On XML Schema Identity Constraints

Sharon Krisher
On XML Schema Identity Constraints

Research Thesis

Submitted in Partial Fulfillment of The Requirements for the Degree of Master of Science in Computer Science

Sharon Krisher

Submitted to the Senate of the Technion - Israel Institute of Technology

Nisan, 5766            HAIFA            April 2006
The Research Thesis Was Done Under The Supervision of Prof. Oded Shmueli in the Department of Computer Science
Contents

Abstract

1 Introduction
  1.1 Terminology
  1.2 Semantics of Key and Keyref Constraints
  1.3 Main Contributions
    1.3.1 Incremental Validation of Key and Keyref Constraints
    1.3.2 Foreign-Key Navigation
  1.4 Related Work
    1.4.1 XML Identity Constraints
    1.4.2 Incremental Validation
    1.4.3 XPath
  1.5 Organization

2 Incremental Validation
  2.1 Motivation
  2.2 Incremental Validation Overview
    2.2.1 Assumptions
    2.2.2 Suggested Operations
  2.3 Data Structures
    2.3.1 Motivation for Maintaining the Data Structures
    2.3.2 Definitions of the Data Structures
    2.3.3 Populating the Data Structures
  2.4 Non-incremental Validation
  2.5 Space Complexity
  2.6 Validation Algorithms
    2.6.1 Changing the Value of a Simple-type Node
    2.6.2 Transactions: Changing the Values of a Set of Simple-type Nodes
    2.6.3 Adding a Subtree
    2.6.4 Deleting a Subtree
# Implementation of Incremental Validation

## Overview

3.1 Validation in XSV

3.2 Implementation Details

3.2.1 Data Structures

3.2.2 Populating the Data Structures

3.2.3 Incremental Validation Functions

3.3 Designing a Fully Incremental Validator

3.4 Chapter Summary

## Experimentation

4.1 Specification of Experiments

4.1.1 Single-value Change

4.1.2 Multiple-value Change

4.1.3 Addition

4.1.4 Deletion

4.2 Results and Analysis

4.2.1 Setup

4.2.2 Single Value Change

4.2.3 Several Value Changes

4.2.4 Addition

4.2.5 Deletion

4.3 Space Consumption

4.4 Chapter Summary

## Foreign Key Navigation

5.1 Motivation

5.2 Foreign-key-navigation Axes Defined

5.3 Evaluation of Foreign-key-navigation Axes

5.3.1 Time Complexity

5.3.2 Space Complexity

5.4 Expressive Power

5.5 Single-field Navigation

5.6 Related Operators

5.6.1 A Generalized Navigation Operator

5.6.2 The Union Operator

5.6.3 An Intersection Operator

5.7 Chapter Summary
6 Conclusion

Bibliography

A A Detailed Proof of Theorem 5.2

B Incremental Validation Implementation: XSV+
   B.1 Single-value Change
   B.1.1 validateFieldChange() 135
   B.1.2 getKeyAndKeyrefUpdates() 136
   B.1.3 updatePath() 137
   B.1.4 updatePathToJoinNode() 137
   B.1.5 updateNodeK() 137
   B.1.6 updateNodeKR() 140
List of Figures

1.1 A simple example of key and keyref constraints. ............... 5
1.2 Semantics of key and keyref constraints. .................... 8

2.1 Key and keyref constraints: an example document. .......... 18
2.2 Physical structure of a KeyInfo structure. .................. 22
2.3 Data structures example. ................................... 25
2.4 Algorithm for changing the value of a simple-type node. .... 30
2.5 UpdateNodes. ............................................ 36
2.6 UpdateNodeK. ............................................. 37
2.7 UpdateNodeKR ............................................. 37
2.8 ProcessChanges. ........................................... 38
2.9 HandleOldKeySequences. ................................... 39
2.10 HandleNewKeySequences. ................................... 39
2.11 Join Nodes - an example. .................................. 45
2.12 BeforeJN. ................................................ 45
2.13 FromJN. .................................................... 46
2.14 UpdateJNNodeK ........................................... 46
2.15 ProcessChangesJN ........................................ 47
2.16 Running example: KeyInfo, ChildrenKeyInfo and KeyrefInfo after the update. ........................................... 48
2.17 UpdateNew. ................................................ 53
2.18 UpdateNodeNew ............................................ 54
2.19 UpdateDeletion .......................................... 57
2.20 UpdateNodeDeletion ...................................... 58

3.1 XSV flow of execution. ...................................... 64

4.1 The schema used for experiments. ............................ 72
4.2 Another view of the schema used for experiments. .......... 73
4.3 XSV vs. XSV+: Setup times (seconds). ........................ 76
4.4 Average time per single value change (seconds), comparison 1. 77
4.5 Average time per single value change (seconds), comparison 2. 77
Abstract

As XML [1] becomes an increasingly popular format of representing information, it becomes more and more important to verify the integrity of this information. XML Schema is a W3C standard of defining constraints the XML documents must conform to. A schema can define the structure of documents that conform to it (i.e., which elements and attributes appear in the document, and the parent-child relationships between them), and the data types of attributes and of simple elements. One of the useful features of XML Schema is the ability to define identity constraints of three kinds: 'key', 'keyref', and 'unique'. In this work we focus on the 'key' and 'keyref' constraints. These constraints are similar to the key and foreign key constraints used in databases. However, the hierarchical nature of XML documents complicates the semantics of these constraints. These semantics are not apparent at first reading of the formal specification [5], and they are explained in detail in the introduction to this thesis.

It is very important to be able to correctly and efficiently validate a document with respect to key and keyref constraints. In our experience, few XML Schema validators validate these constraints in accordance with the formal specification. Those that do, provide only the ability to validate a whole document and not the ability to change a document and check efficiently whether the change violates identity constraints. We explain how to statically validate identity constraints for a given document, and review how it is done (correctly) in the open-source validator XSV. Then, we define several simple operations that change a document (change a value of a node or several nodes, add or delete a sub-tree), and present efficient algorithms for incrementally validating these operations, i.e., algorithms that check whether changing the document will cause identity constraints to be violated. We do this by defining data structures that capture the state of the document with respect to the identity constraints defined in an XML Schema. As we perform a change operation, we update the relevant data structures accordingly. If the operation is found to violate a constraint, all updates to the data structures are rolled back. Otherwise, the data structures are updated, so that they reflect the state of the changed document.

We present an implementation of these algorithms. This implementation is written in Python. It is based on the open-source validator XSV, and extends it with incremental validation capabilities. We perform experiments with our implementation, and show its superiority to validation of identity constraints from scratch using XSV.
Another part of this work involves XPath, a language for navigating in XML documents. XPath includes axes that enable navigation according to axes. These axes navigate from a node to its parent, children or siblings. We suggest adding foreign-key navigation axes. Given a schema that defines a keyref $KR$, we suggest adding two axes, one that allows navigation from a node to a node it references (i.e., its 'parent' according to the keyref), and another that allows navigation from a node to nodes that reference it (i.e., its 'children' according to the keyref). We present a formal definition of such axes. Then, we show an algorithm for evaluating an XPath expression that uses such axes. We define a fragment of XPath that is enriched with such axes, and explore its expressive power. We also discuss the special case in which the keyref definition contains only one field.
Chapter 1

Introduction

Over the last few years, the Extensible Markup Language [1] (XML) has become an increasingly popular format of describing hierarchical data. XML is the lingua franca for information exchange over the web, and is also used to represent the internal data of many software systems. XML Schema [4] is a W3C recommendation that provides a mechanism for defining constraints on XML documents. An XML schema is in itself a valid XML document. It is a description of a type of XML documents, expressed in terms of constraints on the structure and content of documents of that type, above and beyond the basic syntax constraints imposed by XML itself.

One of the useful mechanisms provided by XML Schema is the ability to define identity constraints, including keys and foreign keys. These constraints, called 'key' and 'keyref', are similar to the 'primary key' and 'foreign key' constraints of databases, but their semantics is more complex due to the hierarchical nature of XML documents.

Key and keyref definitions use XPath 1.0 [2] expressions in order to specify paths to relevant nodes of the document. XPath is a language for navigating in an XML document. An XPath expression is a series of navigation steps and predicates. A navigation step leads from a node to its parent, children, descendants or siblings, and may also specify the tag of the desired target node. A predicate may be used to filter the set of nodes returned after the navigation step.

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.

```
<xs:key name="KeyName">
  <xs:selector xpath=XPATH_EXPRESSION/>
  <xs:field xpath=XPATH_EXPRESSION/>
```


