Refining the Infrastructure of τXSchema

Honor's Thesis

Alex Henniges
Department of Computer Science
University of Arizona
Supervisor: Dr. Richard Snodgrass

April 30, 2010

Abstract

With the ever-growing use of XML and the demand for storing time-varying documents, τXSchema is one such method for providing a unified system for temporal documents. This paper studies the current implementation of τXSchema in Java and recommends a new design that reduces redundant code found in the tools by improving upon the infrastructure, resulting in less code that is easier to understand and has the potential to run more efficiently.

Contents

List of Figures

1 Introduction

Extensible Markup Language (XML) is a format for describing data within documents. The use of XML has risen considerably in recent years, due in part to the language's flexibility in describing a wide range of data. In addition, XML's format can be easily read (it closely mimics HTML) and is intuitive. That is, data is represented in an object-oriented design where elements are stored in a hierarchical fashion (see Section 2.1). In addition to storing data, XML can describe how the data should be stored, such as how two elements are related, in a *schema*. W3C defines the most widely used schema language for XML, termed XMLSchema.

However, there are no standardized tools in XML to store the changes that may occur in a set of data. Such information, called temporal data, comes in many forms and is useful in many scenarios. For example, a company may have data on the current address of an employee, but might also want to know the previous addresses of the employee. If this type of data is accessible, then questions can be asked of the data: e.g. how many times has the employee moved in the last two years? For any set of data, this extra information can easily be added into the XML's schema. A more difficult problem is developing a schema that records changes in data for all possible sets of data.

In [1], a solution to this problem is suggested by way of an extension to the W3C schema language, known as τXSchema. The paper recommends that a set of standard XML documents representing the time-varying data is stored in a single additional, *temporal document*. Moreover, the schema or schemas that describe the instances of the data is collected in another document called the *temporal schema*. The temporal document and temporal schema are both described by a standard XMLSchema schema, thus providing an upward-compatible extension for time-varying documents. As part of this system, several tools are provided: squash, schema mapper, and τValidator, which allow the user to describe how the temporal data is presented and organized.

An initial version of most of these tools and the underlying system was implemented in Java. However, the design of this first version had several flaws that made it difficult to complete the implementation of τXSchema. These flaws include a previous terminology for τXSchema and a corresponding doc-

ument set that proved to be cumbersome. This paper will discuss those flaws and then explain in detail the new language and the features of the resulting design. In summary, this paper will show how identifying the commonality found within the tools that support τXSchema and moving this functionality into the infrastructure (that is, the shared classes) enables the tools to require less code, to be more easily understood, and to run more efficiently.

2 Overview

At the start of this project, the goal was to complete the implementation of the temporal validator, as described in [1]. Other important tools, such as squash and schema mapper, were already implemented (see Section 2.5). Also, the functionality that represented temporal elements, known as *items*, was already written in Java.

The early part of the project involved learning XML, reading about τXSchema through a 260-page technical report [1], providing editing comments for this report, working on the τXSchema website, and performing some small, initial changes to the τXSchema system of nearly $10,000$ lines of code, as well as the appropriate test cases for those changes. However, the design of the tools proved to be cumbersome and a large barrier to entry for any new hands on the project. The goal then changed to re-building the implementation of the concepts presented elsewhere [1].

The rest of this section provides helpful background to the τXSchema system.

2.1 XML

XML is a format for encoding documents, very similar to HTML. Tags are used to indicate the beginning and end of objects. These objects are either *elements*, which are entities that hold all sorts of data and even other elements, or *attributes*, which are the data. A small example of an XML file is given in listing 1. Each XML file has a *root element*, in our example `person`, that also contains some preamble-type information, including the schema that validates it. In the example, the `person` element has two child elements, `fname` and `age`. In addition, the `fname` element has an attribute for a `prefix` to the name.

