consistent with the physical annotation.
- UNSQUASH. This tools performs the exact opposite operation of SQUASH: given a temporal XML document and temporal schema, it generates a sequence of non-temporal XML documents, each valid over a particular period of time.
- SCHEMAMAPPER. Once the annotations are found to be consistent, this tool generates the representational schema from the original snapshot schema(s) and the temporal and physical annotations. The representational schema is needed to serve as the schema for a time-varying document.
- τXMLLINT. This tool is used to validate a temporal document against its temporal schema. The validation process includes validating each slice against its schema and validating all slices against given temporal constraints.

In τXSchema we talk about two distinct concepts when dealing with temporal documents. The first is the *language* employed by the user to the describe the temporal data. Here we mean the syntax that the user has at his disposal when telling the tools what data he has and what he wants to happen with the data. The second concept is the *representation* of the data, which is the result of the tools performing some transformation on the temporal data behind the scenes. Here we are talking about the internal format of an output file.

2.6 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 [5], studies on the frequency of data change [5], and temporal query languages [14]. 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 [23]. Grandi has created a bibliography of previous work in this area [16].

There has been a lot of interest in representing time-varying documents. Marian et al. [23] discuss versioning to track the history of downloaded documents. Chien, Tsotras and Zaniolo [5] have researched techniques for compactly storing multiple versions of an evolving XML document. Chawathe et al. [4] described a model for representing changes in semi-structured data and a language for querying over these changes. For example, the diff based approach [2, 6] focuses on an efficient way to store time-varying data and can be used to help detect transaction time changes in the document at the physical level. Buneman et al. [3] provide another means to store a single copy of an element that occurs in many snapshots. Grandi and Mandreoli [17] propose a `<valid>` tag to define a validity context that is used to timestamp part of a document. Finally, Chawathe et al. [4] and Dyreson et al. [10] discuss timestamps on edges in a semi-structured data model.

Some version control tools (which are designed for text files) have also been developed for data and schema varying XML documents (e.g., [22]). 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.

Our focus in this research is to characterize the design space of temporal representations: what are the possible ways to represent a temporal document, and what are the benefits of each? If we could categorize each existing representational approach, then we could more easily evaluate the entire design space. We also are interested in how each design choice affects the versioning of schemas and specifically how each approach allows temporal constraints to be implemented. Related to these tasks is the desire to have a language for easily describing temporal data, independent of the underlying representation on which we concentrate first.

3 Design Goals and Decisions

In this section we focus on the language used to describe temporal systems: temporal documents, temporal schemas, and annotations. Our goal is to extend the existing τXSchema language to provide the user an enhanced capability for describing temporal data and systems independent of the underlying physical representation. We first introduce and define terminology. We then outline our goals for the design of the language and provide a few short examples. From these goals we establish a set of decisions and conclude with a comprehensive example.

3.1 Terminology

This section defines terms relevant to τXSchema.

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

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

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

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

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

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

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

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

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

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

3.2 Design Goals

This section defines a set of high-level goals for the structure of τXSchema.

(a) *Upward compatibility* with established XML designs, techniques, and tools is the most important goal that drives the rest of the design. Conventional documents and conventional schemas should work within τXSchema. As an example, Figure 1 shows a conventional document and conventional schema being validated by XMLLINT [38]. τXMLLINT should be able to produce the same output as the conventional tool given the same input.

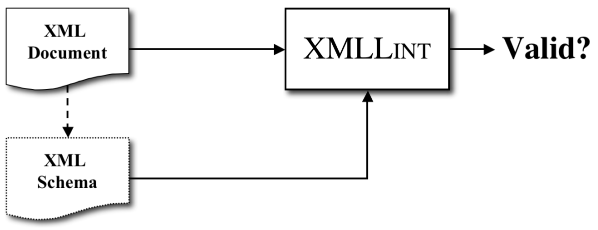


Figure 1: An XML document, which references an XML Schema, being validated by XMLLINT. The solid lines going into XMLLINT indicate that the documents are explicitly input into the tool.

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

(c) *Convenience* and *intuition* should be stressed. It is important to make migrating from a conventional system to τXSchema as easy as possible. Whenever possible, we should adopt existing XML formats, naming schemes, and methodologies; see Listings 1 and 2 for examples.

Listing 1: Conventional XML Schema syntax for one schema to include another.

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

Listing 2: τXSchema syntax to include one temporal schema into another.

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

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

(e) *Substitutability* of the various artifacts of the system is another primary goal. If there is more than one way to describe a temporal artifact, then each way should be permitted anywhere any other way is permitted. For example, both Listings 3 and 4 below contain the same information; the former lists a sequence of conventional documents and their lifetimes while the latter is the squashed version of these same documents.

Listing 3: A way to represent two conventional documents.

```
<temporalRoot>
```

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

Listing 4: Another way to represent two conventional documents.

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

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

</temporalRoot>
```

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

(f) Temporal data can occur at *any level* of the system. This includes the temporal schema and annotations document. For example, temporal schemas should be allowed to reference other temporal schemas, which in turn reference other temporal (or conventional) schemas. The schemas for these schemas should be allowed to be temporal or conventional. Eventually a conventional document or schema is reached, and the process completes; however, no limits or constraints should be placed on the amount or location of temporal data. As another example, Listing 5 shows a temporal schema that references an annotation document that itself is temporal.

Listing 5: A temporal schema references an annotations document that is itself temporal.

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

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

(h) Simple cases should be *simple* for the user to create. For example, if only one conventional schema is present, then the temporal schema is only required to list the URI for this schema, and no further markup is needed for the tools to work correctly. Listing 6 shows such a temporal schema.

Listing 6: The temporal schema should be as simple as possible.

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

3.3 Design Decisions

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

3.3.1 General Decisions

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

(2) The characterization of a temporal document will be achieved by using a `<sliceSequence>` element; each individual conventional document's URI and its associated lifetime will be directly listed in this element via a `<slice>` element and its `location`, `begin`, and optional `end` attributes. See Listing 3 for a simple example. This decision fulfills goal **(f)** because the `<sliceSequence>` element can occur anywhere; thus allowing temporal data to occur anywhere. It also fulfills goal **(d)** because this method allows a simple way to add new versions of documents to the system.

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

(4) In all cases, the default logical annotation is "anything can change" and the default physical annotation is "timestamp is located at root." This decision satisfies goal **(h)** by supplying a useful default annotation set if the user doesn't supply one. See Listing 6 above for an example of a schema that does not specify any annotations.

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

3.3.2 Temporal Schema Decisions

(6) A temporal schema will have the root element `<temporalSchema>` which belongs to the τXSchema namespace. The root element will have two subelements, only the first of which is required.

- `<conventionalSchema>`. In this element, the user will specify the conventional schema(s) that belongs to the system.
- `<annotationSet>`. In this element, the user will specify the annotation(s) that belongs to the system, if any.

Within each of the two elements there will be four separate ways to specify schemas and annotations.

(a) Listing the URI of each conventional document with a `<sliceSequence>` element (see Listing 3)

(b) Including a (conventional or temporal) document with a `<include>` element (see Listing 1 for an example and decision **(9)** for more information)

(c) Placing the text of a document directly in the element (see Listing 4)

(d) Omitting the element altogether. In this case, default behavior will be assumed (see, e.g., decision **(4)**). Default behavior would only apply to annotations documents, as a default conventional schema (by perhaps automatically detecting and creating a schema based on the first conventional document) would be dangerous and might provide unintended semantics.

Providing these different mechanisms allows for substitutability (satisfying goal **(f)**), convenience (satisfying goal **(a)**), and simplicity (satisfying goal **(d)**).

(7) An `<include>` element will be used to include a document into the temporal schema; it has the same semantics of placing the entire actual text of the document into the schema. `<include>` can reference any kind of document, including a conventional schema, a temporal schema, and annotation documents. This element has the effect of removing the root of the included document; see Listings 7 and 8 for an example.

Listing 7: A schema using `<include>`.

```
<temporalSchema>
 ...
 <annotationSet>
  <include schemaLocation="anno.xml"/>
 </annotationSet>
 ...
</temporalSchema>
```

Listing 8: `anno.xml`

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

This decision satisfies goal **(e)** by allowing both the `<include>` element and the actual text of the document to have the exact same semantics and goal **(c)** by keeping consistent syntax as XML Schema `<include>` elements.

(8) Both conventional and temporal schemas can `<include>` any number of conventional and temporal schemas. See Listing 5 for an example. This decision satisfies goal **(f)** by permitting temporal data to occur at any level in the system.

(9) There can be one temporal schema for each independent conventional schema present in the system. In this way, each conventional schema can vary over time independently, as well as have their own logical and physical annotations. See Figure 2 for an example, and Section 3.4 for more information.

3.3.3 Temporal Document Decisions

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

(11) A temporal document will have an `<temporalSchemaSet>` element with subelements `<temporalSchema>` that will specify the URI of each temporal schemas. See Listing 9 for an example.

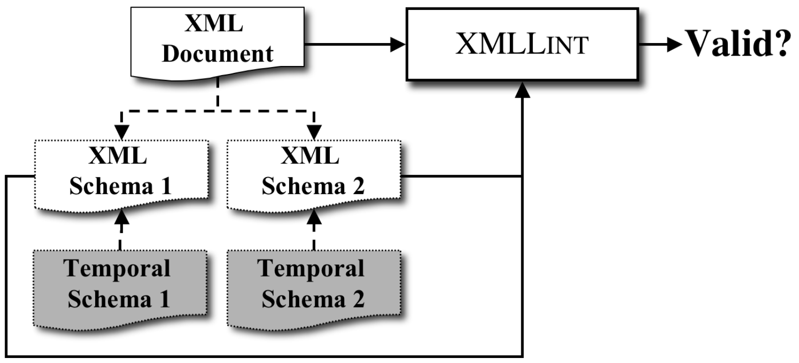


Figure 2: Each conventional schema has a separate corresponding temporal schema.

(12) The root element may have a `<sliceSequence>` element to list a sequence of conventional documents (i.e., slices). We choose the term "sequence" here since the ordering of the slices is important; they must be listed from earliest to latest. See Listing 9.

Listing 9: `exampleTemporalDocument.xml`

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

This decision satisfies goal **(h)** by providing an easy way to SQUASH the documents, goal **(d)** by providing a simple mechanism for adding slices, and goal **(e)** by allowing either the `<sliceSequence>` element or the SQUASH representation to appear in the temporal document.

3.3.4 Annotation Document Decisions

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

3.4 Company Example

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

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

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

Figure 3 depicts the overall scenario.

3.4.1 Initial Configuration

Consider the following scenario which begins on 2008-01-01. The user has a conventional schema which defines a `<Person>` element, which itself has a `<Name>` element, an `<SSN>` element, and an `ID` attribute (see Listing 10).

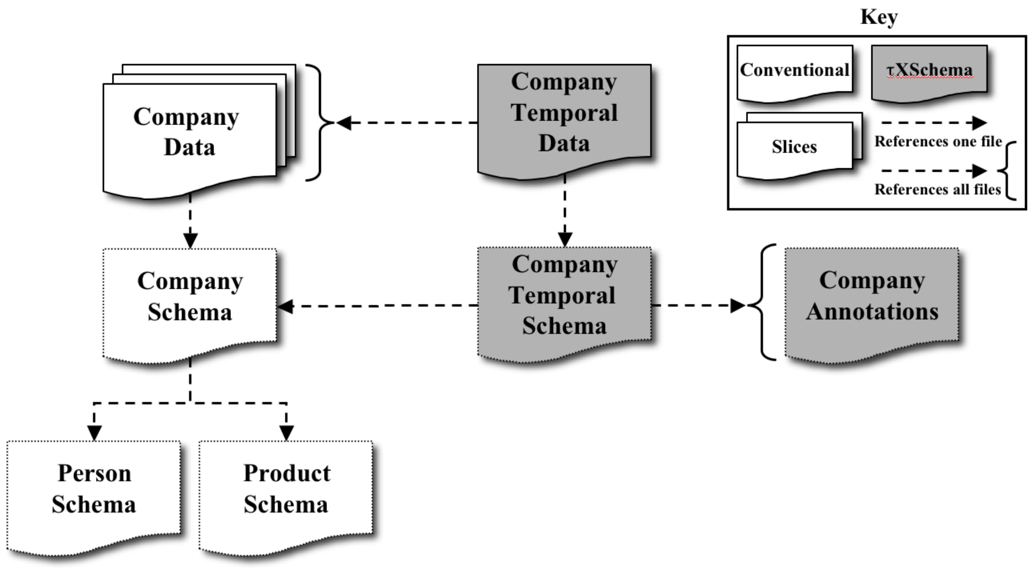


Figure 3: An overview of the end-state of the Company example.

Listing 10: `Company.A.xsd`

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

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

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

</xsd:schema>
```

He also has a conventional document conforming to the schema (see Listing 11).

Listing 11: `data.A.0.xml`

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

  <Person ID="1">
    <Name>Steve</Name>
    <SSN>111-22-3333</SSN>
  </Person>

</Company>
```

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

In the following sections, we will add new versions of the conventional document, add new versions of the conventional schema, break up the conventional schema into multiple subschemas, and specify temporal annotations.

3.4.2 Adding Temporal Data

On 2008-03-17, the user corrects the `<SSN>` element in the conventional document to produce a new version (see Listing 12). The user can now use τXSchema to create temporal documents and use the τXSchema tools to validate these documents.

Listing 12: `data.A.1.xml`

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

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

</Company>
```

The user creates a temporal document that lists both slices of the conventional document with their associated timestamps (see Listing 13).

Listing 13: `temporalDocument.0.1.xml`

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

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

</td:temporalRoot>
```

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

4 Schema Versioning Example

We now extend our design in Section 3 to include schema versioning. Here we must focus on the intricacies of changing namespaces from slice to slice, versioning of subschemas that are included or imported into the main schema, and versioning multiple main schemas. We briefly cover each in turn.

4.1 Company Example Extended

We now extend our example presented in Figure 3 in Section 3.4 to include schema versioning. We use the same conventions and naming schemes as before. Figure 4 depicts the scenario. Here, in addition to the Company Data document varying over time, the Company Schema, Person Schema, Product Schema, and Company Annotation documents will also vary over time. Note that those documents are depicted here as multiple slices.

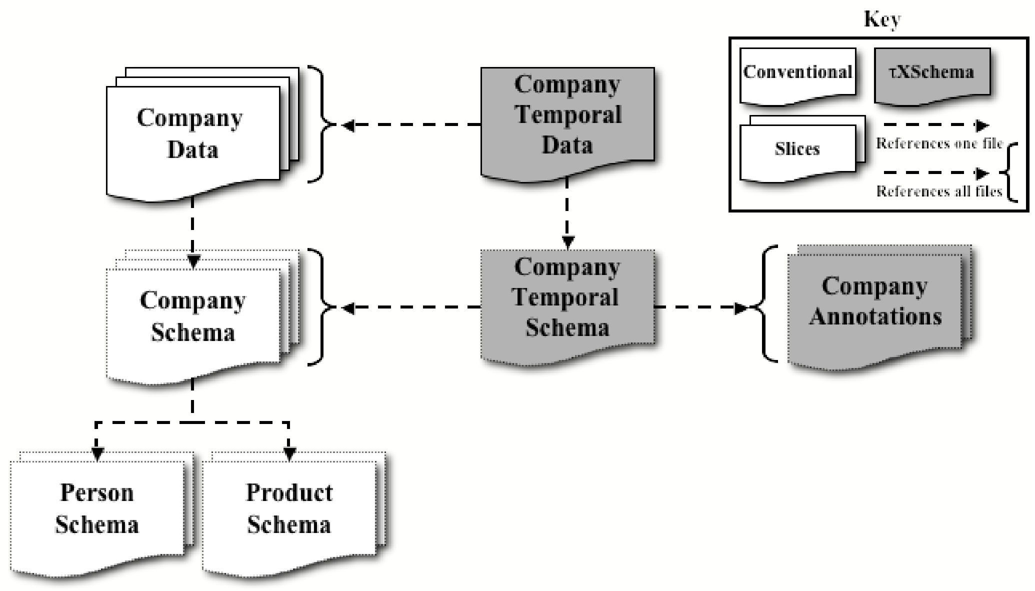


Figure 4: An overview of the end-state of the Company example.

4.2 Changing Schemas

On 2008-05-22, the user decides to make a change to the conventional schema. He changes the `<Name>` element to `<FirstName>`, resulting in a new version of the schema (Listing 14, line 19).

Listing 14: `Company.B.xsd`

```
1 <?xml version="1.0"?>
2 <xsd:schema
3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4   targetNamespace="http://www.company.org"
5   xmlns="http://www.company.org"
6   elementFormDefault="qualified">
7
8   <xsd:element name="Company">
9     <xsd:complexType>
```

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

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

</xsd:schema>
```

The user must therefore update the conventional document to conform to the new schema (see Listing 15, line 5).

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

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Company xmlns="http://www.company.org">
3
4   <Person ID="1">
5     <FirstName>Steve</FirstName>
6     <SSN>123-45-6789</SSN>
7   </Person>
8
9 </Company>
```

The user now creates an explicit temporal schema to represent the changes to the conventional schema. He chooses to use method (a) as described in design decision **(6)** above (see Listing 16). Note that since the user has yet to specify any annotations, the defaults are still in effect.

Listing 16: `temporalSchema.0.xml`

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

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

</ts:temporalSchema>
```

The user must also update the temporal document to include the new slice of the conventional document (Listing 17, line 11).