The key definition includes a selector expression and one or more field expressions. These are expressions over a simple fragment of XPath [2], 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 we call the target node set of s (later we refer to these nodes as the selector-identified nodes 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 ∈ S and for each fᵢ, i = 1..m, fᵢ evaluates to a single, simple-type node in the context of x.
- For each x₁, x₂ ∈ S, if fᵢ(x₁) = fᵢ(x₂) for each i = 1..m then x₁ and x₂ 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:keyref>
```

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.
Figure 1.1: A simple example of key and keyref constraints.

For a simple example of the usefulness of key and keyref constraints, observe the schema depicted (informally) in Figure 1.1. This schema defines a data store of libraries. Each library contains a list of books and a list of checkouts. A book is uniquely identified by the combination of its name and of the name of its author. We want to ensure the following.

1. A specific book appears only once within a library (and may have several copies, specified by its NumberOfCopies element).

2. A checkout must reference a book which is listed in the library.

This is achieved by defining, within the scope of the Library element, the following constraints.

1. A key, whose selector expression is //Book and whose field expressions are ./Name and ./Author.

2. A keyref, whose selector expression is //Checkout and whose field expressions are ./BookName and ./BookAuthor.

1.1 Terminology

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 $KField_1, \ldots, KField_k$ be the
field expressions of $K$. Let $D$ be a document.

- The instances of $e$ in $D$ are called the \textit{scoping nodes} of $K$.

- 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 sometimes called the ”target node set”). Each $x \in S_n$ is a \textit{selector-identified node} (of $n$ and $K$). Note that a selector-identified node of $K$ may have several scoping nodes (if it is reachable from several different scoping nodes via the selector expression).

- Let $x$ be a selector-identified node of $K$. If a node $f$ is returned when we evaluate a field expression in the context of $x$ then we call $f$ a \textit{field} of $x$. We call the sequence of values of the nodes returned when evaluating $KField_1, \ldots, KField_k$ in the context of $x$ the \textit{key-sequence} of $x$. Note that a node $f$ may be a field of several different selector-identified nodes.

These terms are also used for keyrefs.

### 1.2 Semantics of Key and Keyref Constraints

The semantics of keyref references, as described in [5], is quite complex\footnote{And is not apparent at first reading.}.

- 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 a node $n''$ may be considered as being referenced by $n'$ only if $n''$ is a selector-identified node of the key, that has the same field values as $n'$, and at least one of the scoping nodes of $n''$ 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 that may be referenced by $KR$’s (or any other keyref that refers to $K$) selector-identified nodes whose scoping node is $n$. For each such node the table holds the node’s key-sequence (i.e., the values of its fields). In order to construct the table for $n$, we compute the union of the tables of $n$’s children. Also, if $n$ is a scoping node of $K$, we add its selector-identified nodes, and key sequences, to the combined table. Then, 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. If there exists a selector-identified node of $n$ with key-sequence $ks$ then it stays in the table. Note that this conflict resolution may result in an empty table (as a key-sequence that appears only in child tables, and appears there more than once, will not appear in $n$’s table).

To illustrate these points, observe the document depicted in Figure 1.2, where nodes of the document are marked by their tags. Suppose that this document conforms to a schema that defines a key, and a keyref that refers to it. $a$ (i.e., the node whose tag is 'a’) is the scoping node of the keyref, and $e$ is its selector-identified node. The $c$ nodes are the scoping nodes of the key, and their $c_-$ children are their selector-identified nodes (i.e., the selector expression is $./c_-$). The fields of a selector-identified node are its child nodes (whose values are shown in the Figure). Note that there are several $c_-$ nodes with the same key-sequence. This does not violate the key constraint because these $c_-$ nodes do not share a scoping node. There are three different $c_-$ nodes in the document whose key-sequence is (3,4). $e$ references the left-most of these $c_-$ nodes. This is because the key-sequence (3,4) appears in the key tables of both children of $d$. Therefore, this key-sequence does not appear in the table of $d$. Thus, this key-sequence appears only in the table of the left child of $b$. The corresponding $c_-$ node (marked with a circle) percolates all the way up to the table of $a$, and therefore it is the node referenced by $e$. If we were to change the key-sequence of $e$ to (1,2), the keyref constraint would be violated and the document would become invalid. This is because the key-sequence (1,2) does not appear in $d$’s table (since it appears in the tables of both its children), and thus does not percolate up to $a$’s table. This means that if we change $e$’s key-sequence to (1,2), $e$ will not reference any selector-identified node of the key.

### 1.3 Main Contributions

#### 1.3.1 Incremental Validation of Key and Keyref Constraints

- We define data structures that capture the state of a document with respect to key and keyref constraints.

- We define operations that change the document: change the value of a
simple-type node, change several values transactionally, add a subtree or delete a subtree.

- For each operation, we present an efficient algorithm that updates the data structures, and identifies violations of key and keyref constraints as it does so. If a violation is identified, all updates to the data structures are rolled back. Otherwise, after the completion of an algorithm run, the data structures reflect the state of the changed document. These algorithms do not scan the whole document, but rather examine only the parts of the document and of the data structures that may be affected by the change.

- We present an implementation of the algorithms (based on the open-source validator XSV ([38])), and experiments that compare the time it takes to validate a document from scratch after changing it with the time it takes to validate the structural constraints from scratch and validate the key and keyref constraints using our incremental algorithms. The experiment show that incremental validation of key and keyref constraints is much faster than validation of these constraints from scratch.
1.3.2 Foreign-Key Navigation

- We suggest new XPath axes, called foreign-key navigation axes. These axes navigate between selector-identified nodes of key and keyref constraints. We suggest a syntax for these axes and present a formal definition of their semantics.

- We extend a known algorithm (presented in [15]) for evaluating XPath expressions, to allow the evaluation of expressions that use the new axes.

- We define a simple yet expressive fragment of XPath, called $\text{XPath}'$ and explore its expressive power, compared to the expressive power of the fragment $\text{XPath}'_{fk}$, created by adding Foreign-key navigation axes to $\text{XPath}'$. We show that the foreign-key navigation axes add expressive power to $\text{XPath}'$.

- We discuss foreign-key navigation in cases where the key and keyref involved contain only a single field. We show that in many situations, single-field foreign-key navigation can be achieved using regular XPath expressions (i.e., without foreign-key navigation axes).

1.4 Related Work

1.4.1 XML Identity Constraints

Much work has been done on identity constraints for XML. Most papers do not adhere to the XML Schema specification of identity constraints, but rather define identity constraints independently of XML Schema and discuss various properties of these constraints.

In [28], three kinds of simple identity constraints are defined. These are key and foreign key constraints that are close in their semantics to database constraints. They apply to all instances (nodes) of a certain element defined in a DTD. The implication problem (i.e., if certain constraints hold, does it imply that other constraints hold) is studied for these constraints.

In [29], several classes of keys and foreign keys are defined. They are defined in the context of DTDs, and so the target nodes of a key or a foreign key are all instances of a certain element (tag). The fields are attributes defined in the DTD. The consistency problem is studied - given a DTD and a set of constraints, is there a document that conforms to the DTD and satisfies the constraints. This problem is undecidable for keys and foreign keys of more than one field. For one field it is in NP. The consistency problem
is further studied in [30]. The constraints here are more sophisticated than in [29]: there is a class of 'constraints with regular expressions', where the target nodes and the fields may be specified using regular expressions, and a class of 'relative integrity constraints', where a constraint may be defined within the scope of some element defined in the DTD.

In [31], a syntax is proposed for key specification (foreign keys are not dealt with) using regular path expressions, where there is no scope specification but the field expression can navigate to the parent. These keys are proved to be always satisfiable, and a cubic time algorithm for implication is shown.

In [19], a specification for XML keys (but not foreign keys) is defined. These keys are called "relative keys", and are similar to XML Schema keys. A relative key is defined by \( \langle Q, (Q', S) \rangle \), 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. 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.

In [20], relative keys are reasoned about. The satisfiability problem (given a set of constraints, is there a document that satisfies them) and the implication problem (if a set of constraints hold, does that imply that another constraint holds) are studied. It is proven that any finite set of keys is finitely satisfiable. The (finite) implication problem is proven to be finitely axiomatizable, and a polynomial time algorithm (polynomial in the size of keys) for the problem is shown.

\[ \text{1.4.2 Incremental Validation} \]

There has been work regarding incremental validation of XML documents. However, we have not encountered work on incremental validation of XML
Schema key and keyref constraints.

In [32], insertion and deletion operations are defined. XML Schema is not used, but rather a certain kind of tree automata (that represents structural constraints defined in a DTD). Key and keyref constraints are not handled. The methods discussed in [32] are expanded in [33] 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 "percolation" semantics of XML Schema key and keyref constraints (i.e., the complex semantics of references). The context paths have only child steps and so it seems that 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 for it; Deletion of a key node that is referenced is disallowed (because there’s 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 [36], insertion and deletion operations are defined. The updates are validated with respect to several classes of DTDs. Key and keyref constraints are not handled, only structural and attribute constraints. Glushkov automata are used to represent the content model of elements defined in the schema. The subtrees inserted or deleted are assumed to be valid. The algorithms presented take time \( O(n \times \log n) \) to validate structural constraints, where \( n \) is the number of nodes in the document. ID and IDREF constraints are validated in time \( O(|y| \times \log n) \) for insertion of a subtree \( y \) and \( O(|y| \times h \times \log n) \) for deletion of a subtree \( y \), where \( h \) is the height of the document. Checking these constraints is done by maintaining a data structure that holds all ID attribute values (in order to check that they are unique), and also a reference counter for each node \( x \) in the tree, that holds the number of IDREF references to nodes which are descendants of \( x \).

Incremental validation, of insertion and deletion of sub-trees, with respect to key constraints, is introduced in [34]. These key constraints are similar to the key constraints of XML Schema, but not identical. Note that their definition includes an XPath expression that navigates to the scoping nodes,
which is not the case in XML Schema keys. 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 that the "key path" evaluates to, 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 \( S = S_1 \cap \ldots \cap S_p \), where \( p \) is the number of key paths of the key. The key is violated if \( S \) contains more than one node. Keyref constraints are not handled in [34]. Note that while the index structure introduced in [34] 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.

In [35], 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 (which represent XML Schema structural constraints) are handled. Key and keyref constraints are not handled. For DTDs, the time complexity is \( O(m \times \log n) \), where \( m \) is the number of updates and \( n \) is the size of the document. For specialized DTDs, the time complexity is \( O(m \times \log^2 n) \).

Incremental validation, of insertion and deletion of sub-trees, with respect to key constraints, is introduced in [34]. These key constraints are similar to the key constraints of XML Schema, though not identical. Keyref constraints are handled. Validation is done by maintaining a "Key index".

The work in [37] deals with validating a document that conforms to one schema, with respect to another schema. Only structural constraints are considered. "Abstract XML Schemas", that capture these constraints, are defined. The schemas are preprocessed, the document is not. In the algorithm presented, a node is validated in parallel according to a type \( t \) (defined in the original schema) and another type \( t' \) (defined in the new schema). If \( t \) is subsumed by \( t' \) then validation succeeds (and checking the subtrees is
avoided), if they are disjoint then validation fails. Otherwise, subtrees are recursively checked. There is also an algorithm for validation when the tree is changed, and then validated according to the new schema.

1.4.3 XPath

XPath (the XML path language) 1.0 is a W3C recommendation defined in [2]. It is a language for navigating in an XML document and selecting nodes or values. XPath 1.0 expressions are used in XSLT [9] (XML Transformations), which is a language for transforming XML documents into other XML documents. XPath 2.0 [3] expands the basic functionality of path expressions provided by XPath 1.0, and adds features such as types, variables, quantified expressions and 'for' loops. XQuery 1.0 [5] is an extension of XPath 2.0, that adds more functionality to it. Throughout the thesis, we shall focus on XPath 1.0 and refer to it simply as XPath (unless stated otherwise).

XPath expressions are used in the definition of XML Schema identity constraints. The selector expressions and field expressions of these definitions belong to a simple fragment of XPath, called "restricted XPath".

Much work has been done on containment of XPath queries, i.e., deciding whether for every possible input document, the result of one query is contained in the result of the other query. Miklau and Suciu [12] show a sound and complete exponential time algorithm for deciding containment of queries of a fragment of XPath that consists of node tests, the child axis /, the descendant axis //, wildcards * and predicates [ ]. They also show that the problem is CoNP-complete. In [10], Wood studies containment for a basic XPath fragment under DTD constraints. In [11], Deutsch and Tannen introduce a method for deciding containment for a simple XPath fragment under simple integrity constraints. It will be interesting to examine containment in the presence of foreign-key-navigation axes.

The minimization of XPath queries is studied in [14]. The XPath fragment studied includes /, //, [ ] and *. The minimization is with respect to the number of nodes in the tree pattern that represents the query. The problem is co-NP complete. A sound and complete, but exponential, algorithm is shown. A polynomial algorithm is shown for a limited fragment.

The expressive power of several XPath fragments is analyzed in [22]. The largest of these fragments, which is similar to Core XPath (an XPath fragment, defined in [10], that supports commonly used features of XPath), is proven to be equivalent to positive existential first order logic in two free variables (one to represent the context node and another to represent the output of the query). The XPath dialect defined in [24] and [23] by introducing conditional axes is proven to be equivalent to full first order logic.
It is important to observe that the XPath fragments treated in these papers ([22], [23], [24]) do not include comparison of values - the fragments of [22] do not include an equality operator at all and the dialect defined in [23] uses an equality operator only to check the label of a node and not its string value. Thus these fragments (and the logic languages to which they are equivalent) cannot express navigation according to keyref constraints (discussed in Chapter 5), since the comparison of node values is essential to such navigation and to the basic semantics of key and keyref constraints. It is also important to note that the set of labels in these fragments is finite. Node labels can be used to represent values (i.e., a node of label "x" and text value "a" would be represented as a node of label "x" having a child node of label "a"), but if there is a finite set of labels then only text nodes whose text is taken out of a finite domain of strings can be represented.

Another important field of XPath-related research is the evaluation of XPath expressions. In [15], algorithms are presented for processing queries. Their time complexity is polynomial in the sizes of the query and the document. It is linear for a fragment of XPath, called Core XPath, that does not include comparisons, or arithmetic and string operations. Note that the selector and field expressions, used in the definitions of XML Schema keys and keyrefs, belong to Core XPath. Algorithms with improved complexity are presented in [17] and in [18].

To the best of our knowledge, there has not been much work on adding new axes to XPath. One example of such work is [24] and [23], where a new 'conditional axis' is introduced. This axis can be used for queries such as "find a descendant y with @x=val1 such that all the nodes on the path to y have @x=val2". Such queries have similar semantics to that of the temporal logic operator 'until'. This 'conditional axis' adds expressive power to a path language called XCore, which is very similar to Core XPath (defined in [16]).

1.5 Organization

The rest of the thesis is organized as follows.

- In Chapter 2 we discuss incremental validation of key and keyref constraints. First we explain how non-incremental validation of these constraints can be performed. Then we define several operations that change a given document, and present algorithms that check whether such an operation maintains the validity of the document (i.e., algorithms that incrementally validate the changed document).

- In Chapter 3 we discuss an implementation of the algorithms presented
in Chapter 2.

• In Chapter 4, we review experiments done with the implementation. We present results that show the time that can be saved by using incremental validation instead of validation from scratch.

• Foreign-key XPath navigation is discussed in Chapter 5.