Listing 1: An example XML file

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3           xsi:noNamespaceSchemaLocation="./PersonSchema.xsd">
4         <fname prefix="Mr.">Steve</fname>
5         <age>24</age>
6 </person>
```

In order to maintain the consistency and correctness of XML files, schemas are used to describe how the documents ought to be structured. A document like our example in Listing 1 can be described by its corresponding schema, shown in Listing 2. This document can appear complicated, but one of the things that it states is that any `person` element is required to have one and only one `fname` and `age`. If an XML document with the `person` element were to not have one of these items, then it would not be validated as consistent with this schema.

Listing 2: The Person schema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3                   elementFormDefault="qualified"
4                   attributeFormDefault="unqualified">
5
6    <xsd:element name="person">
7      <xsd:complexType mixed="true">
8        <xsd:sequence>
9          <xsd:element ref="fname" minOccurs="1"
10                      maxOccurs="1" />
11         <xsd:element name="age" type="xsd:integer"
12                      minOccurs="1" maxOccurs="1"  />
13       </xsd:sequence>
14     </xsd:complexType>
15   </xsd:element>
16
17   <xsd:element name="fname">
18     <xsd:complexType mixed="true">
19       <xsd:attribute name="prefix" type="xsd:string" />
20     </xsd:complexType>
21   </xsd:element>
22 </xsd:schema>
```

2.2 New Language Design

To represent temporal data, it was proposed that a system of three documents be used: the temporal document, the temporal schema, and the annotation document [1]. The temporal document represents an instance of temporal data. It is the analog of a standard XML document. In fact, a temporal document references the conventional documents, called *slices*, that are changing over time. A temporal document can be written in several forms (see Section 2.4). A temporal document also refers to a temporal schema.

The *temporal schema* is the analog of a standard XML schema. Like the temporal document, the temporal schema refers to a sequence of conven-

tional schemas that are changing over time. Each conventional schema should match the conventional documents valid at times the schema is valid. It is important to note that the temporal schema is not the literal schema for the temporal document, but is used when creating such a schema and for temporal validation. A temporal schema also references one or more annotation documents.

The *annotation document* has no analog to a standard W3C notion. This document is used to describe how the temporal data should be structured and to define temporal constraints. This document is separated into two parts: physical and logical. *Physical annotations* state which elements of a conventional document can change. This information is important when structuring the temporal data, especially when squashing the temporal document (see Section 2.5). The *logical annotations* define how the data can change, such as once an element appears it can not disappear later. An annotation document may also change over time, perhaps when the schema changes, so a temporal schema can reference more than one annotation document.

2.3 Old Language Design

The first design of τXSchema used a different language and set of documents than of those used in the latest definition [1]. Developing a new language was done before considering a redesign of the code, but the change was nontrivial and integral to the proposed new design. We present a brief overview here to help emphasize the improvements in the new design.

The original language of τXSchema recommended the use of a bundle of documents with a chain of containments. The highest level document was the `Config` document. This document provided the path and times of validity of the slices as well as a reference to a `TemporalBundle`. The `TemporalBundle` document maintained all the other documents, including the schemas for each slice and the physical and logical annotations, which were separate documents.

The new language combines the annotations into one document and reorganizes the config and bundle documents into the temporal document and temporal schema. The former change is sensible as it decreases the number

of required documents while still making a logical merge. The latter change helps to improve τXSchema as an extension of XML and XMLSchema. The names of the new documents also better describe their functions.

2.4 Representation Types

A temporal document can be represented in several ways (a more in-depth analysis of each representation is given in Thomas' paper [2]).

SliceSequence This can be thought of as the basic representation of a temporal document, indicating for each slice only the filename and period for which that document is active.

ItemBased In this representation, the temporal document appears most like a standard XML document. Each time-varying element, now an item, contains its multiple instances.

EditBased A temporal document that is in this form records the differences from one version of a document to the next. This is a compact form and can be easy to generate, but can take time to re-interpret to the more understandable ItemBased representation.

SliceBased This representation shows the contents of every slice of the temporal document. It is the most comprehensive but naturally the largest representation as well.

Figure 1 shows two slices of a document and Figure 2 shows the resulting representation, in each of the four types.

```
<!-- 2008-01-01 -->
<person>
  <fname>Steve</fname>
  <age>24</age>
</person>
```

(a) Slice1

```
<!-- 2008-03-17 -->
<person>
  <fname>Steve</fname>
  <age>25</age>
</person>
```

(b) Slice2

Figure 1: Two slices of a simple XML document [2].

```
<tv_root>
  <person>
    <fname>Steve</fname>
    <age_Item>
      <age_Version begin="2008-01-01">
        <age>24</age>
      </age_Version>

      <age_Version begin="2008-03-17">
        <age>25</age>
      </age_Version>
    </age_Item>
  </person>
</tv_root>
```