Listing 17: `temporalDocument.1.1.xml`

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD">
3   <td:temporalSchemaSet>
4     <td:temporalSchema location="./temporalSchema.0.xml"/>
5   </td:temporalSchemaSet>
6
7   <td:sliceSequence>
8     <td:slice location="data.A.0.xml" begin="2008-01-01" />
9     <td:slice location="data.A.1.xml" begin="2008-03-17" />
```

```
    <td:slice location="data.B.1.xml" begin="2008-05-22" />
  </td:sliceSequence>

</td:temporalRoot>
```

4.2.1 Introducing Subschemas

On 2008-07-11, the user decides to split the schema into several smaller subschemas while also adding a new element. The *main* schema (see Listing 18) no longer defines any elements itself, but instead uses the XML Schema `<import>` (line 9) and `<include>` (line 10) elements to reference two subschemas (see Listings 19 and 20).

Listing 18: `Company.C.xsd`

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

Listing 19: `Person.C.0.xsd`

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

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

</xsd:schema>
```

Listing 20: `Product.C.0.xsd`

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

```
 6    elementFormDefault="qualified">
 7
 8    <xsd:complexType name="ProductType">
 9      <xsd:sequence>
10        <xsd:element name="Type" type="xsd:string" minOccurs="1" maxOccurs="1"/>
11      </xsd:sequence>
12    </xsd:complexType>
13
14  </xsd:schema>
```

The user updates the instance document to conform to the new schema paradigm and adds a `<Product>` element (see Listing 21, lines 10–12).

Listing 21: `data.C.2.xml`

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <Company xmlns="http://www.company.org" xmlns:per="http://www.person.org">
 3
 4    <Person ID="1">
 5      <per:FirstName>Steve</per:FirstName>
 6      <per:SSN>111-22-3333</per:SSN>
 7    </Person>
 8
 9    <Product>
10       <Type>Widget</Type>
11    </Product>
12
13  </Company>
```

Note that in this example, the `Company` schema is taking the so-called "heterogeneous" and "homogeneous" namespace approaches for the `Person` and `Product` subschemas, respectively [7]. That is, the Person subschema's namespace is exposed in the `Company` schema, while the `Product` subschema defines the same target namespace as the `Company` schema.

The user then changes the temporal schema (Listing 22, line 8) and the temporal document (Listing 23, lines 4 and 11) to reflect these modifications. Note that the temporal schema only references the `Company` schema, because that schema directly imports `Person` and includes `Product`.

Listing 22: `temporalSchema.1.xml`

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">
 3
 4    <ts:conventionalSchema>
 5      <ts:sliceSequence>
 6        <ts:slice location="Company.A.xsd" begin="2008-01-01" />
 7        <ts:slice location="Company.B.xsd" begin="2008-05-22" />
 8        <ts:slice location="Company.C.xsd" begin="2008-07-11" />
 9      </ts:sliceSequence>
10    </ts:conventionalSchema>
11
12  </ts:temporalSchema>
```

Listing 23: `temporalDocument.1.2.xml`

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD">
 3    <td:temporalSchemaSet>
 4      <td:temporalSchema location="./temporalSchema.1.xml"/>
 5    </td:temporalSchemaSet>
 6
 7    <td:sliceSequence>
 8      <td:slice location="data.A.0.xml" begin="2008-01-01" />
```

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

</td:temporalRoot>
```

4.2.2 Adding Logical Annotations

On 2008-08-04, the user decides to construct an annotation document. Here, the user creates a logical annotation (via the `<item>` element) that specifies that only the `<FirstName>` element can change (see Listing 24).

Listing 24: `annotations.0.xml`

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

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

</as:annotationSet>
```

The user must then update the temporal schema to include the temporal annotation document (see Listing 25, lines 12–16) and the temporal document to point to the new version of the temporal schema (see Listing 26, line 4). τXMLLINT will now check this constraint between conventional document slices over time.

Listing 25: `temporalSchema.2.xml`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">
3
4    <ts:conventionalSchema>
5      <ts:sliceSequence>
6        <ts:slice location="Company.A.xsd" begin="2008-01-01" />
7        <ts:slice location="Company.B.xsd" begin="2008-05-22" />
8        <ts:slice location="Company.C.xsd" begin="2008-07-11" />
9      </ts:sliceSequence>
10   </ts:conventionalSchema>
11
12   <ts:annotationSet>
13     <ts:sliceSequence>
14       <ts:slice location="annotations.0.xml" begin="2008-08-04" />
15     </ts:sliceSequence>
16   </ts:annotationSet>
17
18 </ts:temporalSchema>
```

Listing 26: `temporalDocument.2.3.xml`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD">
3    <td:temporalSchemaSet>
4      <td:temporalSchema location="./temporalSchema.2.xml"/>
5    </td:temporalSchemaSet>
6
```

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

</td:temporalRoot>
```

4.2.3 Temporal Subschemas

On 2008-09-10, the user changes the `Person` subschema (see Listing 27, line 11) to include a `<LastName>` element.

Listing 27: `Person.D.1.xsd`

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

The user must update the `Company` schema (Listing 28, line 9) to reference the new version of the subschema.

Listing 28: `Company.D.xsd`

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

The user must also update the conventional document (Listing 29, line 7) to include a `<LastName>` element.

Listing 29: `data.D.3.xml`

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

  <Person ID="1">
    <per:FirstName>Steve</per:FirstName>
    <per:LastName>Thomas</per:LastName>
    <per:SSN>111-22-3333</per:SSN>
  </Person>

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

</Company>
```

The user then changes the temporal schema (Listing 30, line 9) and temporal document (Listing 31, lines 4 and 12) to reflect these modifications.

Listing 30: `temporalSchema.3.xml`

```xml
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <ts:temporalSchema xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema">
 3
 4   <ts:conventionalSchema>
 5     <ts:sliceSequence>
 6       <ts:slice location="Company.A.xsd" begin="2008-01-01" />
 7       <ts:slice location="Company.B.xsd" begin="2008-05-22" />
 8       <ts:slice location="Company.C.xsd" begin="2008-07-11" />
 9       <ts:slice location="Company.D.xsd" begin="2008-09-10" />
10     </ts:sliceSequence>
11   </ts:conventionalSchema>
12
13   <ts:annotationSet>
14     <ts:sliceSequence>
15       <ts:slice location="annotations.0.xml" begin="2008-08-04" />
16     </ts:sliceSequence>
17   </ts:annotationSet>
18
19
20 </ts:temporalSchema>
```

Listing 31: `temporalDocument.3.3.xml`

```xml
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD">
 3   <td:temporalSchemaSet>
 4     <td:temporalSchema location="./temporalSchema.3.xml"/>
 5   </td:temporalSchemaSet>
 6
 7   <td:sliceSequence>
 8     <td:slice location="data.A.0.xml" begin="2008-01-01" />
 9     <td:slice location="data.A.1.xml" begin="2008-03-17" />
10     <td:slice location="data.B.1.xml" begin="2008-05-22" />
11     <td:slice location="data.C.2.xml" begin="2008-07-11" />
12     <td:slice location="data.D.3.xml" begin="2008-09-10" />
13   </td:sliceSequence>
14
15 </td:temporalRoot>
```

4.2.4 Namespace Changes

One month later (2008-11-13), the user changes the target namespace of the main schema to `"stevescompany.org"` (see Listing 32, lines 4 and 5). Since the `Product` subschema uses the homogeneous namespace paradigm, it also must be updated to the new namespace (see Listing 33, lines 4 and 5). Of course, the user must also update the conventional document (see Listing 34, line 2).

Listing 32: `Company.E.xsd`

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

Listing 33: `Product.E.1.xsd`

```
 1 <?xml version="1.0"?>
 2 <xsd:schema
 3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 4   targetNamespace="http://www.stevescompany.org"
 5   xmlns="http://www.stevescompany.org"
 6   elementFormDefault="qualified">
 7
 8   <xsd:complexType name="ProductType">
 9     <xsd:sequence>
10       <xsd:element name="Type" type="xsd:string" minOccurs="1" maxOccurs="1"/>
11     </xsd:sequence>
12   </xsd:complexType>
13
14 </xsd:schema>
```

Listing 34: `data.E.3.xml`

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <Company xmlns="http://www.stevescompany.org" xmlns:per="http://www.person.org">
 3
 4   <Person ID="1">
 5     <per:FirstName>Steve</per:FirstName>
 6     <per:LastName>Thomas</per:LastName>
 7     <per:SSN>111-22-3333</per:SSN>
 8   </Person>
 9
10   <Product>
11      <Type>Widget</Type>
12   </Product>
13
14 </Company>
```

The user then changes the temporal schema (see Listing 35, line 10) and temporal document (see Listing 36, lines 4 and 13) to reflect his modifications.

Listing 35: `temporalSchema.4.xml`

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

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice location="Company.A.xsd" begin="2008-01-01" />
      <ts:slice location="Company.B.xsd" begin="2008-05-22" />
      <ts:slice location="Company.C.xsd" begin="2008-07-11" />
      <ts:slice location="Company.D.xsd" begin="2008-09-10" />
      <ts:slice location="Company.E.xsd" begin="2008-11-13" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

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

</ts:temporalSchema>
```

Listing 36: `temporalDocument.4.3.xml`

```xml
<?xml version="1.0" encoding="UTF-8"?>
<td:temporalRoot xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TD">
  <td:temporalSchemaSet>
    <td:temporalSchema location="./temporalSchema.4.xml"/>
  </td:temporalSchemaSet>

  <td:sliceSequence>
    <td:slice location="data.A.0.xml" begin="2008-01-01" />
    <td:slice location="data.A.1.xml" begin="2008-03-17" />
    <td:slice location="data.B.1.xml" begin="2008-05-22" />
    <td:slice location="data.C.2.xml" begin="2008-07-11" />
    <td:slice location="data.D.3.xml" begin="2008-09-10" />
    <td:slice location="data.E.3.xml" begin="2008-11-13" />
  </td:sliceSequence>

</td:temporalRoot>
```

4.2.5 Multiple Conventional Schemas

The user now (2008-11-27) wants to create a level of independence between each of the conventional schemas: when a subschema changes, he does not want to have to change the main schema. To do this, he creates a temporal schema for each of the conventional schemas and in the main conventional schema he references the temporal schema (as opposed to the conventional subschema).

Listings 37 and 38 show the new temporal schemas for the Product and Person subschemas, respectively. Listing 39, lines 9 and 12, shows the main conventional schema referencing the new temporal schemas.

Listing 37: `ProductTemporalSchema.xml`

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

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice filename="Product.C.0.xsd" begin="2008-07-11" />
      <ts:slice filename="Product.E.1.xsd" begin="2008-11-13" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

</ts:temporalSchema>
```

Listing 38: `PersonTemporalSchema.xml`

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

  <ts:conventionalSchema>
    <ts:sliceSequence>
      <ts:slice filename="Person.C.0.xsd" begin="2008-07-11" />
      <ts:slice filename="Person.D.1.xsd" begin="2008-09-10" />
    </ts:sliceSequence>
  </ts:conventionalSchema>

</ts:temporalSchema>
```

Listing 39: `Company.F.xsd`

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

  <xsd:import namespace="http://www.person.org"
              schemaLocation="./PersonTemporalSchema.xml" />

  <xsd:include schemaLocation="./ProductTemporalSchema.xml" />

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

</xsd:schema>
```

5 Functionality Placement: Schema vs. Tools

To this point we have focused on how the user describes his temporal documents and their schemas. We now turn to examine where schema constraint functionality is placed and the issues that arise when validating temporal constraints. In this section we focus on the latter.

Before facing these issues, it is convenient to discuss the approach that the τXSchema tools take to validate temporal contraints. Figure 5 shows the overall architecture of the tools as they manage XML documents and their schemas. A sequence of non-temporal documents is input into SQUASH to create a temporal representation; this document can then be validated using τXMLLINT and SCHEMAMAPPER. UNSQUASH can be used to reconstruct the original non-temporal documents from the temporal representation, while RESQUASH can be used to create a new representation (e.g., different timestamp locations) from a given representation.

Figure 6 provides the validation procedure used by τXMLLINT. The first step is to pass the temporal schema into τXMLLINT, which ensures that the logical and physical annotations are consistent with the snapshot schema and with each other. Once the annotations are found to be consistent, SCHEMAMAPPER is invoked to generate a representational schema from the original snapshot schema and the temporal and physical annotations. The representational schema is then used as the schema for the temporal document and input into a conventional validator (in this case, XMLLINT). The next step is to pass the temporal document and the temporal schema to *Temporal Constraint Validator*. This step is to enforce temporal constraints that are not possible to be enforced by the representational schema alone.

A key design decision during the validation of temporal constraints is the placement of functionality: should a constraint be implemented in the representational schema or within the temporal constraint validator? Implementing (or *expressing*, or *enforcing*) constraints in the representational schema may provide faster validation since a conventional validator can be invoked directly, but may result in increased size and complexity of the representational schema. Conversely, implementing constraints in the temporal validator may yield small and compact schemas, but requires more time to perform the validation since the tools must perform checks across all slices sequentially and individually in the worst case.

In the following sections we explore two issues related to the placement of functionality. First, we determine which temporal constraints are possible to be expressed in a representational schema using the item-based representation class[5] and which can only be implemented in the temporal constraint validator. Second, for those constraints that can be implemented in both the schema and the temporal constraint validator, we provide a brief analysis of the tradeoffs between the two placements.

5.1 Constraints

In this section we discuss both *sequenced* (enforced at each point in time) and *non-sequenced* (enforced on the temporal document as a whole) constraints and determine for each whether it is possible to express that constraint in the representational schema. For both sequenced and non-sequenced constraints, we focus on the following classes of constraints.

Identity constraints These constraints restrict uniqueness of elements and attributes in a given document. Identity constraints are defined in the schema document using a combination of `<selector>` and `<field>` sub-elements within an `<key>` or `<unique>` element.

[5]Section 6 introduces and describes the four kinds of representation classes and Section 5.2 outlines how the other three classes affect this analysis.

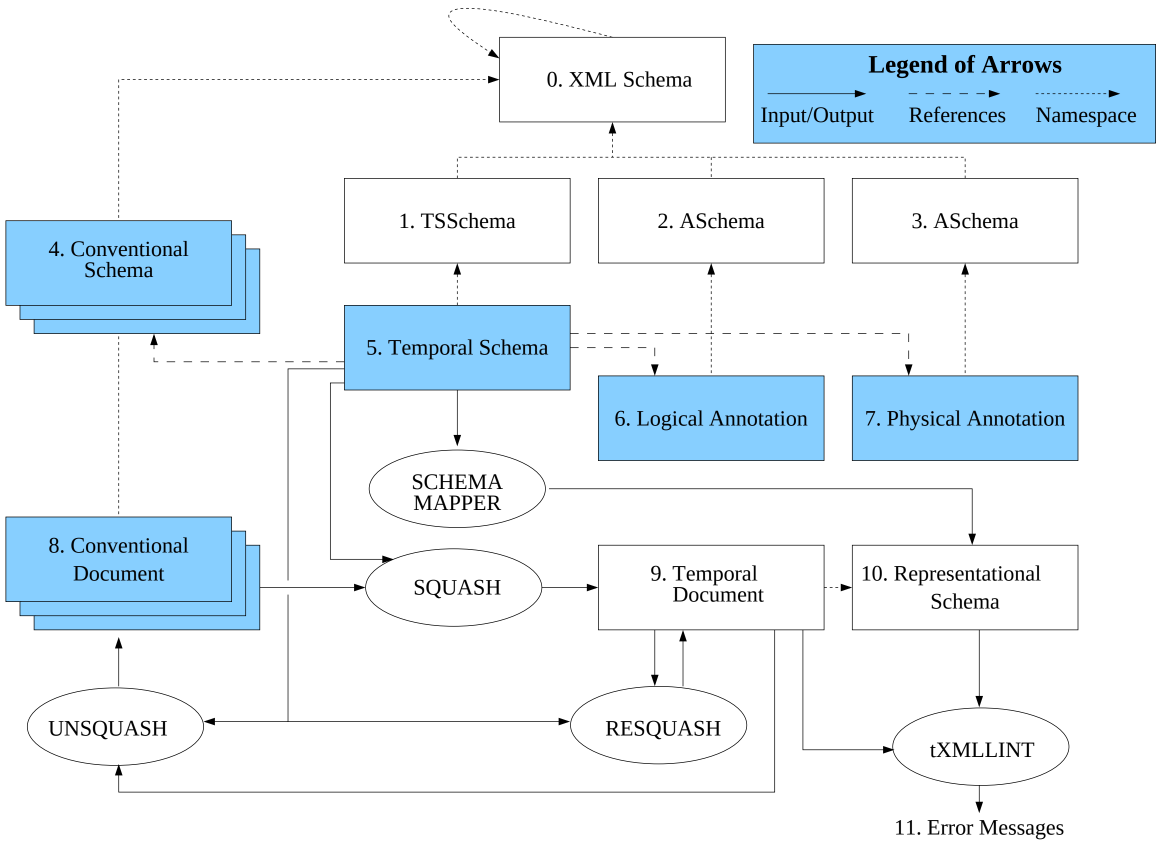


Figure 5: The overall architecture of τXSchema.

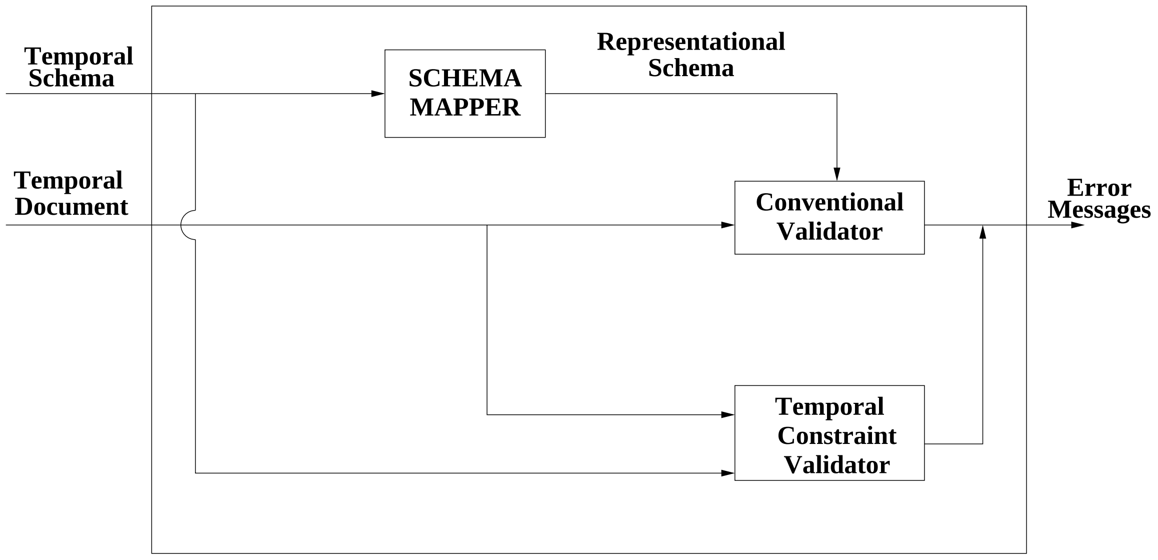


Figure 6: Validating a document with Time-Varying Data: τXMLLINT.

