Incremental Validation of Key and Keyref Constraints

Sharon Krisher and Oded Shmueli
Technion - Israel Institute of Technology
CS Dept., Haifa 32000 Isreal
{shak, oshmu}@cs.technion.ac.il

Abstract. We suggest simple update operations on XML documents. These operations may change the value of a simple-type node, or change several values transactionally. We present efficient algorithms for checking the validity of such operations with respect to key and keyref constraints (XML Schema identity constraints). We discuss the implementation of the algorithms for checking the validity of single/multiple value changes, and present experiments that show its superiority to validation from scratch.

1 Introduction

One of the useful mechanisms provided by XML Schema [2] is the ability to define identity constraints, including keys and foreign keys.

A key definition appears inside an element definition. This element is called the scope of the key. The key definition imposes constraints on the sub-tree of the scoping element. It looks as follows.

```xml
<xs:key name="KeyName">
  <xs:selector xpath=XPATH_EXPRESSION/>
  <xs:field xpath=XPATH_EXPRESSION/>
  ... [possibly more fields]
</xs:key>
```

The key definition includes a selector expression and one or more field expressions. These are expressions over a simple fragment of XPath [1], called "restricted XPath". They do not contain predicates, and in each path the first location step may be "/", but the other steps may only be 'self' or 'child' steps. Also, for a field expression the path may end with an attribute. The selector expression is evaluated, with an instance s of the scoping element as a context node, to produce a set of nodes which is called (in the XML Schema standard) the target node set of s. For each node in the target node set, every field expression must evaluate (relative to the node) to a node set containing exactly one node, of a simple type. Within an instance of the scoping element, there must not exist two distinct nodes of the target node set that have the same sequence of field values. Let K be a key, defined within the definition of an element e, with selector expression Sel and field expressions f1, ..., fm. A document
$D$ is said to satisfy $K$ if and only if for every instance $n$ of $e$ in $D$, the following hold. Let $S$ be the set of nodes obtained from evaluating $Sel$ in the context of $n$ ($S = Sel(n)$). Then

- For each $x \in S$ and for each $f_i$, $i = 1..m$, $f_i$ evaluates to a single, simple-type node in the context of $x$.
- For each $x_1, x_2 \in S$, if $f_i(x_1) = f_i(x_2)$ for each $i = 1..m$ then $x_1$ and $x_2$ are the same node.

A keyref definition is very similar to a key definition. It appears within the definition of a scoping element and specifies selector and field expressions. It looks as follows.

```xml
<xs:keyref name="KeyRefName" refer="KeyName">
   <xs:selector xpath="XPATH_EXPRESSION"/>
   <xs:field xpath="XPATH_EXPRESSION"/>
   ... [possibly more fields]
</xs:key>
```

The "refer" attribute specifies the name of the key constraint that this keyref constraint refers to. Let $n$ be an instance of the scoping element of a keyref. For each node $u$ in the target node set of $n$, there must exist a node $v$, in some target node set of the referred key, that has the same sequence of field values. The exact semantics is explained below.

Observe, for example, the document depicted in Fig. 1(a). We refer to nodes by the names written inside the circles. Node tags are written next to them. Consider a key constraint whose scoping nodes are the $B$ nodes, whose selector is $(./C | ./B/C)$ and whose fields are ./f and ./g. It ensures that within a $B$ node, there are no two child or grandchild $C$ nodes with the same combination of $f$ and $g$ values (i.e., in this document, $c_1$ and $c_3$ must have unique $(f, g)$ values, because they are a child and a grandchild of $b_1$). Consider a keyref constraint (that refers to this key) whose scoping nodes are the $B$ nodes, whose selector is ./E and whose fields are ./f and ./g. It ensures that if an $E$ node appears as a child of some $B$ node then there is some $C$ node, within the scope of this $B$ node or one of its descendant $B$ nodes, that has the same $f$ and $g$ values as the $E$ node.

XML Schema keys are similar to "relative keys", defined in [4]. A relative key is defined by $(Q, (Q', S))$, where $Q$ and $Q'$ are path expressions (similar to restricted XPath expressions) and $S$ is a set of path expressions. The analogy to XML Schema keys is as follows. $Q$ selects the scoping nodes (instances of the scoping element) of the key. $Q'$ is the selector expression, and $S$ is the set of field expressions. Defining the scope by using a path expression $Q$ does not affect the algorithms that we present (since we can easily find the scoping nodes by evaluating $Q$ instead of finding the instances of the scoping element). However, there are significant differences in the semantics of XML Schema keys and relative keys.
In an XML Schema key, a field expression must evaluate to a set that contains exactly one node, and the node must be of a simple type. In a relative key, the set may contain any number of nodes (and may be empty), and these nodes may be of any type.

An XML Schema key is violated if there are two distinct target nodes \( t_1, t_2 \) within the same scoping node such that for every field expression \( f_i \), \( f_i(t_1) = f_i(t_2) \). A relative key is violated if there are two distinct target nodes \( t_1, t_2 \) within the same scoping node such that for every field expression \( f_i \), \( f_i(t_1) \cap f_i(t_2) \neq \emptyset \). Equivalence of nodes is determined according to their string values.

Our algorithms can be adapted to support relative keys instead of XML Schema keys. The main difference would be in the way that we store and compare key-sequences (sequences of field values).

**Basic Concepts**

Let \( K \) be a key defined within the definition of an element \( e \) in a schema \( S \). Let \( KSel \) be the selector expression of \( K \). Let \( KField_1, ..., KField_k \) be the field expressions of \( K \). Let \( D \) be a document.

1. The instances of \( e \) in \( D \) are called the *scoping nodes* of \( K \).
2. Let \( n \) be a scoping node. Let \( S_n \) be the set of nodes which is the result of evaluating \( KSel \) in the context of \( n \) (\( S_n \) is called the "target node set" in the XML Schema standard). Each \( x \in S_n \) is a *selector-identified node* (of \( n \) and \( K \)). A selector-identified node \( x \) of a key \( K \) may have several scoping nodes \( n_i \). In this case we can say that \( x \) is a selector-identified node of \( K \), and we can also say that \( x \) is a selector-identified node of \( K \) and \( n_i \), for each such \( n_i \). So, \( x \) is a selector-identified node if and only if \( x \) belongs to the target node set of at least one scoping node.
3. Let \( x \) be a selector-identified node of \( K \). If a node \( f \) is returned when evaluating a field expression in the context of \( x \) then we call \( f \) a *field* of \( x \). We call the sequence of values of the nodes returned when evaluating \( KField_1, ..., KField_k \) in the context of \( x \) the *key-sequence* of \( x \). These terms are also used for keyrefs.

The semantics of foreign key references, as described in [3], is complex\(^1\):

- These references are local to a scoping node of the keyref. Suppose \( n' \) is a selector-identified node of a keyref scoping node \( n \). Then \( n' \) is referencing nodes that are selector-identified nodes of the key (and have the same field values as \( n' \)), and whose scoping node is either \( n \) or a descendant of \( n \).
- In a valid document, every selector-identified node of a keyref references (within a scoping node) exactly one selector-identified node of a key. To ensure this, there is a mechanism that resolves conflicts. Let \( n \) be a scoping node of a keyref \( KR \) that refers to a key \( K \). There is a table, associated with \( n \), which holds \( K \)'s selector-identified nodes \( u \), in the subtree rooted at \( n \), that may be referenced by \( KR \)'s selector-identified nodes whose scoping node is \( n \). For each such node \( u \), the table holds \( u \)'s key-sequence (i.e., the values of its fields). Such tables are in fact associated with all nodes. They can be created in a bottom-up traversal of the tree. For a node \( n \), we create the table as follows. If \( n \) is a scoping node of the key, we add its selector-

\(^1\) And is not apparent at first reading.
identified nodes to the table. Then, we compute the union of the tables of n’s children, and add it to n’s table. If the combined table contains two or more rows with the same key-sequence ks (and different nodes), this is considered a conflict. The conflict is resolved as follows. All nodes with key-sequence ks that were added from the children’s tables are removed. Note that this may result in adding no entry for ks.