(a) Item

```
<tv_root>

  ...

  <sliceSequence>
    <slice location="./slice1.xml"
        begin="2008-01-01" end="
        2008-03-17" />
    <slice location="./slice2.xml"
        begin="2008-03-17" />
  </sliceSequence>

  ...
</tv_root>
```

(b) SliceSequence

```
<tv_root>
  <timestamp begin="2008-01-01">
    <person>
      <fname>Steve</fname>
      <age>24</age>
    </person>
  </timestamp>
  <timestamp begin="2008-03-17">
    <person>
      <fname>Steve</fname>
      <age>25</age>
    </person>
  </timestamp>
</tv_root>
```

(c) Slice

```
<tv_root>
  <timestamp begin="2008-03-17">
    <person>
      <fname>Steve</fname>
      <age>25</age>
    </person>
  </timestamp>

  <timestamp begin="2008-01-11" >
    <change lines="3">
      &lt;age&gt;24&lt;/age&gt;
    </change>
  </timestamp>
</tv_root>
```

(d) Edit

Figure 2: The four representation types [2].

2.5 Tools

The operations that we would like to perform on temporal documents can be categorized into three tools:

Squash/ReSquash. Squashing a temporal document refers to taking multiple slices and 'squashing' them into one document. When we want to convert from one representation type to another, we refer to it as resquashing (or possibly unsquashing) the document. Often, a certain representation is more effective for one application than another in both time and the complexity of the implementation. The standard concept of squash is therefore converting from SliceSequence to ItemBased.

SchemaMapper. When a temporal document is being squashed, there is a check that the documents are valid. The tool SchemaMapper will create a standard schema document, known as the representational schema, to validate against the squashed document. SchemaMapper uses the temporal schema and annotation document to create the representational schema.

TemporalValidator. When storing temporal data, there are some constraints we may like to have that can not be implemented using XML's standard validation tools, even using squashed documents and a representational schema. For example, one may want to enforce the rule that an employee's email address can not change more than once per year. The TemporalValidator extends XML's validator to include temporal constraints. The TemporalValidator first creates a representational schema using SchemaMapper and performs a standard XML validation on the temporal document. It then checks for any temporal constraints, which can be found in the annotation document.

3 Former Design

The Unified Modeling Language (UML) diagram in Figure 3 at the end of this section shows the structure of the code at the beginning of this project. The UML shows the classes in τXSchema and their dependencies to one another. The classes at the bottom of Figure 3, such as `Item` and `BaseItem`, as well as `Primitive` represent the fundamental classes of the system that maintain the temporal data.

There are 17 classes or interfaces in total that are used for the tools. The procedure to use the tool `Squash`, for example, would be to start with the `main` function found in `Squash`, which then goes to the representation factory to create a `DoSquashing` object to perform the actual squash function. The rest of the functionality is found in `DoSquashing`, such as parsing the temporal document and temporal schema and creating the output document. As a result, despite breaking the tool into multiple classes, 596 lines of code can be found in the `DoSquashing` class.

Furthermore, the class `DoSVSquashing`, which performs a squash in cases where the schema changes over time, is distinguished from the standard squash. In the same vein, the functionality to reverse a squash is treated as a completely different tool. Because all of the tools are composed as such, each tool must parse the temporal documents and schema on its own and instantiate similar variables. An example would be that both Squash and SchemaMapper must parse the temporal document to access the annotation document in order to build the `LogicalAnnotationValidator` (LAV) and `PhysicalAnnotationValidator` (PAV). The parsing procedure requires about 50 lines of code and is somewhat complex. Because this code was featured in both classes, when the document structure was changed from bundles the same code had to be modified in both places. These types of changes have the potential to introduce consistency errors.