	Identity	Referential	Cardinality	Datatype
Sequenced	×	×	✓	✓
Non-sequenced	×	×	×	×

Table 1: The classes of constraints that can be implemented in a representational schema in the general case.

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

Cardinality constraints The cardinality of elements in XML documents is restricted by the use of `minOccurs` and `maxOccurs` in the XML Schema document. The cardinality of attributes is restricted using `optional`, `required`, or `prohibited`.

Datatype restrictions Datatype definitions can restrict the structure and content of elements, and the content of attributes. For example, a datatype definition can restrict the content of an element `<age>` to be between 0 and 100.

Table 1 provides a sneak-peak summary of which of the eight classes of constraints we claim can be implemented in the representational schema in the general case. We now provide an argument for each cell of this table in turn.

5.1.1 Sequenced Constraints

In this section we examine whether each class of sequenced constraints can be enforced by a representational schema. Given a conventional XML Schema constraint, we define the corresponding logical semantics in XML Schema in terms of a *sequenced constraint*. For example, a conventional (cardinality) constraint, “There should be between 0 and four 4 URLs for each supplier,” has the following sequenced constraint: “There should be between 0 and 4 website URLs for each supplier *at every point in time*.”

For each sequenced constraint below we use the following approach. If we claim that the constraint can be enforced by a representational schema, we outline a method that can be used by the τXSchema tools to transform the sequenced constraint syntax into standard XML Schema syntax. If, on the other hand, we claim that the constraint cannot be enforced by a representational schema, we provide a counter example that illustrates the specific shortcoming of XML Schema that forbids the constraint to be enforced.

Identity Constraints We claim that identity constraints of elements and attributes *cannot* be enforced in a representational schema in the general case. To see this, consider the following example that begins with Listings 40 and 41. In this example, we require `<zip>` elements to have unique `code` attributes via an identity constraint named `zipUnique`.

Listing 40: XML Schema `<unique>`.

```
...
   <xs:unique name="zipUnique">
     <xs:selector xpath="zip"/>
     <xs:field xpath="@code"/>
   </xs:unique>
...
```

Listing 41: Unique `code`s (slice 1).

```
...
  <zip code="85721"> Tucson,  AZ </zip>
  <zip code="85001"> Phoenix, AZ </zip>
...
```

Now suppose the user were to change the `code` for Tucson to be the same as Phoenix (violating the conventional schema's identity constraint), and then back again (see Listings 42 and 43).

Listing 42: Slice 2 (invalid).

```
...
  <zip code="85001"> Tucson,  AZ </zip>
  <zip code="85001"> Phoenix, AZ </zip>
...
```

Listing 43: Slice 3 (valid).

```
...
  <zip code="85721"> Tucson,  AZ </zip>
  <zip code="85001"> Phoenix, AZ </zip>
...
```

Assuming that the physical annotations place the timestamps at the `<zip>` element level, the above actions would create an item-based representation similar to the one shown in Listing 44.

Listing 44: Squashed version of the three slices.

```
...
  <zip_RepItem>
    <zip_Version begin="1" end="1">
      <zip code="85721"> Tucson, AZ </zip>
    </zip_Version>
    <zip_Version begin="2" end="2">
      <zip code="85001"> Tucson, AZ </zip>
    </zip_Version>
    <zip_Version begin="3">
      <zip code="85721"> Tucson, AZ </zip>
    </zip_Version>
  </zip_RepItem>

  <zip_RepItem>
    <zip_Version begin="1">
      <zip code="85001"> Phoenix, AZ </zip>
    </zip_Version>
  </zip_RepItem>
 ...
```

With this representation, there is no way to create an identity constraint in XML Schema that can detect that both `code` values at time 2 are the same. If the constraint were constructed to restrict all `./zip_RepItem/zip_Version/zip/@code` values[6] to be unique, this would fail since at times 1 and 3, Tucson has a `code` value of 85701, and this is legal in our temporal constraint. If the constraint were constructed to require all `./zip_Version/zip/@code` within each `./zip_RepItem` to be identical, this would also fail since the user is allowed to change the zip code from slice to slice.

One could imagine extending the constraint shown in Listing 40 to include key specification fields `begin` and `end` for the valid times associated with each version of the `zip` element, as shown in Listing 45 on lines 29 and 30, with the corresponding attributes in the element specification.

[6]This is XPath code [39].

Listing 45: XML Schema `<unique>` with additional fields.

```
...
  <xs:unique name="zipUniqueAttempt">
    <xs:selector xpath="zip"/>
    <xs:field xpath="@code"/>
    <xs:field xpath="@begin"/>
    <xs:field xpath="@end"/>
  </xs:unique>
...
```

As long as the `begin` and `end` attributes are maintained in proper order (which can be checked by τXMLLINT), the keys will uniquely identify each key within each snapshot. Listing 46 below shows an example where the addition of such attributes will achieve the desired functionality. Here, the conventional validator would detect that the `zip` elements on lines 95 and 104 are in violation of the unique constraint, which is indeed correct.

Listing 46: Squashed document with multiple changes

```
 89 ...
 90   <zip_RepItem>
 91     <zip_Version begin="1" end="1">
 92       <zip code="85721" begin="1" end="1"> Tucson, AZ </zip>
 93     </zip_Version>
 94     <zip_Version begin="2" end="2">
 95       <zip code="85001" begin="2" end="2"> Tucson, AZ </zip>
 96     </zip_Version>
 97   </zip_RepItem>
 98 
 99   <zip_RepItem>
100     <zip_Version begin="1" end="1">
101       <zip code="85001" begin="1" end="1"> Phoenix, AZ </zip>
102     </zip_Version>
103     <zip_Version begin="2" end="2">
104       <zip code="85001" begin="2" end="2"> Phoenix, AZ </zip>
105     </zip_Version>
106   </zip_RepItem>
107  ...
```

However, this approach will not succeed in the general case because it only enforces uniqueness at the interval end points and not anywhere within the interval. For example, consider the excerpt from a squashed document shown in Listing 47. We see that the elements on lines 95 and 101 conflict our desired constraint, but since the `begin` attributes are distinct, XML does not detect an error.

Listing 47: Squashed document with multiple changes

```
 89 ...
 90   <zip_RepItem>
 91     <zip_Version begin="1" end="1">
 92       <zip code="85721" begin="1" end="1"> Tucson, AZ </zip>
 93     </zip_Version>
 94     <zip_Version begin="2" end="2">
 95       <zip code="85001" begin="2" end="2"> Tucson, AZ </zip>
 96     </zip_Version>
 97   </zip_RepItem>
 98 
 99   <zip_RepItem>
100     <zip_Version begin="1" end="2">
101       <zip code="85001" begin="1" end="2"> Phoenix, AZ </zip>
102     </zip_Version>
103   </zip_RepItem>
104  ...
```

We are thus forced to conclude that XML Schema lacks the sufficient capability to discriminate time boundaries in a way that would allow sequenced identity constraints to be enforced.

Referential Integrity Constraints We claim that referential integrity constraints *cannot* be implemented in a representational schema. The argument is similar to that for identity constraints: there is no way to create a constraint in XML Schema that can both satisfy referential integrity and time issues. Consider the example shown in Listings 48 and 49.

Listing 48: A referential constraint.

```
...
<!-- Defines a key named "pNumKey" -->
<key name="pNumKey">
  <selector xpath="states/state"/>
  <field xpath="@id"/>
</key>

<!-- Says that the "state" attribute -->
<!-- in <zip><city></zip> elements   -->
<!-- must match a pNumKey.           -->
<keyref name="stateMatcher"
        refer="r:pNumKey">
  <selector xpath="regions/zip/city"/>
  <field xpath="@state"/>
</keyref>
...
```

Listing 49: Squashed document.

```
...
<regions_RepItem>
  <regions_Version begin="1" end="1">
    <regions>
      <zip code="85701">
        <city state="1"/>
      </zip>
    </regions>
  </regions_Version>
  <regions_Version begin="2" end="3">
    <regions>
      <zip code="85701">
        <city state="6"/>
      </zip>
    </regions>
  </regions_Version>
</regions_RepItem>

<states_RepItem>
  <states_Version begin="1" end="2">
    <states>
      <state id="1">Arizona</state>
      <state id="2">California</state>
  </states>
 </states_Version>
 <states_Version begin="3" end="3">
   <states>
     <state id="6">Arizona</state>
     <state id="2">California</state>
   </states>
 </states_Version>
</states_RepItem>
  ...
```

Here, we have a constraint that says *"A city element's state attribute must match an existing state element's id attribute at every point in time."* The squashed document shows that this constraint is satisfied at times 1 and 3, but violated at time 2 since Arizona will point to a non-existent state `id`. To construct an XML Schema that could describe this situation, one would need to be able to somehow discriminate between different `<regions_Version>` elements according to their `begin` and `end` attributes, but there is no such way to accomplish this without the help from a procedural language like XQuery [42]. We are again forced to conclude the XML schema lacks sufficient mechanisms to enforce a referential constraint.

Cardinality Constraints We claim that the cardinality of both elements and attributes *can* be enforced in the representational schema. Consider an element e which has created a logical item i. If the lowest timestamp is located at a ancestor or descendent of e, then no change to the definition of e from the original schema is necessary, only a direct copy into the representational schema. If a timestamp is located at i, then the cardinality constraint information must be moved from e up to

i in the representational schema. Since there must be one item for each original element, ensuring that we have a particular number of items is the same as ensuring that we have a particular number of original elements.

Listings 50 and 51 below show an example constraint: *"The element* `<supplier>` *can occur exactly 1 or 2 times."* We first assume that the physical timestamps–specified in the temporal schema– are placed at a predecessor or successor element of the `<supplier>` element. In this case the specification of the `<supplier>` element requires no modification in the representational schema.

Listing 50: Snapshot schema 1.

```
...
  <xs:element name="supplier"
          minOccurs="1" maxOccurs="2">
    ...
  </xs:element>
...
```

Listing 51: Representational schema 1.

```
...
  <xs:element name="supplier"
          minOccurs="1" maxOccurs="2">
    ...
  </xs:element>
...
```

Listings 52 and 53 show the same example as above, except now the physical timestamps are located at the level of the `<supplier>` element. In this case, the transformation pushes the constraints up to the `<supplier_RepItem>` element.

Listing 52: Snapshot schema 2.

```
...
  <xs:element name="supplier"
          minOccurs="1" maxOccurs="2">
    ...
  </xs:element>
...
```

Listing 53: Representational schema 2.

```
...
  <xs:element name="supplier_RepItem"
          minOccurs="1" maxOccurs="2">
    ...
    <xs:element name="supplier_Version">
      ...
      <xs:element name="supplier">
      ...
      </xs:element>
    </xs:element>
  </xs:element>
...
```

Datatype Constraints We claim that datatype definitions of both elements and attributes *can* be enforced in the representational schema. This can be achieved by copying the datatype definition for each element in the original schema into the representational schema. Since datatype restrictions are not affected by the location of timestamps, the transformation is trivial in all cases. See Listings 54 and 55 for an example of the datatype constraint: *"The element* `<age>` *must have a value between 0 and 100, inclusive, at all times."* No changes to the constraint must be made in the transformation.

Listing 54: Datatype snapshot schema.

```
...
  <xs:element name="age">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0"/>
        <xs:maxInclusive value="100"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
...
```

Listing 55: Datatype rep. schema.

```
...
  <xs:element name="age">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0"/>
        <xs:maxInclusive value="100"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
...
```

5.1.2 Non-sequenced Constraints

Non-sequenced constraints are constraints applied to a temporal element as a whole (including the lifetime of the data entity) rather than individual time slices. Non-sequenced constraints are not defined on snapshot XML Schema equivilants. An example of a non-sequenced (cardinality) constraint is: “There should be no more than 10 URLs for each supplier *in any year*.”

We claim that in general it is *not* possible to enforce non-sequenced constraints within a representational schema. Since non-sequenced constraints can reference arbitrary sections of time that don’t necessarily correspond to slice lifetimes or schema change (*schema wall*) boundaries, it is impossible to use XML Schema to isolate and thus validate these sections. For example, consider the simple non-sequenced cardinality constraint: *“There should be two or three unique suppliers in any given year.”* If the document were changed at intervals that were less than one year in duration, we could have a representation that looked similar to Listing 56.

Listing 56: Squashed version. One day equals one unit of time.

```
...
<suppliers_RepItem>
  <suppliers_Version begin="1" end="1">
    <supplier id="1">IBM</supplier>
    <supplier id="2">HP</supplier>
  </suppliers_Version>
  <suppliers_Version begin="2" end="100">
    <supplier id="1">IBM</supplier>
    <supplier id="3">Sun</supplier>
  </suppliers_Version>
    <suppliers_Version begin="100" end="600">
    <supplier id="3">Sun</supplier>
    <supplier id="4">Apple</supplier>
  </suppliers_Version>
</suppliers_RepItem>
  ...
```

It is easy to see that there are in fact four suppliers between the times 1 and 365, violating our example contraint. However, there is no way to construct an XML Schema to successfully validate this, since we would need some way to accumulate the number of unique `<supplier>`s across `<supplier_Version>` and then check this number against the constraint; but there is no such way to perform this accumulation in XML Schema.

However, we do note that there exist specific circumstances in which non-sequenced constraints *may* be validated. Again consider the non-sequenced cardinality constraint: *“There should be 2 or 3 unique suppliers in any given year.”* Also suppose that the timestamps were placed at some element above the `<supplier>` element and that slices were created exactly once per year.

The result will be a representation that closely mimics the individual slices. We see that it is possible to create a reprsentational schema to enforce this constraint (Listings 57 and 58).

Listing 57: Item-based temporal representation #1.

```
...
  <company_RepItem>
    <company_Version begin="1" end="2">
      <company>
        <suppliers>
          <supplier id="123"/>
          <supplier id="456"/>
        </suppliers>
       </company>
    </company_Version>
     ...
...
```

Listing 58: Non-sequenced representational schema #1.

```
...
  <xs:element name="company_RepItem"
    ...
    <xs:element name="company_Version">
      ...
      <xs:element name="supplier">
          minOccurs="2" maxOccurs="3">
      ...
      </xs:element>
    </xs:element>
  </xs:element>
...
```

In this case, we are guaranteed to have one `<suppliers>` element per year. Thus, validating each element in each company version will validate the constraint.

As another example, consider the non-sequenced cardinality constraint: *"There should be between 2 and 4 players on the team in any given year."* If the slices happen to have a one-to-one correspondence with the boundaries for a year, and the timestamp happens to be at or above the `<team>` element, then we could have the following representational schema.

Listing 59: Item-based temporal representation #2.

```
...
<team_RepItem>
  <team_Version begin="1" end="1">
    <team>
      <player>Steve</player>
      <player>Bob</player>
      <player>Mark</player>
      <player>Paul</player>
    </team>
  </team_Version>
  <team_Version begin="2" end="2">
    <team>
      <player>Steve</player>
    </team>
  </team_Version>
</team_RepItem>
  ...
```

Listing 60: Non-sequenced representational schema #2.

```
...
  <xs:element name="player"
          minOccurs="2" maxOccurs="4">
    ...
  </xs:element>
...
```

In general, we see that such special cases can be constructed when both of the following conditions are met.

- Placing the physical timestamp at or above the highest element that is involved in the constraint.
- Versioning the conventional document so that the lifetime of each slice matches the time unit specified by the constraint (e.g., if the constraint involves one year, then there would be exactly one slice per year).

Clearly, these situations are of limited practical use since they are constricting and unlikely to occur naturally. Nevertheless, one might argue that the tools could simply adopt the following strategy. "If a special case occurs, place the functionality in the representational schema; otherwise, place the functionality in the tools." We argue that this process would add complexity that is not justified by the marginal performance gains, especially when there are multiple constraints defined and only some would meet the special-case criteria.

5.2 Functionality of Other Representation Classes

In the above sections we considered whether constraints could be expressed in an XML schema using the item-based representation class. (Section 6 introduces and describes the four kinds of representation classes.) We now provide a brief commentary on the ability of each of the remaining three representation classes to express constraints. Briefly, the remaining three representation classes provide the same or worse level of capability as the item-based class.

The slice-based class allows the same set of constraints to be expressed as the item-based class. This is because the slice-based class is a special case of the item-based class; it possesses no unique characteristics and thus the same limitations apply. The reference-based class also allows the same set to be expressed. This can be seen by viewing the reference-based class as an optimized, but similar version of the item-based class. The reference-based class has the same structure as the item-based class (e.g., items, versions, physical timestamps); the only difference is that the reference-based class avoids data duplication by providing multiple references to subtrees that occur more than once. This process does not gain the reference-based class any benefits that can be used to enforce constraints. The edit-based class is not able to express any temporal constraints within the representational schema since it reduces changes to the XML tree to simple text content that cannot reliably be parsed and examined.

5.3 Placement of Functionality

For those constraints that are possible to implement within either the representational schema or the tools (i.e., sequenced cardinality and datatype constraints), the question remains: where should the functionality be placed? To address this question, we provide a discussion below.