Given a document that conforms to a schema that defines identity constraints, it is important to be able to efficiently check whether changing the document violates the constraints. We define two update operations on XML documents. One enables changing the value of a single node and the other enables changing the values of several nodes transactionally. We assume that these operations are executed on a document D that is valid with respect to an XML Schema S that defines a key constraint K and a keyref constraint KR. For simplicity, we assume one key and one keyref constraint. Our algorithms can be easily extended to handle multiple key and keyref constraints. We denote the sizes (in terms of the number of characters in the text representation) of the document and the schema by |D| and |S|, respectively. We assume that the document has an in-memory DOM-like representation [5] (that uses Node objects), and that every node in the document has a unique node id (i.e., node identifier). We present algorithms that validate update operations with respect to the key and keyref constraints that are defined in the schema. Given a document, we create data structures that reflect the state of the document with respect to K and KR. As update operations are executed, these data structures are maintained accordingly.

Related work There has been work regarding incremental validation of XML documents. But, we have not encountered work on incremental validation of XML Schema key and keyref constraints. Also, we have not encountered work regarding incremental transactions (i.e., incremental validation of a set of changes, that are to be executed in a transaction).

In [7], insertion and deletion operations are defined. XML Schema is not used, but rather a certain kind of tree automata. Key and keyref constraints are not handled. The methods discussed in [7] are expanded in [8] to support key and foreign key constraints. These constraints are not the ones defined in XML Schema, although they are similar. Unlike the constraints of XML Schema, these constraints include a path expression that navigates to scoping nodes (which are called ‘context nodes’), and a foreign key constraint must have the same context node path as the key constraint that it refers to. These constraints do not have the complex “percolation” semantics of XML Schema key and keyref constraints. The context paths have only child steps and therefore scoping nodes cannot be descendants of each other. For validation from scratch, a kind of tree automaton is used, and field values are carried up the tree in a bottom up traversal. Algorithms are presented for incremental validation of insertion and deletion of a subtree. The simplified semantics of these constraints enables relatively simple algorithms. From the point of insertion or deletion, p, the algorithm simply finds the single context node p’ and performs checks on this node; Deletion of a
key node that is referenced is disallowed (because, due to the simple semantics, there is no possibility that after the deletion a keyref node will reference a different node). During initial validation, a data structure is created. This structure keeps all context nodes of the key, for each one it keeps its target nodes and their key-sequences, and a reference-count for each target node.

In [11], insertion and deletion operations are also defined. The updates are validated with respect to several classes of DTDs. Key and keyref constraints are not handled, only structural and attribute constraints. In [10], operations are defined for replacing a label of a node, inserting a new leaf node and deleting a leaf node from the document. DTDs and specialized DTDs are handled, but neither key nor keyref constraints are not handled.

In [14], a logic based language for defining constraints is introduced. The constraints are very expressive, but no data structures and/or complexity results are given.

Incremental validation, of insertion and deletion of sub-trees, with respect to key constraints, is introduced in [9]. These key constraints are similar to the key constraints of XML Schema, but not identical. Their definition includes an XPath expression that navigates to the scoping nodes, which is not the case in XML Schema keys (in which no such expression is given, and the scoping nodes are the instances of the element in which the key constraint is defined). Also, it allows a selector-identified node to have several values for each field (the key is violated if there are two nodes that share at least one value for each field). Validation is done by using an index structure. The first level of the index is a key constraint. The second is a scoping node (called ”context node” here). The third is a ”key path”, which is a path that appears in a field of the key constraint. The fourth level is the ”key value” (which is a value to which the ”key path” evaluates, for at least one target node). Target nodes are grouped, according to the ”key values”, into equivalence classes called ”key value sharing classes” (KVSC). That is, for every key value (that corresponds to a specific key path, within a specific scoping node of a specific key constraint), the KVSC is a set of target nodes (selector-identified nodes of the scoping node) that have the same key value for this key path. When a new target node $t$ is added to a scoping node, the validator checks if there is an existing target node $t'$ such that $t$ shares some key value with $t'$ for every key path. This is done as follows. For each key path $P_i$ of the key, the union $S_i$ of the KVSCs that $t$ belongs to (for $P_i$ ) is computed. Let $S$ be $S_1 \cap \ldots \cap S_p$, where $p$ is the number of key paths of the key. The key is violated if $S$ contains more that one node. Keyref constraints are not handled in [9]. Note that while the index structure introduced in [9] facilitates efficient validation of key constraints, it does not come close to containing the information needed to validate keyref constraints, and it is not apparent how (and if) it can be easily extended to support the semantics of keyref constraints. This is due to the complex semantics of keyref references, described above.
2 Data Structures

The following data structures are the basis for our incremental algorithms. We have the global data structures \textit{KeySelIdent}, \textit{KeyRefSelIdent} and \textit{FieldInfo}, and also data structures associated with nodes: \textit{KeyInfo}, \textit{ChildrenKeyInfo} and \textit{KeyrefInfo}.

**Motivation for maintaining the data structures**

The data structures are designed to enable efficient monitoring as the document changes. The \textit{FieldInfo} structure enables us to know which selector-identified nodes are affected by changing a simple-type node, i.e., which key-sequences change as the value of the node changes. The \textit{KeySelIdent} and \textit{KeyRefSelIdent} structures allow us to check whether a given node is a selector-identified node, and if so, of which scoping nodes. They also allow us to keep the key-sequence of each selector-identified node only once, even if the node has several scoping nodes.

For a keyref scoping node \( n \), the \( n.\textit{KeyrefInfo} \) structure is quite straightforward - it holds \( n \)'s selector identified nodes and their key-sequences. That way, if the key-sequence of one of these nodes changes, or if the key-sequence of a referenced node changes, we do not have to re-calculate these key-sequences, but rather update them (if needed), and check the validity of the references.

The \( x.\textit{KeyInfo} \) structures (for each node \( x \)) are needed in order to check the validity of references. If \( x \) is a keyref scoping node then, in order for the document to be valid, each key-sequence in \( x.\textit{KeyInfo} \) must also appear in \( x.\textit{KeyInfo} \). \( x.\textit{KeyInfo} \) is also important if \( x \) is a key scoping node, since we can make sure that, following a change in the document, there are no duplicate key-sequences in \( x.\textit{KeyInfo} \) (if there are, then the key is violated). \( x.\textit{KeyInfo} \) is maintained even if \( x \) is not a key or keyref scoping node, because the content of a node's \textit{KeyInfo} structure is affected by the \textit{KeyInfo} structures of the node's children (as described in the definition of the \textit{KeyInfo} structure, and dictated by the semantics of XML Schema constraints).

The \( x.\textit{ChildrenKeyInfo} \) structures (for each node \( x \)) allow us to easily update the \textit{KeyInfo} structure of a node \( x \) following an update to one or more of its children's \textit{KeyInfo} structures. For each key-sequence \( ks \) that appears in the \textit{KeyInfo} structure of at least one child of \( x \), \( x.\textit{ChildrenKeyInfo} \) tells us in which children's \textit{KeyInfo} structures \( ks \) appears, and to which selector-identified nodes it belongs. Thus, when the entry for \( ks \) in some child changes, we can update the entry for \( ks \) in \( x.\textit{KeyInfo} \) according to the current states of \( x.\textit{ChildrenKeyInfo} \) and \( x.\textit{KeyInfo} \). Basically, \( x.\textit{ChildrenKeyInfo} \) saves us the trouble of having to check the \textit{KeyInfo} structures of all of \( x \)'s children when the structure of one of the children is updated.

**Logical Data Structures**

- \textit{KeySelIdent}: Contains a record for every selector-identified node of the key\(^2\)
  
  \( K \). The record contains the node (or, more precisely, a reference/pointer to

\(^2\) In order to keep information regarding several key constraints, we would have a structure \textit{KeySelIdent}[\textit{key}] for every key constraint \textit{key}.  


the node), its key-sequence (a tuple of values), its scoping nodes\(^3\) and also its lowest and highest scoping node (i.e., the first and last node if we sort the scoping nodes of this node according to height).

- **Keyref SelIdent:** Similar to KeySelIdent, but for the keyref KR.

- **FieldInfo:** A search tree over node identifiers. For every node \(x\), we keep references to the records of KeySelIdent where \(x\) is a field of the selector-identified node, and also references to the records of Keyref SelIdent where \(x\) is a field of the selector-identified node. For each such record we also keep the position of the field in the corresponding key-sequence.