The commonality of the tools suggests that there should be a way to add to the infrastructure so that all of the tools access the same classes. At the same time, the large number of classes dedicated to the tools should be merged to help identify more similarities in the code. For example, the multiple representations of Squash can be grouped in a more efficient way. This would also decrease confusion and make it more clear how a tool arrives

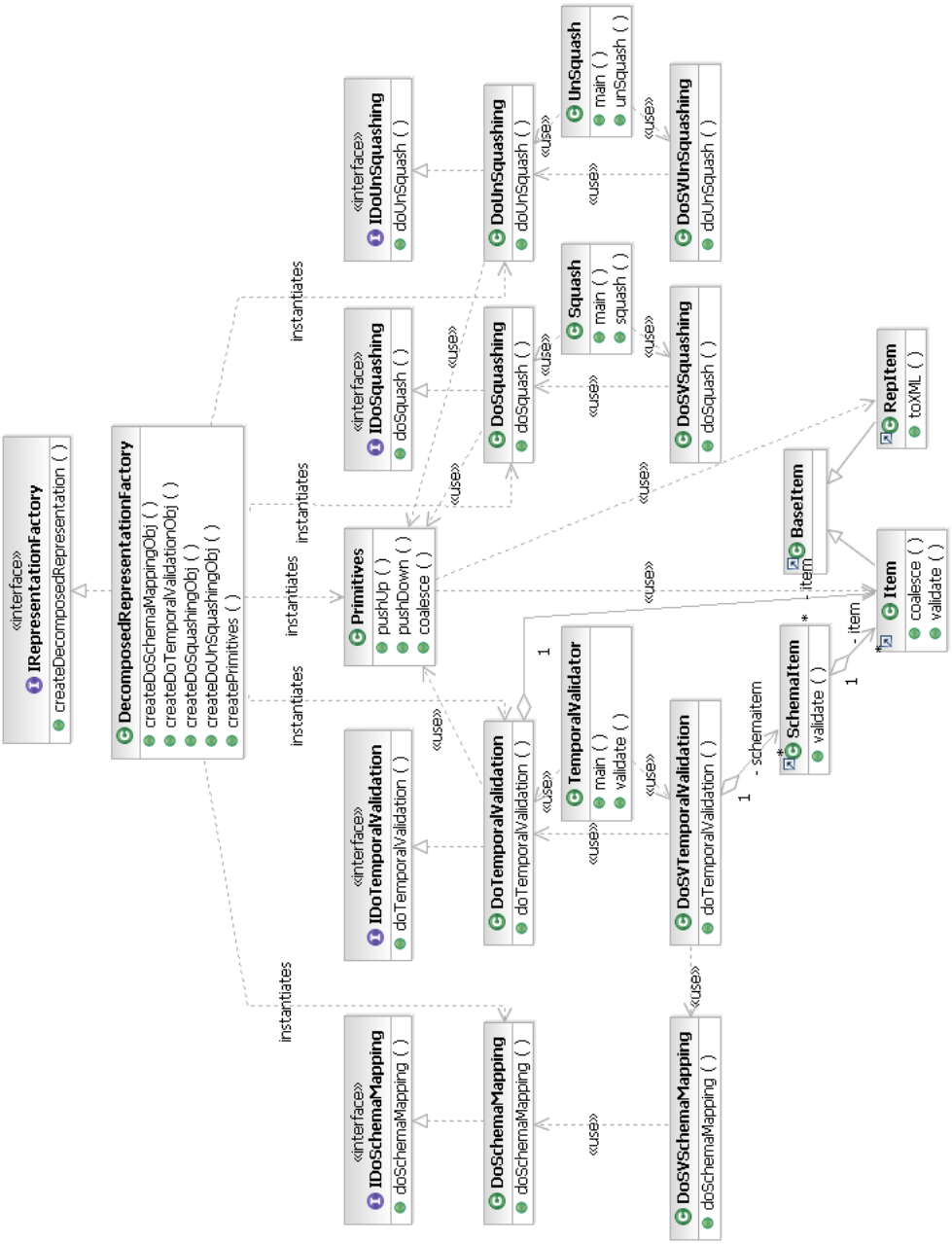

Figure 3: The UML diagram for the former design.

at its output. The rest of this thesis discusses an improvement to the design and concludes with an evaluation of the new design.

4 New Design

The UML diagram for the new design is Figure 4. Our original new design went through more than six iterations and multiple conference calls. Discussions focused on building a larger infrastructure to decrease the size of the tools, to improve options for the input and output of data, and to ensure that the new design could make use of older code. In addition, early implementation was done to test the feasibility of the new structure. We now walk through the individual parts of that design.