Consider the model of validation used by τXMLLINT shown in Figure 6. First, the temporal document is validated against the representational schema using a conventional validator (i.e., XMLLINT). Then the *Temporal Constraint Validator* is invoked to explicitly and exhaustively check all temporal constraints. This module uses DOM to parse and traverse each slice and manually checks each constraint present in the logical annotation set. From the description of these steps we draw two simple observations. First, the conventional validator is always invoked on the temporal document, no matter which constraints are being implemented in the representational schema. Second, temporal constraints which are "hard" to implement are done so using DOM. Thus, since the conventional validator is empirically much faster than DOM, and is being invoked anyway, we argue that all constraints, when possible, should be implemented within the representational schema. This will provide much better performance in terms of time required, and as we have shown in the previous sections, will not greatly increase the complexity of the representational schema. Furthermore, SCHEMAMAPPER will not require extensive modifications in order to create a schema that can enforce these constraints, since the transformation is trivial in most cases and relatively simple in the rest.

For these reasons, we conclude that the functionality of sequenced cardinality and datatype constraints be placed within the representational schema and not within the temporal constraint validator.

6 Representation Classes

In this section we present the design space for temporal representations in XML. We first introduce and describe some aspects relating to schema versioning. We then characterize the general design space. Then we elaborate on the edit-based, item-based, and slice-based approaches.

6.1 Schema Versioning Considerations

In the following sections our focus is on the different approaches to representig temporal data. However, central to each approach is the method of handling schema versioning. This section briefly summarizes some concepts that are present in each method.

A key idea that first appeared in a paper on temporal aggregation [35] is that of *schema-constant periods (SCP)*. It is possible, even with versioned schemas having themselves versioned schemas, to identify contiguous periods of time when there are no schema changes, anywhere. These are termed as schema-constant periods. These periods are non-overlapping and continuous; between the periods are schema change *walls*. Now, during these periods the data may be (and probably is) versioned, but at least we have a fixed base schema and fixed 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 outlined by Joshi [19], all we have to do is validate across schema changes.

For each of the representation classes below, we will show how schema change walls are handled. However, for simplicity, most examples will contain a single schema constant period without loss of generality.

6.2 Design Space

Researchers have proposed and evaluated many different temporal representations on an individual basis [3, 5, 15, 19, 22, 29]. We have found that all extant representations can be categorized into one of four categories depending on the decisions made to the following two considerations. The first consideration is whether the resulting representation will keep the entire content of a slice explicitly or use some sort of compression which requires slices to be reconstructed but eliminates data duplication. We call this decision *Direct* or *Indirect*. The second consideration is whether the resulting representation will explicitly capture the changes to the XML tree or will only capture changes to the file itself without any knowledge of XML. We call this decision *Itemed* or *Flat*. As these considerations are orthogonal, they induce four possible classes of representations, as shown in Table 2. The Flat classes do not consider XML structure and instead treat the file as a flat text file; the Itemed classes use XML structure within the representation. The Direct classes maintain full versions of each slice in the representation which may result in data duplication; the Indirect classes use some sort of compression which requires slices to be reconstructed but alleviates all data duplication. We now name and briefly describe each class in turn.

The *slice-based* representation class maintains the full text content of each slice throughout the entire history of a document and does not use knowledge of XML structure. Each new slice is simply appended to the representation with the appropriate timestamp. In this way, a full history of slices is maintained in a single representation. Two advantages of the slice-based scheme are its simplicity and its ease of implementation: no processing or logic is needed to add a new

	Flat	**Itemed**
Direct	*Slice-based*	*Item-based*
Indirect	*Edit-based*	*Reference-based*

Table 2: The design space of temporal representations and the resulting classes.

slice. Another advantage is the ease of reconstructing an arbitrary slice: one must simply find the requested timestamp and select the corresponding slice. The major disadvantage of this scheme is the size of the resulting representation: it will grow linearly with the size and number of slices and will thus require both a larger amount of disk space and a large amount of memory for processing. We elaborate on this in Section 6.3.

The *edit-based* representation class maintains the full text content of only the most recent slice, storing reverse edit scripts for each additional slice. It does not use knowledge of XML structure; instead it uses the well-known `diff` tool to compute the text differences between two slices. This scheme can often result in an extremely compact representation but requires extra processing to reconstruct and validate past slices. More detail is given in Section 6.4.

The *item-based* representation class creates and maintains an item for each time-varying XML element. It keeps each slice intact and uses knowledge of XML structure. An *item* is a collection of versions that in concert represent the same real-world entity. It is a logical entity that evolves over time through various slices. This scheme is compact and allows for fast validation. More detail is given in Section 6.5.

The *reference-based* representation class is similar to the item-based representation in that it uses the concept of items. However, the reference-based scheme depends on key identifiers (via XML Schema `<key>` elements) in each element node. Here, every item is present as a child of a top level element, and then each slice representation makes references to one or more of these items. Thus, this scheme factors out common data items to avoid duplication, but as a result the slices do not remain fully intact. This scheme provides similar size performance to the item-based scheme on average and performs similarly in validation time.

Listings 61–66 show two slices of an XML document and the resulting representation in each of the four classes of representations. In each case, SQUASH is used to transform the two slices into the single representation, and UNSQUASH can be used to recreate the original two slices from each of the four representations. Table 2 summarizes the design space.

Listing 61: Slice on 2008-01-01.

```
<a>
  <b>foo</b>
  <c>bar</c>
</a>
```

Listing 62: Slice on 2008-03-17.

```
<a>
  <b>foo</b>
  <c>bar2</c>
</a>
```

Listing 63: Slice-based representation.

```
<tv_root>
  <timestamp begin="2008-01-01">
    <a>
      <b>foo</b>
      <c>bar</c>
    </a>
  </timestamp>
  <timestamp begin="2008-03-17">
    <a>
      <b>foo</b>
      <c>bar2</c>
    </a>
  </timestamp>
</tv_root>
```

Listing 64: Edit-based representation.

```
<tv_root>
  <timestamp begin="2008-03-17">
    <a>
      <b>foo</b>
      <c>bar2</c>
    </a>
  </timestamp>

  <timestamp begin="2008-01-11" >
    <change lines="3">
      &lt;c&gt;bar&lt;/c&gt;
    </change>
  </timestamp>
</tv_root>
```

Listing 65: Item-based representation.

```
<tv_root>
  <a>
    <b>foo</b>
    <c_Item>
      <c_Version begin="2008-01-01">
        <c>bar</c>
      </c_Version>

      <c_Version begin="2008-03-17">
        <c>bar2</c>
      </c_Version>
    </c_Item>
  </a>
</tv_root>
```

Listing 66: Reference-based representation.

```
<tv_root>
  <a>
    <b>foo</b>
    <c_Item itemRef="1"/>
  </a>
  <c_Item itemID="1"/>
    <c_Version begin="2008-01-01">
      <c>bar</c>
    </c_Version>
    <c_Version begin="2008-03-17">
      <c>bar2</c>
    </c_Version>
  </c_Item>
</tv_root>
```

6.3 Slice-Based Representation

As mentioned previously, the *slice-based* representation class maintains the full text content of each slice throughout the entire history of a document and does not use knowledge of XML structure. Each new slice is simply appended to the representation with the appropriate timestamp. In this way, a full history of slices is maintained in a single representation, albeit with obvious data duplication.

Since any data change will result in a new copy of the entire XML tree, and any schema change will result in a new wall, we can expect the size of the representation to grow linearly with the number of slices. In some real-world scenarios where each slice is on the order of kilobytes and the number of slices is measured in the thousands, this size growth can become problematic for both disks (for storage) and memory (for parsing and processing). However, this approach is extremely simple and so it is often a researcher's initial idea.

This approach can be thought of as a special case of the item-based scheme with the physical timestamp placed at the root. With this strategy, the slice-based scheme can be trivially implemented into the τXSchema tools.

6.4 Edit-Based Representation

The *edit-based* scheme (also called *diff-based* or *delta-based*) is proposed and described in several research papers [3, 5, 23]. Briefly, this representation maintains the most recent version of the document and then only the edits necessary to transform each slice into the previous (see Listing 64).

The edit-based scheme has the potential of minimizing the representation size in some cases, and on average, will result in significantly smaller representation than the slice-based approach. Another advantage of the edit-based scheme is the relative simplicity of the construction of the representation.

However, the edit-based approach suffers from high processing overhead to reconstruct early versions, since the edit script has to be applied for every slice in between. It is also difficult to make time-traveling queries [14] since either the entire version history must first be reconstructed, or a complex analysis of the edit scripts must be performed. Also, it is important to note that the edit scripts are saved as text only and not XML trees; as a result, it is difficult or impossible to validate temporal constraints directly on the edit scripts.

We have implemented this representation in the τXSchema tools with the following approach. First, we take the most recent slice in its entirety and place it as a subelement of a `<timestamp>` element. Then, for each successive slice, we run `diff -e` to compute the difference. `diff` is a standard command line tool that computes the difference between two files. The `-e` option formats the output into an `ed` script. We encode the output of `diff -e` as follows.

Listing 67: `diff` output.

```
6c
    <quantity>2</quantity>
.
```

Listing 68: Edit-based encoding.

```
<change line="6">
    &lt;quantity&gt;2&lt;/quantity&gt;
</change>
```

We create a `<change>`, `<add>`, or `<delete>` element depending on the operation specified by `diff` (line 1 of Listing 67), along with the `lines` attribute. The text content of the new element is set to be the output of `diff` with special characters (e.g., angle brackets) encoded to retain valid XML syntax (i.e., "<" is replaced with"`&lt;`"). To recreate an arbitrary slice, we iteratively apply the `patch` tool on the `diff` output.

One important but easily overlooked detail with this approach is the issue of white space. Since white space is an important and necessary characteristic of `diff` output, it should be captured and maintained in the resulting representation. However, the default behavior of DOM and other parsers is to ignore whitespace in elements that don't explicitly set the `xml:space` attribute to `preserved` [43], which could cause problems during the `patch`ing stage. For example, consider the simple document shown in Listing 69. After being parsed by DOM and then written to disk, the resulting file could look something similar to the document shown in Listing 70.

Listing 69: Original document.

```
<a>
  <b>foo</b>
</a>
```

Listing 70: Parsed and output by DOM.

```
<a><b>foo</b>
</a>
```

Since the edit-based representation outputs the latest slice in its entirety and then uses this as the starting point for the reverse edit scripts, it is crucial that newlines be preserved. To achieve this, we introduce a filter that is applied to the conventional documents before they are parsed by DOM; the filter encodes each newline in the original document with a `<tXnl>` element (meaning "τXSchema new line"). See Listing 71 and 72 for an example filter output.

Listing 71: Original document.

```
<a>
  <b>foo</b>
</a>
```

Listing 72: After filter and DOM mangling.

```
<a> <tXnl/><b>foo</b><tXnl/>
</a>
```

We then apply the reverse filter after writing the latest slice to disk (that is, the filter removes all newlines and then replaces all `<tXnl>` with newlines) to ensure that the newly written file matches the newline structure of the original, and thus the reverse edit scripts will work correctly.

To implement temporal validation of this representation, we use the following approach. For each schema-constant period, we UNSQUASH the temporal document into individual slices and then SQUASH them into a slice-based representation. We can then use the tools to validate the new slice-based representation. This allows us to use the same (unmodified) tools to validate all three representation classes, which promotes software reuse and reduces complexity. We argue that this approach is necessary in order to validate some temporal constraints, since it is impossible to do so using the edit scripts alone. However, there is some performance degradation due to the extra steps involved. This tradeoff is quantified in Section 8.

Note that the above approach must treat each schema-constant period separately, as opposed to the entire representation, due to the nature of the edit-based scheme. Section 6.4.2 below elaborates on this idea.

Through this implementation strategy, we are able to fully capture, reproduce, and validate the changes between slices with the help of commonly available tools and create a practically useful edit-based representation.

6.4.1 Capturing Namespaces

Since the edit-based approach does not represent data changes in an XML format (rather, these changes are captured in encoded text within an XML element), the issue of capturing namespaces does not arise. The output of `diff` does not indicate what kind of change occurred on a line (e.g., namespace change versus element change), only what lines changed and the new values of those lines. Thus, when namespace changes occur, we cannot detect them explicitly and so we handle them in the same manner as any other schema change.

6.4.2 Schema Versioning

Changes to the schema are handled with the following approach. For each schema change, we represent a wall by inserting a new `<schemaVersion`X`>` element. Each `<schemaVersionX>` is populated with a set of one or more `<tv:timestamp>` sub-elements: the first such sub-element contains the entire version of the most recent slice in the current schema-constant period, and each additional sub-element contains the edit script produced by `diff`. Listing 73 shows an example edit-based representation with schema versioning.

Listing 73: Edit-based representation with schema versioning.

```
<sv_root>
<schemaItem>
  <schemaVersion0>
    <tv:timestamp begin="2008-03-17">
      <rep0:person>
          ...
      </rep0:person>
    </tv:timestamp>
    <tv:timestamp begin="2008-01-11" >
      <change line="3">...</change>
    </tv:timestamp>
    <tv:timestamp begin="2008-01-07" >
      <change line="8">...</change>
    </tv:timestamp>
    ...
  </schemaVersion0>
  <schemaVersion1>
    <tv:timestamp begin="2008-04-23">
      <rep0:person>
           ...
      </rep0:person>
    </tv:timestamp>
    <tv:timestamp begin="2008-04-10" >
      <change line="3">...</change>
    </tv:timestamp>
    ...
  </schemaVersion1>
</sv_root>
```

The design of this representation is such that each SCP is exactly similar in syntax and format to the schema-constant case. The representational schema must then be constructed so that for each SCP the contents of the base schema are inserted as a sub-element of a timestamp element. Timestamp elements are also permitted to have change, add, or delete sub-elements to encode the edit scripts.

6.5 Item-Based Representation

The *item-based* scheme [19] creates and maintains an item for each time-varying element and was the original representation type implemented in τXSchema. An *item* is a collection of XML elements that in concert represent the same real-world entity. It is a logical entity that evolves over time through various slices. An item can occur at any level in the XML tree hierarchy, and is specified by the user via a physical annotation. Every occurrence of the actual temporal element from the snapshot document is replaced by its corresponding item. The item-based representation has the following features.

- Only elements are time-varying and can have versions. The immediate content (text and attributes), is considered to be an integral part of an element and therefore does not have a separate time-varying lifetime.
- A version of an element is created when any change the immediate content or attributes of an element change. This includes text content, sub elements, comments, and processing instructions. The change must be observable through DOM: only changes observable through DOM create a new version. This is implies that whitespace and attribute ordering does not create a new version. In contrast, since the edit-based representation uses `diff` to observe changes, whitespace and attribute ordering would create a new edit script in the representation.
- If an element is glued but remains unchanged, then the lifetime of the current version of the element is extended; no new version is created. This implies that versions are coalesced [19].
- The timestamp that represents the version's lifetime is a N-dimensional temporal element. It may include now, until changed, and/or indeterminate times.

We now extend the item-based representation to explicitly capture schema versioning and namespaces with elements. These tasks are accomplished with the use of additional elements and namespaces in the resulting representation. Briefly, each element e in the original document with namespace ns will take on the following form in the representation,

`<ns:e>` $\Rightarrow$ `<rep`X_Y_`ns:e>`

where X and Y are unique integers for each schema-constant period and namespace change, respectively. The following sections describe this production in more detail and provide examples.

6.5.1 Capturing Namespaces

A conventional document may have more than one namespace, with each namespace associated with a different schema. Further, these namespaces may change from slice to slice without a schema changing (i.e., the namespace mapping in the conventional document changes, but the conventional schema does not change). To capture and reproduce such situations, each namespace in each slice must be mapped by the tools to a unique namespace in the representation. Consider the two slices shown in Listings 74 and 75.

Listing 74: Slice on 2008-01-01.

```
<a>
  <ns1:b>foo</ns1:b>
  <ns2:c>bar</ns2:c>
</a>
```

Listing 75: Slice on 2008-03-17.

```
<a>
  <ns1:b>foo1</ns1:b>
  <ns2:c>blah</ns2:c>
</a>
```

In this simple example, three different namespaces are used and remain constant between slices (specifically, the default namespace, `ns1` and `ns2`). The resulting representation (Listing 76) has one corresponding namespace for each of the original namespaces. Here, the `rep` namespace corresponds to the default namespace used by `<a>` in the slices.

Listing 76: Item-based representation of Listings 74 and 75.

```
<tv_root>
  <rep:a>
    <!-- rep_ns1 is a newly created namespace -->
    <rep_ns1:b_RepItem>
      <rep_ns1:b_Version begin="2008-01-01">
        <ns1:b>foo</ns1:b>
      </rep_ns1:b_Version>
      <rep_ns1:b_Version begin="2008-03-17">
        <ns1:b>foo1</ns1:b>
      </rep_ns1:b_Version>
    </rep_ns1:b_RepItem>

    <!-- rep_ns2 is a newly created namespace -->
    <rep_ns2:c_RepItem>
      <rep_ns2:c_Version begin="2008-01-01">
        <ns2:c>bar</ns2:c>
      </rep_ns2:c_Version>
      <rep_ns2:c_Version begin="2008-03-17">
        <ns2:c>blah</ns2:c>
      </rep_ns2:c_Version>
    </rep_ns2:c_RepItem>
  </rep:a>
</tv_root>
```

Since no naming conflicts are present in this scenario, the unique integer Y is not necessary and is therefore omitted. Similarly, the unique integer X for the SCP is omitted. To illustrate a scenario where a naming conflict occurs, consider the following two slices.

Listing 77: Slice on 2008-01-01.

```
<a>
  <ns1:b>foo</ns1:b>
  <rep_ns1:c>bar</rep_ns1:c>
</a>
```

Listing 78: Slice on 2008-03-17.

```
<a>
  <ns1:b>foo1</ns1:b>
  <rep_ns1:c>bar</rep_ns1:c>
</a>
```

Since `<ns1:b>` is time-varying, we need to create an item and thus a new namespace for the item. We can not use `rep_ns1` in the resulting representation because that namespace already exists in the original slice and would cause confusion. Instead, we create a new namespace rep_0_ns1, as shown in Listing 79.

