APPENDIX
I ALGORITHMS

In this appendix we provide the major algorithms for checking sequenced and non-sequenced constraints. This material elaborates on the summary of the τXMLLINT implementation provided in the paper.

```
CheckSequencedConstraints(constraints,
                          temporal_document):
foreach constraint in constraints do
    temporal_document ←
        FilterTemporalDocument(
            constraint, temporal_document)
    foreach t in ExtractTimePoints(
                    temporal_document) do
        slice ← ExtractSlice(t, temporal_document)
        if ¬Validate(slice) then
            return false
        end if
    end
end
return true
```

Algorithm 1: Checking Sequenced Constraints

Algorithm 1 first shrinks the input $temporal\_document$ by removing irrelevant nodes in the $FilterTemporalDocument$ routine, resulting in a small fraction of the original document. The relevant nodes are determined by the constraints, particularly the values of the `selector` and `field` elements. The nodes in the document that are not referenced by these elements will not be kept in the filtered document. It then iterates through the time points, which are the times when changes occur in the document, computed by routine $ExtractTimePoints$.

```
ExtractSlice(at_time, node):
if IsVersionNode(node) then
    foreach child in node.getChildNodes() do
        if child.begin_time ≤ at_time ∧
           at_time < child.end_time then
            return ExtractSlice(at_time, child)
        end
    end
    return nil
else
    slice ← node
    foreach child in node.getChildNodes() do
        extract ← ExtractSlice(at_time, child)
        if extract ≠ nil then
            slice.addChild(extract)
        end if
    end
    return slice
end
```

Algorithm 2: Slicing

A recursive routine $ExtractSlice$, presented as Algorithm 2, is then invoked to extract the corresponding slice. The versions within a version node are assumed to be contiguous; hence, only one child will ever be extracted. If *nil* is returned from the root, then there is a problem with the timestamps and τXMLLINT will return false.

After the slices are extracted, the routine $Validate$ is invoked to verify the sequenced constraints. In our current implementation, we utilize the DOM based validation facility provided by Java.

The number of $time\_periods$ (slices) and the size of each slice dominate the overall complexity of checking sequenced constraints. Since the $ExtractSlice$ routine traverses the entire document tree, its complexity in the worst case (when at_time is greater than all the $begin_time$s) is O(n), where n is the number of nodes in the document. Assuming the number of time points in the temporal document is m, the complexity of Algorithm 1 is O($n \cdot m$).

```
CheckNonSequencedConstraints(constraints,
                             temporal_document):
foreach constraint in constraints do
    evaluation_windows ← GetEvaluationWindows(
        constraint.evaluation_window_size,
        constraint.slide_size, temporal_document)
    foreach eval_window in evaluation_windows do
        results ← ExtractNodes(
                      eval_window,
                      constraint.identifier,
                      constraint.xpath,
                      temporal_document)
        if ¬ValidateSpecificConstraint(
                constraint, results) then
            return false
        end
    end
end
return true
```

Algorithm 3: Checking Non-Sequenced Constraints

```
ExtractNodes(evaluation_window, identifier,
             xpath_query, temporal_document):
result ← new map
candidate_nodes ← XPath.Evaluate(
    identifier, xpath_query, temporal_document)
candidate_nodes ← FilterNodesbyEvalWindow(
        evaluation_window, candidate_nodes)
foreach can_node in candidate_nodes do
    result.add(can_node.identifier, can_node.value)
end
return result
```

Algorithm 4: Extracting Related Nodes for Non-Sequenced Constraints

Algorithm 3 validates all the non-sequenced constraints. For each constraint, its evaluation windows are first computed based on (i) the period of the temporal document, (ii) the evaluation window size (the length of each period in the temporal document during which this constraint applies), and

(iii) the slide size (the distance between the begin times of successive evaluation windows).

$ExtractNodes$ extracts only the nodes from the temporal documents that are relevant to a constraint, within each evaluation window. We discuss this routine shortly. $ValidateSpecificConstraint$ is then utilized to examine whether the extracted nodes violate the constraint. This routine, which checks all four types of non-sequenced constraints, is straightforward. For a cardinality constraint, the routine groups each distinct key and accumulates the count of occurrences of each key. Similarly, for a unique constraint, distinct keys are grouped. But in this case, if a key's count is more than one, this constraint is considered to be violated. In checking a referential constraint, the routine evaluates the XPath expression of the conventional constraint that is referenced by the temporal constraint. The existence of the values of the nodes that are being checked is examined against the values returned from evaluating the conventional constraint. Finally, in verifying a datatype constraint, each pair of consecutive values specified by the constraint are examined to determine whether the value transition rules are violated.