4.1 Temporal Map

Definition. *A* ***temporal map*** *is a collection of objects indexed by time periods with the following properties:*

1. *Insertion: When inserting an object into the temporal map, if the time period intersects the period of an existing object, the periods should be decomposed into disjoint intervals, and the newly inserted object should replace the old object over the time period given.*

2. *Retrieval: An object can be retrieved from the map by providing a time instance or a period. If a period is given that is not a subset of a period in the index,* `null` *is returned. That is, the provided period cannot intersect two periods of the map.*

Defining and using a temporal map aids in the storage of temporal data for the τXSchema system. By making use of generics as introduced in Java 1.5, we can use temporal maps to hold many different kinds of objects, such as the conventional schemas in the temporal schema. Because we are often interested in an instance of time, such as the state of a document on a specific date, the temporal map allows for retrieval of the document without requiring knowledge of the entire period during which the document is constant.

4.2 Document Classes

In the DOM implementation of XML in Java, there is a document object that contains the root node of the specific XML document and other information. We would like to add additional information so that it becomes a temporal

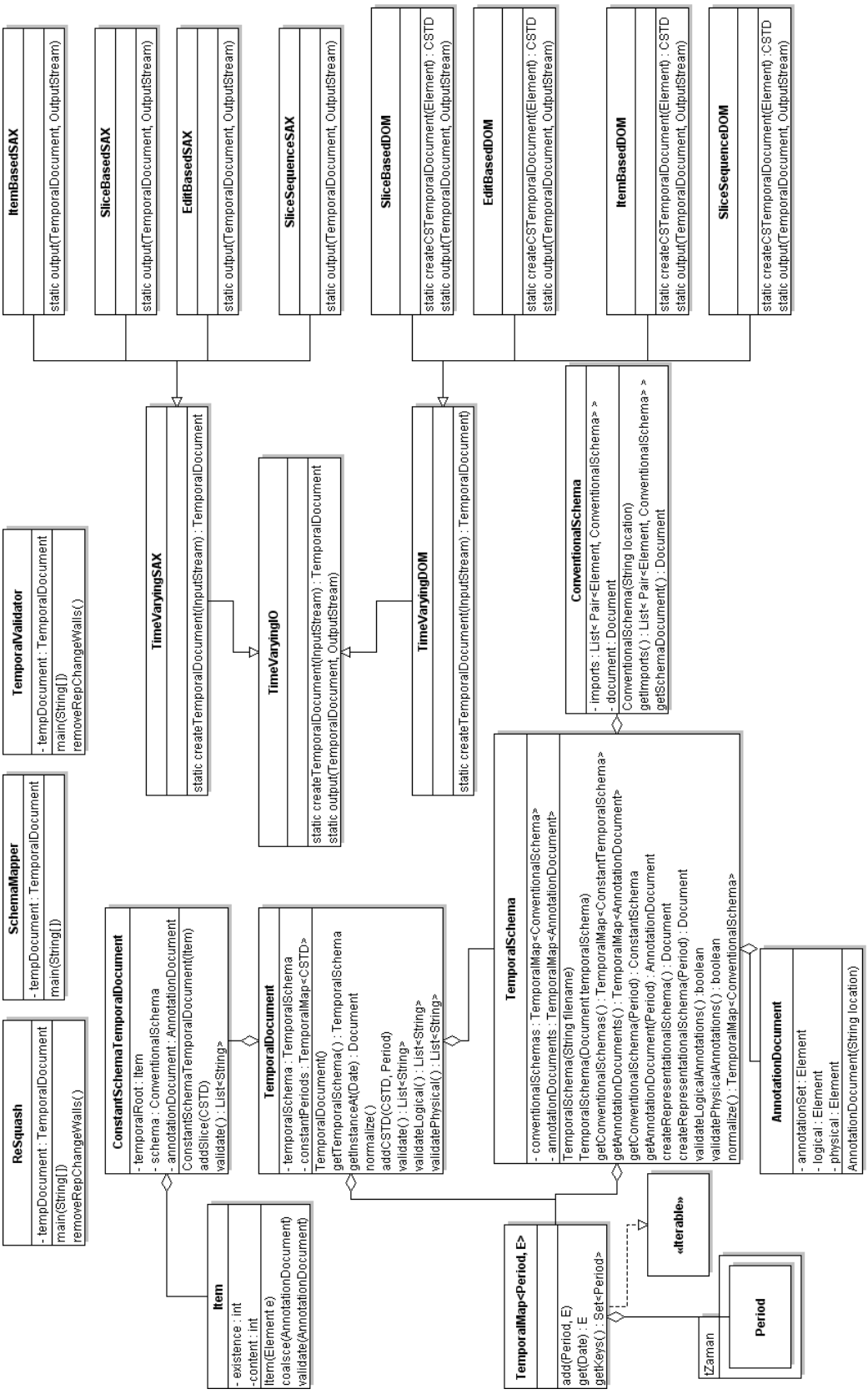

Figure 4: The UML diagram for the new design.