- \(x.\text{KeyInfo}\): For a node \(x\), \(x.\text{KeyInfo}\) contains records of the form \((n, ks, isSelectorIdentified)\), where:
  - \(n\) is a selector-identified node of the key \(K\) that is a descendant of \(x\), or \(x\) itself (in case \(x\) is both a scoping node and a selector-identified node of itself).
  - \(ks\) is the key-sequence of \(n\).
  - \(isSelectorIdentified\) is a Boolean value.

\(x.\text{KeyInfo}\) is the union of two sets. One set addresses the case where \(x\) is a scoping node of \(K\). The other set addresses the case where \(x\) may or may not be a scoping node of \(K\). It contains only records whose key-sequences do not appear in records of the first set, and each such key-sequence appears in a record of the \(\text{KeyInfo}\) structure of exactly one child of \(x\). Formally, 
\[
x.\text{KeyInfo} = \{(n, ks, True) \mid x\ \text{is a scoping node of} \ K, \ n\ \text{is a selector-identified node with key-sequence} \ ks\ \text{in the scope of} \ x\} \cup \{(n, ks, False) \mid \text{There is no selector-identified node of} \ K\ \text{with key-sequence} \ ks\ \text{and scope} \ x, \ \text{there is a child} \ z\ \text{of} \ x\ \text{and a Boolean} \ b\ \text{such that} (n, ks, b) \in z.\text{KeyInfo} \text{and there is no child} \ z' \neq z\ \text{and a Boolean} \ b'\ \text{such that} (n, ks, b') \in z'.\text{KeyInfo}\}\.
\]

Note that if \(x\) is a scoping node of \(K\), records with \(isSelectorIdentified = True\) contain the selector-identified nodes for which \(x\) is a scoping node. The key constraint dictates that there are no two such records with the same key-sequence. If \(x\) is a scoping node of a keyref that refers to \(K\), a reference to a key-sequence \(ks\) is only valid if there is a record with this key-sequence in \(x.\text{KeyInfo}\) (regardless of its \(isSelectorIdentified\) value). Thus, the \(\text{KeyInfo}\) structures are mainly used to verify foreign key references.

The definition of \(x.\text{KeyInfo}\) seems somewhat complicated. This is due to the conflict resolution mechanism which is defined in the XML Schema standard and discussed in Section 1.

- \(x.\text{ChildrenKeyInfo}\): For a node \(x\), it contains an entry for every key-sequence that appears in the \(\text{KeyInfo}\) structure of at least one child of \(x\). In the entry for a key-sequence \(ks\) there is a set of tuples of the form \((\text{child}, \text{nodeInTable})\), where \(\text{child}\) is a child node of \(x\) and \(\text{child.\text{KeyInfo}}\) contains a record \((\text{nodeInTable}, ks, b), b \in \{\text{True, False}\}\). This information helps in updating the \(\text{KeyInfo}\) structures when the document is modified. Let \(x.\text{ChildrenKeyInfo}[ks]\) denote the entry for a key-sequence \(ks\). It is considered as \text{null} if there is no such entry. We observe:

\(^3\) A node may be selector-identified by more than one scoping node.
• If a tuple \((z, n)\) appears in \(x.\text{ChildrenKeyInfo}[ks]\) it means that \(n\) is a valid candidate to appear in \(x.\text{KeyInfo}\), i.e., it 'survived' competition from other nodes in the sub-tree of \(z\) and appears in \(z.\text{KeyInfo}\).

• If \(|x.\text{ChildrenKeyInfo}[ks]| = 1\) and either \(x\) is not a scoping node of \(K\), or \(x\) is a scoping node of \(K\) but has no selector-identified node with key-sequence \(ks\), then the 'candidate' that appears in \(x.\text{ChildrenKeyInfo}[ks]\) also appears in \(x.\text{KeyInfo}\) (with \(\text{isSelectedIdentified} = \text{False}\)).

- \(x.\text{KeyrefInfo}\): For a scoping node \(x\) of the keyref \(KR\), \(x.\text{KeyrefInfo}\) contains all records of the form \((n, ks)\) where \(n\) is a selector-identified node (of \(KR\)) in the scope of \(x\) and \(ks\) is the key-sequence of \(n\). For non \(KR\)-scoping-nodes, \(x.\text{KeyrefInfo}\) is \(\text{null}\).

The document uniquely determines the content of these structures. The incremental algorithm needs to ensure this by making the necessary updates to the structures as update operations are performed on the document.

In addition to the aforementioned data structures, for every node \(x\) we keep a set of key-sequences \(x.\text{RemovedSequences}\). If an algorithm updates \(x.\text{KeyInfo}\), \(x.\text{RemovedSequences}\) is the set of key-sequences that appear in \(x.\text{KeyInfo}\) prior to the update and do not appear there following the update. After updates to the \(\text{KeyInfo}\) structures are performed, for every node \(x\) which is a keyref scoping node we need to verify that there are no nodes that reference the key-sequences in \(x.\text{RemovedSequences}\). The \(\text{RemovedSequences}\) sets are only used during the execution of certain update algorithms and are initialized as empty when such an algorithm is executed. Fig. 1(b) shows a portion of\(^4\) the data structures for the document depicted in Fig. 1(a) (with a key constraint whose scoping nodes are the \(B\) nodes, whose selector is \(./C \mid ./B/C\) and whose fields are \(./f\) and \(./g\), and a keyref constraint whose scoping nodes are the \(B\) nodes, whose selector is \(./E\) and whose fields are \(./f\) and \(./g\)).

Physical Data Structures

In order to enable efficient searches according to key-sequences in the data structures, we keep the \(\text{KeyInfo}\), \(\text{ChildrenKeyInfo}\) and \(\text{KeyrefInfo}\) structures in the following physical structure. The structure is a search tree over the values of the first field in a key-sequence. Each leaf is associated with a search tree over the values of the second field, and so on. So, in a leaf of value \(v\) in the first tree, there is a tree that holds information only for key-sequences where the value of the first field is \(v\). A search in this multi-search-tree allows access to the actual records that are stored in it. Note that in \(x.\text{KeyInfo}\), several records may be stored for every key-sequence (of selector-identified nodes that have this key-sequence).

Moreover, we maintain a search tree over node identifiers for each \(\text{KeyInfo}\) and \(\text{KeyrefInfo}\) structure, in order to efficiently search for records according to the identifier of a selector-identified node. Also, for a node \(y\) and a key-sequence \(ks\), the set of nodes \(y.\text{ChildrenKeyInfo}[ks]\) is maintained as a search tree over the node identifiers of \(y\)'s children, in order to efficiently access the tuple stored in \(y.\text{ChildrenKeyInfo}[ks]\) for a specific child \(x\), if such a tuple exists.

\(^4\) The \(\text{KeyInfo}\) structures that are not shown contain no records.
We use a succinct representation of strings as integers based on a TRIE structure. Using this representation, the time to compare two key-sequences is \( \text{number of fields} \times \text{time to compare integers} = O(\left| S \right| \log N) = O(\left| S \right| \log |D|) \), where \( N \) is the number of text nodes in the document\(^5\). For practical purposes, we assume that comparing two integers is done in \( O(1) \), and then the time to compare key-sequences is\(^6\) \( O(\left| S \right|) \).

In Section A of the appendix, we explain how key and keyref constraints can be validated from scratch.

3 Changing the value of a simple-type node

We define an update operation \( \text{update}(f, \text{newval}) \) where \( f \) is some simple-type node and \( \text{newval} \) is the value to be assigned to it. We assume that \( \text{newval} \) is different than the current value of \( f \), otherwise the change has no effect.

The idea behind the algorithm: Since selector expressions do not include predicates, changing the value of a simple-type node can only change the key-sequences of existing selector-identified nodes (and cannot change the sets of selector-identified nodes). These nodes can be found via lookup in the \text{FieldInfo} structure. Since the selector and field expressions are restricted XPath expressions, we know that the affected selector-identified nodes, and their scoping nodes, are all on the path from the root to the changed simple-type node. Thus, we need to traverse this path, bottom-up, and update the data structures

\(^5\) Which is no more than the size of the document.
\(^6\) We use \( |S| \) here as an upper bound on the number of fields in a key-sequence.
(KeyInfo, ChildrenKeyInfo and KeyrefInfo) associated with the nodes along the path. The result of updating the KeyInfo structure of a node \( x \) serves as input for updating the ChildrenKeyInfo structure (and subsequently the KeyInfo structure) of its parent \( y \). For every key scoping node along the path, we check that the key is not violated. That is, if \( s \) is a key scoping node, \( n \) is a selector-identified node of \( s \), and the key-sequence of \( n \) changes from \( ks \) to \( ks' \), we check that the key-sequence \( ks' \) does not already appear in \( s.KeyInfo \). For every keyref scoping node along the path, we check that the keyref is not violated. That is, if \( s \) is a keyref scoping node, for every key-sequence in \( s.KeyrefInfo \) that changes we check that the new key-sequence appears in \( s.KeyInfo \), and for every key-sequence that is removed from \( s.KeyInfo \) we check that it is not referenced in \( s.KeyrefInfo \).