• We conclude in Chapter 6.
Chapter 2

Incremental Validation

2.1 Motivation

As the popularity of XML increases, it becomes more and more important to validate XML documents efficiently, with respect to XML Schema. Current commercial XML validators enable validation of complete documents, but do not offer a mechanism for manipulating a document and incrementally validating the changes. Some work has been done regarding incremental validation of structural constraints, and also of non-XML Schema identity constraints. We present algorithms that perform incremental validation with respect to XML Schema key and keyref constraints, and handle the semantics dictated by the XML Schema specification. We define several operations that change the content of a document, i.e., add or remove nodes, or change the value of existing nodes. We present algorithms that check whether performing such an operation would violate key or keyref constraints, and change the document only if the operation maintains the validity of the document with respect to the constraints. These algorithms traverse only the parts of the document that may be affected by the change operation, and not the whole document. They maintain, in an efficient manner, data structures that hold information relevant to the validation of key and keyref constraints.

2.2 Incremental Validation Overview

We define several update operations on XML documents.
2.2.1 Assumptions

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 \[39\] (that uses Node objects), and that every node in the document has a unique node identifier.

2.2.2 Suggested Operations

We suggest the following operations. For each one we present an algorithm that performs the operation on the document, but undoes any changes if the operation causes a violation of the key or keyref constraints.

- Changing the value of a simple-type node: \( \text{ChangeValue}(f, \text{newval}) \), where \( f \) is some simple-type node and \( \text{newval} \) is the value to be assigned to it.

- Changing the values of a set of simple-type nodes:
  \( \text{ChangeValues}((f_1, \text{newval}_1), ..., (f_m, \text{newval}_m)) \), where for \( 0 \leq i \leq m \), \( f_i \) is some simple-type node and \( \text{newval}_i \) is the value to be assigned to it. This operation is transactional, i.e., either all changes are made or no change is made (if performing all changes leaves the document in an invalid state).

- Adding a subtree: \( \text{AddSubTree}(p, T, i) \), where \( p \) is a node in the document \( D \) and \( T \) is a data tree. The root of \( T \) is to be added as the \( i \)'th child of \( p \).

- Deleting a subtree: \( \text{Delete}(t) \), where \( t \) is some node. The operation deletes the subtree \( T \), rooted at \( t \), from the document.

We demonstrate the data structures and algorithms on the document depicted in Figure 2.1. 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 b1). 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.

2.3 Data Structures

We keep data structures that capture the state of the document with respect to key and keyref constraints. Our algorithms maintain these structures.

2.3.1 Motivation for Maintaining the Data Structures

The data structures are designed to enable efficient monitoring as the document changes. The KeyFieldInfo and KeyrefFieldInfo structures enable 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 KeySelIdent and 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 \( n.KeyrefInfo \) 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 \( x.KeyInfo \) 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 \( x.KeyrefInfo \) must also appear in \( x.KeyInfo \). \( x.KeyInfo \) 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 \( x.KeyInfo \) (if there are, then the key is violated). \( x.KeyInfo \) is maintained even if \( x \) is not a key or keyref scoping node, because the content of a node’s \( KeyInfo \) structure is affected by the \( KeyInfo \) structures of the node’s children (as described in the definition of the \( KeyInfo \) structure, and dictated by the semantics of XML Schema constraints).

The \( x.ChildrenKeyInfo \) structures (for each node \( x \)) allow us to easily update the \( KeyInfo \) structure of a node \( x \) following an update to one or more of its children’s \( KeyInfo \) structures. For each key-sequence \( ks \) that appears in the \( KeyInfo \) structure of at least one child of \( x \), \( x.ChildrenKeyInfo \) tells us in which children’s \( KeyInfo \) 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 \( x.KeyInfo \) according to the current states of \( x.ChildrenKeyInfo \) and \( x.KeyInfo \). Basically, \( x.ChildrenKeyInfo \) saves us the trouble of having to check the \( KeyInfo \) structures of all of \( x \)'s children when the structure of one of the children is updated.

### 2.3.2 Definitions of the Data Structures

Note that the definition of the data structures allows for multiple key and keyref constraints. Each data structure has an instance for each key or keyref constraint, depending on the data structure. For example, for each key constraint \( K_i \) defined in the schema, there is a data structure \( KeySN[K_i] \). However, our algorithms assume a single key constraint and a single keyref constraint. Thus, for example, the algorithms use \( KeySN \) to refer to \( KeySN[K] \), where \( K \) is the single key constraint defined in the schema.

- **KeySN** (short for **KeyScopingNodes**): This is a global data structure, i.e., exists per document, and not per node. For each key \( K \),
KeySN[K] contains the scoping nodes of K. In order to enable checking efficiently whether a given node is a scoping node, KeySN[K] is maintained as a search tree over node identifiers. For a scoping node n, we also keep the distance (i.e., number of edges in the document tree) of n from the root of the document, and denote it KeySN[K][n].

- KeyrefSN (short for KeyrefScopingNodes): This is a global data structure. For each keyref KR, KeyrefSN[KR] contains the scoping nodes of KR. In order to enable checking efficiently whether a given node is a scoping node, KeyrefSN[KR] is maintained as a search tree over node identifiers. For a scoping node n, we also keep the distance (i.e., number of edges in the document tree) of n from the root of the document, and denote it KeyrefSN[KR][n].

- KeySelIdent (short for KeySelectorIdentifiedNodes): This is a global data structure. For each key K, KeySelIdent[K] contains a record (s, ks, SN) for each selector-identified node s of K, where ks is the key-sequence of s and SN is a list of s’s scoping nodes, ordered by distance from the root (all these scoping nodes are ancestors of s, and therefore on the path from the root to s). In order to enable fast access to the record of a node s, KeySelIdent[K] is maintained as a search tree over node identifiers.

- KeyrefSelIdent (short for KeyrefSelectorIdentifiedNodes): This is a global data structure. For each keyref KR, KeyrefSelIdent[KR] contains a record (s, ks, SN) for each selector-identified node s of KR, where ks is the key-sequence of s and SN is a list of s’s scoping nodes, ordered by their distance from the root (all these scoping nodes are ancestors of s, and therefore on the path from the root to s). In order to enable fast access to the record of a node s, KeyrefSelIdent[KR] is maintained as a search tree over node identifiers.

- KeyFieldInfo: This is a global data structure. For each key K, KeyFieldInfo[K] contains a tuple (f, occurrences) for every node f that appears as a field of some selector-identified node. occurrences is a set of tuples of the form (s, i), where s is a selector-identified node of K such that f is the i’th field of s (i.e., the i’th field expression of K evaluates to {f} in the context of s). KeyFieldInfo[K] is maintained as a search tree over node identifiers (to facilitate searching for field nodes).

- KeyrefFieldInfo: This is a global data structure. For each keyref KR, KeyrefFieldInfo[KR] contains a tuple (f, occurrences) for ev-
Every node $f$ that appears as a field of some selector-identified node. 

occurrences is a set of tuples of the form $(s, i)$, where $s$ is a selector-identified node of $KR$ such that $f$ is the $i$'th field of $s$ (i.e., the $i$'th field expression of $K$ evaluates to $\{f\}$ in the context of $s$). $KeyrefFieldInfo[KR]$ is maintained as a search tree over node identifiers (to facilitate searching for field nodes).

- $x.KeyInfo$: This is a local data structure, i.e., there are multiple instances of this data structure, associated with document nodes. This data structure exists for each node in the document, though it may be empty for some of the nodes. For a key $K$ and a node $x$, $x.KeyInfo[K]$ 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 the case where $x$ is both a scoping node and a selector-identified node of itself).
  
  - $ks$ is the key-sequence of $n$.
  
  - $isSelectorIdentified$ is a Boolean value.

There are no two records with the same key-sequence in $x.KeyInfo$. $x.KeyInfo[K] = \{(n, ks, True) | x$ is a scoping node of $K, n$ is a selector-identified node with key-sequence $ks$ in the scope of $x\} \cup \{(n, ks, False) | There is no selector-identified node of $K$ with key-sequence $ks$ and scope $x$, there is a child $z$ of $x$ and a Boolean $b$ such that $(n, ks, b) \in z.KeyInfo[K]$ and there is no child $z' \neq z$ and a Boolean $b'$ such that $(n, ks, b') \in z'.KeyInfo[K]\}$

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 $x.KeyInfo[K]$ (regardless of its $isSelectorIdentified$ value). Thus, the $KeyInfo$ structures are used to verify foreign key references.

In order to allow searching for a selector-identified node in the $KeyInfo$ structures, we maintain a search tree over node identifiers for each $x.KeyInfo[K]$ structure. In order to also allow searching for key-sequences, we maintain (for each $x.KeyInfo[K]$ structure) a multi-search-tree over key-sequences. The multi-search-tree 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 (of $x.KeyInfo[K]$) that are stored in it. For example, if a node $A$ is a scoping node of $K$ and has three selector-identified nodes $B_1, B_2, B_3$ whose key-sequences are $(1,2,3), (1,4,5), (2,3,4)$ respectively, then the structure for $A.KeyInfo[K]$ looks as in Figure 2.2.

- $x.ChildrenKeyInfo$: This is a local data structure, i.e., there are multiple instances of this data structure, associated with document nodes. This data structure exists for each node in the document, though it may be empty for some of the nodes. For a key $K$ and a node $x$, $x.ChildrenKeyInfo[K]$ contains an entry for every key-sequence $ks$ that appears in the $KeyInfo[K]$ structure of at least one child of $x$. The entry contains a set of tuples of the form $(\text{child}, \text{nodeInChild})$, where $\text{child}$ is a child node of $x$ and $\text{child.KeyInfo}[K]$ contains a record $(\text{nodeInChild}, ks, b), b \in \{\text{True, False}\}$. This information helps in updating the $KeyInfo$ structures when the document is modified. For a key-sequence $ks$, let $x.ChildrenKeyInfo[K][ks]$ denote the entry with key-sequence $ks$ in $x.ChildrenKeyInfo[K]$. It is considered null if there is no such entry. We observe the following:

- If a tuple $(z, n)$ appears in $x.ChildrenKeyInfo[K][ks]$ it means that $n$ is a valid candidate to appear in $x.KeyInfo[K]$, i.e., it
survived’ competition from other nodes in the sub-tree of $z$ and appears in $z.KeyInfo[K]$.

- If $|x.ChildrenKeyInfo[K][ks]| = 1$ and $x$ has no selector-identified nodes with key-sequence $ks$ then the ‘candidate’ that appears in $x.ChildrenKeyInfo[K][ks]$ also appears in $x.KeyInfo[K]$.

$x.ChildrenKeyInfo[K]$ is a multi-search-tree (over key-sequences). For a record $(ks, occurrences)$ in $x.ChildrenKeyInfo[K]$, occurrences is maintained as a hash-table, that maps a child of $x$ to a node in the child’s $KeyInfo$ structure.

- $x.KeyrefInfo$: This is a local data structure, i.e., there are multiple instances of this data structure, associated with document nodes. This data structure exists for each node in the document, but $x.KeyrefInfo[KR]$ is NULL if $x$ is not a scoping node of the keyref $KR$. For a keyref $KR$ and a node $x$ which is a scoping node of $KR$, $x.KeyrefInfo[KR]$ 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 $n$. Observe that unlike the case for $K$, $x.KeyrefInfo[KR]$ is maintained only for scoping nodes of $KR$ and NOT for other nodes. For other nodes, $x.KeyrefInfo[KR]$ is NULL.

For each $x.KeyrefInfo[KR]$ structure, we maintain a search tree over node identifiers, and also a multi-search-tree over key-sequences. The entry for a key-sequence $ks$ in the multi-search-tree may contain several records $(n_i, ks)$. This is because several selector-identified nodes of the same scoping node of $KR$ may have the same key-sequence.

**Lemma 2.1** If a record $(n, ks, b)$ appears in $x.KeyInfo[K]$ then either $b = True$ or there is a node $y$ which is a descendant of $x$ such that a record $(n, ks, True)$ appears in $y.KeyInfo[K]$ and a record $(n, ks, False)$ appears in the $KeyInfo[K]$ structure of every node along the path from $y$ to $x$.

**Proof** Let $BaseNodes$ be the set of nodes $n$ such that $n.KeyInfo[K]$ is not empty, and all descendants of $n$ have empty $KeyInfo[K]$ structures. Let $dist(x)$ be the maximum value in $\{distance(x, n) | n \in BaseNodes\}$, where $distance(x, n)$ is the number of edges on the path from $x$ to $n$. Note that $dist(x)$ is defined for each node $x$ such that $x.KeyInfo[K]$ is not empty. We prove the Lemma by induction on $dist(x)$.

Basis: $dist(x) = 0$. That is, $x \in BaseNodes$, which means all descendants of $x$ have empty $KeyInfo[K]$ structures. Since $x$ cannot get records from the child structures, $x.KeyInfo[K]$ can only contain records with $b = True$.  


Induction step: We assume correctness for nodes with $\text{dist} \leq k$ and prove for a node $x$ with $\text{dist}(x) = k + 1$. Suppose $x.\text{KeyInfo}[K]$ contains a record $(n, ks, \text{False})$. By definition of $\text{KeyInfo}[K]$, this means that there is a child $c$ such that a record $(n, ks, b)$ appears in $c.\text{KeyInfo}[K]$. Since $c$ is a child of $x$, $\text{dist}(c) \leq k$ (for every path from $c$ to a node in $\text{BaseNodes}$, there is a path from $x$ to the same node, through $c$. Thus, $\text{dist}(x)$ is at least $\text{dist}(c) + 1$). Therefore by the induction hypothesis, either $b = \text{True}$ or there is a node $d$ which is a descendant of $c$ (and of $x$) such that a record $(n, ks, \text{True})$ appears in $d.\text{KeyInfo}[K]$ and a record $(n, ks, \text{False})$ appears in the $\text{KeyInfo}[K]$ structure of every node along the path from $c$ to $d$. This proves the claim for $x$.

The document uniquely determines the content of these structures. The incremental algorithms need 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 $x$ and key $K$ we keep a set of key-sequences $x.\text{RemovedSequences}[K]$. If an algorithm updates $x.\text{KeyInfo}[K]$, $x.\text{RemovedSequences}[K]$ is the set of key-sequences that appear in $x.\text{KeyInfo}[K]$ 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 $x.\text{RemovedSequences}[K]$. The $\text{RemovedSequences}$ sets are only used during the execution of certain update algorithms and are initialized to be empty when such an algorithm is executed.

Figure 2.3 shows a portion of the data structures for the document depicted in Figure 2.1 (and repeated in 2.3).

Note: We use a succinct representation of strings as integers based on a TRIE structure. Let $N$ be the number of text nodes in the document. We map each string value of a text node in the document to an integer in the range $1..N$. In the tree structure that represents the document, for each text node we save both the string value and the integer. We also save the mapping (from strings to integers) in a TRIE structure. This is a tree in which every node represents a string prefix, and has outgoing edges for all possible ‘next character’ of some string (that exists in the document). We keep the integer values in the nodes that represent the corresponding string values. We maintain an array $\text{MapInfo}[1..N]$. $\text{MapInfo}[i]$ is the number of text nodes whose value is mapped to $i$. Suppose we update the value of some text node $tn$ whose string value is $str_1$ and whose integer value is $i$.

\[ \text{The KeyInfo structures that are not shown contain no records.} \]
Let the new string value be \( str_2 \). We decrease \( MapInfo[i] \) by 1. We look for \( str_2 \) in the TRIE structure. If it is there, associated with an integer \( j \), then we increment \( MapInfo[j] \) by 1 and associate \( j \) with the text node. If \( str_2 \) is not in the TRIE, we associate it with \( k \), where \( k \) is the lowest integer such that \( MapInfo[k] = 0 \). We increase \( MapInfo[k] \) by 1, insert this mapping to the TRIE and associate the string value \( str_2 \) and the integer value \( k \) with the text node \( tn \).

Using this representation, the time to compare two key-sequences is \((\text{number of fields}) \cdot (\text{time to compare integers}) = O(|S|) \cdot O(\log |N|) = O(|S| \log |D|)\), where \( N \) is the number of text nodes in the document. For practical purposes, we assume that comparing two integers is done in \( O(1) \), and then the time to compare key-sequences is \( O(|S| \log |D|) \).

### 2.3.3 Populating the Data Structures

- **KeySN and KeyrefSN**: We obtain the scoping nodes of each key and keyref constraint. This is done as follows. We use an algorithm that is similar to a validation algorithm presented in [36]. Given a schema \( S \), we construct finite deterministic automata that represent the structural constraints imposed by the schema, and use these automata to locate the scoping nodes through a traversal of the document. The algorithm populates \( KeySN[K] \) (respectively, \( KeyrefSN[KR] \)) for each

\[\text{Figure 2.3: Data structures example.}\]
key (respectively, keyref) constraint \( K \) defined in \( S \).

Given a schema \( S \), we assign unique names to all anonymous complex types that appear in the schema. Then, for each complex type \( T \) we construct a regular expression \( R_T \) that represents the structure of \( T \). The symbols of \( R_T \) are the elements (tags) that appear in \( T \). We ignore attributes, since we assume the documents on which we evaluate queries are valid (so there is no need to go over the attributes. Also, attributes clearly cannot be scoping nodes). For each regular expression \( R_T \) we construct a Glushkov automaton \( A_{R_T} \) (see 2.2.1 in [36]).

Aside from the initial state, the states of \( A_{R_T} \) are positions in \( R_T \). Each such position corresponds to an element \( e \) that appears in \( T \) (\( e \) being the element’s tag). With every state we associate the type \( T \) of the corresponding element and also the key constraints \( \{ K_i \} \) and keyref constraints \( \{ K_{R_i} \} \) that are defined in the element. Note that there may be several elements with the same tag \( e \) and different types, defined in different complex types. We consider all simple types as type \( T_{\text{simple}} \). After constructing the automata, we create a mapping of every global (i.e., top level) element \( e \) of type \( T \) to the automaton \( A_{R_T} \) that corresponds to the type \( T \). We also save sets of the keys and keyrefs defined in the element. Note that there are no two global elements with the same name. Therefore we can simply save a mapping of element names (tags) to automata (and to lists of keys and keyrefs).