To provide the routine $ValidateSpecificConstraint$ with the proper input, routine $ExtractNodes$ (Algorithm 4) evaluates XPath expressions specified by the constraints and filters the document to produce the *items*, each identified by an *identifier*, to be validated by the non-sequenced constraints. As mentioned above, this algorithm performs XPath evaluation and document content filtering, returning a mapping from identifiers to the values associated with each identifier. By evaluating the XPath expressions from the constraints, a set of candidate nodes is extracted from the temporal document. These nodes are then processed to retain only the nodes in the current evaluation window. For each *candidate_node*, the distinct *identifiers* are grouped and stored in the *result* variable.

The complexity of Algorithm 4 is determined by the number of candidate nodes n in the document as well as the number of time points m. The overall complexity is thus O($n \cdot m$). The complexity of Algorithm 3 is determined primarily by the number of evaluation windows l as well as the complexity of Algorithm 4, and is thus O($l \cdot n \cdot m$).

This worst case behavior is consistent with the experimental results given in Section 9.3. Concerning Figure 5, for sequenced constraints, the number of nodes (n) and the number of evaluation windows (l) are both fixed, with the number of slices (m) varied on the x-axis, implying the observed linear growth in total execution time. Concerning Figure 6, for non-sequenced constraints, the number of nodes and number of evaluation windows are still fixed, except for cardinality constraints, which has a smaller window size, resulting in an increasing number of evaluation windows as the number of slices increases. Thus referential integrity and identity constraints exhibit linear behavior, while cardinality constraints exhibit quadratic behavior.

The implementation of τXMLLINT can be obtained from http://cgi.cs.arizona.edu/apps/tauXSchema/.

APPENDIX
II CONSTRAINTS USED IN EVALUATION

During our evaluation, we used the following three non-sequenced constraints. To produce fair execution time results, when we evaluated one of the constraints, we deactivated the other two in the annotations document.

```
<!-- Non-sequenced Identify Constraint: -->
<!-- Item IDs are unique for books and may
<!-- not ever be re-used. -->
<item target="item">
  <nonSeqKey name="bookIDKey" dimension="validTime"
      evaluationWindow="36500">
    <selector xpath="." />
    <field xpath="@id" />
  </nonSeqKey>
</item>
```

```
<!-- Non-sequenced Referential Integrity: -->
<!-- A related item should refer to a valid
<!-- item (possibly not currently an item in print). -->
<item target="item">
  <nonSeqKeyref name="relatedItemRI" refer="itemID">
    <selector xpath="." />
    <field xpath="related_items//related_item//item_id" />
  </nonSeqKeyref>
  <itemIdentifier name="item_id"
      timeDimension="transactionTime">
    <field path="@id"/>
  </itemIdentifier>
</item>
```

```
<!-- Non-sequenced Cardinality: -->
<!-- In any calendar year, an item may have up
<!-- to 6 authors. -->
<item target="item">
  <nonSeqCardinality name="bookAuthorsNSeq" maxOccurs="6"
      dimension="validTime" evaluationWindow="365"
      slideSize="365">
    <selector xpath="." />
    <field xpath="authors//author/@author_id" />
  </nonSeqCardinality>
  <itemIdentifier name="item_id"
      timeDimension="transactionTime">
    <field path="@id"/>
  </itemIdentifier>
</item>
```

Additionally, we used the following four sequenced constraints. Again, to produce fair execution time results, when evaluating one of the constraints, we deactivated the other three in the schema.

```
<!-- Sequenced Cardinality: -->
<!-- An item must have between 1 and 4 authors. -->
<xs:element ref="author" minOccurs="1" maxOccurs="4"/>
```

```
<!-- Sequenced Identify Constraint: -->
<!-- Item ISBNs are unique. -->
<xs:unique name="ISBNUnique">
  <xs:selector xpath=".//item/attributes"/>
  <xs:field xpath="ISBN"/>
</xs:unique>
```

```
<!-- Sequenced Referential Integrity: -->
<!-- A related item should refer to a valid item -->
<xs:key name="itemID">
  <xs:selector xpath=".//item"/>
  <xs:field xpath="@id"/>
</xs:key>
<xs:keyref name="itemIDRef" refer="itemID">
  <xs:selector xpath=".//item/related_items/related_item"/>
  <xs:field xpath="item_id"/>
</xs:keyref>
```

```
<!-- Sequenced Datatype: -->
<!-- The number_of_pages must be of type short. -->
<xs:element name="number_of_pages" type="xs:short"/>
```