3.1 The Algorithm

**Input:** A schema \( S \), a document \( D \) (represented in memory), a node \( f \) and a value \( newval \).

**Output:** A result - VALID or INVALID.

**Pre-conditions:** \( D \) is valid with respect to \( S \). \( f \) is a simple-type node in \( D \). The data structures corresponding to \( D \) (as described in Section 2) have been created, and are correct (i.e., reflect the state of \( D \)).

**Post-conditions:** The data structures are correct. If the result is INVALID then the document is unchanged (identical to the input document). If the result is VALID, the value of \( f \) is \( newval \) and the document is otherwise unchanged.

In order to simplify the description of the algorithm, we assume that when the algorithm determines that the update is invalid, it performs a rollback of all changes to the data structures, and exits with output INVALID. We indicate this by writing exit(INVALID) in the pseudo-code. The algorithm is depicted in Fig. 2. It consists of two stages. In the first stage we find which selector-identified nodes of \( K \) and \( KR \) are affected by the update. In the second stage we traverse the path from the changed node to the root, and update the data structures of each node. Since selector expressions do not include predicates, changing the value of a simple-type node cannot change the sets of selector-identified nodes (it only changes the key-sequences of the selector-identified nodes).

```plaintext
Input: Schema \( S \), document \( D \) with corresponding data structures, node \( f \), value \( newval \).
Output: result - VALID or INVALID.

Compute \( K\)Updates and \( K\)RUpdates;
UpdateNodes(K\)Updates, K\)RUpdates, f);
// If we got here without exiting with output INVALID, then everything is ok.
return VALID.
```

**Fig. 2.** Algorithm for changing the value of a simple-type node.

```plaintext
Input: K\)Updates, K\)RUpdates, f) 
Output: None.

currentNode = f;
currentChild = null;
changesInChild = \emptyset;
while (currentNode \neq null) {
    changesInNode = UpdateNode(currentNode, currentChild, changesInChild, K\)Updates);
    UpdateNode(currentNode, K\)RUpdates);
    currentChild = currentNode;
    changesInChild = changesInNode;
    currentNode = currentNode.parent;
}
```

**Fig. 3.** UpdateNodes.
1. Finding affected selector-identified nodes. We search for $f$ in $FieldInfo$ to determine which selector-identified nodes are affected by the update, i.e., nodes for which $f$ is a field. We also update the key-sequences stored in the relevant records of $KeySelIdent$ and $KeyRefSelIdent$. Following these searches we have a set of key-sequence updates of the form (node, old key-sequence, new key-sequence) for $K$ and for $KR$. We call these $KUpdates$ and $KRUpdates$. A node may appear only once in $KUpdates$ (respectively, $KRUpdates$). All the nodes that appear in $KUpdates$ (respectively, $KRUpdates$) are on the same path from the root to $f$, since they are all ancestors of $f$.

2. Updating the data structures of nodes along the path to the root. This stage is executed in the procedure $UpdateNodes(KUpdates, KRUpdates, f)$ as depicted in Fig. 3. This procedure, which is the heart of the algorithm, uses the function $UpdateNode_K$ and the procedure $UpdateNode_KR$, which we now describe, in order to update the data structures associated with a single node.

$UpdateNode_K(y, x, changesInChild, KUpdates)$
This function updates $y.KeyInfo$ and $y.ChildrenKeyInfo$, and also inserts the appropriate key-sequences into $y.RemovedSequences$. These updates are done based on the set of key-sequence changes $KUpdates$ and on the changes made to the $KeyInfo$ structure of y’s child node $x$. These changes are passed in the set $changesInChild$. These are tuples of the form $(ks, n)$, where $ks$ is a key-sequence whose record in $x.KeyInfo$ has changed\(^7\) and $n$ is the node that appears in the new record for $ks$ in $x.KeyInfo$ (if there is one). If there is no such new record (i.e., a record has been removed), $n$ is $null$. A key-sequence may only appear once in $changesInChild$. The function $UpdateNode_K$ returns the set of the changes it has made (in the same format as $changesInChild$), which is later used for updating the parent of $y$. We denote this set by $changes$.

It is initialized to $\emptyset$ and is returned as the result at the end of the function execution. Since a key-sequence $ks$ can only appear once in $changes$, we use the syntax $changes[ks] := n$ to denote that addition of the tuple $(ks, n)$ to $changes$. This means that if a tuple $(ks, n')$ already exists in $changes$, it is replaced by $(ks, n)$. The function is depicted in Fig. 4. It uses the functions $ProcessChanges$, $HandleOldKeySequences$ and $HandleNewKeySequences$, depicted in Figs. 6, 7 and 8, respectively.

For example, suppose that the value of $c3.f$ in the document of Fig. 1(a) is changed from 1 to 5. In this case, $c3$ is the only affected selector-identified node. $KUpdates = \{(c3, (1, 2), (5, 2))\}$. $b3.KeyInfo$ is updated with the new key-sequence of $c3$. This change is passed on to the execution of $UpdateNode_K$ on $b1$, i.e., $changesInChild = \{((1, 2), null), ((5, 2), c3)\}$. As $changesInChild$ is processed, $b1.ChildrenKeyInfo$ is updated. Furthermore, the record $(c3, (5, 2), False)$ is added to $b1.KeyInfo$. The record $(c3, (1, 2), True)$ is not yet removed from $b1.KeyInfo$ (since it appears with $True$). Then, $KUpdates$ is processed. Since

\(^7\)I.e., if $r_1$ is the record for $ks$ in $x.KeyInfo$ before $UpdateNode_K(x,...)$ is executed (possibly $r_1 = null$), and $r_2$ is the record for $ks$ in $x.KeyInfo$ after $UpdateNode_K(x,...)$ is executed (possibly $r_2 = null$), then $r_1 \neq r_2$. 

11
c3 is a selector-identified node of b1, \((c3, (1, 2), (5, 2)) \in RelevantUpdates\). As a result, the records \((c3, (1, 2), True)\) and \((c3, (5, 2), False)\) are removed and the record \((c3, (5, 2), True)\) is added to \(b1.KeyInfo\).

**UpdateNode\(_K\)\(_R\)**

This procedure updates the \(Keyref\_Info\) structure of a node according to the received key-sequence changes (\(KR\_Updates\)) and also checks whether all references are valid. It is depicted in Fig. 5.

### 3.2 Complexity and Correctness

There are \(O(h)\) tuples in \(K\_Updates\) and \(KR\_Updates\) (where \(h\) is the height of the document tree), and we update the data structures of \(O(h)\) nodes. All lookups are done in search trees (or multi-search-trees) of size at most \(O(|D|)\). Therefore, the complexity of the algorithm is \(O(|S|h^2\log|D|)\). A reasonable assumption is that \(h = O(\log|D|)\), and then the complexity is \(O(|S|\log^3|D|)\). In the worst case, \(h = O(|D|)\) and the complexity is \(O(|S||D|^2\log|D|)\). Note that in most real-world cases, a simple-type node serves as a field of only one selector-identified node, and then the complexity is only \(O(|S|\log|D|)\). Also note that since our data structures use search trees, searching within them takes logarithmic time. Using hash tables, the \(\log|D|\) factors may be replaced by expected \(O(1)\) time.

Theorem 1, based on the Lemmas in Section B of the appendix, establishes correctness.