Listing 79: Item-based representation of Listings 77 and 78.

```
<tv_root>
  <rep:a>
    <rep_0_ns1:b_RepItem>
      <rep_0_ns1:b_Version begin="2008-01-01">
        <rep_0_ns1:b>foo</rep_0_ns1:b>
      </rep_0_ns1:b_Version>
      <rep_0_ns1:b_Version begin="2008-03-17">
        <rep_0_ns1:b>foo1</rep_0_ns1:b>
      </rep_0_ns1:b_Version>
    </rep_0_ns1:b_RepItem>

    <rep_rep_ns1:c>bar</rep_rep_ns1:c>
  </rep:a>
</tv_root>
```

In this scenario, the value 0 happens to be the first unique integer that relieves the naming conflict. If there already exists a namespace rep_0_ns1 in the original document, then the representation would try rep_1_ns1. If rep_1_ns1 already exists, this process would iterate until no conflicts remain.

6.5.2 Schema Versioning

To achieve schema versioning in the item-based representation class, we introduce one level of abstraction. For each schema-constant period, we create a new representational schema in the normal way and define a unique namespace for this schema of the form <rep$X$_Y> as described above. Then, in the main representational schema, we import the representational schema for each SCP and define a new element <schemaItem>. Sub-elements of this element correspond to each SCP and thus each <rep$X$_Y> namespace. Listings 80 and 81 below show an simple example of schema versioning and Listing 82 shows the representational schema.

Listing 80: Version 1 of a simple schema.

```
...
<element name="athlete">
  <complexType mixed="true">
    <sequence>
      <element name="athName"
               type="string"/>
    </sequence>
    <attribute name="athID"/>
    <attribute name="age" />
  </complexType>
</element>
...
```

Listing 81: Version 2 of a simple schema.

```
...
<element name="athlete">
  <complexType mixed="true">
    <sequence>
      <element name="athName"
               type="string"/>
    </sequence>
    <attribute name="athNumber"/>
    <attribute name="age" />
  </complexType>
</element>
...
```

Note that the files rep0.xsd and rep1.xsd correspond to the representational schemas created for the two SCPs. Each defines the namespace and elements corresponding to its SCP.

The temporal document then has one <schemaVersion$X$> element for each SCP and the representation proceeds in the normal way. Listing 83 shows the temporal document created in the scenario.

Listing 82: Representational schema.

```xml
<?xml version="1.0"?>
<xsd:schema targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema"
           xmlns="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/TVchema"
           xmlns:rep0="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema0"
           xmlns:rep1="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema1"
           elementFormDefault="qualified">

<xsd:import namespace="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema"
            schemaLocation="TVSchema.xsd">

<xsd:import namespace="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema0"
            schemaLocation="rep0.xsd">

<xsd:import namespace="http://www.cs.arizona.edu/tau/tauXSchema/RepSchema1"
            schemaLocation="rep1.xsd">

<xsd:element name="tv_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:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 83: Temporal document.

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

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

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

                ...
                </rep1:athlete_RepItem>
            </rep1:tv_root>
        </schemaVersion1>
    </schemaItem>
</tv_root>
```

7 Implementation

In this section we present the modifications and enhancements to the tools that are required to support different representation types. We also discuss the methodology used by τXMLLINT and its relationship to the representation used. We first provide a brief overview of the tools and their implementation and conclude with a list of general enhancements made to the project.

7.1 Overview

The tools are open-source and beta versions are available [36] and the full implementation and architecture is described by Joshi [19]. Figure 7 shows the overall architecture of the tools as a UML class diagram [26]. The architecture consists of three packages: `tau.xml` for the interface of each tool; `tau.time` for classes that handle time; and `tau.util` for utility classes common to all tools and classes.

The tools have been implemented in Java using the DOM API [41]. The DOM API was chosen over SAX API due to its ability to create an object-oriented hierarchical representation of the XML document in main memory which can be navigated and manipulated at run-time. This capability has proven to be extremely useful in all of the tools.

Figure 5 shows the overall architecture of the tools as they manage XML documents and their schemas. A sequence of non-temporal documents is input into SQUASH to create a temporal representation; this document can then be validated using τXMLLINT and SCHEMAMAPPER. UNSQUASH can be used to reconstruct the original non-temporal documents from the temporal representation, while RESQUASH can be used to create a new representation (e.g., different timestamp locations) from a given representation.

7.2 τXMLLINT

Figure 6 provides the validation procedure used by τXMLLINT. The first step is to pass the temporal schema into τXMLLINT, which ensures that the logical and physical annotations are consistent with the snapshot schema and with each other. Once the annotations are found to be consistent, SCHEMAMAPPER is invoked to generate a representational schema from the original snapshot schema and the temporal and physical annotations. The representational schema is then used as the schema for the temporal document and input into a conventional validator (in this case, XMLLINT). The next step is to pass the temporal document and the temporal schema to *Temporal Constraint Validator*. This step is to enforce temporal constraints that are not possible to be enforced by the representational schema alone.

The algorithm for τXMLLINT is given in Figure 8. τXMLLINT is able to check for the following types of temporal constraints.

Content Constant Content of an element cannot vary over time.

Cardinality Constant Cardinality of an element cannot vary over time.

Existence Constant The element cannot disappear and reappear again.

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

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

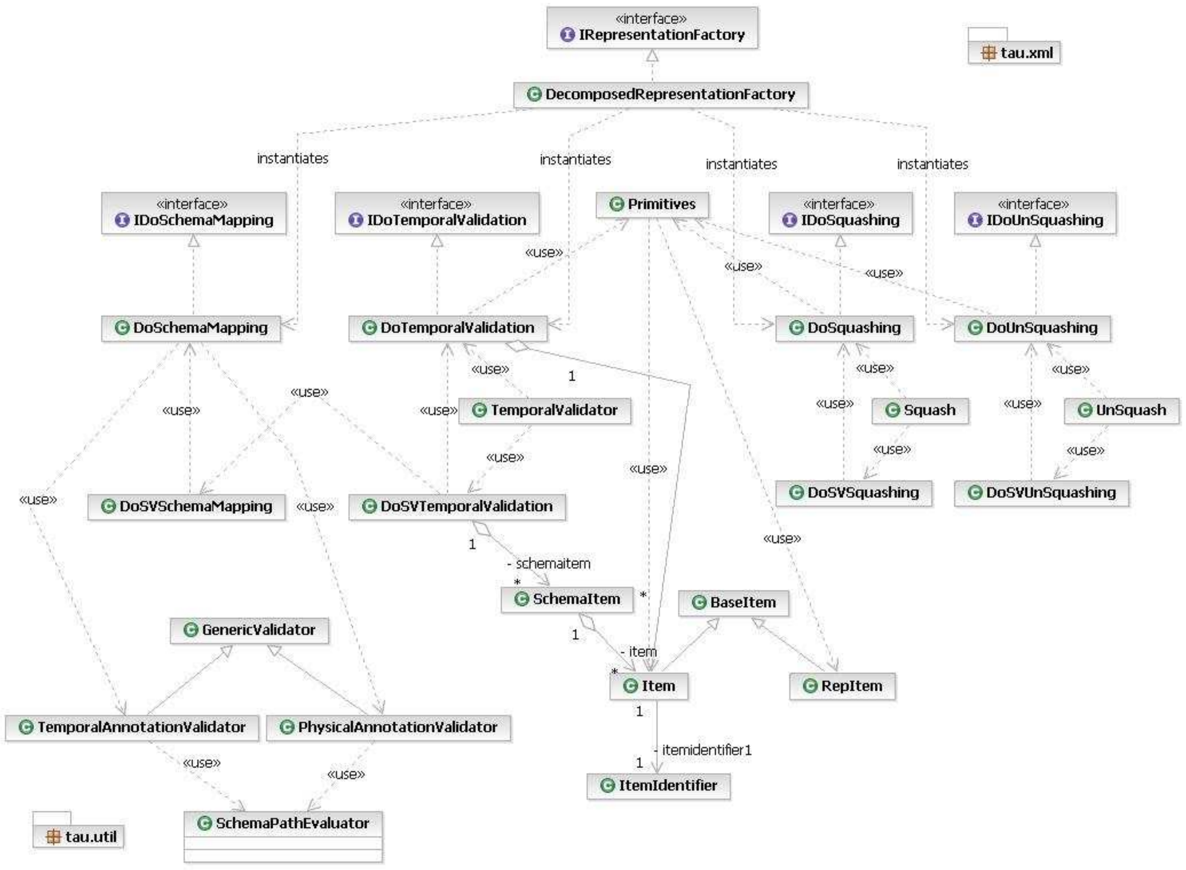


Figure 7: The overall class diagram for the tools of τXSchema.

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

τXMLLINT enforces each temporal constraint using a simple brute-force approach. For example, for an item that has a content-constant constraint, the *Temporal Constraint Validator* loops through each version of the item and determines whether the nodes are DOM-equivalent.

7.3 Tool Modifications and Extensions

In order to implement different classes of representations (as described in Section 6), the tools were reorganized to abstract the details of the representation so that these details may vary freely without affecting the rest of the code. This was achieved by introducing abstract factory methods [13] in each of the tools (SQUASH, UNSQUASH, SCHEMAMAPPER, τXMLLINT) in place of the original methods; each abstract factory method would then call the appropriate concrete method based on the type of representation specified by the user. Figures 9 and 10 show the placement of the abstract factory method.

The changes to allow the edit-based representation to be used within the tools are described in detail in Section 6.4, and the changes to the item-based representation are described in Section 6.5.

Figure 8: Algorithm: τXMLLINT

```
//Inputs
// snapshotSchema - Parsed snapshot schema document
// logicalAnnotation - Parsed logical 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
```

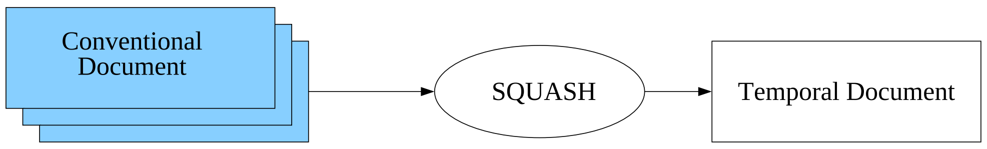


Figure 9: SQUASH before abstract factory methods were added.

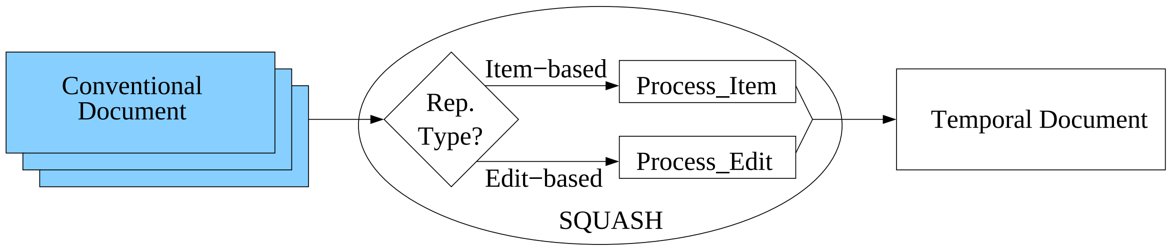


Figure 10: SQUASH after abstract factory methods were added.

7.3.1 Schema Versioning

To implement schema versioning, the representational schema had to be generalized so that each schema-constant period corresponds to a new namespace. Each namespace begins with the root element of the slice and describes all changes within the SCP in the same way as described in previous sections.

In particular, the tools are constructed in a such a way that schema versioning is handled by a different Java class than schema-constant cases. This allows flexibility in the way that schema-versioning is implemented and provides an abstraction to the developer. Both the schema-versioning and schema-constant classes implement the same interface so that the other parts of the tools do not need to be aware of the fact that schemas are changing. The schema-versioning classes make use of the functionality within schema-constant classes for each SCP; this again reduces code duplication and promotes software reuse.

Figure 11 shows the method for validating a temporal document with a time-varying schema. (This method was also presented in Section 5 as Figure 6.) To validate such a document, τXMLLINT applies the conventional validator to the document, using the representational schema produced by SCHEMAMAPPER. It then determines the times when the schema changes, thus determining schema-constant periods. For each such period, the time-varying data checker is invoked to check the temporal integrity constraints over the time-varying data, with the single base schema, temporal annotation, and physical annotation. Then the *temporal constraint checker* glues across the schema change walls and performs the temporal checks across these walls.

The framework for cross-wall validation is described in detail by Joshi [19] and our implementation closely follows his design. Briefly, the tools must consider the following issues that arise with schema versioning.

Accommodating evolving keys. When a schema-change wall is encountered, items across the wall need to be associated. This process is called *cross-wall* gluing or *bridging*. This becomes especially tricky if either the snapshot key (specified in the base schema) upon which an item is defined, or if the item identifier itself (specified in the logical annotation) changes. The solution

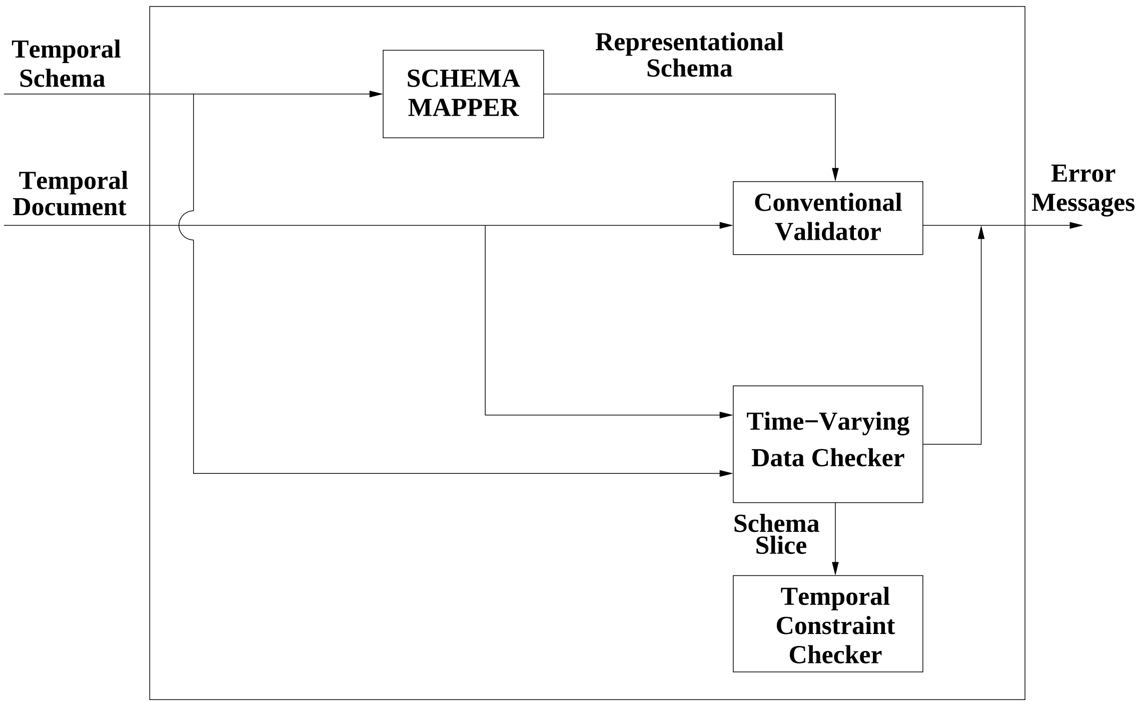


Figure 11: Validating a document with Time-Varying Data and a Time-Varying Schema.

is to use an `<itemIdentifierCorrespondence>` element to determine the type of mapping desired, 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.

Accommodating gaps. When gaps appear in the lifetime of an item, the process of finding the correspondence between the items from corresponding SCPs becomes more difficult. To handle the case of gaps, it has been decided to create a new item when the element reappears; that is, the first item is not virtually extended across a schema change wall, since it is difficult or impossible to know a priori if an item that reappears is the same item or a different item. Thus, when gaps appear, the tools create a new item and both the original item and new item are treated like every other item in the evaluation.

Semantics for mixed data and schema changes. When a data change co-exists with a schema change within a single transaction, τXSchema places no restrictions on the types of changes allowed; this is the most general and flexible design. However, given two schemas, it becomes very difficult to find the differences between them and validate the versions. Thus, versions of an item across schema change walls are not validated if a schema change is detected for it. This decision leads to no additional work needed from the tools.

Non-sequenced constraints. Here we consider temporal constraints such as *"An item can only be changed 3 times per year."* Note that it has been decided to consider non-sequenced constraints only within a SCP since the user might introduce unintended complexity when specifying new constraints in the middle of the year: should the constraint be checked only from that point forward or from the beginning of the document's lifetime? This means that the tools must verify each constraint within each SCP separately, which is relatively simple. For example, for the above temporal constraint, the tools must simply check the number of versions of an item and make sure that it is less than or equal to the specified amount. Note that only the theoretical design exists for some non-sequenced constraints; full implementation is left for future work.

We note that the complete set of features and functionality described above is not yet implemented by the tools. As we describe in Section 9.2, more work is needed to complete the implementation.

7.3.2 Project Support

We also enhanced the overall organization and environment of the tools, documentation, examples and test cases, and build scripts to more closely match standard industry practices. These enhancements were performed in the Eclipse Platform version 3.3.2 to allow full integration of CVS, the use of the tools' Ant build files, and a robust debugging and testing environment.

- A standard command-line interface was given to each of the tools. This interface takes the form `tool_name [options] inputfile ...` where `[options]` include options common to all tools (e.g., level of debug output) as well as options specific to each tool (e.g., output file names).
- A robust and powerful logging capability was added using Apache's LOG4J [1]. This allows the user to specify dynamically the level of debug output provided by the tools. Specific debug output was then added at various places in the system (e.g., τXMLLINT now says why a temporal document would fail validation, as opposed to just silently failing). The amount and type of debug output is specified at invocation via a command line option. For example, specifying `-debug 0` results in no debug output being displayed, while `-debug 4` provides exhaustive debug output, such as what methods are being called, values of specific variables, and which slice is currently being parsed.
- The project directory structure and organization were improved. They are now more consistent with other popular open-source projects.
- Version control for the project was implemented with CVS [28]. CVS allows multiple developers to easily work on the source code at the same time and logically merge their changes into a central repository.
- A project wiki page (`http://cgi.cs.arizona.edu/apps/tauXSchema`) was created for bug tracking and user/developer collaboration [37]. The wiki was created using PMWIKI 2.1.27. A dynamic list of bugs and issues was also created to aid the software development effort.
- Enhanced maintainability was enabled by extracting common functionality of the tools into a single `Common` class. This class performs activities such as XML tree traversal, checking DOM node types, file I/O, and date formatting. In general, if a piece of code or logic (however trivial) was needed in more than one place, we abstracted it into the Common class.