Our algorithm is based on the validation algorithm presented in section 3.1 of [36]. The algorithm performs a depth-first traversal of the document tree. At each stage it keeps an automaton for every “open” element (i.e., an element whose sub-tree has not yet been fully traversed). The current states of the running automata are saved in a stack. Upon encountering the root node \( r \) (that is, the topmost user defined element in the document), we start running the corresponding automaton (which we find according to the element name). As this automaton encounters the first child (i.e., sub-element) of \( r \), it moves from its initial state to a new state \( q \). Recall that when constructing the automaton, we associated with \( q \) a type \( T \). If \( T \) is not \( T_{\text{simple}} \), we start running the corresponding automaton \( A_{R_T} \). We continue traversing the tree. After we finish running an automaton for some node (i.e., traverse the node’s sub-tree), we make a transition in the automaton of the parent node.

We populate the data structures as we go along. For the root node \( r \) (that is, the topmost user defined element), we have a set of keys \( S_K \) and a set of keyrefs \( S_{KR} \) that we access according to the tag of \( r \).
(recall that we save these sets after creating the automata). For every key $k \in S_K$, we add the root node to $KeySN[k]$, with value 0 (i.e., $KeySN[k][r] = 0$. For a node $n$, $KeySN[k][n]$ is the distance of $n$ from the root.). For every keyref $kr \in S_{KR}$, we add the root node to $KeyrefSN[kr]$. When, during the traversal of the document, we reach a non-root node $n$, this is done in a transition to some state $q$ (possibly a transition from $q$ to $q$) of the automaton run at $n$’s parent node. Let $S_{Kq}$ and $S_{KRq}$ be the sets of keys and keyrefs, respectively, associated with $q$. Then, for every key $k \in S_{Kq}$, we add $n$ to $KeySN[k]$. For every keyref $kr \in S_{KRq}$, we add $n$ to $KeyrefSN[kr]$. The value for $n$’s entry is $n$’s distance from the root (we keep track of this information as we traverse the document).

In creating regular expressions to represent complex types, we handle various features of XML Schema:

- **Sequences** are represented as concatenations in the regular expression. For example a sequence of an element $e_1$ with minOccurs=0, maxOccurs=unbounded, and of an element $e_2$ with minOccurs=1, maxOccurs=2, is represented as $e_1^*e_2(\epsilon+e_2)$. If the aforementioned sequence appears with maxOccurs=unbounded then the regular expression would be $(e_1^*e_2(\epsilon+e_2))^*$. 

- A choice in the schema is represented using ’+’ in the regular expression.

- If an xs:all appears in the schema, we need to write all possibilities in the regular expression. For example an xs:all between elements $e_1$, $e_2$ and $e_3$ is translated into the expression $e_1e_2e_3 + e_1e_3e_2 + e_2e_1e_3 + e_2e_3e_1 + e_3e_1e_2 + e_3e_2e_1$.

- If a type contains a reference to a model group then we write the content explicitly and then generate the regular expression.

- If a type contains an <xs:element ref="e"/> , for some global element e, e appears in the regular expression and e’s type is associated with the corresponding state of the automaton.

- **Substitution groups**: Suppose some (global) element $e_1$ is the head of a substitution group whose members are the (global) elements $e_2,...,e_k$. Whenever a ’ref’ to $e_1$ appears in some type, instead of using $e_1$ in the corresponding regular expression, we use $(e_1 + e_2 + ... + e_k)$.

- If a type $T'$ derives from a type $T$, in order to construct the regular expression $R_{T'}$ we write the content of $T'$ explicitly and
then translate it into a regular expression.

– If an element $e$ appears in the document with an $xsi$:type specification, $xsi$:type=$T$, we use the automaton $A_{R_T}$ instead of the automaton to be used according to the description of the algorithm above. Note that $xsi$:type is the only mechanism through which an element appears in the document with a different type than the type specified in the schema (specifically, with a type derived from the one specified in the schema).

– If a type contains an $xs$:any element, it means that any well formed XML element may appear. Let $l_1, \ldots, l_n$ be the tags of elements that appear in the schema. Then $xs$:any is represented in the regular expression by $(l_1 + \ldots + l_n + \gamma)$, where $\gamma$ represents an ‘unknown’ symbol. With the state that corresponds to the $\gamma$ symbol in the corresponding automaton, we associate the type $T_{\text{any}}$ and a NULL automaton. At run time, if we get to this state then we do not go down the current branch, since anything may appear there. In a pre-processing stage, we replace any tag of the document that does not appear in the schema with $\gamma$.

Now we discuss the time complexity of the algorithm. The construction of a Glushkov automaton is quadratic in the size of the regular expression. Therefore the first stage of our algorithm, in which we translate a schema into automata, can be done in time $O(|S|^4)$. According to [36], running the algorithm for a document $D$ takes time $O(|D|\log|S|)$. Also, the scoping nodes need to be inserted into KeySN and KeyrefSN. Each insertion takes time $O(\log|D|)$, and there are $O(|D|)$ insertions for each key / keyref constraint ($O(|S||D|)$ for all constraints). Therefore the complexity of obtaining the scoping nodes is $O(|S|^4) + O(|D|\log|S|) + O(|S||D||\log|S||D|)$. For a fixed schema, the complexity is $O(|D||\log|D|)$.

• KeySelIdent: For a key $K$ with a selector expression $Sel$, we create an automaton that represents $Sel^{-1}$ (see Section 5.5 for the definition of a reverse expression). We run it in a bottom up traversal of the document tree. That is, an instance of the automaton is created for every leaf of the tree, and it climbs along the path to the root. In each state, we store the nodes that we encountered so far which are currently in this state. In each node we wait for the automata from all children to arrive before continuing up, so that we go over each node only once.

---

4The algorithm presented in [36] handles DTDs and a basic subset of XML Schema, while we handle also more advanced features of XML Schema. However, our algorithm’s execution is very similar.
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. When 
discovering that a node \( n \) is a selector-identified node of a node \( s \), we 
add the record \((n, [], \{s\})\) (the key-sequence will be calculated later) to 
KeySelIdent[\(K\)], or, if a record \((n, [], SN)\) already exists, we add \( s \) to 
\( SN \). In order to calculate the key-sequences, for each field expression 
\( f \) we create an automaton that represents the reverse expression \( f^{-1} \).

We run it in a bottom up traversal. When we reach a selector-identified 
ode, we know that nodes in accepting states are its fields (in a valid 
document, there should be only one such field for a selector-identified 
ode, and it should be a simple-type node). We add the field value to 
the key-sequence in KeySelIdent.

Complexity:

- When running the \( Sel^{-1} \) automaton, we go over each node once. 
  For each node, we check if it is a scoping node (by searching in 
  KeySN). This takes time \( O(|D| \log |D|) \).

- For each selector-identified node \( n \), we access KeySelIdent for 
each scoping node of \( n \) (this is because when we reach each of 
these scoping nodes, \( n \) is in an accepting state, and we need to 
either add a record to KeySelIdent or modify an existing record). 
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 \( O(|D|^2) \) accesses 
to KeySelIdent, and \( O(|D|^2 \log |D|) \) complexity \( O(|S||D|^2 \log |D|) \) 
for all keys in the schema). 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 \( O(\log |D| \times \sum_{x=1}^{h} b^x \times x) \), which is smaller 
than \( O(\log |D| \times h \times \sum_{x=1}^{h} b^x) \). Since \( \sum_{x=1}^{\log_b |D|} b^x = O(|D|) \), we get 
\( O(|D| \log^2 |D|) \) \( O(|S||D| \log^2 |D|) \) for all keys in the schema). From 
here on, weʼll use the term average case to denote the case where 
the tree has a branching factor of \( b \) and height \( O(\log_b |D|) \).

- Since there are \( O(|D|) \) selector-identified nodes, and each one has 
\( O(|S|) \) fields, calculating the key-sequences and saving them in 
KeySelIdent takes time \( O(|S||D| \log |D|) \) \( O(|S|^2|D| \log |D|) \) 
for all keys in the schema). This is the time it takes to run the 
\( f^{-1} \) automata (for each field expression \( f \), and accessing the 
KeySelIdent data structures for each (selector-identified node,
field node) pair (that is, $O(|S||D|)$ times).

The complexity of populating the KeySelIdent structure, for all keys in the schema, is $O(|S||D|^2 \log |D| + |S|^2 |D| \log |D|)$ in the worst case, and $O(|S||D| \log^2 |D| + |S|^2 |D| \log |D|)$ in the average case.

- **KeyInfo**: We populate these structures in two stages.

  1. First, each selector-identified node of the key needs to be inserted into the KeyInfo structure of each of its scoping nodes (this is done based on the KeySelIdent structure).

  2. 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 (and if the node is a scoping node, also the records for its selector-identified nodes), while removing duplicate key-sequences (that were added from the children’s structures). For a node $v$, $v$.KeyInfo[$K$] is calculated as follows. We insert the records from all children structures of $v$ into a temporary search tree, tmpKeyInfo, which is ordered by key-sequences (has the same structure as $v$.KeyInfo[$K$]). If we try to add a record with a key-sequence that already appears in tmpKeyInfo, this means that the key-sequence appears in more than one child structure. Thus, we save this key-sequence in a separate hash-table duplicateKeySequences, and we do not insert the record. Then, we insert the records from tmpKeyInfo into $v$.KeyInfo[$K$]. However, we disregard the records whose key-sequence appears in duplicateKeySequences. Also, if the key-sequence of a record already appears in $v$.KeyInfo[$K$] (due to a selector-identified node of $v$), we do not add the record.

**Complexity:**

- First stage (inserting each selector-identified node into the KeyInfo structures if its scoping nodes): each insertion takes time $O(|S| \log |D|)$ (since KeyInfo is a multi-search-tree). A selector-identified node has at most $O(h)$ scoping nodes, and thus the number of insertions is at most $O(h|D|)$. Thus, the complexity of this stage (for a single key constraint) is $O(|S|h|D| \log |D|)$.

- Seconds stage: For a node $v$, let the total number of records in $v$’s children structures be $N(v)$.
The time it takes to create $\text{tmpKeyInfo}$ for $v$ is $O(|S| \cdot N(v) \cdot logN(v))$, since we insert $N(v)$ records into the multi-search-tree $\text{tmpKeyInfo}$.

For all nodes in the document, we get $O(\sum_{v \in D} |S| \cdot N(v) \cdot logN(v)) = O(|S| \cdot log|D| \cdot \sum_{v \in D} N(v))$. Since each node has at most $O(h)$ ancestors, $\sum_{v \in D} N(v)$ is at most $O(h \cdot |D|)$, and thus we get $O(|S| \cdot h \cdot |D| \cdot log|D|)$.

To this complexity, we need to add the complexity of making the insertions of records (from the $\text{tmpKeyInfo}$ structures) into the $\text{KeyInfo}$ structures. Each selector-identified node is inserted into at most $h$ $\text{KeyInfo}$ structures, and each insertion takes time $O(|S| \cdot log|D|)$, since there are at most $O(|D|)$ records in a $\text{KeyInfo}$ structure (a node may appear only once in a single $\text{KeyInfo}$ structure). Thus, the complexity of insertions is $O(|S| \cdot h \cdot |D| \cdot log|D|)$.

Therefore, the complexity of populating the $\text{KeyInfo}$ structures of a key is $O(|S| \cdot h \cdot |D| \cdot log|D|)$, which is $O(|S| \cdot |D|^2 \cdot log|D|)$ in the worst case, or $O(|S| \cdot |D| \cdot log^2|D|)$ in the average case. For all keys in the schema, the complexity is $O(|S|^2 \cdot |D|^2 \cdot log|D|)$ (or $O(|S|^2 \cdot |D| \cdot log^2|D|)$).

- **KeyrefSelIdent** and **KeyrefInfo**: This is similar to creating the $\text{KeySelIdent}$ and $\text{KeyInfo}$ structures. The difference is that the union operations are not needed. The complexity is still $O(|S|^2 \cdot |D| \cdot log|D|) / O(|S|^2 \cdot |D| \cdot log^2|D|)$.

- **ChildrenKeyInfo**: As we go over the $\text{KeyInfo}[K]$ structures of the children of a node $n$ (in order to populate $n.\text{KeyInfo}[K]$), we also insert tuples into $n.\text{ChildrenKeyInfo}[K]$. Each selector-identified node may appear in the $\text{ChildrenKeyInfo}[K]$ structures of its ancestors. As calculated above, this amounts to $O(|D|^2)$ insertions in the worst case, or $O(|D| \cdot log|D|)$ insertions on average. The complexity for a key $K$ is $O(|S| \cdot |D| \cdot log^2|D|) / O(|S| \cdot |D| \cdot log^2|D|)$.

- **KeyFieldInfo**: As we calculate the key-sequence of a selector-identified node $s$ (when we create the $\text{KeySelIdent}[K]$ structure), if the $i$'th field expression evaluates to a node $f$ then we add the record $(f, \{(s, i)\})$ to $\text{KeyFieldInfo}[K]$, or add $(s, i)$ to the occurrences of $f$ if $f$ already appears in $\text{KeyFieldInfo}[K]$. There are $O(|D|)$ field nodes, and each one may be a field of its ancestors. Therefore (similarly to the above calculations), we access $\text{KeyFieldInfo}[K] \cdot O(|D|^2)$ times in the worst case, or $O(|D| \cdot log|D|)$ times on average, which takes time.
in the worst case, or $O(D)$ we get $O(D)$ which is an upper bound on the complexity of this problem. Given a document $(\text{which is an upper bound on the complexity of this problem})$, the problem of validating key and keyref constraints from scratch (i.e., non-incremental validation).

To the best of our knowledge, there are currently no complexity results for the creation of these structures will fail). Creating the structures takes time $O(D)$ in the worst case, or $O(D)$ on average (as explained in Section 2.3.3). For a keyref $KR$ and a scoping node $n$ of $KR$, as we create $n.KKeyrefInfo[KR]$, we search for every key-sequence of $n.KKeyrefInfo[KR]$ in $n.KKeyInfo[K]$, where $K$ is the key that $KR$ refers to. Each such search takes time $O(S)$ on average. As explained in Section 2.3.3 the total number of occurrences of selector-identified nodes in scoping nodes is $O(D)$ in the worst case, $O(D)$ on average. Since there are $O(S)$ keyref constraints, the complexity of performing these searches is $O(S)$ in the worst case, $O(S)$ on average.

Therefore, the complexity of validation is $O(S)$ in the worst case ($O(S)$ on average), or $O(D)$ on average for a fixed schema. 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 [27], a keyref constraint is defined by $C(B.lB \subseteq A.lA)$, where $A$, $B$, $C$ are elements defined in a DTD, $lA$ is a child element of $A$ and $lB$ 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 $b.lB = a.lA$. 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 much more complicated.

### 2.5 Space Complexity

The non-incremental validation algorithm described above, as well as the incremental validation algorithms, use the data structures described in Section 2.3. The sizes of these data structures, for a document \( D \) that conforms to a schema \( S \), are as follows.

- **KeySN**: For each key constraint, \( KeySN \) contains an entry for each scoping node. Thus its size is \( O(|S||D|) \).
- **KeyrefSN**: Same as \( KeySN \) - \( O(|S||D|) \).
- **KeySelIdent**: Contains an entry for each selector-identified node. Since the entry contains a list of scoping nodes, its size is \( O(h) \). Thus, the size of this data structure (for all key constraints) is \( O(|S||D|h) \).
- **KeyrefSelIdent**: Same as \( KeySelIdent \) - \( O(|S||D|h) \).
- **KeyFieldInfo**: Contains an entry for each node that serves as a field of another node. The entry contains the occurrences of the field (at most \( O(h) \) occurrences). Thus, the size of this data structure (for all key constraints) is \( O(|S||D|h) \).
- **KeyrefFieldInfo**: Same as \( KeyFieldInfo \) - \( O(|S||D|h) \).
- **KeyInfo**: Each selector-identified node may have an entry in the \( KeyInfo \) structure of each of its ancestors. Thus the size of these structures is \( O(|S||D|h) \).
- **KeyrefInfo**: Each selector-identified node has an entry in the \( KeyInfo \) structure of each of its scoping nodes (at most \( O(h) \) scoping nodes). Thus the size of these structures is \( O(|S||D|h) \).
- **ChildrenKeyInfo**: For each record of the \( KeyInfo \) structure of a node \( n \), there is a corresponding record in the \( ChildrenKeyInfo \) structure of \( n \)’s parent. Thus, the size of the \( ChildrenKeyInfo \) structures is \( O(|S||D|h) \).
If we consider the size of a key-sequence as $O(S)$, then the size of the KeyInfo structures is $O(|S|^2|D|h)$. This is also the total size of the data structures. In the worst case, this amounts to $O(|S|^2|D|^2)$. Note however, that the worst case is very uncommon. In the case where $h = O(log|D|)$, the size of the data structures is $O(|S|^2|D|log|D|)$. It is also important to note that in most real cases, a field node has only a single selector-identified node, a selector-identified node has only a single scoping node, and a scoping node cannot be a descendant of another scoping node. This greatly reduces the size of the data structures.

### 2.6 Validation Algorithms

Each algorithm performs an operation that changes the document. Changes are rolled back if performing the operation creates an invalid document. Validity is checked incrementally, i.e., without validating the changed document in its entirety. For simplicity, we assume one key constraint $K$ and one keyref constraint $KR$. Therefore, we refer to the data structures without explicitly indicating a key or keyref constraint (i.e., we use $x.KeyInfo$ to refer to $x.KeyInfo[K]$). Our algorithms can be easily extended to handle multiple key and keyref constraints.

#### 2.6.1 Changing the Value of a Simple-type Node

We define an update operation $update(f, newval)$ where $f$ is some simple-type node and $newval$ is the value to be assigned to it. We assume that $newval$ is different that 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 KeyFieldInfo and KeyrefFieldInfo structures. 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 (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 \( s.KeyrefInfo \).

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 2.3) 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 Figure 2.4. 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).

1. **Finding affected selector-identified nodes.** We search for \( f \) in \( KeyFieldInfo \) and \( KeyrefFieldInfo \), 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
Figure 2.4: Algorithm for changing the value of a simple-type node.

2. Updating the data structures of nodes along the path to the root. This stage is executed in the \texttt{UpdateNodes(KUpdates, KRUpdates, f)} function, depicted in Figure 2.5. This function, which is the heart of the algorithm, uses the functions \texttt{UpdateNode}_K and \texttt{UpdateNode}_{KR}, which we now describe, in order to update the data structures associated with a single node.

\texttt{UpdateNode}_K(y, x, changesInChild, KUpdates)

This function updates \texttt{y.KeyInfo} and \texttt{y.ChildrenKeyInfo}, and also inserts the appropriate key-sequences into \texttt{y.RemovedSequences}. These updates are done based on the set of key-sequence changes \texttt{KUpdates} and on the changes made to the \texttt{KeyInfo} structure of \texttt{y}'s child node \texttt{x}. These changes are passed in the set \texttt{changesInChild}. These are tuples of the form \((ks, n)\), where \texttt{ks} is a key-sequence whose record in \texttt{x.KeyInfo} has changed and \texttt{n} is the node that appears in the \texttt{new} record for \texttt{ks} in \texttt{x.KeyInfo} (if there is one). If there is no such new record (i.e., a record has been removed), \texttt{n} is \texttt{null}. A key-sequence may only appear once in \texttt{changesInChild}. The function \texttt{UpdateNode}_K returns the set of the changes it has made (in the same format as \texttt{changesInChild}), which is later used for updating the parent of \texttt{y}. We denote this set by \texttt{changes}. It is initialized to \(\emptyset\) and is returned as the result at the end of the function execution. Since a key-sequence \texttt{ks} can only appear once in \texttt{changes}, we use the syntax \texttt{changes[ks] := n} to denote the addition of the tuple \((ks, n)\) to \texttt{changes}. This means that if a tuple \((ks, n')\) already exists in \texttt{changes}, it is replaced by \((ks, n)\). The function is depicted in Figure 2.6. It uses the functions \texttt{ProcessChanges}, \texttt{HandleOldKeySequences} and \texttt{HandleNewKeySequences}, depicted in Figs.\footnote{\texttt{I.e.}, if \(r_1\) is the record for \texttt{ks} in \texttt{x.KeyInfo} before \texttt{UpdateNode}_K(x, ...) is executed (possibly \(r_1 = \texttt{null}\)), and \(r_2\) is the record for \texttt{ks} in \texttt{x.KeyInfo} after \texttt{UpdateNode}_K(x, ...) is executed (possibly \(r_2 = \texttt{null}\)), then \(r_1 \neq r_2\).}