**Theorem 1.** If the update violates \(K\) or \(KR\) then this violation is discovered during the execution of the algorithm. If the update does not violate \(K\) or \(KR\) then the state of the data structures, after the execution of the algorithm, corresponds to the state of the document after the update. \(\square\)

---

\(^8\) \(O(h^2\log|D|)\) for a fixed schema.
Input: A node \(y\), a child node \(x\) of \(y\), a set of changes in \(x\), \(changesInChild\), changes in \(y\) so far - \(changes\) (passed by reference).
Output: None.

\[
\text{ProcessChanges}(y, x, changesInChild, ByRef changes) \{
\text{for each } (ks, n) \in changesInChild \{ \\
\text{if the set } y.ChildrenKeyInfo[ks] \text{ contains a tuple } (x, n') \\
\text{// We know that the entry for } ks \text{ in } x.KeyInfo \text{ has changed,} \\
\text{// which means that the tuple } (x, n') \text{ in } y.ChildrenKeyInfo[ks] \\
\text{// is not up to date.} \\
\text{remove this tuple;} \\
\text{if } n \neq \text{null} \\
\text{// if } n \text{ is null, it means that there is now no entry for } ks \text{ in } x.KeyInfo \\
\text{// and thus no reason to add to } y.ChildrenKeyInfo[ks]. \\
\text{// This is not the case here...} \\
\text{add the tuple } (x, n) \text{ to } y.ChildrenKeyInfo[ks]; \\
\text{if } \exists z \text{ such that } (z, ks, True) \in y.KeyInfo \{ \\
\text{if } \exists z \text{ such that } (z, ks, False) \in y.KeyInfo \{ \\
\text{// Here we remove the old record.} \\
\text{// Later we will check if a (different) record should be added} \\
\text{// according to } y.ChildrenKeyInfo[ks], \text{...} \\
\text{remove } (z, ks, False) \text{ from } y.KeyInfo; \\
\text{changes[ks]} := \text{null;} \\
\} \\
\text{if } y.ChildrenKeyInfo[ks] \text{ has exactly one tuple } (z, nodeInTable) \{ \\
\text{add the record } (nodeInTable, ks, False) \text{ to } y.KeyInfo; \\
\text{changes[ks]} := \text{nodeInTable;} \\
\} \\
\} \\
\}
\]

Fig. 6. \text{ProcessChanges}.

4 Transactions - Multi Simple-value Updates

We define an update operation \(update((f_1, newval_1), ..., (f_m, newval_m)), m > 1\), where for \(0 \leq i \leq m\), \(f_i\) is some simple-type node and \(newval_i\) is the value to be assigned to it. Note that simply doing these updates in order by using the algorithm of Section 3 is wrong, as an INVALID update may be 'corrected' by a later one (that is, performing only the first update leaves the document in a temporary invalid state). As we present the algorithm, we demonstrate it on the document depicted in Fig. 1(a). Recall the definitions of key and keyref constraints for this document, presented in Section 1. The scoping nodes of the key are the \(B\) nodes. The selector of the key is \((./C \mid ./B/C)\) and the fields are \(./f\) and \(./g\). The scoping nodes of the keyref are the \(B\) nodes. The selector of the keyref is \(./E\) and the fields are \(./f\) and \(./g\). The relevant KeyInfo, \(ChildrenKeyInfo\) and \(KeyrefInfo\) structures are depicted in Fig. 1(b). We perform the following update: \((c.e.f,6), (c1.f,6), (c2.f,5), (c3.f,5)\).

The idea behind the algorithm: As in the case of a single change, we can find the affected selector-identified nodes according to the FieldInfo structure. The difference is that we have to consider all value changes when we calculate the new key-sequence for an affected selector-identified node. From every changed node, we begin to move up the tree and update the structures associated with nodes. As we progress along the path from a changed node \(f_{i_1}\) to the root, we can update the structures as in the single-change case, as long as we don't reach a node \(x\) which is also an ancestor of some other changed node \(f_{i_2}\). We call such
a node a join node. In order to update its structures, we need to first calculate the changes along the path to \( x \) from each \( f_i \), \( i = 1..m \), such that the changed node \( f_i \) is a descendant of \( x \). Thus, we update the data structures in a layered manner. First, we update the nodes that have only one \( f_i \) descendant. Then we move up the tree and update nodes with two descendant \( f_i \)'s (these are join nodes of rank 2). From these nodes we continue up the tree, until we reach join nodes of rank 3 or higher, and so forth. Along the path from a join node of rank \( r \) to a join node of a higher rank (\( r + 1 \) or more), data structures are updated similarly to the way they are updated in the single-change case. Only when we reach a join node of rank \( r + 1 \) or higher, do we have to use a slightly different method of updating data structures, in order to integrate changes from several paths.

### 4.1 The Algorithm

**Input:** A schema \( S \), a document \( D \) (represented in memory), tuples \( (f_1, newval_1), \ldots, (f_m, newval_m) \), \( m > 1 \), where for \( 0 \leq i \leq m \), \( f_i \) is a node and \( newval_i \) is the value to be assigned to it.

**Output:** A result - VALID or INVALID.

**Pre-conditions:** \( D \) is valid with respect to \( S \). For \( 1 \leq i \leq m \), \( f_i \) is a simple-type node in \( D \). The data structures corresponding to \( D \) (as described in Section 2) have been created, and are correct (i.e., reflect the state of \( D \)).

**Post-conditions:** The data structures are correct. If the result is INVALID then the document is unchanged (identical to the input document). If the result is VALID, the value of \( f_i \) is \( newval_i \) for each \( 1 \leq i \leq m \), and the document is otherwise unchanged.

1. **Finding affected nodes.** For each \( 1 \leq i \leq m \), we search for \( f_i \) in \( FieldInfo \) to determine which selector-identified nodes are affected by the change in the value of \( f_i \), i.e., nodes for which \( f_i \) is a field. Note that changing values of nodes does not change which nodes are selector-identified nodes. We also
update the key-sequences stored in the relevant records of KeySelIdent and Keyref SelIdent. We update the key-sequences according to all changes that affect them. Note that a key-sequence may contain several fields whose values are changed. After these searches, we have a set of key-sequence updates for $K$ (respectively, $KR$), denoted by $KUpdates$ (respectively, $KRU pdates$), of the form $(\text{node}, \text{old key-sequence}, \text{new key-sequence})$. A node may appear only once in $KUpdates$ (respectively, $KRU pdates$). In our running example, we perform the update $(e,f,6)$, $(c1,f,6)$, $(c2,f,5)$, $(c3,f,5)$. So, $KRU pdates = \{(e, (1,2), (6,2))\}$ and $KUpdates = \{(c1, (4,2), (6,2)), (c2, (3,2), (5,2)), (c3, (1,2), (5,2))\}$.

Let $KUpdates_i$ (resp., $KRU pdates_i$) be the set of key-sequence updates $(n, ks, ks') \in KUpdates$ (resp., $KRU pdates$) such that $f_i$ is a field of $n$.

2. Finding Join Nodes. A node $n$ is a Join Node if it is on the paths of at least two $f_i$ nodes to the root. In other words, the KeyInfo structure of $n$ may need to be updated according to changes of at least two fields. Denote the set of Join Nodes by $JN$. We find these nodes as follows. With each node $v$ on the path from some $f_i$ to the root, we associate an integer $\text{counter}[v]$, initially 0 ($\text{counter}$ can be implemented as a hash-table, keyed by node objects). For each $1 \leq i \leq m$, for each node $v$ on the path from $f_i$ to the root (including $f_i$ and the root), we increment $\text{counter}[v]$ by 1. $JN$ contains all nodes $n$ such that $\text{counter}[n] \geq 2$. If $n \in JN$ and $\text{counter}[n] = k$, we say that the rank of $n$ is $k$, denoted $\text{rank}(n) = k$. Next, if $n_1 \in JN$, $n_2 \in JN$, $\text{rank}(n_1) = \text{rank}(n_2)$ and $n_1$ is an ancestor of $n_2$, then we remove $n_1$ from $JN$. This ensures that once we reach a node $n \in JN$, the next Join Node on the path to the root has a higher rank.
In the running example, \( JN = \{b_1\} \) and \( \text{rank}(b_1) = 4 \). Fig. 9 shows another example, with a larger set of Join Nodes. In it, the \( f_i \) nodes appear in grey and the Join Nodes appear in black. The rank of each Join Node is next to it.