Together, these changes add a level of usability to the overall system.

8 Evaluation of Representation Classes

This section presents a detailed empirical analysis of each representation class in a variety of scenarios. The goal is to determine how each class performs with respect to four metrics: size of representation, time to construct the representation, time to validate the representation, and time to reconstruct an arbitrary slice. We first describe the motivation for this evaluation and present the methodology used in our experiments. We then analyze the results of the experiments and we conclude with general observations and recommendations.

8.1 Motivation

When choosing a representation for a temporal XML document, we consider several characteristics in the decision making process. Consider the following examples.

- A user wants to transmit a document across the country on the internet. Here, the *size of the representation* is the most important feature.
- A user makes frequent updates to a file, resulting in frequent creations of the representation. Here, the *time taken to create the representation* will be the most important feature.
- A user wants to frequently select different versions of a document, an operation called temporal slicing. Here, the *time taken to extract the original documents* is the most important feature.
- A user makes frequent updates to a file and must always validate the document to ensure correctness. Here, both the *time taken to create* and *validate the representation* are the most important features.

It is not clear whether any single representation class can best meet the needs of every user in every scenario. Our aim is to quantify the features of each class so that informed decisions can be made by the user, taking into account their particular needs.

8.2 Methodology

Given the above motivation, we will address the following questions about each representation class. Is the size of the representation linear in the number and size of slices, or does it provide some level of compression? Does the overhead of the representation result in a large amount of time required to squash and unsquash? Can we validate the representation quickly enough to allow practical use?

To answer these questions, we first extended the τXSchema tools to support each representation type[7] and then ran a set of experiments to test and evaluate each representation. In these experiments, we were interested in how the representation would respond to several independent variables: the amount of change from slice to slice, the types of changes within each slice, the number of slices in the system, and the size of each individual slice. To quantify the changes, we measured several dependent variables: the size of the resulting representation, the time taken to create the representation from the original documents, the time taken to extract the original documents from the representation, and the time taken to validate the representation against the original schema(s) and temporal constraints. Tables 3 and 4 summarize the experiment.

We have created tools[8] to help build temporal cases dynamically based on the four experiment parameters and to automate the run process and output the results. The experiments were executed

[7]Due to time and tool constraints, we could not evaluate the reference-based representation. However, it is believed that this representation would perform similarly to the item-based representation under typical circumstances.

Independent Variables		
Name	Expressed as	Values
Slice Size	Number of elements	$10, 20, 40, 80, ..., 2000$
Number of Slices	Number of files	$10, 20, 40, 80, ..., 2000$
Amount of Change	Percentage of changed elements	$0, 2, 4, 8, 16, 32, 64$
Type of Change	Percent value change vs. new element	$(0, 100), (25, 75), (50, 50), (75, 25), (0, 100)$

Table 3: The independent variables considered in the experiments.

Dependent Variables	
Name	Measured by
Representation size	Kilobytes on disk of the representation
Time taken to squash files	Seconds of execution
Time taken to unsquash files	Seconds of execution
Time taken to validate files	Seconds of execution

Table 4: The dependent variables measured in the experiments.

on a machine running Ubuntu 8.10 with a 2.83 GHz Intel Core2 quad-core processor and 8 GB main memory. The testing scripts were created in Perl and the data was analyzed in Matlab.

Note that all of the following experiments were run with the temporal constraint functionality placed at the tool level as opposed to the representational schema level as described in Section 5. Also note that schema-versioning is not considered in the following examples due to the complexity it adds to the creation of large, random scenarios. However, we believe that these results will provide a good initial understanding of the behavior of each representation class since each schema change results in a new wall, with the representation structure remaining the same within each wall. Thus, a schema change will have the same effect across all representation types.

We note here that the execution times presented in the following sections contain both I/O and execution times—the entire execution time of the process. Since caching was not disabled during the experiments, it is possible that some experiments unfairly report smaller execution times; this might happen if an experiment involved reading a file that was already in the O/S cache.

8.3 Initial Sensitivity to Parameters

Initial experiments were run to test the sensitivity of each representation type to the variables that controlled the amount and type of change in each slice. In particular, we performed runs with the amount of change set to 1%, 2%, 4%, ..., 64% and with the type of change set to (0% new version, 100% new item), (25%, 75%), (50%, 50%), (75%, 25%), and (100%, 0%). We also varied the size of each slice (values of 10 and 100) and number of slices (values of 10 and 200). For each combination of parameters, we ran 30 repetitions for each representation class and took the minimum result (since we are interested in the performance of the tools in isolation, but the experiments were conducted on a time-shared machine with background processes running). Interestingly, the results showed that all representation classes were relatively insensitive to the change-related parameters. This is likely because for each representation class, the amount of computation overhead for each additional change per file is small when compared to the execution

[8] See Appendix B for details.

of the entire tool (e.g., file I/O for each slice, DOM parsing of each file, etc). Appendix C provides the detailed results of these executions.

As a consequence of these results, for each of our experiments described in the following sections, we held the amount of change constant at 32% and the type of change constant at (75%, 25%), which represent intermediate values for each.

8.4 SQUASH Results

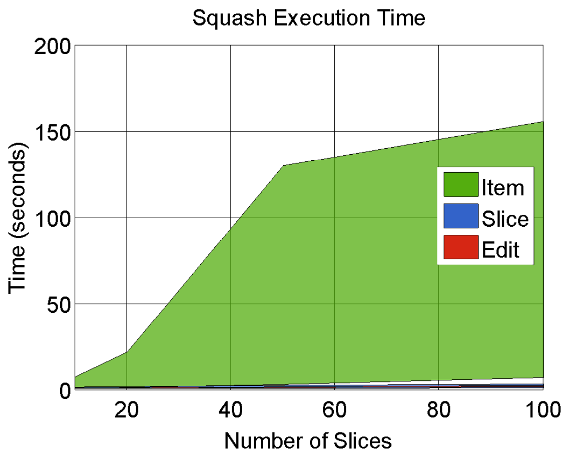
We first consider the amount of time required for each representation class to squash a temporal document. Figure 12 displays the results with all classes on the same plot (with a band stretching across the document size parameter for each class) while Figure 13(a)–13(c) shows each representation class individuallly. We immediately see that the item-based scheme is particularly sensitive to the parameters and even modest increases result in a large increase in time. We now briefly investigate why this happens.

Figures 14(a) and 14(b) show the execution traces of the SQUASH algorithm for the item-based and edit-based representation classes, respectively. Note that in each case, only the steps that required a significant portion of the execution time are shown. Table 5 summarizes the actual exectution time for each step in both classes for three simple scenarios. We immediately see that for the item-based representation, the bulk of the execution time is being spent on task 1.2.2 (physical to temporal conversion), specifically in the push down operation. The push down operation recursively calls itself to "push down" the items from the root node down to the `<part>` elements; this algorithm involves merging similar versions of an item into a single version. In these executions, the push down operation is called a total of 342, 642, and 1282 times for the three input sets, respectively. The edit-based scheme does not have the concept of items, and thus avoids the penalty of the push down operation. Further, the `diff -e` call that the edit-based scheme employs between each slice is relatively inexpensive: on the order of 0.001 seconds to compare two files with 20 elements each, 0.03 seconds to compare files with 1000 elements each, and 0.2 seconds to compare files with 10,000 elements each, and 1.65 seconds to compare files with 100,000 elements each.

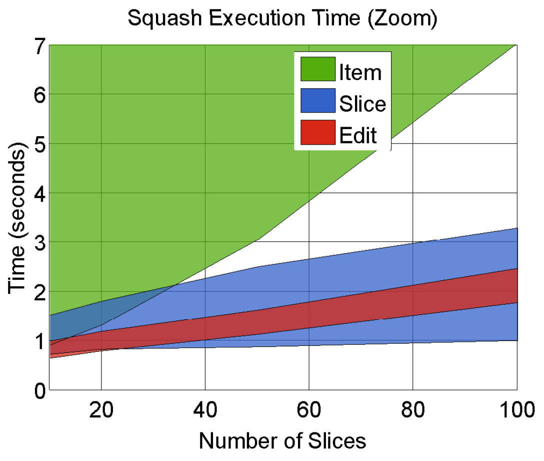
Step #	**Item-based**			**Edit-based**			**Slice-based**		
	(10, 10)	(10, 20)	(20, 20)	(10, 10)	(10, 20)	(20, 20)	(10, 10)	(10, 20)	(20, 20)
1	2.15	3.99	9.59	0.85	0.99	1.09	0.91	1.13	1.70
1.1	0.02	0.01	0.02	0.02	0.02	0.01	0.01	0.01	0.01
1.2	1.63	3.47	9.04	0.33	0.46	0.56	0.39	0.59	1.15
1.2.1	0.06	0.10	0.14	0.28	0.43	0.51	0.07	0.10	1.14
1.2.2	1.43	3.21	8.46	-	-	-	0.19	0.24	0.14
1.2.2.1	0.57	1.06	2.38	-	-	-	0.01	0.02	0.04
1.2.2.2	0.80	2.08	5.67	-	-	-	0.03	0.05	0.05
1.2.2.3	0.04	0.05	0.09	-	-	-	0.14	0.17	0.19
1.2.3	0.10	0.11	0.39	-	-	-	0.08	0.18	0.61

Table 5: The execution times (in seconds) in SQUASH for each task, broken up by representation type and shown for three different input sets. In these runs, the amount of change was set to .32 and the type of change was set to $(75\%, 25\%)$.

In order to further investigate the time required for the slice-based and edit-based schemes under heavier conditions we increased the magnitude of the parameters and the results are shown

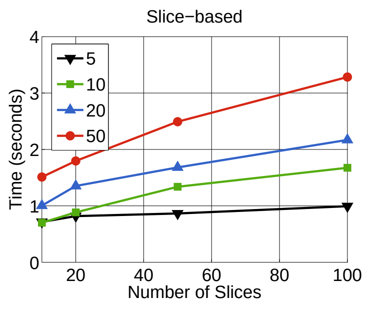


(a) Full view. The item-based class requires orders of magnitude more time to squash than the slice-based and edit-based classes.

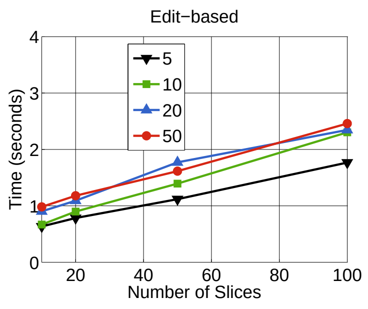


(b) Zoom view of the slice-based and edit-based classes.

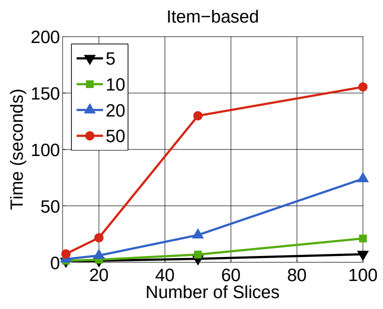
Figure 12: Time required to squash a temporal document. The three band colors correspond to the different representation types. Each band stretches across $\{5, 10, 20, 50\}$ elements per slice.



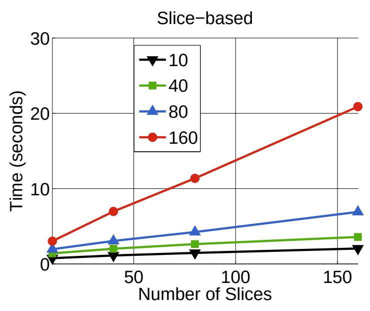
(a) Slice-based



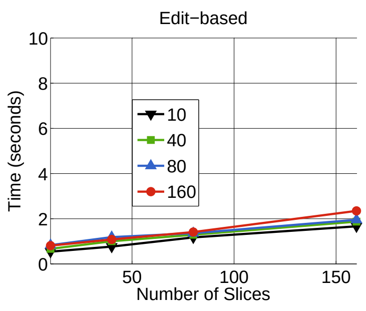
(b) Edit-based



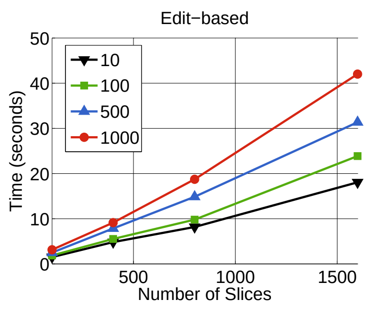
(c) Item-based



(d) Slice-based



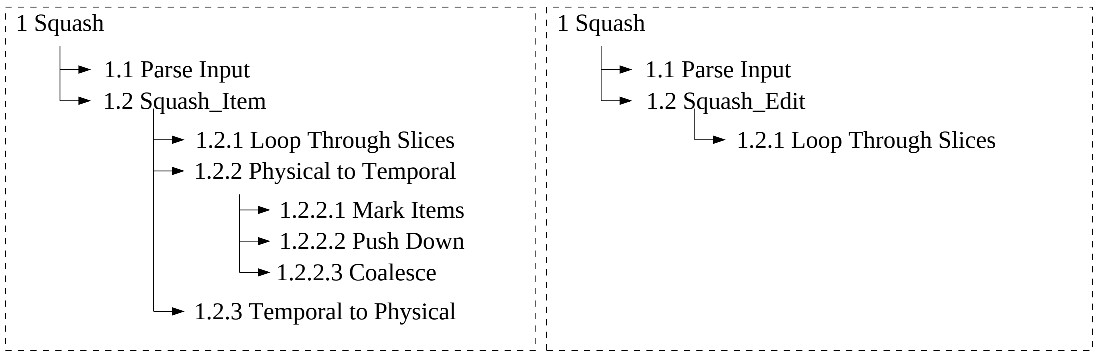
(e) Edit-based



(f) Edit-based (larger)

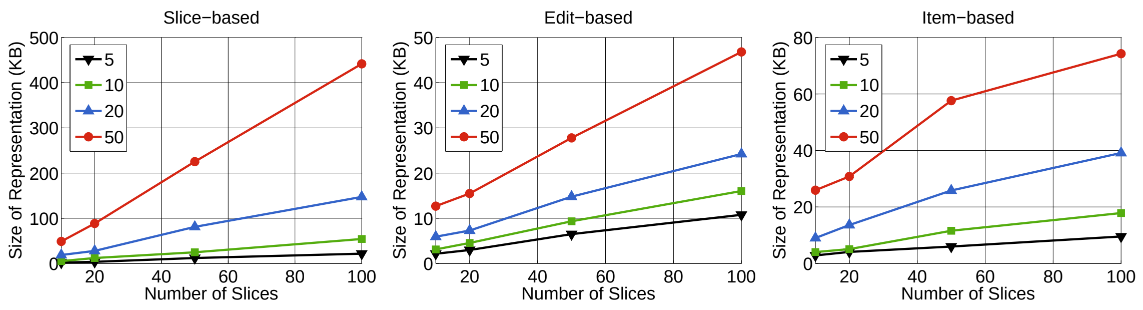
Figure 13: Time required to squash a temporal document. Here, the lines correspond to different document sizes, shown in number of elements.

in Figures 13(d)–13(f). We see again that the edit-based scheme has better performance when compared to the slice-based scheme under the same parameters. Figure 13(f) shows the parameter



(a) Item-based (b) Edit-based

Figure 14: The main methods (in terms of time) entered during the execution of SQUASH.



(a) Slice-based (b) Edit-based (c) Item-based

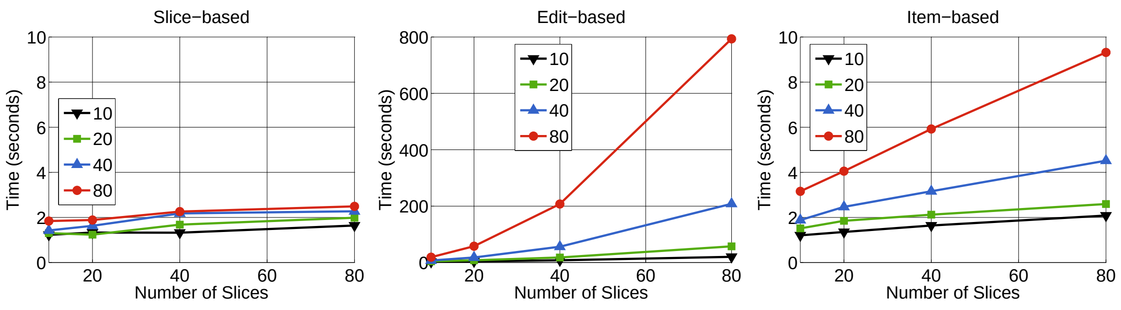
Figure 15: Size of the resulting temporal document. Note the different scales on the y-axis.

set that first starts to stress the performance of the edit-based scheme (i.e., the parameters that first cause the execution time to show larger than linear growth); these parameters are roughly 10x the parameters that stressed the slice-based scheme.

Figures 15(a)–15(c) show the size on disk of the resulting temporal document. As expected, the slice-based scheme grows linearly with the number of slices and the size of the representation; this is because this scheme keeps the entire unmodified slice in the representation. However, the edit- and item-based schemes are able to provide a large amount of compression and keep the file size relatively low.