Input: A node y, a node x, a set of changes changesInChild, a set of node updates KUpdates.
Output: A set of changes changes.

changes := ∅;
// changes is passed by reference, and may be changed within ProcessChanges().
ProcessChanges(y, changesInChild, changes);
HandleOldKeySequences(y, RelevantUpdates, changes);
HandleNewKeySequences(y, RelevantUpdates, changes);
y.RemovedSequences := ∅;
for each (ks, null) ∈ changes
  // null means that the entry in y.KeyInfo has been removed.
y.RemovedSequences += ks;
return changes;

Input: node, KUpdates.
Output: None.
if node.KeyrefInfo ̸= null {
  for each tuple (n, ks, ks′) in KUpdates
    if n appears in a record r ∈ node.KeyrefInfo
      // Once the key-sequence of n changes to ks', there will be no node
      // for n to reference
      exit(INVALID);
    }
  node.RemovedSequences := null; //reset
}

Figure 2.6: UpdateNodeK.

Figure 2.7: UpdateNodeKR.

For example, suppose that the value of c3.f in the document of Figure 2.1 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 UpdateNodeK on b1, i.e., $changesInChild = \{( (1, 2), null), ( (5, 2), c3)\}$. As changesInChild is processed, b1.ChildrenKeyInfo is updated. 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 c3 is a selector-identified node of b1, $(c3, (1, 2), (5, 2)) ∈ RelevantUpdates$. First we process the ‘old’ key-sequence (1, 2), thus the record $(c3, (1, 2), True)$ is removed. Then we process the ‘new’ key-sequence (5, 2), thus the record $(c3, (5, 2), False)$ is removed and the record $(c3, (5, 2), True)$ is added to b1.KeyInfo.

UpdateNodeKR(y, KUpdates)
This function updates the KeyrefInfo structure of a node according to the received key-sequence changes ($KUpdates$) and also checks whether all references are valid. It is depicted in Figure 2.7.

Complexity
We consider the complexity of:

- Finding affected nodes. We access the KeyFieldInfo and KeyrefFieldInfo data structures according to the node id of f. Since these are search trees over $O(|D|)$ node identifiers, this takes time $O(log|D|)$.

- Executing UpdateNodes. We execute UpdateNodeK and UpdateNodeKR
Therefore, the complexity of the algorithm is $O(|S|h^2 \log |D|)$, or $O(h^2 \log |D|)$ for a fixed schema. On average, $h = \log |D|$ and we get $O(|S|\log^3 |D|)$. 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 $O(|S|h\log |D|)$. 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.

Figure 2.8: ProcessChanges.
Input: $y$, RelevantUpdates, changes (passed by reference).
Output: None.

HandleOldKeySequences($y$, RelevantUpdates, ByRef changes) {
    for each $(n, ks, ks') \in$ RelevantUpdates {
        Remove the record $(n, ks, True)$ from $y.KeyInfo$.
        if $y.ChildrenKeyInfo[ks]$ contains exactly one tuple $(child, nodeInTable)$ {
            add the record $(nodeInTable, ks, False)$ to $y.KeyInfo$.
            changes[ks] := nodeInTable;
            // If an entry for ks already exists in changes, it is replaced.
            // Such an entry may have been added by ProcessChanges.
            } else 
                changes[ks] := null; // This indicated deletion from $y.KeyInfo$.
        }
    }

Figure 2.9: HandleOldKeySequences.

Input: $y$, RelevantUpdates, changes (passed by reference).
Output: None.

HandleNewKeySequences($y$, RelevantUpdates, ByRef changes) {
    for each $(n, ks, ks') \in$ RelevantUpdates {
        Let $r$ be the record for $ks'$ in $y.KeyInfo$ (null if there is no such record);
        if $r \neq$ null {
            // The new key-sequence already appears in $y.KeyInfo$.
            if $r$.isSelectorIdentified = True
                // This means that $r$ existed prior to processing the
                // update, since the previous stages can only add records
                // with isSelectorIdentified = False.
                exit(INVALID);
                // All changes done to the data structures are rolled back
                // and the algorithm exits.
            } else {
                remove $r$ from $y.KeyInfo$.
                // We know that $n$ is a selector-identified node of $y$,
                // since $(n, ks, ks')$ is in RelevantUpdates.
                add the record $(n, ks', True)$ to $y.KeyInfo$.
                changes[ks'] := $n$;
            }
        }
    }

Figure 2.10: HandleNewKeySequences.

Correctness

The algorithm’s correctness follows from the following Lemmas.

**Lemma 2.2** If the update violates $K$ then the violation is discovered during the execution of UpdateNodes.

**Proof** If the update violates $K$ then there is a node $y$ which is a scoping node of $K$ and nodes $n_1, n_2$ which are selector-identified nodes of $K$ such that: (1) $y$ is a scoping node of $n_1$ and $n_2$. (2) After the update, $n_1$ and $n_2$ have the same key-sequence. The violation is discovered when we execute UpdateNode$_K$ for $y$. Before the update, these nodes have different key-sequences (since the document is valid prior to the update). Therefore,
at least one of these nodes appears in $K_{\text{Updates}}$. If only one appears, in a tuple $(n_i, ks, ks')$, we discover the violation as we process this tuple (in $\text{HandleNewKeySequences}$) and check for the existence of a record for $ks'$ with $\text{isSelectorIdentified} = \text{True}$ (before inserting a record $(n_i, ks', \text{True})$ into $y.\text{KeyInfo}$). If both nodes appear in $K_{\text{Updates}}$, we discover the violation as we process the second of their corresponding tuples in $K_{\text{Updates}}$ (during the execution of $\text{HandleNewKeySequences}$).

**Lemma 2.3** If during the execution of $\text{UpdateNodes}$ it is decided that the update violates $K$ then the update indeed violates $K$.

**Proof** Such a decision is reached during the execution of $\text{UpdateNode}_K$ for some node $y$, as we process a tuple $(n, ks, ks')$ and discover that a record for $ks'$ with $\text{isSelectorIdentified} = \text{True}$ already exists in $y.\text{KeyInfo}$. There are two possibilities. (1) The existing record was already there prior to this execution of $\text{UpdateNode}_K$, and the node in it does not appear in $K_{\text{Updates}}$ (had it appeared in $K_{\text{Updates}}$, we would have already removed this record, since in $\text{UpdateNode}_K$ we deal with the 'old' key-sequences before dealing with the 'new' key-sequences). (2) This record was added as part of the update (i.e., the key-sequence of the node in the record has changed). This means that the update changes the key-sequence of two selector-identified nodes within the same scope to the same value. In both cases, the update is indeed invalid.

**Lemma 2.4** If the update does not violate $K$ in the subtree rooted at a node $y$ then after executing $\text{UpdateNode}_K(y, ..., y.\text{KeyInfo}$ and $y.\text{ChildrenKeyInfo}$ correspond to the state of the updated document, and $y.\text{RemovedSequences}$ is accurate (i.e., $y.\text{RemovedSequences}$ contains key-sequences that appear in $y.\text{KeyInfo}$ before the update and do not appear there following the update).

**Proof** In order to prove this, we prove the following claim: After executing $\text{UpdateNode}_K(y, ..., y.\text{KeyInfo}$ and $y.\text{ChildrenKeyInfo}$ are correct (i.e., correspond to their definitions) and the changes returned from $\text{UpdateNode}_K$ are accurate. We prove this claim by induction on the height of the node $y$. Basis: $y = ns$, where $ns$ is the first key scoping node on the path from $f$ to the root that has a selector-identified node which is affected by the change. This means that $\text{changesInChild} = \emptyset$ and $y.\text{ChildrenKeyInfo}$ is not changed. $y.\text{ChildrenKeyInfo}$ need not be changed, since there are no scoping nodes of affected nodes below $ns$, and therefore, according to Lemma 2.1, the $\text{KeyInfo}$ structures of children of $ns$ do not need to be changed. The function $\text{UpdateNode}_K$ only processes the tuples in $K_{\text{Updates}}$.
which are ‘relevant’, i.e., tuples where the node is a selector-identified node of $y$. First we remove the old records of these nodes (i.e., the record $(n, ks, True)$ for a tuple $(n, ks, ks')$). Only then do we deal with the ‘new’ key-sequences. This ensures that there is no collision between the old key-sequence of some node and the new key-sequence of another node. After removing an ‘old’ record $(n, ks, True)$, we add a record according to $y.\text{ChildrenKeyInfo}[ks]$ if addition is needed. Then, for each tuple $(n, ks, ks')$ we add the record $(n, ks', True)$ to $y.\text{KeyInfo}$. If $y.\text{KeyInfo}$ already contains a record for $ks'$ with $\text{isSelectorIdentified} = False$, the new node is inserted in its stead.

This sequence of operations adheres to the definitions of the data structures and leaves them in a correct state. The set $changes$ that is returned from the function call is maintained during the function’s execution. Every time we change the record for a key-sequence $ks$ in $y.\text{KeyInfo}$, we change the tuple for $ks$ in $changes$. Therefore this set accurately reflects the changes made in the function.

Induction step: Assume the induction hypothesis holds for a node $x$ of height $h$ and prove it holds for $x$’s parent $y$ at height $h + 1$. By hypothesis, $changesInChild$ reflects the changes made to $x.\text{KeyInfo}$ during the execution of $\text{UpdateNode}_K(x, ...)$. Therefore, $y.\text{ChildrenKeyInfo}$ is updated correctly. As we update $y.\text{ChildrenKeyInfo}$, we may also add records with $\text{isSelectorIdentified} = False$ to $y.\text{KeyInfo}$ or remove such records from $y.\text{KeyInfo}$. This is because the existence of these records is based on the state of the children, which has now changed. After processing $changesInChild$, we proceed, as in the execution of $\text{UpdateNode}_K(ns, ...)$, to process $KUpdates$. This works the same as in $\text{UpdateNode}_K(ns, ...)$. Note that adding a record when processing $changesInChild$ does not prevent adding a record in this stage (of processing $KUpdates$), since only records with $\text{isSelectorIdentified} = False$ are added as we process $changesInChild$.

Also note that there is no course of execution in which we add a record for some key-sequence $ks$ that does not appear in $y.\text{KeyInfo}$ and afterwards remove the record without replacing it with another record for $ks$. Therefore, if a tuple $(ks, null)$ appears in $changes$ then indeed $ks$ is a key-sequence that appears in $y.\text{KeyInfo}$ prior to the execution of $\text{UpdateNode}_K(y, ...)$. and does not appear in $y.\text{KeyInfo}$ after the execution.

When executing $\text{UpdateNode}_K(y, ...)$, $y.\text{RemovedSequences}$ is set to contain key-sequences $ks$ that appear in a tuple $(ks, null)$ in $changes$. The existence of such a tuple $(ks, null)$ implies that we have removed the key-sequence from $y.\text{KeyInfo}$ and therefore the key-sequence needs to appear in $y.\text{RemovedSequences}$.

**Lemma 2.5** If during the execution of $\text{UpdateNodes}$ it is decided that the
update violates $KR$ then the update indeed violates $KR$.

**Proof** Such a decision is reached when $UpdateNode_{KR}$ is executed for a $KR$ scoping node $x$, such that $K$ is not violated in the subtree rooted at $x$ (since such a violation would be discovered before discovering a violation of $KR$), and one of the following occurs. (1) There is a $KR$ selector-identified node $n$ of $x$, whose new key-sequence is $ks'$, and $ks'$ does not appear in $x.KeyInfo$. (2) $x.KeyrefInfo$ contains, after updating it according to $KRUpdates$, a record $(n, ks)$ where $ks$ appears in $x.RemovedSequences$. When $UpdateNode_{KR}$ is executed, $x.KeyInfo$ and $x.RemovedSequences$ are correct (according to Lemma 2.4). Therefore, this is indeed a violation of $KR$. 

**Lemma 2.6** If the update violates $KR$, and does not violate $K$, then the violation is discovered, during the execution of $UpdateNodes$.

**Proof** If an update violates $KR$ then there is a scoping node of $KR$, $y$, and a selector-identified node of $KR$ within the scope of $y$, $n$, such that after the update $n$ references a key-sequence that does not appear in $y.KeyInfo$. According to Lemma 2.4 $y.RemovedSequences$ and $y.KeyInfo$ are correct before $UpdateNode_{KR}$ is executed. Thus, the violation is discovered in $UpdateNode_{KR}(y, KRUpdates)$.

**Lemma 2.7** If the update does not violate $K$ or $KR$ 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 $KRUpdates$, and changes the key-sequences stored in the $KeyrefInfo$ data structures according to $KRUpdates$. This updates the $KeyrefInfo$ data structures as required.

**Theorem 2.1** 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.

**Proof** Follows from the above Lemmas.
2.6.2 Transactions: Changing the Values of a Set of Simple-type Nodes

We define an update operation \( \text{update}((f_1, \text{newval}_1), ..., (f_m, \text{newval}_m)) \) where for \( 0 \leq i \leq m \), \( f_i \) is some simple-type node and \( \text{newval}_i \) is the value to be assigned to it. Note that simply doing these updates in order by using the algorithm of Section 2.6.1 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 Figure 2.1. Recall the definitions of key and keyref constraints for this document, presented in Section 2.2. The scoping nodes of the key are the \( B \) nodes. The selector of the key is \( ./C|.B/C \) 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 \( \text{KeyInfo} \), \( \text{ChildrenKeyInfo} \) and \( \text{KeyrefInfo} \) structures are depicted in Figure 2.3. We perform the following update: \((e.f,6), (c_1.f,6), (c_2.f,5), (c_3.f,5)\).

The idea behind the algorithm: As in the case of a single change, we can find the affected selector-identified nodes according to the KeyFieldInfo and KeyrefFieldInfo structures. 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 \((r + 1 \text{ or more})\), 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.
The Algorithm

Input: A schema $S$, a document $D$ (represented in memory), tuples $(f_1, newval_1)$
$,...,(f_m, newval_m)$, $m > 1$, 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 2.3) have been created, and are correct (i.e., reflect the state of $D$).

Post-conditions: The data structures are correct. If the result is IN-
VALID 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 KeyFieldInfo and KeyrefFieldInfo, 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 KeyrefSelIdent. 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, $KRUpdates$), of the form
(node, old key-sequence, new key-sequence). A node may appear only once in
$KUpdates$ (respectively, $KRUpdates$). In our running example, we perform
the update $(e.f,6)$, $(c_1.f,6)$, $(c_2.f,5)$, $(c_3.f,5)$. Therefore, $KRUpdates =$
$\{(e,(1,2),(6,2))\};$
$KUpdates = \{(c_1,(4,2),(6,2)),(e_2,(3,2),(5,2)),(c_3,(1,2),(5,2))\}.$

Let $KUpdates_i$ (resp., $KRUpdates_i$) be the set of key-sequence updates
$(n, ks, ks') \in KUpdates$ (resp., $KRUpdates$) 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
counter[$v$], initially 0 (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 counter[$v$] by 1. $JN$ contains
all nodes $n$ such that counter[$n$] $\geq 2$. If $n \in JN$ and counter[$n$] = $k$, we say
that the rank of $n$ is $k$, denoted rank($n$) = $k$. Next, if $n_1 \in JN$, $n_2 \in JN$,
rank($n_1$) = rank($n_2$) and $n_1$ is an ancestor of $n_2$, then we remove $n_1$ from
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 function BeforeJN, depicted in Figure 2.12. This function uses the functions UpdateNode\(_K\) and UpdateNode\(_{KR}\), described in section 2.6.1. 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 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 \( n_i.KeyInfo \) in \( n_i.changes \) (bold line, Figure 2.12). This is used when, in the next stage of the algorithm, we update the Join Node that we have reached.

In this stage of our running example, we update nodes along the path from each changed field to \( b1 \) (which is the only Join Node). Since the only node with a non-empty KeyrefInfo structure is \( b1 \), no changes are made to KeyrefInfo structures in the UpdateNode\(_{KR}\) calls of this stage. We now describe how the KeyInfo and ChildrenKeyInfo structures are updated.
Input: \( JN \) (passed by reference), \( K\text{Updates} \).
Output: None.
\[ r := 2; \]
\[ \text{while } JN \neq \emptyset \{ \]
\[ \text{for each } n \in JN \text{ s.t. } \text{rank}(n) = r \{ \]
\[ \text{changesInNode} := \text{UpdateNode}_{K}(n, K\text{Updates}); \]
\[ \text{remove } n \text{ from } JN; \]
\[ \text{ProcessChanges}_{JN}(n, \text{changesInNode}); \]
\[ \text{currentNode} := n\text{.parent}; \]
\[ \text{changeInChild} := \text{changesInNode}; \]
\[ // \text{Continue up the tree, until we reach} \]
\[ // \text{either the next join node (which is guaranteed} \]
\[ // \text{to have a higher rank) or the root.} \]
\[ \text{while } (\text{currentNode} \in JN \text{ and} \]
\[ \text{currentNode} \neq \text{null} \{ \]
\[ \text{changesInNode} := \text{UpdateNode}_{K}(\text{currentNode}, \text{currentChild}, \text{changesInNode}, K\text{Updates}); \]
\[ \text{ProcessChanges}_{JN}(\text{currentNode}, \text{currentChild}, \text{changesInNode}, \text{changesInChild}); \]
\[ \text{currentNode} := \text{currentNode}\text{.parent}; \}
\[ // \text{Save changes for updating} \]
\[ // \text{the parent Join Node} \]
\[ \text{currentChild.changes} := \text{changesInChild}; \]
\[ r++; \}
\[ \} \]

Figure 2.13: FromJN.

Figure 2.14: UpdateJNNode_{K}.

On the path starting at \( e.f \), we update only \( e \). Since \( e \) is not a scoping node, \( e.KeyInfo \) and \( e.KeyRefInfo \) remain empty. The same holds for \( c1 \), on the path starting at \( c1.f \).

On the path starting at \( c2.f \), \( c2 \)'s structures are not changed. When we execute \( \text{UpdateNode}_{K}(b2, \ldots) \), \( \text{RelevantUpdates} = (c2, (3, 2), (5, 2)) \) and therefore we remove the record \((c2, (3, 2), True)) \) from \( b2.KeyInfo \) and add the record \((c2, (5, 2), True)) \). Then \( \text{UpdateNode}_{K}(d, b2, \ldots) \) is executed, with \( \text{changesInChild} = \{((3, 2), null), ((5, 2), c2)\} \). Since \( d \) is not a scoping node, \( \text{RelevantUpdates} \) is empty and so we only need to process \( \text{changesInChild} \). We remove the tuple \((b2, c2) \) from \( d.ChildrenKeyInfo[(3, 2)] \) and remove the record \((c2, (3, 2), False)) \) from \( d.KeyInfo \). We add the tuple \((b2, c2) \) to \( d.ChildrenKeyInfo[(5, 2)] \) and add the record \((c2, (5, 2), False)) \) to \( d.KeyInfo \). We set \( \text{d.changes} \) to \( \{((3, 2), null), ((5, 2), c2)\} \).

On the path starting at \( c3.f \), \( c3 \)'s structures are not changed. When we execute \( \text{UpdateNode}_{K}(b3, \ldots) \), \( \text{RelevantUpdates} = (c3, (1, 2), (5, 2)) \) and therefore we remove the record \((c3, (1, 2), True)) \) from \( b3.KeyInfo \) and add the record \((c3, (5, 2), True)) \). We set \( b3\text{.changes} \) to \( \{((1, 2), null), ((5, 2), c3)\} \).

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 function FromJN, depicted in Figure 2.13. This function uses the functions \( \text{UpdateNode}_{K} \) (depicted in Figure 2.6) and \( \text{UpdateNode}_{KR} \).
Input: A node y; changes in y so far - changes.
Output: None.