3. Updating nodes: First stage. In this stage, for each \( i \), we update the KeyInfo and KeyRefInfo structures of nodes on the path from \( f_i \) to the root. However, we do not climb all the way to the root, only up to the first Join Node that we encounter, as a Join Node needs to receive updates from two or more field changes. This stage is executed in the procedure BeforeJN, depicted in Fig. 10. This procedure uses the function \( \text{UpdateNode}_K \) and the procedure \( \text{UpdateNode}_{KR} \), described in section 3. Note that within this stage we update nodes that are on the path of only one \( f_i \) to the root. Therefore, when we update such a node we know that it will not be updated later due to changes of other \( f_i \)'s. Thus, invalid references found in this stage indicate an invalid update operation (exit(INVALID) in the code of \( \text{UpdateNode}_{KR} \)). Also note that for each \( i \), if \( n_i \) is the last node that we update on the path from some \( f_i \) to the root, then we save the changes made to \( n_i.\text{KeyInfo} \) in \( n_i.\text{changes} \) (bold line, Fig. 10). This is used when, in the next stage of the algorithm, we update the Join Node that we have reached.

4. Updating nodes: Second stage. In this stage we update the Join Nodes we reached so far and continue to move up the tree. We advance gradually, each time updating nodes up to the next Join Node. This stage is executed in procedure FromJN, depicted in Fig. 11. This procedure uses function \( \text{UpdateNode}_K \), depicted in Fig. 4, and procedure \( \text{UpdateNode}_{KR} \) of Fig. 5. It also uses the function \( \text{UpdateJNNode}_K \) (see bold line in Fig. 11), depicted in Fig. 12. This function is very similar to \( \text{UpdateNode}_K \). The difference is that \( \text{UpdateNode}_K \) receives changes from exactly one child. \( \text{UpdateJNNode}_K \), on the other hand, needs to handle changes propagated through possibly several children. These changes are processed in the function \( \text{ProcessChangesJN} \), depicted in Fig. 13 (instead of calling \( \text{ProcessChanges} \) as in \( \text{UpdateNode}_K \)).
In this stage of our running example, we update the Join Node $b_1$. First we call $UpdateJNNode_{K}(b_1, KUpdates)$. In this function call, we first process the changes in $b_1$’s children (i.e., $x.changes$ for each child $x$). After updating $b_1.ChildrenKeyInfo$, we remove $(c_2, (3, 2), False)$ from $b_1.KeyInfo$. The key-sequence $(1, 2)$ is not yet removed from $b_1.KeyInfo$, since it appears with $isSelectorIdentified = True$.

We set $b_1.ChildrenKeyInfo[(5, 2)]$ to $\{(d, c_2), (b_3, c_3)\}$, and therefore we do not yet add the key-sequence $(5, 2)$ to $b_1.KeyInfo$. Then, we process $RelevantUpdates^9$. $RelevantUpdates = \{(c_1, (4, 2), (6, 2)), (c_3, (1, 2), (5, 2))\}$. Thus, we remove $(c_1, (4, 2), True)$ and $(c_3, (1, 2), True)$, and add $(c_1, (6, 2), True)$ and $(c_3, (5, 2), True)$. $b_1.RemovedSequences$ is set to $\{(1, 2), (3, 2), (4, 2)\}$. Then, we call $UpdateNode_{K}(b_1, KUpdates)$. In this function call, we successfully verify that the key-sequence $(6, 2)$ appears in $b_1.KeyInfo$, replace the record $(e, (1, 2))$ in $b_1.KeyrefInfo$ with $(e, (6, 2))$ and verify that $b_1.KeyrefInfo$ does not contain any key-sequences that appear in $b_1.RemovedSequences$. Note that if $e,f$ was not changed from 1 to 6 then this check would fail and the update would be invalid. The $KeyInfo$, $ChildrenKeyInfo$ and $KeyrefInfo$ structures following the update are depicted in Fig. 14.

**Complexity** In order to perform $m$ changes (as a transaction), the data structures ($KeyInfo$, $ChildrenKeyInfo$ and $KeyrefInfo$) of at most $O(m \times h)$

---

9 Recall that these are the tuples $(n, ks, ks’) \in KUpdates$ such that $n$ is a selector-identified node of $b_1$. 

---

Fig. 11. From JN.

Fig. 12. UpdateJNNode_{K}. 

---

Input: $JN$ (passed by reference), $KUpdates$, $KRIUpdates$.

Output: None.

c := 2;
while $JN \neq \emptyset$ {
for each $y \in JN$ s.t. rank(y) = c {
changesInNode = $UpdateJNNode_y(n, KUpdates)$;
remove $n$ from $JN$;
$UpdateNode_y(n, KRIUpdates)$;
}
changesInChild := changesInNode;
}
CurrentChild := $n$.parent;
changesInChild := changesInParent;
// Continue up the tree, until we reach
// either the next join node (which is guaranteed
// to have a higher rank) or the root.
while (CurrentNode \notin JN and
CurrentNode \neq null) {
changesInNode := 
$UpdateNode_y$(CurrentNode, CurrentChild, changesInChild, KUpdates);
$UpdateNode_y$(CurrentNode, KRIUpdates);
CurrentChild := $n$.parent;
}
// Save changes for updating
// the parent join node
CurrentChild.changes := changesInChild;
}
c++;
Input: A node \( y \), changes in \( y \) so far - \( \text{changes} \).
Output: None.

\[
\text{ProcessChangesJN}(y, \text{ByRef changes}) \{
\text{changedSequences} := \emptyset;
\text{for each } c \text{ in } y.\text{childNodes} \{ \\
\quad \text{for each } (ks, n) \in c.\text{changes} \{ \\
\quad\quad \text{if } ks \notin \text{changedSequences} \\
\quad\quad \quad \text{changedSequences} += ks; \\
\quad\quad \text{if } y.\text{ChildrenKeyInfo}[ks] \text{ contains a tuple } (c, n') \\
\quad\quad \quad \text{remove this tuple}; \\
\quad\quad \text{if } n \neq \text{null} \\
\quad\quad \quad \text{add the tuple } (c, n) \text{ to } y.\text{ChildrenKeyInfo}[ks]; \\
\quad \} \\
\quad \text{for each } c \text{ in } y.\text{childNodes} \\
\quad c.\text{changes} := \text{null}; \\
\quad \text{for each } ks \text{ in } \text{changedSequences} \\
\quad \text{if there is no } x \text{ such that } (x, ks, \text{True}) \in y.\text{KeyInfo} \\
\quad \quad \text{remove } (x, ks, \text{False}) \text{ from } y.\text{KeyInfo}; \\
\quad \quad \text{changes}[ks] = \text{null}; \\
\quad \} \\
\text{if } y.\text{ChildrenKeyInfo}[ks] \text{ has exactly one tuple } (c, \text{nodeInTable}) \{ \\
\quad \text{add the record } (\text{nodeInTable}, ks, \text{false}) \text{ to } y.\text{KeyInfo}; \\
\quad \text{changes}[ks] = \text{nodeInTable}; \\
\} \\
\}
\]

Fig. 13. \text{ProcessChangesJN}.

<table>
<thead>
<tr>
<th>n</th>
<th>ks</th>
<th>aSelectorIdentified</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>6,2</td>
<td>true</td>
</tr>
<tr>
<td>c3</td>
<td>5,2</td>
<td>true</td>
</tr>
</tbody>
</table>

| e | 6,2 |

<table>
<thead>
<tr>
<th>b1.\text{KeyInfo}</th>
<th>b1.\text{ChildrenKeyInfo}</th>
<th>b1.\text{KeyrefInfo}</th>
<th>d.\text{KeyInfo}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5,2)</td>
<td>(b3,c3),(d,c2)</td>
<td>c2</td>
<td>5,2</td>
</tr>
<tr>
<td>b2.\text{KeyInfo}</td>
<td>b3.\text{KeyInfo}</td>
<td>d.\text{ChildrenKeyInfo}</td>
<td></td>
</tr>
<tr>
<td>c2</td>
<td>5,2</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>c3</td>
<td>5,2</td>
<td>true</td>
<td>(5,2)</td>
</tr>
</tbody>
</table>

Fig. 14. Running example: \text{KeyInfo}, \text{ChildrenKeyInfo} and \text{KeyrefInfo} after the update.

Nodes need to be updated. In order to update each one, we need to perform at most \( O(m \ast h) \) lookups (of key-sequences and of selector-identified nodes). Since each lookup (in a search tree) takes at most \( O(|S|\log |D|) \), the complexity is \( O(m^2 h^2 |S|\log |D|) \).