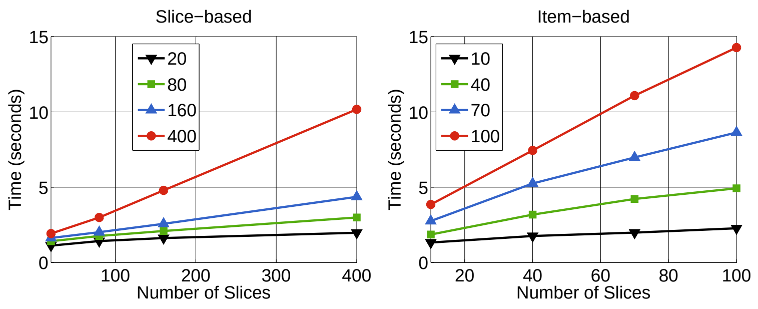
8.5 τXMLLINT Results

Here we considered the validation time required for a single temporal cardinality constraint. Figures 16(a)–16(c) show the amount of time required to validate scenarios with modest number of slices and size for each slice. We see that while the slice-based and item-based schemes can handle these parameter ranges with relative ease, the edit-based scheme takes orders of magnitude more time. This is due to the nature of the current implementation of the edit-based scheme: the temporal document is first unsquashed into a series of slices; then these slices are squashed into an item-based representation with the timestamp at the root; finally, the item-based representation is validated in the normal way. Although this implementation benefits from the reuse of several



(a) Slice-based (b) Edit-based (c) Item-based

Figure 16: Time required to validate the temporal document. Note the different scales on the time axis; the edit-based scheme takes orders of magnitude longer.



(a) Slice-based (b) Item-based

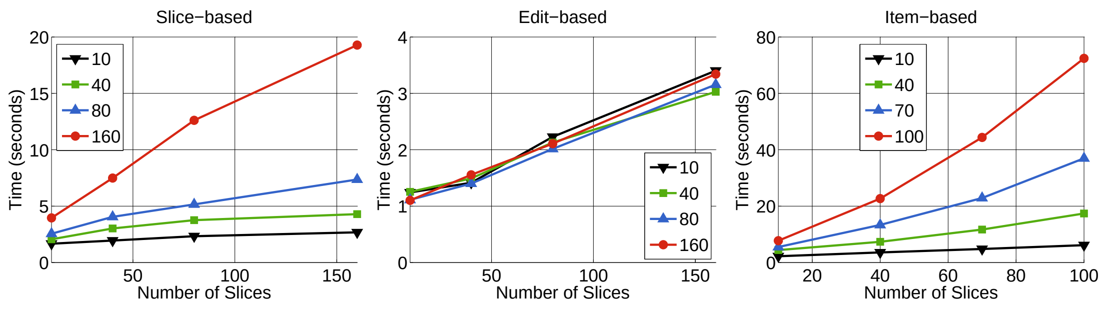
Figure 17: Time required to validate the temporal document. Note the different scales on the x-axis. The slice-based scheme can handle roughly four times the number of slices within the same time period.

existing software modules and is logically correct, it suffers from both high overhead and the bad performance of the item-based squashing module. These factors add up to significant values, with the majority of the time coming again from the push down operation of the item-based scheme (70% of the total execution time).

Figure 17 shows the slice-based and item-based schemes under heavier conditions to illustrate the magnitude of the parameters that first cause a noticable increase in execution time. We see that the slice-based scheme can handle roughly four times the number of slices as the item-based scheme within the same time period. This is because the slice-based scheme requires no pre-processing before validation can begin, while the item-based scheme must undergo a number of operations to be in the correct form for validation.

8.6 UNSQUASH Results

Figure 18 shows the amount of time required to extract all slices from the temporal document. We see that, like the SQUASH results, the edit-based scheme has the best performance, while the item-based scheme requires significantly more time than the other two schemes. An execution analysis similar to that for SQUASH was performed and the same conclusions were reached. In particular, during the unsquash operation, the item-based scheme must perform the opposite of



(a) Slice-based (b) Edit-based (c) Item-based

Figure 18: The amount of time required to extract all slices from a temporal document. Note the different x and y axes.

Representation	SQUASH Time		SQUASH Size		UNSQUASH Time		τXMLLINT Time	
	Rank	Ratio	Rank	Ratio	Rank	Ratio	Rank	Ratio
Slice-based	2	1.1	3	3.9	2	2.6	1	-
Edit-based	1	-	1	-	1	-	3	41.1
Item-based	3	15.7	2	1.5	3	13.0	2	1.6

Table 6: The overall results of the analysis. The *Rank* columns indicate the performance of this representation when compared to the other two (e.g., a rank of 2 means it was the second best). The *Ratio* column indicates how much worse this representation performed compared to the top ranking representation, measured as the average ratio between the two representations.

merging items and pushing down timestamps: it must push up timestamps and unmerge items. These operations result in a huge number of recursive calls and loops for every item. In contrast, the edit-based scheme only needs to run the `patch` command for each edit script. The slice-based scheme must simply loop though the temporal document and extract each slice with a copy-and-paste-like method; no preprocessing is needed. Even still, it does not perform as well as the edit-based scheme.

8.7 Representation Conclusions and Recommendations

The above results show that, as anticipated, no representation scheme provides the best performance under all conditions. In particular, while the edit-based scheme shows the fastest time to squash and unsquash as well as the smallest representation size, it suffers from tremendous overhead during validation[9]. On the other hand, the slice-based scheme can be squashed, unsquashed, and validated quickly but the resulting representation is very large. Further, the item-based scheme results in a smaller representation and can be validated quickly, but suffers from a large amount of time to squash and unsquash. Table 6 summarizes the findings.

Our current recommendation would be to use the edit-based scheme for all activities that do not require temporal validation (although under this assumption, a conventional validator may still

[9]It should be noted that although the current implementation of edit-based validation is not very efficient, it would be difficult to find any implementation that would be. The problem lies in the inability to enforce complex temporal constraints by examining the edit scripts alone. This implies that the first step during validation would always have to be reconstructing the original slices. Only then could the validation process—whatever that may be—begin.

make sense to validate the last instance and the one representation in its entirety), and to use the slice-based scheme in all cases that do require temporal validation. However, if improvements can be made to the item-based squashing and unsquashing methodologies (i.e., the “push up” and “push down” operations) in terms of execution time, then the item-based scheme would become most attractive in all scenarios. In this case, additional analysis would be required to study the effects of using an item-based scheme in its original form versus a hybrid between the item-based and the edit-based. For example, one could imagine storing the representation on disk using the edit-based method, while converting into the item-based scheme for all in-memory operations (e.g, adding or extracting slices, enforcing temporal constraints). Further, one might consider using a mix of representation types for different parts of the timeline. For example, a user could use the slice-based scheme for the five most recent slices, the item-based scheme for the next 300 slices, and the edit-based scheme for all other slices. This might allow efficient extraction of recent files, efficient validation for all files in the recent past, and efficient storage for files not likely to be queried often. The time and space tradeoffs for these options are left for future work.

9 Summary and Future Work

In this work we have explored several aspects relating to the representation of temporal data and their schemas in XML. Our goal was to evaluate disparate representational approaches to provide an initial characterization of the design space. We addressed this goal through language design, qualitative and quantitate analysis, and tool enhancement.

9.1 Summary

First, we have extended the language of τXSchema and enhanced the overall organization. We have introduced temporal schemas and temporal documents that are flexible, robust, modular, and easy-to-use. Our design is upward compatible with conventional tools and extremely robust in that temporal data can occur at any level of the system: *anything can change*. We are able to capture both document and schema changes and provide the user a simple way to do so. Our design provides an expressive capability to describe temporal data in many situations.

Second, we have shown which types of temporal constraints may be enforced by a representational schema and which may not. In particular, we have shown that only sequenced cardinality and datatype constraints may be enforced via a representational schema. For these two types of constraints, we have provided a qualitative analysis of the benefits of each placement that generally conclude that, when possible, functionality should be placed within the representational schema and not within the tools. These analyses can be used as guidelines by future developers and researchers.

Third, we have provided an initial characterization of the design space for temporal representations which resulted in the definition of four orthogonal representation classes: slice-based, edit-based, item-based, and reference-based. We have shown how to implement the slice-, edit-, and item-based representations in a practically useful way using commonly-available tools where possible. Our evaluation of these classes shows that no representation performs the best under all conditions; instead, each is best suited for a subset of scenarios. The edit-based scheme provides the fastest means of creating and extracting the representation, as well as the most compact representation. However, this scheme suffers from a large overhead penalty when validating temporal constraints. In contrast, both the item-based and slice-based schemes can be validated quickly but require more time for creation and extraction. In general, we recommend that the item-based representation be used as long as performance enhancements can first be applied to the algorithms.

A fourth contribution was significant implementation and project organization. We implemented schema versioning into the τXSchema tools, along with additional representation types. We provided a number of project enhancements, such as a uniform structure and interface, logging capabilities, version control, wiki page for collaboration, and code reorganization to meet common software engineering paradigms.

Overall, we have uncovered some of the issues relating to representing temporal data and proposed methods to address these issues. Our analyses will help future researchers and users understand the trade-offs involved with each representation class. Users concerned with transmitting files will opt for the edit-based scheme while users concerned with validation time will choose the item-based scheme. Temporal system designers might consider a hybrid approach between the item-based and edit-based.

9.2 Future Work

Several avenues exist to enhance our understanding of the issues regarding representations that this thesis presented. The main concerns would be tool implementation, followed by a more rigorous evaluation technique.

9.2.1 Tool Implementation

In their current implementation, the tools suffer from a few obvious inefficiencies which greatly impact execution time and memory requirements. As an example, consider the implementation of the item-based representation class. As noted in Section 8, the tools currently rely on certain "push-up" and "push-down" routines for squashing a series of slices; the repetitive application of these routines resulted in much larger execution times when compared to other representation classes. Other similar improvements can be made to significantly impact the efficiency of the tools.

The functionality placement analysis could also be extended to more fully understand the impact of the validation time of specific temporal constraints, allowing the tools to be fine-tuned for performance. For example, does a sequenced cardinality constraint validate faster than a non-sequenced referential constraint? What are the characteristics of faster-validating constraints? How can we use these results to increase the performance of the slower-validating constraints?

Another enhancement involves the completion of τXMLLINT so that the complete set of temporal constraints are functional and the new language enhancements are fully implemented. At present, only a subset of such constraints are functional. Also, in order to achieve the desired upward compatibility with the conventional version of XMLLINT, additional command-line options need to be implemented in τXMLLINT so that it closely mimics the behavior of the conventional tool.

As mentioned previously, the current implementation of τXMLLINT uses a DOM parser to parse the input XML documents. However, it may be advantageous to instead use a Simple API for XML (SAX) parser [25]. While DOM parsers provide certain advantages to the programmer—the most important of which is a full tree representation of the data that allows continuous random access to the entire tree—SAX parsers can provide more memory-efficient handling of large documents. The amount of memory required for SAX parsers is typically much smaller than that of a DOM parser, and SAX parsers also significantly outperform DOM parsers in terms of execution time.

In order for the τXSchema tools to change from a DOM parser to a SAX parser, we would unfortunately be required to significantly change the current implementation. Currently, the tools create a DOM `Document` object for each file in the temporal system; this `Document` is then passed around and used within each of the tools directly. For example, certain elements are directly extracted from the `Document` by name or value, depending on the context of the program, to determine the next execution path. However, since SAX does not provide these same mechanisms of direct access to data, an entirely different approach must be taken to achieve the same effects. These changes would need to implemented throughout the entire code base, resulting in a significant effort. More investigation is needed to determine the actual amount of changes required, and thus the feasibility of such a change.

9.2.2 Empirical Evaluation

This research presents an initial characterization of the performance of the four representation classes. In order to achieve a higher level of rigor for our analysis, it is necessary to address shortcomings in both our measurement methodology and choice of experiments. First, standard XML benchmark data sets, such as DBLP [20] and other very large data sets, should be considered. The main difficulty here would be creating and managing a set of versions of these large files so that the τXSchema tools can be applied.

We could improve our analysis so that disk and memory caching issues are addressed; namely, in the current results, it was possible for two experiments to be run on the same file and the second experiment would receive the benefit of the file being cached. Disabling such caching will provide a more consistent measuring technique and also allow us to accurately discriminate between I/O time and execution time.

We can extend the type of experiments that we run by including scenarios in which slices experience a large variation in size (i.e., number of elements in the document) and XML complexity (i.e., type of tree, depth of tree, number of references) over time.

We can further test the representation classes by considering incremental scenarios in which a single slice was squashed into an already-squashed representation.

We claim in Section 8.2 that a schema change wall will have the same effect on execution time and representation size across all four representation classes. Work needs to be done to verify this claim and fully understand the effect of schema-versioning.

Figure 20(b) in Appendix C shows a small, unexpected "hump" in the squash execution time for the edit-based class; more work is needed to understand this anomaly.

9.2.3 Other

In the current version of Section 5 (Functionality Placement: Schema vs. Tools), we argue our points using an intuitive, less formal approach and we include examples and counterexamples as the main technique for illustrating our arguments. However, we would like to move towards a more rigorous approach by creating a formal environment to describe XML Schema data modeling, capabilities, and constraint enforcement functionality. In such an approach we would be able to propose and formally prove various theorems to better illustrate our arguments.

Work can be done to consider indexing issues for each representation class, and additional empirical experiments can be performed to quantify the performance of each class.

We can extend our idea of representation classes to also include database-oriented representations, as opposed to only considering text-based representations.

Further enhancements to τXSchema in general are suggested by Joshi [19].

References

[1] APACHE. Apache's log4j, official website. `http://logging.apache.org/log4j/1.2/index.html`, Viewed December 2, 2008.

[2] BIRSAN, D., SLUIMAN, H., AND FERNZ, S.-A. Xml diff and merge tool, 1999.

[3] BUNEMAN, P., DAVIDSON, S., FAN, W., HARA, C., AND TAN, W. Keys for XML. *Computer Networks 39*, 5 (2002), 473–487.

[4] CHAWATHE, S., ABITEBOUL, S., AND WIDOM, J. Representing and querying changes in semistructured data. In *14th International Conference on Data Engineering* (Orlando, FL, USA, 1998), IEEE Computer Society, pp. 4–13.

[5] CHIEN, S.-Y., TSOTRAS, V. J., AND ZANIOLO, C. Efficient schemes for managing multiversionxml documents. *The VLDB Journal 11*, 4 (2002), 332–353.

[6] COBENA, G., ABITEBOUL, S., AND MARIAN, A. Detecting changes in xml documents. In *18th International Conference on Data Engineering* (San Jose, California, 2002), IEEE Computer Society, pp. 41–52.

[7] COSTELLO, R. L. Zero, one, or many namespaces?, August 2006.

[8] COSTELLO, R. L., AND UTZINGER, M. Impact of xml schema versioning on system design.

[9] DYRESON, C., SNODGRASS, R. T., CURRIM, F., AND CURRIM, S. Schema-mediated exchange of temporal xml data. In *25th International Conference on Conceptual Modeling* (Tucson, AZ, USA, 2006), Conceptual Modeling - ER 2006. 25th International Conference on Conceptual Modeling. Proceedings (Lecture Notes in Computer Science Vol. 4215), Springer-Verlag, pp. 212–27.

[10] DYRESON, C., SNODGRASS, R. T., CURRIM, F., CURRIM, S., AND JOSHI, S. Weaving temporal and reliability aspects into a schema tapestry. *Data Knowl. Eng. 63*, 3 (2007), 752–773.

[11] DYRESON, C. E., LING LIN, H., AND WANG, Y. Managing versions of web documents in a transaction-time web server. In *WWW '04: Proceedings of the 13th international conference on World Wide Web* (New York, NY, USA, 2004), ACM, pp. 422–432.

[12] GABRIEL, J. How to version schemas. In *Proceedings of XML 2004-Conference and Exhibition* (November 2004).

[13] GAMMA, E. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] GAO, D., AND SNODGRASS, R. T. Temporal slicing in the evaluation of xml queries. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases* (2003), VLDB Endowment, pp. 632–643.

[15] GERGATSOULIS, M., AND STAVRAKAS, Y. Representing changes in xml documents using dimensions. In *Database and Xml Technologies*, vol. 2824 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin, Berlin, 2003, pp. 208–222. ISI Document Delivery No.: BY03G HEIDELBERGER PLATZ 3, D-14197 BERLIN, GERMANY.

[16] GRANDI, F. A bibliography on temporal and evolution aspects in the world wide web. Tech. Rep. Technical Report TR-75, TimeCenter, September 2003.

[17] GRANDI, F., AND MANDREOLI, F. The valid web: its time to go. Tech. Rep. Technical Report TR-46, TimeCenter, October 1999.

[18] JENSEN, C. S., AND SNODGRASS, R. T. Temporal data management. Tech. Rep. Technical Report TR-17, TimeCenter, June 1997.

[19] JOSHI, S. τxschema - support for data- and schema-versioned xml documents. Master's thesis, Computer Science Department, University of Arizona, August 2007.

[20] LEY, M. The dblp computer science bibliography, 2009. `http://www.informatik.uni-trier.de/~ley/db`, Viewed February 23, 2009.

[21] LIENTZ, B. P. Issues in software maintenance. *ACM Comput. Surv. 15*, 3 (1983), 271–278.

[22] MARIAN, A. Detecting changes in xml documents. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering* (Washington, DC, USA, 2002), IEEE Computer Society, p. 41.

[23] MARIAN, A., ABITEBOUL, S., COBENA, G., AND MIGNET, L. Change-centric management of versions in an xml warehouse. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2001), Morgan Kaufmann Publishers Inc., pp. 581–590.

[24] MCKENZIE, E., AND SNODGRASS, R. T. An evaluation of relational algebras incorporating the time dimension in databases. *ACM Computing Surveys 23*, 4 (December 1991), 501–543.

[25] MEGGINSON, D. Simple API for XML (SAX), April 2004. `http://www.saxproject.org/`, Viewed January 26, 2009.

[26] OMG. Unified modeling language (uml), v1.5, 2003-03-01 2003.

[27] OZSOYOGLU, G., AND SNODGRASS, R. T. Temporal and real-time databases:a survey. *IEEE Transactions on Knowledge and Data Engineering 7*, 4 (August 1995), 513–532.

[28] PRICES, D. R. Concurrent Versions System (CVS), Official website, December 2006.

[29] RIZZOLO, F., AND VAISMAN, A. A. Temporal xml: modeling, indexing, and query processing. *The VLDB Journal The International Journal on Very Large Data Bases 17*, 5 (2008), 1179–1212.