```csharp
ProcessChangesJN(y, ByRef changes) {
    changedSequences := ∅;
    for each c in y.childNodes
        for each (ks, n) ∈ c.changes
            if ks /∈ changedSequences
                changedSequences += ks;
            if y.ChildrenKeyInfo[ks] contains a tuple (c, n')
                remove this tuple;
            if n ≠ null
                add the tuple (c, n) to y.ChildrenKeyInfo[ks];
    }
    for each c in y.childNodes
        c.changes := null;
    for each ks in changedSequences
        if there is no x such that (x, ks, True) ∈ y.KeyInfo
            if ∃ such that (x, ks, False) ∈ y.KeyInfo
                change[x] := null;
            if y.ChildrenKeyInfo[ks] has exactly one tuple (c, nodeInC)
                add the record (nodeInC, ks, False) to y.KeyInfo;
                changes[ks] := nodeInC;
        }
    }
}
```

Figure 2.15: ProcessChangesJN.

(depicted in Figure 2.7). It also uses the function UpdateJNNodeK (see bold line in Figure 2.13), depicted in Figure 2.14. This function is very similar to UpdateNodeK. The difference is that UpdateNodeK receives changes from exactly one child. UpdateJNNodeK, on the other hand, needs to handle changes propagated through possibly several children. These changes are processed in the function ProcessChangesJN, depicted in Figure 2.15 (instead of calling ProcessChanges as in UpdateNodeK).

In this stage of our running example, we update the Join Node b1. First we call UpdateJNNodeK(b1, KUpdates). In this function call, we first process the changes in b1’s children (i.e., x.changes for each child x). After updating b1.ChildrenKeyInfo, we remove (c2, (3, 2), False) from b1.KeyInfo. The key-sequence (1, 2) is not yet removed from b1.KeyInfo, since it appears with isSelectorIdentified = True.

We set b1.ChildrenKeyInfo[(5, 2)] to {(d, c2), (b3, c3)}, and therefore we do not yet add the key-sequence (5, 2) to b1.KeyInfo. Then, we process RelevantUpdates,