5 Implementation

We implemented the algorithm for changing the value of a simple-type node (Section 3) and the algorithm for changing the values of several nodes (Section 4). The implementation is based on a Python-written open source validator, XSV [13] (XML Schema Validator). XSV is also distributed as an application, that performs validation, given URLs of a document and a schema. Internally, XSV loads the document and schema into memory, where they are represented based on an object model, with classes such as Schema, Document and Element. Then it performs in-memory validation, using its \text{validate} function. In
order to validate key and keyref constraints, XSV keeps a data structure called keyTabs for every node of the document (an instance of the Element class). This structure is very similar to our KeyInfo structure. It is a dictionary that maps a key constraint to a keyTab. The keyTab is a dictionary that maps key-sequences (represented as tuples of values) to nodes. Validation of key and keyref constraints for a node is done in the validateKey module of XSV.

We created a "modified" version of the XSV code, called XSV+:

- We added a function, validate_NoKeys. It is identical to validate, except that it does not call the functions that validate key and keyref constraints.
- We added a module that defines global variables needed to perform validation, and also structures that keep rollback information (i.e., keep the changes performed during an update operation, so that they can be undone if the update turns out to be invalid).
- We modified the validateKey module of XSV so that it inserts information into the data structures we defined, i.e., the global structures, and the keyTabs, KeyrefInfo and ChildrenKeyInfo structures. We changed the keyTab structure so that a tuple (n, isSelectorIdentified) and not just a node n is saved for a given key-sequence.
- We added a function called validateFieldChange, that validates a single change incrementally. It receives a node (an instance of the Element class) and a new value. If the change turns out to be invalid, it raises an exception and rolls back the updates performed. Otherwise, it updates all the data structures. We also added a function called validateFieldChanges, that receives a list of changes, to be made in a transaction.

In order to enable fast lookups in the data structures, we made extensive use of Python dictionaries. Python dictionaries are hash structures, that hold (key,value) pairs, where various objects may serve as keys (for each such object, a hash-code is generated). For example, KeySelIdent is a dictionary keyed by objects that represent key constraints (instances of the class Key). For a key constraint K, KeySelIdent[K] is a dictionary keyed by objects that represent selector-identified nodes (instances of the class Element). For a node n, KeySelIdent[K][n] is a tuple of values that represents the key-sequence of n. Similarly, n.keyTabs[K] holds a keyTab (KeyInfo, in our algorithms) containing information for a key K. n.keyTabs[K] may contain an entry n.keyTabs[K][ks] for a key-sequence (i.e., a tuple of values) ks. If the entry exists, the value stored in it entry is of the form (n', isSelectorIdentified).

5.1 Experiments

We performed a series of single-change operations. For each change, we: (1) Performed the change on the in-memory representation of the document. (2) Validated the document using XSV, yet skipped the validation of key and keyref constraints (i.e., performed only structural validation). This was done by calling validate_NoKeys. Denote the time for this stage by $T_{struct}$. (3) Checked
key and keyref constraints incrementally. Denote the time for this stage by $T_{\text{inc}}$ (the total time to validate the change is $T_{\text{struct}} + T_{\text{inc}}$. The changes were written to a text file (for each change, the "identity" attribute of the field, and the new value), and were then read from the file and validated using XSV (i.e., validation from scratch) in the following manner. We performed an initial loading (from a URL) and validation of the document. Then, for each change in the file, we performed the change on the in-memory representation of the document, and validated the document from scratch in memory (measuring only the validation time). Denote this time $T_{\text{scratch}}$.

The difference between $T_{\text{scratch}}$ and $T_{\text{struct}}$ is a good estimate for the time XSV spends on checking the key and keyref constraints (i.e., the time it takes to validate key and keyref constraints from scratch). Comparing this to $T_{\text{inc}}$ shows the time saved by checking key and keyref constraints incrementally.

![Schema Used for Testing](image)

**Fig. 15.** The schema used for testing

All tests were done on a P-4 2.8GHz PC with 1GB RAM. Testing was done on documents that conform to the schema depicted in Fig. 15. We generated three conforming documents, of sizes 346KB, 682KB and 1.41MB. When generating a document, the number of child nodes at each level was chosen randomly in the following manner. The generation function received a size parameter. The number of 'a' elements was chosen uniformly from the range 1..size. For each 'a' element, the number of 'd' child elements was chosen in the same manner. For
each ‘d’ element, the number of ‘b’ child elements was also chosen in the same way. For each ‘b’ element, the number of ‘c’ child elements was chosen uniformly from the range 1..2*\textit{size}. For every ‘c’ element, the number of ‘c’ child elements was chosen uniformly from the range 1..2*\textit{size}. For every ‘a’ element, 4*\textit{size} key sequences were chosen, and \textit{size} ‘e’ elements were generated for each one. The documents mentioned above were generated with \textit{size} parameters 4, 5 and 6, respectively. Node values were chosen so there would be a fair amount of ”cancellations”, i.e., a key sequence appearing in the \textit{KeyInfo} structures of siblings. For each document, we generated a sequence of 120 single value changes (20 changes as a warm-up, 100 changes for measuring purposes) and performed the changes using XSV+ (stages (1)-(3) above) and XSV.

The fields to be changed and the new values were selected at random, in a way that induced diversity in the experiments. Each change had a 2/3 probability to change a key field and a 1/3 probability to change a keyref field.

Changes to key fields were selected thus:

- With 80% probability: Choose a key field uniformly from all key fields. The new value is either chosen from the existing values of fields in the document or chosen to be 1 + the maximum existing value of a field in the document (50% probability for each case). Such changes are usually valid, since most selector identified nodes of the key are not referenced and therefore can be safely changed (as long as the new key sequence does not violate the key constraint).

- With 20% probability: Choose a change so that it would most likely fail. Choose a keyref field randomly. Find the corresponding key node, and change its corresponding field. The new value is either chosen from the existing values of fields in the document or chosen to be 1 + the maximum existing value of a field in the document.

Changes to keyref fields were selected as follows. Choose a keyref field randomly. For example, suppose it is an \textit{f1} field. Let \textit{p} be its parent node. Choose the new value as follows.

- With 80% probability: Try to perform a valid change. Within the ancestor ‘a’ node, search for key selector-identified nodes that have a key-sequence with the same \textit{f2} value as \textit{p} but a different \textit{f1} value. If such nodes exist, choose the new \textit{f1} value randomly from their \textit{f1} values. Otherwise (this change will be invalid), choose the value from the ones that currently exist in the document.

- With 20% probability: Choose a random value from the values that exist in the document. Thus, the change will most likely be invalid.

The setup time, i.e., the time to perform the initial validation given the URLs of a document and a schema, is slightly larger in XSV+ than in XSV. This is because XSV+ collects more information during validation. For the 346KB document, setup time is 25.9 seconds in XSV and 26.1 seconds in XSV+ (a 0.77% increase). For the 682KB document, it is 47.7 seconds in XSV and 49.9 seconds in XSV+ (a 4.6% increase). For the 1.41MB document, it is 90.9 seconds in XSV and 97.75 seconds in XSV+ (a 7.5% increase).
Fig. 16. Incremental validation vs. validation from scratch: average time per change (seconds).

<table>
<thead>
<tr>
<th>Document Size</th>
<th>Incremental</th>
<th>From Scratch</th>
</tr>
</thead>
<tbody>
<tr>
<td>346KB</td>
<td>0.0065</td>
<td>5.81</td>
</tr>
<tr>
<td>682KB</td>
<td>0.0077</td>
<td>12.39</td>
</tr>
<tr>
<td>1.41MB</td>
<td>0.0066</td>
<td>22.66</td>
</tr>
</tbody>
</table>

Fig. 17. Transactional changes: average time per transactional change (seconds), for a 346KB document.

<table>
<thead>
<tr>
<th>Transaction Size</th>
<th>Incremental</th>
<th>From Scratch</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.6359</td>
<td>5.81</td>
</tr>
<tr>
<td>150</td>
<td>0.93</td>
<td>5.81</td>
</tr>
<tr>
<td>200</td>
<td>1.33</td>
<td>5.81</td>
</tr>
<tr>
<td>300</td>
<td>2.2376</td>
<td>5.81</td>
</tr>
<tr>
<td>700</td>
<td>10.55</td>
<td>5.81</td>
</tr>
</tbody>
</table>

Fig. 16 shows the average time of validating key and keyref constraints (in seconds), per change, using incremental validation (i.e., $T_{inc}$), as compared to using validation from scratch (i.e., $T_{scratch} - T_{struct}$). The incremental validation time is significantly shorter, even if we take into account the larger setup time of XSV+. The results show an improvement of three orders of magnitude.