document. W3C only provides interfaces withing their package, so one option would be to implement the provided *Document* interface. However, we would have to re-implement many methods that do not need to be changed and this would also be a complicated procedure. Another option would be to maintain a document object within the class for a temporal document. This was considered in early implementations, but ultimately deemed unnecessary. Instead, we proceed by introducing three new classes.

TemporalDocument. This is the object we want to represent a temporal XML document. We first simplify the situation by considering periods where the schema for this document remains constant, known as schema constant periods (SCP). An element of a document that changes over time is an *item*. The previous design implemented the `Item` class in Java, and we now use it here for our `TemporalDocument`.

A `TemporalDocument` thus contains a `TemporalMap` of slice documents, which we call `ConstantSchemaTemporalDocument`s (CSTD) that are indexed by the SCPs. Each CSTD has a root `Item` which may have `Item` children. In this way, our `TemporalDocument` mirrors that of DOM's document object. Furthermore, a CSTD has information on the schema and annotations that describe it. We can also merge two CSTDs with adjacent periods using the `Item`'s `merge` function. Lastly, the `TemporalDocument` also references a `TemporalSchema` object.

TemporalSchema. Given the usefulness of the `TemporalDocument` object, it makes sense to also have an object to store information about temporal schemas. The `TemporalSchema` object has a `TemporalMap` of annotation documents. We also want to allow a temporal schema to imports other schemas just as a standard schema would. A complication arises if one of these imported schemas is itself temporal. We propose the following solution.

A `TemporalSchema` begins with a list of its imported schemas, all of which are assumed to be temporal. This recursively creates a tree of imported `TemporalSchema`s. The root `TemporalSchema` then calls the member function `normalize()` on itself. This function performs a post-order traversal of the tree, calling normalize on each child `TemporalSchema`. When called on a leaf, the leaf becomes a `ConventionalSchema` and passes its SCPs to the parent. The parent integrates the SCPs of its children to form one timeline

of SCPs. The parent then in turn passes this timeline to its parent and becomes a `ConventionalSchema`. At the end of this procedure, only the root should be temporal, and this `TemporalSchema` should have a `TemporalMap` of `ConventionalSchema`s.

The `TemporalSchema` provides information (via `AnnotationDocument`s) to the `TemporalDocument` when a document is being validated. Part of this process involves creating the representational schema, so naturally this functionality is provided here.

AnnotationDocument. Representing the annotation document is the simplest of the three. An `AnnotationDocument` contains DOM elements of the logical and physical annotations. Validation of a temporal document uses the logical annotations to check for temporal constraints and physical annotations to determine that only the correct elements are changing over time. The functionality for using these annotations to insert timestamps and create the appropriate items is moved from the two classes LAV and PAV and into this class.

All three document classes allow for more efficient code then the previous design. Before, each tool had its own set of the above objects. In the new design, we separate the annotation from the validator, placing both logical and physical annotations in the `AnnotationDocument`, and implementing validation within the `TemporalDocument` through the `validate` function. Furthermore, a tool now only needs to maintain the `TemporalDocument` and access to the other documents is simple.

Use of the document classes can also save computation time from repetitive traversals of the DOM document object that can occur from using several tools. Now, the time to construct a `TemporalDocument` needs to happen just once and it can then be passed from tool to tool.

4.3 Interface Classes

With this new design, we have moved the maintenance of documents out of the tools. The last commonality that can be found amongst the tools involves the input and output of τXML documents. Each tool is written with

code to read and parse these documents as well as code to format and write documents. In our redesign, we move this functionality into a new set of I/O classes.

We first acknowledge another option for reading XML documents in Java, that of SAX. SAX reads documents fundamentally different from DOM, and thus we would like to make this option available in our implementation. We thus have at the most abstract level the class `TimeVaryingIO` with the abstract function to create a `TemporalDocument` given a temporal document file. This function is implemented in both `TimeVaryingDOM` and `TimeVaryingSAX`. From this point forward we focus on DOM and assume that a SAX implementation would be similar.