[30] RODDICK, J. F. Schema evolution in database systems: an annotated bibliography. *SIGMOD Rec. 21*, 4 (1992), 35–40.

[31] RODDICK, J. F. A survey of schema versioning issues for database systems. *Information and Software Technology 37*, 7 (1995), 383–393.

[32] SJØBERG, D. Measuring schema evolution. Tech. Rep. FIDE/92/36, University of Glasgow, 1992.

[33] SJØBERG, D. Quantifying schema evolution. *Information and Software Technology 35*, 1 (1993), 35–44.

[34] SNODGRASS, R. T. Temporal object oriented databases: A critical comparison. In *Modern Database Systems: The Object Model, Interoperability and Beyond*, W. Kim, Ed. Addison-Wesley/ACM Press, 1995, pp. 386–408.

[35] SNODGRASS, R. T., GOMEZ, S., AND MCKENZIE, E. Aggregates in the temporal query language tquel. *IEEE Trans. on Knowl. and Data Eng. 5*, 5 (1993), 826–842.

[36] SNODGRASS, R. T., AND THOMAS, S. W. TAU Project, Computer Science Department at the University of Arizona, January 2009.

[37] THOMAS, S. W. Temporal XML Schema wiki, July 2008. `http://cgi.cs.arizona.edu/apps/tauXSchema`, Viewed January 26, 2009.

[38] VEILLARD, D. The xml c parser and toolkit of gnome. `http://xmlsoft.org`, Viewed June 26, 2008.

[39] W3C. XML Path Language (XPath), November 1999. `http://www.w3.org/TR/xpath`, Viewed October 12, 2008.

[40] W3C. XML Schema Part 0: Primer, May 02 2001.

[41] W3C. Document Object Model (DOM) Level 2 HTML Specification Version 1.0, November 08 2002. `http://www.w3.org/DOM/`, Viewed November 2008.

[42] W3C. XQuery 1.0: An XML Query Language, January 2007. `http://www.w3.org/TR/xquery`, Viewed October 12, 2008.

[43] W3C. Extensible Markup Language (XML) 1.0, August 2006. `http://www.w3.org/TR/REC-xml`, Viewed August 25, 2008.

A Schemas

A.1 TSSchema: Schema for Temporal Schema

Listing 84: TSSchema: Schema for Temporal Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema  targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema"
            xmlns:ts="http://www.cs.arizona.edu/tau/tauXSchema/TSSchema"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            version="December 5, 2008">

  <xs:include schemaLocation="./SliceSequence.xsd"/>

  <xs:element name="temporalSchema">
    <xs:complexType>
      <xs:sequence>

        <xs:element name="conventionalSchema" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:choice>
              <xs:element name="sliceSequence" type="ts:sliceSequenceType"/>
              <xs:element name="include"       type="ts:includeType"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>

        <xs:element name="annotationSet" minOccurs="0" maxOccurs="1">
          <xs:complexType>
            <xs:choice>
              <xs:element name="sliceSequence" type="ts:sliceSequenceType"/>
              <xs:element name="include"       type="ts:includeType"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>

      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A.2 ASchema: Schema for Annotation Schema

Listing 85: ASchema: Schema for Annotation Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema  targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/ASchema"
            xmlns:a="http://www.cs.arizona.edu/tau/tauXSchema/ASchema"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            version="December 5, 2008">


  <xs:element name="annotationSet">
    <xs:complexType>
      <xs:all>

        <xs:element name="logical"  type="a:logicalType"  minOccurs="0" maxOccurs="1" />
        <xs:element name="physical" type="a:physicalType" minOccurs="0" maxOccurs="1"/>

      </xs:all>
    </xs:complexType>
  </xs:element>


  <xs:complexType name="logicalType">
      <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="a:kindType"      use="optional"/>
                    <xs:attribute name="content"
                                  type="a:contentType"   use="optional"/>
                    <xs:attribute name="existence"
                                  type="a: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="a:kindType"      use="optional"/>
                  <xs:attribute name="content"   type="a:contentType"   use="optional"/>
                  <xs:attribute name="existence" type="a: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="a:keyrefTypeII" use="optional"/>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="field" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="path" type="xs:string" use="required"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name"
                                type="xs:string"           use="optional"/>
                  <xs:attribute name="timeDimension"
                                type="a:timeDimensionType" use="optional"/>
                </xs:complexType>
              </xs:element>
              <xs:element name="attribute" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="validTime" minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="contentVaryingApplicability"
                                      minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                              <xs:attribute name="begin"
                                            type="xs:string" use="optional"/>
                              <xs:attribute name="end"
                                            type="xs:string" use="optional"/>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="frequency" type="xs:string" minOccurs="0"/>
                        </xs:sequence>
                        <xs:attribute name="kind"
                                      type="a:kindType"    use="required"/>
                        <xs:attribute name="content"
                                      type="a:contentType" use="optional"/>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="transactionTime" minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
```

```
                              <xs:element name="frequency" type="xs:string" minOccurs="0"/>
                            </xs:sequence>
                          </xs:complexType>
                        </xs:element>
                      </xs:sequence>
                      <xs:attribute name="name" type="xs:string" use="optional"/>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
                <xs:attribute name="target" type="xs:anyURI" use="required"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>

  <!--  Simple Types used by the logical annotations above -->
   <xs:simpleType name="kindType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="state"/>
      <xs:enumeration value="event"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="keyrefTypeII">
    <xs:restriction base="xs:string">
      <xs:enumeration value="snapshot"/>
      <xs:enumeration value="itemIdentifier"/>
    </xs:restriction>
    <!-- II in "keyrefTypeII" stands for ItemIdentifier -->
  </xs:simpleType>
  <xs:simpleType name="contentType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="constant"/>
      <xs:enumeration value="varying"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="existenceType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="constant"/>
      <xs:enumeration value="varyingWithGaps"/>
      <xs:enumeration value="varyingWithoutGaps"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="timeDimensionType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="validTime"/>
      <xs:enumeration value="transactionTime"/>
      <xs:enumeration value="bitemporal"/>
    </xs:restriction>
  </xs:simpleType>


  <xs:complexType name="physicalType">
      <xs:sequence>
        <xs:element name="include" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="annotationLocation" type="xs:anyURI"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="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="a:timeDimensionType" use="optional"/>
                  <xs:attribute name="stampBounds"
                                type="a:stampType"         use="optional"/>
                </xs:complexType>
              </xs:element>
              <xs:element name="orderBy" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="field" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:choice>
                          <xs:element name="target" type="xs:string"/>
                          <xs:element name="time">
                            <xs:complexType>
                              <xs:attribute name="dimension" type="a:timeDimensionType"/>
                            </xs:complexType>
                          </xs:element>
                        </xs:choice>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="target"
                          type="xs:string"           use="required"/>
            <xs:attribute name="dataInclusion"
                          type="a:dataInclusionType" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>


  <xs:simpleType name="stampType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="step"/>
      <xs:enumeration value="extent"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="dataInclusionType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="expandedEntity"/>
      <xs:enumeration value="referencedEntity"/>
      <xs:enumeration value="expandedVersion"/>
      <xs:enumeration value="referencedVersion"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

A.3 SliceSequenceSchema: Schema for Slice Sequences

Listing 86: SliceSequenceSchema: Schema for Slice Sequences

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

    <xs:complexType name="sliceSequenceType">
        <xs:sequence>
           <xs:element name="slice" minOccurs="1" maxOccurs="unbounded">
             <xs:complexType>
               <xs:attribute name="location" type="xs:string" use="required"/>
               <xs:attribute name="begin"    type="xs:date"   use="required"/>
               <xs:attribute name="end"      type="xs:date"   use="optional"/>
             </xs:complexType>
           </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="includeType">
       <xs:attribute name="schemaLocation" type="xs:string" use="required"/>
    </xs:complexType>
</xs:schema>
```

A.4 TDSchema: Schema for Temporal Document

Listing 87: TDSchema: Schema for Temporal Document

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema  targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/TDSchema"
            xmlns:td="http://www.cs.arizona.edu/tau/tauXSchema/TDSchema"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            version="December 5, 2008">

  <xs:include schemaLocation="./SliceSequence.xsd"/>

  <xs:element name="temporalDocument">
    <xs:complexType>
      <xs:sequence>

        <xs:element name="temporalSchemaSet" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="temporalSchema" minOccurs="1" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="location" type="xs:string"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="sliceSequence" type="td:sliceSequenceType"/>

      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A.5 TVSchema: Schema for Timestamps

Listing 88: TVSchema: Schema for Timestamps

```
<xsd:schema targetNamespace="http://www.cs.arizona.edu/tau/tauXSchema/TVSchema"
            xmlns:tv="http://www.cs.arizona.edu/tau/tauXSchema/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>
```

B Evaluation Tools

B.1 Slice Generator

Listing 89: Slice Generator script

```perl
#!/usr/bin/perl -w
#################################################################
#
# Author:  Stephen W. Thomas
# Date:    Fall 2008
# Purpose: To generate a large number of XML slices. The
#          output of this script is a set of sliceXX.xml
#          files along with config.xml file used in the
#          tools.
#
##################################################################

if ($#ARGV < 3){
   print "Usage: $0 amountChange docSize changeKind numSlices\n";
   exit 1;
}

$amountChange = $ARGV[0];
$docSize      = $ARGV[1];
$changeKind   = $ARGV[2];
$numSlices    = $ARGV[3];

# Set begin date
$lastDate       = "2008-01-01";

my $numParts    = $docSize;

# Probability that any given element is changed
my $pNewVersion = $amountChange * $changeKind;

# Probability that a new item (element) is created
my $pNewItem    = $amountChange * (1.0-$changeKind);

#Initialize values
$totalPossible = $numParts * (10*$numSlices);
my @partQuant;
for ($j=0; $j < $totalPossible; ++$j){
    $partQuant[$j] = $j;
}

# Print header of config document
open(CONFIG, ">config.xml");
print CONFIG "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n";
print CONFIG "<config bundle=\"./test_bundle.xml\"
             xmlns=\"http://www.cs.arizona.edu/tau/tauXSchema/ConfigSchema\">\n";

# Print each slice to disk, and add the name of the slice to the config file
for ($i = 0; $i <= $numSlices; ++$i){
    printLargeFile($i);
    addSnapshotToConfig($i);
}
print CONFIG "</config>\n";


# Adds an entry into the config file
sub addSnapshotToConfig{
    $outfile    = "slice$i.xml";
    $date1      = $lastDate;
    $lastDate   = incrementDate($date1);
    print CONFIG
```

```
      "  <snapshot file=\"$outfile\" beginDate=\"$date1\" endDate=\"$lastDate\"/>\n";
}


# Increments a simple date format "YYYY-MM-DD" with wrappint
sub incrementDate{
   $firstDate = shift(@_);
   $incMonth = 0;
   $incYear  = 0;

   @ar = split("-", $firstDate);
   $firstYear  = $ar[0];
   $firstMonth = $ar[1];
   $firstDay   = $ar[2];

   $newDay = $firstDay + 1;
   if ($newDay > 28){
     $newDay   = 1;
     $incMonth = 1;
   }
   $newMonth = $firstMonth + $incMonth;
   if ($newMonth > 12){
     $newMonth   = 1;
     $incYear = 1;
   }
   $newYear = $firstYear + $incYear;

   return sprintf "%4d-%02d-%02d",  $newYear,$newMonth,$newDay;
}


#####################################################################
# In each additional slice, we may add more elements to each section.
# We may also change the content of existing elements.
#####################################################################
sub printLargeFile{

    # Should we increase the number of <part> elements?
    if (rand() <= $pNewItem){
      $numParts = $numParts + rand(10);
    }

    $outfile = "slice$i.xml";
    open(OUT, ">$outfile");
    print OUT "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n";
    print OUT "<root>\n";

    print OUT "<parts>\n";

    # Print out every <part> element that we have.
    for ($j=0; $j < $numParts; ++$j){

        # Should we create a new "version" of this element?
        # A new version is created by just changing the value
        # of the <quantity> subelement.
        if (rand() <= $pNewVersion){
            $partQuant[$j]++;
        }

        print OUT "  <part>\n";
        print OUT "    <name>part_$j</name>\n";
        print OUT "    <id>$j</id>\n";
        print OUT "    <quantity>$partQuant[$j]</quantity>\n";
        print OUT "  </part>\n";
    }
    print OUT "</parts>\n";

    print OUT "</root>\n";
}
```

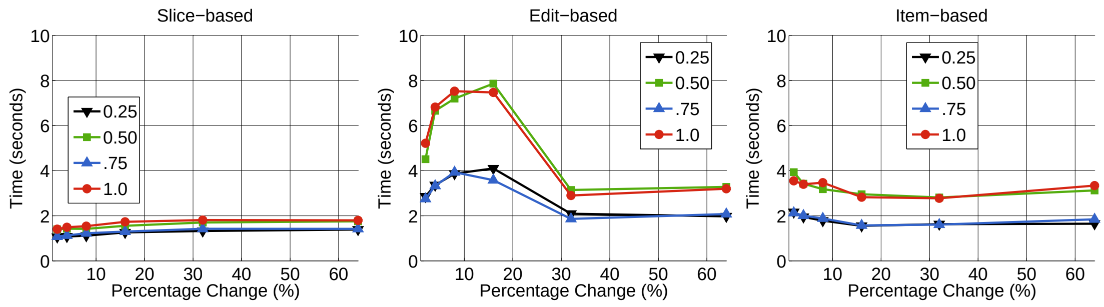
B.2 Scenario Tester

Listing 90: AllRuns script

```perl
#!/usr/bin/perl
##############################################################
#
# Author:  Stephen W. Thomas
# Date:    Fall 2008
# Purpose: To execute a large set of runs and output results
#          to stdout.
#
###############################################################

use Time::HiRes qw( gettimeofday tv_interval );

my $N = 30;

# Define the variables and their values
my @amountChanges = (0, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64);
my @docSizes      = (1000, 4000, 16000, 64000);
my @degrees       = (0, .25, .5, .75, 1);
my @numSlices     = (100, 1000, 5000, 10000);

my ($t1, $t2, $squashTime, $squashSize, $valTime, $unsquashTime);

# Full factorial design space. Run all cases.
foreach my $amountChange (@amountChanges) {
  foreach my $docSize (@docSizes) {
    foreach my $degree (@degrees) {
      foreach my $numSlice (@numSlices) {

        $squashTime   = 0;
        $squashSize   = 0;
        $valTime      = 0;
        $unsquashTime = 0;

        for (my $i=0; $i<$N; ++$i){

            #print "Producing test case $amountChange $docSize $degree $numSlice \n";
            system("generator.pl $amountChange $docSize $degree $numSlice");


            # Run Squash
            $t1 = [gettimeofday];
            system("squash config.xml");
            $squashTime += tv_interval($t1);
            $squashSize += `ls -l squashed.xml | awk '{ print\$5 }'`;


            # Run tXMLLint
            $t1 = [gettimeofday];
            system("txmllint config.xml squashed.xml > /dev/null");
            $valTime += tv_interval($t1);

            # Run UnSquash
            $t1 = [gettimeofday];
            system("unsquash config.xml squashed.xml");
            $unsquashTime += tv_interval($t1);

            system("rm slice*");
            system("rm squashed.xml");

        }

        # Get averages
        $squashTime   = $squashTime   / $N;
        $squashSize   = $squashSize   / $N;
        $valTime      = $valTime      / $N;
        $unsquashTime = $unsquashTime / $N;

        # Output results
        printf "%12.5f %12.5f %12.5f %12.5f %12.5f %12.5f %12.5f %12.5f\n",
            $amountChange, $docSize,  $degree, $numSlice,
            $squashTime, $squashSize,
            $valTime, $unsquashTime;
          }
        }
  }
}
```

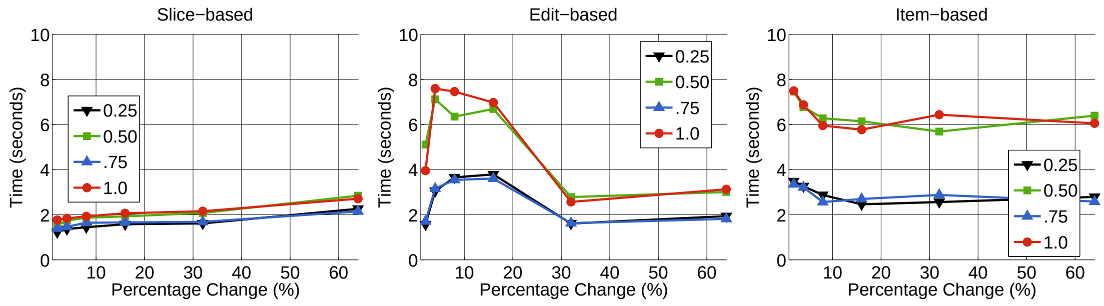
C Initial Sensitivity to Parameters

This appendix provides details on the results of the initial experiments executed. The goal was to determine whether or not the dependent variables of interest were sensitive to the amount of change and type of change between each slice. Figure 19 shows the results of SQUASH in these scenarios. We see that for each representation type, as the percentage of change increases, the time required to squash the document does not increase significantly. Figure 20 shows the results of a larger scenario, but the trends exhibited by the results are similar.



(a) Slice-based (b) Edit-based (c) Item-based

Figure 19: Time required to squash 10 slices, each with about 10 elements. The slice-based and item-based representation schemes show almost no difference in performance between both percentage change increases (x-axis) or type of change increased (lines). The edit-based scheme shows some small variation, but no general trend is evident and the absolute amount of change is small.



(a) Slice-based (b) Edit-based (c) Item-based

Figure 20: Time required to squash 100 slices, each with about 200 elements (20 slices with 20 elements in the case of the item-based scheme). Again, the slice-based and item-based representation schemes show almost no difference in performance between both percentage change increases (x-axis) or type of change (lines). The edit-based scheme shows some small variation, but no general trend is evident and the absolute amount of change is small.

Both UNSQUASH and τXMLLINT show similar trends. In effort to reduce to number of experiments run, we conclude that the type and frequency of change is not a large factor in representation performance, and thus we can fix these parameters for the remainder of the experiments.