We performed similar testing for the algorithm for changing several simple-type nodes as a transaction (all or nothing). We ran tests on a 346KB document. Each test is a sequence of transactional changes (validated by performing all changes on the in-memory representation of the document, then calling validate_NoKeys to validate structural constraints and validateFieldChanges to validate key and keyref constraints incrementally), and each transactional change consists of $T_{ranSize}$ random simple-type node value changes, where $T_{ranSize} \in \{100, 150, 200, 300, 700\}$. Here, $T_{inc}$, $T_{struct}$ and $T_{scratch}$ refer to validating a transactional change (one that changes the values of several simple-type nodes). Fig. 17 shows the average time (in seconds), per transactional change, of validating key and keyref constraints incrementally, for each value of $T_{ranSize}$, compared with the time to validate key and keyref constraints from scratch (which is independent of $T_{ranSize}$). It shows that for reasonable transaction sizes, incremental validation is much faster than validation from scratch. The incremental checks become slower as $T_{ranSize}$ increases, and eventually incremental validation becomes slower than validation from scratch.

6 Conclusion

We present data structures that capture the state of a document with respect to key and keyref constraints defined in a schema. We suggest a simple update operation that changes the value of a simple-type node. We also suggest a ”transactional” update operation - changing values of several simple-type nodes in the same update operation. For each operation we present an efficient algorithm that validates the update incrementally with respect to key and keyref constraints. If the constraints are not violated, the algorithm updates the data structures to reflect the state of the updated document. Otherwise, the changes are undone in order to preserve consistency. The methods of this paper can be integrated with an incremental structure validator such as the ones in [10–12].
A Validation of Key and Keyref constraints from scratch

To the best of our knowledge, there are currently no complexity results for this problem. Validation can be done by creating the KeyInfo structures and making sure that for every scoping node of a keyref, all references are valid. Finding the scoping nodes can be done in time $O(|S|^4 + |D|\log |S|)$ by representing the schema using finite deterministic automata and running the automata on the document. The $|S|^4$ factor stems from the complexity of constructing Glushkov automata to represent the types defined in the schema (see [11] for details). Finding selector-identified nodes can be done in time $O(|D|)$ by creating an automaton that represents the reverse selector expression and running it in a bottom up traversal of the tree. In each state, we store the nodes that we encountered so far which are currently in this state. When we reach a scoping node, we know that each node stored in an accepting state is a selector-identified node of this scoping node. Then, in a similar way, we can find the field nodes of each selector-identified node of the key needs to be inserted into the KeyInfo structure of each of its scoping nodes. Such an insertion takes time $O(|\log D|)$. A selector-identified node whose depth (i.e., distance from the root) is $x$ can have at most $x$ scoping nodes.
The worst case complexity is if the height of the document tree is \( h = O(|D|) \). Then, if there are \( O(|D|) \) selector-identified nodes and each one has \( O(|D|) \) scoping nodes (its ancestors), we get \( O(|D|^2) \) insertions and \( O(|D|^2 \log |D|) \) complexity. On the other hand, if we assume that the tree has a branching factor of \( b \) and height \( O(\log_b |D|) \) then the complexity is \( O(\log |D| \cdot \sum_{x=1}^{b^h} b^x \cdot x) \), which is smaller than \( O(\log |D| \cdot h \cdot \sum_{x=1}^{b^h} b^x) \).

Then, in order to create the KeyInfo structures, we need to traverse the document tree bottom-up and, for each node, create the union of its children’s KeyInfo structures, while removing duplicate key-sequences. Since these structures are search trees, sorted according to key-sequences, this can be done in time linear in the total number of entries in the structures. Since a selector-identified node can appear only in the KeyInfo structures of its ancestors, the number of entries in all KeyInfo structures is at most \( O(h \cdot |D|) \). If \( h = O(\log |D|) \), we get \( O(|D| \log |D|) \).

In order to check references, for each scoping node \( s \) of the keyref and for each key-sequence \( ks \) in \( s.Keyref\_Info \), we need to search for \( ks \) in \( s.Key\_Info \). We perform at most \( O(h \cdot |D|) \) such searches, and each one takes time \( O(|S| \log |D|) \), so we get \( O(|S| |D| \log |D|) \). If \( h = O(\log |D|) \), we get \( O(|S| |D| \log^2 |D|) \).

Thus, under the assumption that the branching factor is roughly uniform and \( h = O(\log |D|) \), validation can be done in time \( O(|S| |D| \log^2 |D|) \), or \( O(|D| \log^2 |D|) \) if the schema is fixed. The worst case complexity is \( O(|D|^2 \log |D|) \).

This complexity may seem high, but it stems from the complex semantics of foreign key references in XML Schema. Simpler constraints can be checked more efficiently. For example, in [6], a keyref constraint is defined by \( C(B, l_B \subseteq A, l_A) \), where \( A, B, C \) are elements defined in a DTD, \( l_A \) is a child element of \( A \) and \( l_B \) is a child element of \( B \). The semantics of the constraint is that for each subtree rooted at a \( C \) node, for each \( B \) node \( b \) there is an \( A \) node \( a \) such that \( b.l_B = a.l_A \). Such a constraint can be easily checked by collecting the \( l_B \) and \( l_A \) values during a bottom-up sweep of the document. At every \( C \) node, the constraint is checked by making sure that every value in the set of \( l_B \) values appears in the set of \( l_A \) values. Checking the validity of a document with respect to an XML Schema keyref constraint is more complicated (since the key and keyref may have different scoping elements, and the subtree rooted at a keyref scoping node may contain several key scoping nodes).

## B Lemmas for the proof of Theorem 1

**Lemma 1.** If the update violates \( K \) then the violation is discovered during the execution of Update\_Nodes.

**Lemma 2.** If during the execution of Update\_Nodes it is decided that the update violates \( K \) then the update indeed violates \( K \).

**Lemma 3.** If the update does not violate \( K \) then after executing Update\_Nodes, the Key\_Info and Children\_Key\_Info data structures, for all nodes, correspond to the state of the updated document. Also, for each node \( y \), \( y.Removed\_Sequences \) is accurate before Update\_Node\_KR\(_y\) is executed (i.e., \( y.Removed\_Sequences \) contains key-sequences that appear in \( y.Key\_Info \) before the update and do not appear there following the update).

**Lemma 4.** If during the execution of Update\_Nodes it is decided that the update violates KR then the update indeed violates KR.
Proof. Such a decision is reached when UpdateNode\textsubscript{KR} is executed for a KR scoping node \textit{x}, and one of the following occurs. (1) There is a KR selector-identified node \textit{n} of \textit{x}, whose new key-sequence is \textit{ks}', and \textit{ks}' does not appear in \textit{x.KeyInfo}. (2) \textit{x.KeyrefInfo} contains, after updating it according to \textit{KRUdates}, a record \((\textit{n}, \textit{ks})\) where \textit{ks} appears in \textit{x.RemovedSequences}. When Update\textsubscript{KR} is executed, \textit{x.KeyInfo} and \textit{x.RemovedSequences} are correct (according to Lemma 3). Therefore, this is indeed a violation of KR.

\begin{flushright} \Box \end{flushright}

Lemma 5. If the update violates KR then the violation is discovered, during the execution of UpdateNodes.

Proof. If an update violates KR then there is a scoping node of KR, \textit{y}, and a selector-identified node of KR within the scope of \textit{y}, \textit{n}, such that after the update \textit{n} references a key-sequence that does not appear in \textit{y.KeyInfo}. According to Lemma 3, \textit{y.RemovedSequences} and \textit{y.KeyInfo} are correct before UpdateNode\textsubscript{KR} is executed. Thus, the violation is discovered in UpdateNode\textsubscript{KR}(\textit{y}, \textit{KRUdates}).

\begin{flushright} \Box \end{flushright}

Lemma 6. If the update does not violate KR then after executing UpdateNodes, the KeyrefInfo data structures, of all nodes, correspond to the state of the updated document.

Proof. UpdateNodes traverses all KR scoping nodes of nodes in \textit{KRUdates}, and changes the key-sequences stored in the KeyrefInfo data structures according to \textit{KRUdates}. This updates the KeyrefInfo data structures as required.

\begin{flushright} \Box \end{flushright}