When `TimeVaryingDOM` is given a temporal document, it checks the representation type through an attribute, then for every schema constant period, passes the root element to the appropriate representation class, which will create a CSTD. Lastly, `TimeVaryingDOM` will combine the CSTDs to form a `TemporalDocument` object. To perform the opposite function, that is, to take a `TemporalDocument` object and output it to a file, one calls the function `output` to the desired representation class.

A tool such as squash now becomes much easier to implement. An example of the sequence of function calls to perform a squash is given in Figure 5. In words, squash would give the temporal document, likely in `SliceSequence` form, to `TimeVaryingDOM` to create a `TemporalDocument` and then ask `ItemBasedDOM` to output the `TemporalDocument` in the Item-Based representation. ReSquash would be the same only where the input is in some other representation. The code that was originally in the squash tool is thus moved to a more intuitive location and in a way that all other tools can also access the same functionality without having to replicate code.

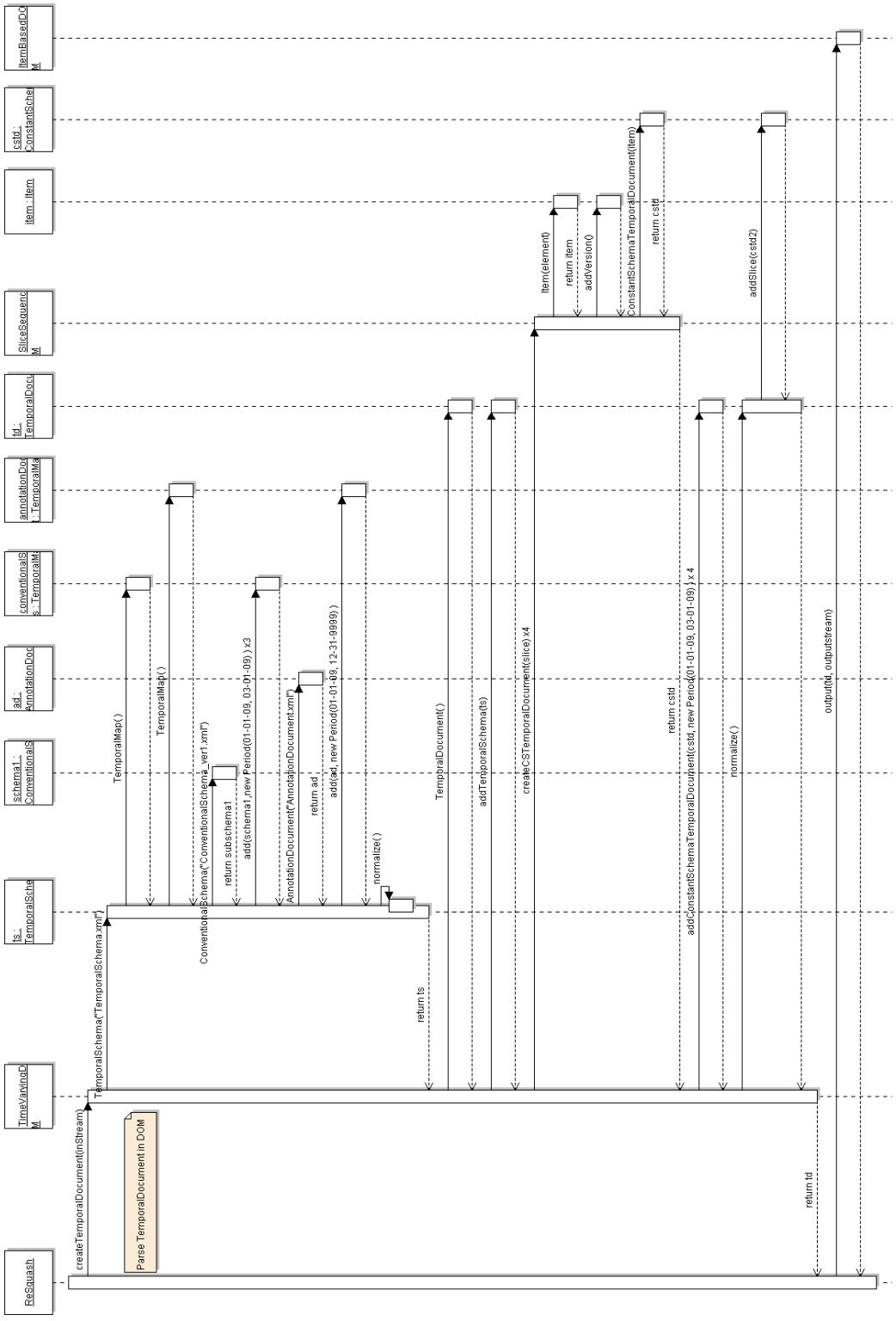

Figure 5: A sequence diagram of a document being squashed.