```
RelevantUpdates = {(c1, (4, 2), (6, 2)), (c3, (1, 2), (5, 2))}.
```

Thus, we remove (c1, (4, 2), True) and (c3, (1, 2), True), and add (c1, (6, 2), True) and (c3, (5, 2), True). b1.RemovedSequences is set to {(1, 2), (3, 2), (4, 2)}.

6Recall that these are the tuples (n, ks, ks') ∈ KUpdates such that n is a selector-identified node of b1.
Then, we call $UpdateNode_{KR}(b_1, KRUpdates)$. 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 Figure 2.16.

**Complexity**

In order to perform $m$ changes (as a transaction), the data structures ($KeyInfo$, $ChildrenKeyInfo$ and $KeyrefInfo$) of at most $O(m \times h)$ nodes need to be updated. In order to update each one, we need to perform at most $O(m \times h)$ 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 $O(m^2h^2|S|\log|D|)$.

**Correctness**

**Lemma 2.8** If the update violates $K$ then the violation is discovered, either during the execution of BeforeJN or during the execution of FromJN.

**Proof** If the update violates $K$ then there is a node $y$ which is a scoping node of $K$ and nodes $n_1, n_2$ which are selector-identified nodes of $K$ such that: (1) $y$ is a scoping node of $n_1$ and $n_2$. (2) After the update, $n_1$ and $n_2$ have the same key-sequence. The violation is discovered when we execute $UpdateNode_{K}(y, ...)$ or $UpdateJNNode_{K}(y, ...)$. Before the update, these nodes have different key-sequences (since the document is valid prior to the update). Therefore, at least one of these nodes appears in $KUpdates$. If only one appears, in a tuple $(n_i, ks, ks')$, we discover the violation as we process this tuple and check for the existence of a record for $ks'$ with $isSelectorIdentified = True$ (before inserting a record $(n_i, ks', True)$ into $y.KeyInfo$). If both nodes appear in $KUpdates$, we discover the violation as we process the second of their corresponding tuples in $KUpdates$.  

Figure 2.16: Running example: $KeyInfo$, $ChildrenKeyInfo$ and $KeyrefInfo$ after the update.
Lemma 2.9 If it is decided that the update violates $K$ then the update indeed violates $K$.

**Proof** Such a decision is reached during the execution of $UpdateNode_K$ or $UpdateJNNode_K$ for some node $y$, as we process a tuple $(n, ks, ks')$ and discover that a record for $ks'$ with $isSelectorIdentified = True$ already exists in $y.KeyInfo$. There are two possibilities. (1) The existing record was already there prior to this execution of $UpdateNode_K$ or $UpdateJNNode_K$, and the node in it does not appear in $KUpdates$ (had it appeared in $KUpdates$, we would have already removed this record, since we deal with the ‘old’ key-sequences before dealing with the ‘new’ key-sequences). (2) This record was added as part of the update (i.e., the key-sequence of the node in the record has changed). This means that the update changes the key-sequence of two selector-identified nodes within the same scope to the same value. In both cases, the update is indeed invalid.

Lemma 2.10 If the update does not violate $K$ in the subtree rooted at a node $y$ then after executing $UpdateNode_K(y, ...)$ or $UpdateJNNode_K(y, ...)$, $y.KeyInfo$ and $y.ChildrenKeyInfo$ correspond to the state of the updated document, and $y.RemovedSequences$ is accurate (i.e., $y.Removedsequences$ contains key-sequences that appear in $y.KeyInfo$ before the update and do not appear there following the update). Also, the set of changes returned from $UpdateNode_K(y, ...)$ or $UpdateJNNode_K(y, ...)$ is correct.

**Proof** We prove the lemma by induction on $y$’s rank $r$.

Basis: $r = 1$ (if $r = 0$ then there is no change in the subtree rooted at $y$). This means that $y$ is not a join node, and its data structures are updated during the execution of $BeforeJN$. That is, $y$’s data structures are updated as the path from a single field is traversed. This is very similar to the algorithm of Section 2.6.1 and the proof of correctness for this case is very similar to the one in Lemma 2.4.

Induction Step: Assume the induction hypothesis holds for nodes of rank $\leq r$ and prove it holds for node $y$ of rank $r + 1$.

- If $y$ is a join node, its data structures are updated in the call to $UpdateJNNode_K(y, ...)$. This function works similarly to $UpdateNode_K$. The difference is that it does not receive a set of changes made to the $KeyInfo$ structure of a specific child. Rather, it processes the changes made to the $KeyInfo$ structures of all children. The rank of each child $c$ of $y$ is at most $r$. Thus, by the induction hypothesis, $c.changes$ is correct (since these are the changes returned by $UpdateNode_K(c, ...)$ or $UpdateJNNode_K(c, ...)$). Therefore, $UpdateJNNode_K(y, ...)$ updates
the data structures correctly (the proof is similar to the one regarding $UpdateNode_{K}$, in Lemma 2.4).

- If $y$ is not a join node, then it is on the path from a join node $x$ of rank $r+1$ to a join node of rank $\geq r+2$, or to the root. $y$’s data structures are updated in the call to $UpdateNode_{K}(y,\ldots)$. As explained above, the data structures of $x$ are updated correctly. Then, the path from $x$ to $y$ is updated, much in the same way as the path from a field to the root is updated in the algorithm of Section 2.6.1. The proof of correctness is similar to the one in Lemma 2.4.

**Lemma 2.11** If it is decided that the update violates $KR$ then the update indeed violates $KR$.

**Proof** Such a decision is reached when $UpdateNode_{KR}$ is executed for a $KR$ scoping node $x$, such that $K$ is not violated in the subtree rooted at $x$ (since such a violation would be discovered before discovering a violation of $KR$), and one of the following occurs. (1) There is a $KR$ selector-identified node $n$ of $x$, whose new key-sequence is $ks'$, and $ks'$ does not appear in $x.KeyInfo$. (2) $x.KeyrefInfo$ contains, after updating it according to $KRUpdates$, a record $(n,ks)$ where $ks$ appears in $x.RemovedSequences$. When $UpdateNode_{KR}$ is executed, $x.KeyInfo$ and $x.RemovedSequences$ are correct (according to Lemma 2.10). Therefore, this is indeed a violation of $KR$.

**Lemma 2.12** If the update violates $KR$, and does not violate $K$, then the violation is discovered.

**Proof** If an update violates $KR$ then there is a scoping node of $KR$, $y$, and a selector-identified node of $KR$ within the scope of $y$, $n$, such that after the update $n$ references a key-sequence that does not appear in $y.KeyInfo$. According to Lemma 2.10, $y.RemovedSequences$ and $y.KeyInfo$ are correct before $UpdateNode_{KR}$ is executed. Thus, the violation is discovered in $UpdateNode_{KR}(y,\ldots)$.

**Lemma 2.13** If the update does not violate $K$ or $KR$ then after executing $BeforeJN$ and $FromJN$, the $KeyrefInfo$ data structures, of all nodes, correspond to the state of the updated document.

**Proof** $UpdateNode_{KR}$ is called for each ancestor of a changed field. For $KR$ scoping nodes, key-sequences stored in the $KeyrefInfo$ data structures are changed according to $KRUpdates$. This updates the $KeyrefInfo$ data structures as required.
Theorem 2.2 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.

Proof Follows from the above Lemmas.

2.6.3 Adding a Subtree

We define an operation $AddSubTree(p, T, i)$, where $p$ is a node in the document $D$ and $T$ is a data tree. Let $root(T)$ be the root of $T$. $root(T)$ is to be added as the $i$’th child of $p$.

The idea behind the algorithm: Adding a subtree can add new scoping nodes (and selector-identified nodes of these scoping nodes) or add new selector-identified nodes to existing scoping nodes. It cannot change the fields of existing selector-identified nodes (if a new node is a field of an existing one, it means that in the context of the existing node, a field expression evaluates to more than one node, which is not allowed). We need to identify scoping nodes and selector-identified nodes in the new subtree. We also need to verify that existing selector-identified nodes do not have new fields because of the addition. Then, we need to update the data structures associated with the nodes on the path from the point of insertion to the root (only those nodes may be affected).

The Algorithm and its Complexity

Input: A schema $S$, a document $D$ (represented in memory), a node $p$ in $D$, a data tree $T$ and an integer $i$.

Output: A result - VALID or INVALID.

Pre-conditions: $D$ is valid with respect to $S$. $p$ has at least $i$ children ($0..i−1$). The data structures corresponding to $D$ (as described in Section 2.3) 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, $T$ appears as a subtree of $p$, where $root(T)$ is the $i$’th child of $p$, and the document is otherwise unchanged.

- Add the string values in $T$ to the TRIE structure and update $MapInfo$. This takes time $O(|T| \ast log(|D| + |T|))$.

- Identify scoping nodes (of $K$ or $KR$) in $T$. This can be done in time $O(|S|^4 + |T| \ast log|S|)$, as explained in Section 2.3.3. The $O(|S|^4)$ part
stems from representing the schema using automata. Since this part is done once for a schema, we do not count it here and therefore this stage takes time $O(|T| \ast \log |S|)$. We also need to add the scoping nodes of $T$ to $KeySN$ and $KeyrefSN$, which takes time $O(|T| \ast \log |D|)$.

- Identify selector-identified nodes in $T$. Keep them in the data structures $NewKeySelIdent$ and $NewKeyrefSelIdent$, which are defined the same as $KeySelIdent$ and $KeyrefSelIdent$, respectively, with the exception that they contain only selector-identified nodes that belong to $T$. The key-sequences of the nodes will be calculated in the next stage of the algorithm. Let $Sel_K$ and $Sel_{KR}$ be the selector expressions of $K$ and $KR$, respectively. There are two ways to do this. 1. For every scoping node of $K$, execute $Sel_K$. Since there are $O(|D| + |T|)$ such scoping nodes, and $Sel_K$ is of size $O(|S|)$, this takes time $O(|S| \ast (|D| + |T|)^2)$ (see [15]). 2. For every node in $T$, execute $Sel_{K}^{-1}$ (the reverse expression of the selector $Sel_K$). See Section 5.5 for the definition of a reverse expression, and the proof that one exists for every XPath' expression. Note that XPath' is a superset of restricted XPath.) and for each node in the result set check if it is a scoping node. This takes time $O(|T| \ast (|S| \ast (|D| + |T|) + h_{new} \ast \log |D|))$, where $h_{new}$ is the height of the new document tree (after adding $T$), which is at most $|D| + |T|$. This is done similarly for $KR$. $Sel_{K}^{-1}$ is the reverse expression of $Sel_K$.

- Identify fields in $T$. Let $f_i$ be the field expressions of $K$. For every node in $T$, we execute $f_i^{-1}$ for every field expression $f_i$. If the result set of such an execution contains selector-identified nodes that do not belong to $T$ (that is, nodes that are found in $KeySelIdent$ or $KeyrefSelIdent$), then the update is INVALID. Otherwise, we save the information in $NewKeySelIdent$ and in $KeyFieldInfo$. This is done similarly for $KR$. We consider the complexity of this stage. For $O(|T|)$ nodes, we evaluate $O(|S|)$ expressions, each of size $O(|S|)$, where the evaluation is done on a document of size $O(|D| + |T|)$. Therefore, this takes time $O(|S|^2 \ast |T| \ast (|D| + |T|))$

- Create the $KeyInfo$, $ChildrenKeyInfo$ and $KeyrefInfo$ structures for the nodes of $T$. This can be done in time $O(|S| \ast |T|^2 \log |T|)$, as explained in Section 2.3 (where these structures are created for a document $D$). As these structures are created, we also make sure that for each scoping node of $K$ in $T$, there is no key-sequence that appears in more than one selector-identified node (otherwise, the update is INVALID). This is detected as records are inserted into the $KeyInfo$
Input: $p$, $t$
Output: None.