5 Evaluation

The changes to the language and document structure was completed and tested with the τXSchema system. The changes in the documents allowed for a more intuitive language, for example, a temporal document maintains the conventional documents and a temporal schema maintains the conventional schemas. Additionally, those changes influenced the direction of the new design. Furthermore, the previously working test cases for τXSchema were translated to the new style and were found to work as before.

The entire implementation of this design was not completed by the end of the project. Initial versions of the `TemporalMap` and document classes were created, but the I/O classes have not been implemented. That said, a number of advantages are readily clear from these changes. For one, there are only three classes, `ReSquash`, `SchemaMapper`, and `TemporalValidator` now dedicated solely to the tools compared to 17 (`DecomposedRepresentationFactory`, `Squash`, `DoSquashing`, `DoSVSquashing` for just one form of Squash) in the old design. Furthermore, the lines of code in one these classes is much less than their former counterparts. For example, in the sequence diagram given in Figure 5, only several lines of the procedure would actually be written within the `Squash` class (just the first and last function calls).

A third advantage is that the parsing of a temporal document is only performed once. Currently, a tool runs τXSchema once then the program ends. However, future use could involve situations where multiple tools are used within one session. Therefore, after the above `Squash` is performed, one can squash the document into other representation types or perform other tools like validation in reduced time. This is one example as to how this design is better than the old system at handling unknown, future situations. As another example, suppose it is determined that a tool that takes as input several temporal documents is needed. For this to be implemented in the old design, all of the data and variables for both documents would have to be maintained within the tool. Instead, the tool can simply carry the two `TemporalDocument` objects which contain their respective data, and moreover, that data is easily accessed. This aspect becomes more valuable if the tool were to, say, take as input a variable number, perhaps an array, of temporal documents. A direct corollary is that the proposed design is more accessible to new programmers working with the τXSchema system.

6 Conclusion & Future Work

The goal of this project was to improve the design of the τXSchema architecture, in particular the design of the tools, so that the code is more understandable as it is passed from one programmer to the next and that the code is more robust so that current and future additions to the the system are easier. The design described in this paper, though never fully implemented, includes many improvements to the previous design. The temporal document classes mimic and extend those of the document object model, much in the same way that τXSchema is designed to extend XMLSchema. Thus, we have now shown through this reorganization that the commonality within the tools can be moved into the infrastructure and that the end result is a cleaner project that is easier to understand and has the potential to run more efficiently.

The implementation of this design should aid in the completion of the tools such as the TemporalValidator, as was the original goal for this project. This design should ultimately make τXSchema a good choice for maintaining temporal data. As XML continues to be used in a wide variety of situations, τXSchema will provide a general way to maintaing time-varying data that extends XMLSchema in an intuitive way.

There are several aspects to the project that are worth spending more time on.

- A more robust implementation of the new design is still needed. This would then be followed by thorough comparisons to the old design, such as number of lines of code, number of classes, and speed of the tools.
- Developing the concept of a `TemporalMap` further and improving the efficiency of the implementation. For example, it may be better that a retrieval returns a list of objects when a given period interests several keys to the map.
- Explore all of the cases involved in the `normalize` function of `TemporalSchema` and nested schema versioning.
- Develop SAX versions of the input and output classes.

References

[1] Faiz Currim, Sabah Currim, Curtis Dyreson, Shailesh Joshi, Richard T. Snodgrass, Stephen W. Thomas, and Eric Roeder. τXSchema: Support for Data- and Schema-Versioned XML Documents. Technical report, Department of Computer Science, University of Arizona, 2009.

[2] Stephen W. Thomas. Implementation and Evaluation of Temporal Representations in XML. Master's thesis, Department of Computer Science, University of Arizona, 2009.