```plaintext
changesInChild := ∅;
currentNode := $p$;
currentChild := $t$;
For each record $(n, ks, b)$ in $t.KeyInfo$
  changesInChild[ks] := n;
while (currentNode ≠ null) {
  changesInNode := UpdateNodeNew(currentNode, currentChild, changesInChild);
  currentChild := currentNode;
currentNode := currentNode.parent;
  changesInChild := changesInNode;
}
```

Figure 2.17: $Update_{New}$.

We now consider the complexity of this stage. We execute $Update_{Node_{New}}$ $O(h)$ times. In each such execution, the most time consuming operations are the searches of key-sequences in $KeyInfo$ structures. There are $O(|T|)$ such searches (because there are at most $O(|T|)$ selector-identified nodes of $K$ or $KR$ in $T$), and each one takes time $O(|S| \ast \log |D|)$ (searching within $O(|S|)$ levels of nested search trees). Thus the complexity of executing $Update_{New}$ is $O(|S| \ast |T| \ast h \ast \log |D|)$, which is at most $O(|S| \ast |T| \ast |D| \ast \log |D|)$.

- Add the records of $NewKeySelIdent$ to $KeySelIdent$.
- Add the records of $NewKeyrefSelIdent$ to $KeyrefSelIdent$.

Thus the complexity of the algorithm is at most $O(|S| |T|^2 \log |T| + |T|^2 \log |D| + |S|^2 |T|^2 + |S| |T||D| \log |D| + |S|^2 |T||D|)$.

**Correctness**

**Lemma 2.14** If $K$ or $KR$ are violated within $T$ then this is discovered in the algorithm. Otherwise, before the call to $UpdateNodes_{New}$, the data struc-
Input: A node $y$, a node $x$, a set of changes $\text{changesInChild}$.
Output: A set of changes.
changes := $\emptyset$;
// changes is passed by reference, and may be changed within $\text{ProcessChanges}()$.
$\text{ProcessChanges}(y, x, \text{changesInChild}, \text{changes})$;
for each $(n, ks, SN) \in \text{NewKeySelIdent}$
    if $y \in SN$
        Let $t$ be the record for $ks$ in $y$.KeyInfo (null if there is no such record);
        if $t \neq \text{null}$ and $t$.isSelectorIdentified = True
            exit(INVALID);
        if $t \neq \text{null}$
            remove $t$ from $y$.KeyInfo;
            add the record $(n, ks, \text{True})$ to $y$.KeyInfo;
            changes[ks] := $n$;
    }
for each $(n, ks, SN) \in \text{NewKeyrefSelIdent}$
    if $y \in SN$
        if no record for $ks$ exists in $y$.KeyInfo
            exit(INVALID);
        add the record $(n, ks)$ to $y$.KeyrefInfo;
    }
for each $(ks, \text{null}) \in \text{changes}$
    if a record for $ks$ exists in $y$.KeyrefInfo
        exit(INVALID);
return changes;

Figure 2.18: $\text{UpdateNodeNew}$.

The data structures for $T$ are created, and then references are checked. This is basically validation from scratch of $T$, and it is done in much the same way as described in Section 2.4.

Lemma 2.15 If $K$ is not violated within the subtree rooted at $y \notin T$ then after updating $y$.ChildrenKeyInfo and $y$.KeyInfo (in $\text{UpdateNodeNew}(y, ...)$, before going over $\text{NewKeyrefSelIdent}$), these data structures are correct, i.e., reflect the state of the subtree after the addition of $T$. Also, the changes returned from $\text{UpdateNodeNew}(y, ...)$ are accurate.

Proof We prove this claim by induction on the height of the node $y$.
Basis: $y = p$. In this case, $\text{changesInChild}$ contains tuples $(n, ks)$ for all $(n, ks, b) \in$ root$(T).\text{KeyInfo}$. This information is entered into $y$.ChildrenKeyInfo (since $y$ has a new child). Upon changing $y$.ChildrenKeyInfo[ks] for some key-sequence $ks$, we update $y$.KeyInfo if needed. Here we only add or remove records with $\text{isSelectorIdentified} = \text{False}$, since the existence of these records is based on the state of the children, which has now changed. Then, we process $\text{NewKeySelIdent}$. If a record $(n, ks, SN)$ appears in $\text{NewKeySelIdent}$ and $y \in SN$, this means that $n$ is a new selector-identified
node of y with key-sequence ks. Therefore, we remove the record for ks in y.KeyInfo if one exists and insert the record (n, ks, True). This sequence of operations adheres to the definitions of the data structures and leaves them in a correct state. The set changes that is returned from the function call is maintained during the function’s execution. Every time we change the record for a key-sequence ks in y.KeyInfo, we change the tuple for ks in changes. Therefore this set accurately reflects the changes made in the function.

Induction step: Assume the induction hypothesis holds for a node x of height h and prove it holds for x’s parent y at height h + 1. If K is not violated in y’s subtree then it is not violated in x’s subtree. Therefore, by hypothesis, changesInChild reflects the changes made to x.KeyInfo during the execution of UpdateNodeNew(x, ...). The update is performed much in the same way as in the induction basis.

**Lemma 2.16** If K is violated in a scoping node y \( \notin T \) then this is discovered in UpdateNodeNew(y, ...).

**Proof** A violation in y can occur if there is a node in T which is a selector-identified node of y and has the same key-sequence as another selector-identified node of y (either in T or not). For every new selector-identified node (a node in NewKeySelIdent) of y, we make sure that y does not already have a selector-identified node with the same key-sequence (i.e., a record in y.KeyInfo for this key-sequence with isSelectorIdentified = True), and then we add a new record to y.KeyInfo (and move on to the next node in NewKeySelIdent). Thus, the violation will be discovered.

**Lemma 2.17** If y is a scoping node of KR, K is not violated in the subtree rooted at y and KR is not violated in y, then y.KeyrefInfo is correct after UpdateNodeNew(y, ...).

**Proof** Adding T to the document cannot change the records that already exist in y.KeyrefInfo. For each added node which is a KR selector-identified node of y, a record needs to be added to y.KeyrefInfo. This is done by iterating NewKeyrefSelIdent and adding the record (n, ks) to y.KeyrefInfo if a tuple (n, ks, SN) appears in NewKeyrefSelIdent and y \( \in SN \).

**Lemma 2.18** If y is a scoping node of KR, K is not violated in the subtree rooted at y and KR is violated in y then this is discovered in UpdateNodeNew(y, ...).

**Proof** If adding T causes KR to be violated in y, this means that there is a KR selector-identified node of y that has no referenced node after the addition. According to Lemma 2.15 y.KeyInfo is updated correctly. Thus, the
violation is discovered once we update $y.KeyrefInfo$ (i.e., add records according to $T$) and check if each key-sequence in (the updated) $y.KeyrefInfo$ has a corresponding key-sequence in (the updated) $y.KeyInfo$.

**Lemma 2.19** If adding $T$ violates $K$ or $KR$ then this is detected by the algorithm.

**Proof** As we identify fields in $T$, we make sure that the addition of $T$ does not add fields to nodes in $D$. Other possible violations of $K$ or $KR$ are detected according to Lemma 2.14, Lemma 2.16 and Lemma 2.18.

**Lemma 2.20** If adding $T$ does not violate $K$ or $KR$ then after executing the algorithm, all data structures reflect the state of the document following the addition.

**Proof** The $KeyInfo$, $ChildrenKeyInfo$ and $KeyrefInfo$ data structures are correct according to Lemma 2.15 and Lemma 2.17. Updating the global data structures ($FieldInfoK$, $FieldInfoKR$, $KeySelIdent$, $KeyrefSelIdent$, $KeySN$ and $KeyrefSN$), as described in the algorithm, is straightforward.

**Theorem 2.3** The algorithm for adding a sub-tree works correctly, i.e., if adding a sub-tree $T$ violates $K$ or $KR$ then this is discovered, and if $K$ and $KR$ are not violated then all data structures are updated correctly, so as to reflect the state of the document following the addition.

**Proof** Follows from Lemma 2.19 and Lemma 2.20.

**Adding a Simple-type Node**

This is a simple case of adding a sub-tree, where $|T| = 1$. Thus the algorithm for adding a sub-tree can be used, with complexity $O(|D| \log |D||S| + |D||S|^2)$. Note that if we add a single node $n$, $n$ can be a selector-identified node of $K$ (respectively, $KR$) only if there is only one field in $K$ and $KR$, and the field expression of $K$ (respectively, $KR$) evaluates to $n$ when executed in the context of $n$. This is because a field of a selector-identified node can only be the node itself or a descendant of it.

**2.6.4 Deleting a Subtree**

We define an operation $Delete(t)$, where $t$ is some node. The operation deletes the sub-tree $T$, rooted at $t$, from the document.
The Algorithm and its Complexity

- We traverse the nodes of $T$ and insert their identifiers into a search tree $NODES_T$, in order to enable us to easily check whether a node $n$ belongs to $T$. This stage takes time $O(|T| \log |T|)$.

- We search for each $n \in T$ in $KeyFieldInfo$ and $KeyrefFieldInfo$. For each $s$ such that $n$ is a field of $s$ (according to $KeyFieldInfo$ or $KeyrefFieldInfo$), if $s \notin T$ then the operation is `INVALID`. This is because in such a case, removing $T$ will cause a field expression to evaluate to $\emptyset$ on $s$ (as it evaluates to a single node before the removal, and the field expression does not contain predicates). A search in $KeyFieldInfo$ or $KeyrefFieldInfo$ takes time $O(\log |D|)$, and yields $O(h)$ selector-identified nodes. The identifiers of these selector-identified nodes are searched for in $NODES_T$, in order to determine if the nodes belong to $T$. Therefore the complexity of this stage is $O(|T|(|\log |D| + h\log |T|)|)$.

- We update the data structures ($KeyInfo$, $ChildrenKeyInfo$ and $KeyrefInfo$) of the nodes along the path from $t.parent$ to $root(D)$. This is done in function $UpdateDeletion$ depicted in Figure 2.19. In order to update the data structures of a single node, this function uses the function $UpdateNodeDeletion$, depicted in Figure 2.20.

We now consider the complexity of this stage. We execute $UpdateNodeDeletion$ $O(h)$ times. In each such execution, we process at most $T$ child updates (since changes involve only key sequences that appear in $T$), which takes time $O(|S| |T| \log |D|)$ (since we need to search for the key-sequences in the $KeyInfo$ structure). It takes time $O(\log |D|)$ to check whether $y$ is a scoping node. If it is a scoping node of $K$ then we need to look for $T$'s nodes in $y.KeyInfo$, which takes time $O(|T| \log |D|)$. If it is a
Input: A node y, a node x, a set of changes changesInChild.
Output: A set of changes.

\[
\text{changes} := \emptyset; \\
// \text{changes is passed by reference, and may be changed within ProcessChanges()};
\]

ProcessChanges(y, x, changesInChild, changes);

if \( y \in \text{KeySN} \)
for every node s in NODES_T
if a record \((s, ks, True)\) appears in y.KeyInfo {
remove the record from y.KeyInfo;
changes[ks] := null;
if y.ChildrenKeyInfo[ks] contains a single tuple \((x, c)\) {
add the record \((c, ks, False)\) to y.KeyInfo;
changes[ks] := c;
}
}

if \( y \in \text{KeyrefSN} \)
for every node s in NODES_T
if a record \((s, ks)\) appears in y.KeyrefInfo {
remove the record from y.KeyrefInfo;
for every \((ks, null)\) \in changes
if y.KeyrefInfo contains a record for ks {
exit(INVALID);
}
}

return changes;

Figure 2.20: UpdateNodeDeletion.

scoping node of KR, we need to look for T’s nodes in y.KeyrefInfo, and we also need to look for the key sequences that were removed from y.KeyInfo in y.KeyrefInfo. Since there are at most \(|T|\) such key sequences, this takes time \(O(|S||T|\log D)|\). Therefore, each execution of UpdateNodeDeletion takes time \(O(|S||T|\log D)|\) and the complexity of executing UpdateDeletion is \(O(|S||T| \ast h \ast \log D)|\).

- For each \( n \in T \), we need to remove n’s entries in KeyFieldInfo, KeyrefFieldInfo, KeySN, KeyrefSN, KeySelIdent and KeyrefSelIdent. Since these are all search trees over node identifiers, the complexity of this stage is \(O(|T|\log D)|\).

The complexity of the algorithm is \(O(|S||T| \ast h \ast \log D)|\), or \(O(|T| \ast h \ast \log D)|\) for a fixed schema.

Correctness

Lemma 2.21 if K is not violated within the subtree rooted at \( y \not\in T \) then after updating y.ChildrenKeyInfo and y.KeyInfo \( \text{in UpdateNodeDeletion}(y, ...) \), these data structures are correct, i.e., reflect the state of the subtree after the deletion of \( T \). Also, the changes returned from UpdateNodeDeletion(y, ...) are accurate.

Proof We prove this claim by induction on the height of the node y.
Basis: \( y = p \). In this case, changesInChild contains tuples \((ks, null)\) for all
\((n, ks, b) \in t.KeyInfo\). This information is updated in \(y.ChildrenKeyInfo\) (since \(y\) lost its child \(t\)). Upon changing \(y.ChildrenKeyInfo[ks]\) for some key-sequence \(ks\), we update \(y.KeyInfo\) if needed. Here we only add or remove records with \(isSelectorIdentified = False\), since the existence of these records is based on the state of the children, which has now changed. Then, we remove, from \(y.KeyInfo\), the records of \(y\)'s selector-identified nodes (of \(K\)) that were removed from the document. According to the updated \(y.ChildrenKeyInfo\) structure, we may replace a removed record with a new record. This sequence of operations adheres to the definitions of the data structures and leaves them in a correct state. The set \(changes\) that is returned from the function call is maintained during the function's execution. Every time we change the record for a key-sequence \(ks\) in \(y.KeyInfo\), we change the tuple for \(ks\) in \(changes\). Therefore this set accurately reflects the changes made in the function.

Induction step: Assume the induction hypothesis holds for a node \(x\) of height \(h\) and prove it holds for \(x\)'s parent \(y\) at height \(h + 1\). By hypothesis, \(changesInChild\) reflects the changes made to \(x.KeyInfo\) during the execution of \(UpdateNodeDeletion(x, ...\)). The update is performed much in the same way as in the induction basis.

**Lemma 2.22** If \(K\) is violated then this is discovered in the algorithm.

**Proof** \(K\) can only be violated if the deletion of \(T\) removes fields of nodes which aren’t in \(T\) (because then after the deletion there will be a selector-identified node with a field expression that evaluates to an empty set). We check this in the second stage of the algorithm (before calling \(UpdateDeletion\)).

**Lemma 2.23** If \(y\) is a scoping node of \(KR\), \(K\) is not violated in the subtree rooted at \(y\) and \(KR\) is not violated in \(y\), then \(y.KeyrefInfo\) is correct after \(UpdateNodeDeletion(y, ...\))

**Proof** For each deleted node which is a \(KR\) selector-identified node of \(y\), a record needs to be deleted from \(y.KeyrefInfo\). This is done by iterating \(NODEST\) and looking for the nodes in \(y.KeyrefInfo\).

**Lemma 2.24** If \(y\) is a scoping node of \(KR\), \(K\) is not violated in the subtree rooted at \(y\) and \(KR\) is violated in \(y\) then this is discovered in \(UpdateNodeDeletion(y, ...\))

**Proof** If deleting \(T\) causes \(KR\) to be violated in \(y\), this means that there is a \(KR\) selector-identified node of \(y\) that has no referenced node after the deletion. According to Lemma 2.21 \(y.KeyInfo\) is updated correctly. Thus, the
violation is discovered once we update $y\. KeyrefInfo$ (i.e., delete records according to $T$) and check if each key-sequence in (the updated) $y\. KeyrefInfo$ has a corresponding key-sequence in (the updated) $y\. KeyInfo$. Note: $KR$ is also violated if a $KR$ selector-identified node which not in $T$ has a field in $T$. This is detected in the second stage of the algorithm (before calling $Update\_Deletion$).

**Lemma 2.25** If deleting $T$ violates $K$ or $KR$ then this is detected by the algorithm.

**Proof** Follows from Lemma 2.22 and Lemma 2.24.

**Lemma 2.26** If deleting $T$ does not violate $K$ or $KR$ then after executing the algorithm, all data structures reflect the state of the document following the addition.

**Proof** The $KeyInfo$, $ChildrenKeyInfo$ and $KeyrefInfo$ data structures are correct according to Lemma 2.21 and Lemma 2.23. Updating the global data structures ($FieldInfoK$, $FieldInfoKR$, $KeySelIdent$, $KeyrefSelIdent$, $KeySN$ and $KeyrefSN$), as described in the algorithm, is straightforward.

**Theorem 2.4** The algorithm for deleting a subtree works correctly, i.e., if deleting a subtree $T$ violates $K$ or $KR$ then this is discovered, and if $K$ and $KR$ are not violated then all data structures are updated correctly, so as to reflect the state of the document following the deletion.

**Proof** Follows from Lemma 2.25 and Lemma 2.26.

**Note:** In all these algorithms, if a change to the document turns out to be invalid during the course of the update operation then all changes to the data structures must be rolled back. Therefore, when changes are made to the data structures, the original data is saved in special data structures. This feature is not described here, though it was included in the implementation (Chapter 3).

**2.7 Chapter Summary**

In this Chapter we presented data structures that represent the state of a document with respect to key and keyref constraints. We explained how,
given a document and an XML schema, these data structures can be populated. We presented an algorithm that validates key and keyref constraints for a document (from scratch).

Furthermore, we presented a framework for incrementally validating such constraints. In this framework, the data structures are initially populated for the document. Then, operations that change the document may be executed. For each such operation, we presented an algorithm that traverses only the relevant parts of the document, and updates the data structures. While updating the structures, any violation of key or keyref constraints is detected. If there are no violations, the document is changed and the data structures correspond to the state of the changed document. We presented algorithms for changing the value of a simple-type node, changing the values of several nodes transactionally, adding a sub-tree and deleting a sub-tree.
Chapter 3

Implementation of Incremental Validation

3.1 Overview

We implemented the algorithms described in Chapter 2. The implementation is based on an existing validator, called XSV [38] (XML Schema Validator). XSV is an open-source validator written in Python. It 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 validate function, that receives an Element and a Schema object. 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. Our implementation uses a modified version of XSV in order to load the document into memory and create the necessary data structures. Incremental validation is performed on the in-memory representation of the document. We have implemented the algorithms so that they support any number of key and keyref constraints. The data structures keep information for all constraints defined in the schema (and allow access according to a key or keyref constraint). When the data structures of a node are updated, we update the information pertaining to all keys defined in the schema. If a key sequence ks is removed from n.keyTabs[K] (where K is a key constraint), we check that there is no reference to ks in n.KeyrefInfo[KR] for every keyref KR that refers to K.
3.1.1 Validation in XSV

When given the URLs of a document and a schema, XSV loads them into memory, where they are represented as a Document and a Schema object, respectively. Each type defined in the schema is represented by a Type object. Each complex type is associated with an FSM object, that represents a finite state machine that corresponds to the type’s content model. The document element and the schema object are passed to the validate function, and it calls the validateElement function. The content model of an element is validated in the function validateContentModel. This model can be empty, text only, or complex (i.e., an element with children, possibly an element with mixed content). In the latter case, validation is done in the validateElementModel function. This function applies the appropriate FSM to the element, and assigns types to the child elements according to the FSM. After validating the content model, validateElement calls validateChildTypes, which in turn calls validateElement for each child element. After validating the content model and the child elements (if there are any), validateElement calls validateKeys. This function receives an element node $n$, and an element declaration $decl$, that contains a list of all key and keyref constraints which are defined in the corresponding schema element (i.e., $n$ is a scoping node of each key and keyref in $decl$). For each key $K$ in $decl$, this function creates a keyTab, $n.keyTabs[K]$. The keyTab is a dictionary that maps key-sequences (i.e., tuples of values) to selector-identified nodes. $n.keyTabs[K]$ is created by evaluating the selector expression of $K$ in the context of $n$, and evaluating the field expressions for each resulting selector-identified node. For each resulting selector-identified node $s$, whose key-sequence is $ks$, $n.keyTabs[K]$ maps $ks$ to $s$. A validation error is reported if a field expression does not evaluate to a single simple-type node, or if two selector-identified nodes of $n$ have the same key-sequence. Then, information from the keyTabs of $n$’s children is propagated to $n.keyTabs$. For each child $c$, key $K'$, key-sequence $ks'$ and node $n'$, such that $c.keyTabs[K'][ks'] = n'$, $(ks', n')$ is propagated to $n.keyTabs[K']$ (i.e., $n.keyTabs[K'][ks'] = n'$) if the following conditions are satisfied.

1. There is no other child $c'$ of $n$ such that $ks'$ appears in $c'.keyTabs[K']$.
2. $ks'$ does not already appear in $n.keyTabs[K']$ (i.e., there is no selector-identified node of $n$ (for key $K'$) whose key-sequence is $ks'$.

Note that $n.keyTabs[K']$ may contain key-sequences from child keyTabs, even if $n$ is not a scoping node of $K'$.

The keyTabs are created bottom up. Since validateElement calls validateKeys only after validating the child elements, validateKeys is called for $n$ only af-
After creating $n$’s children (and the children’s keyTabs have been created).

After creating $n$’s keyTabs, validateKeys checks the keyref references. For each keyref $KR$ in decl, the selector expression of $KR$ is evaluated in the context of $n$, and the field expressions are evaluated for each resulting selector-identified node. For each resulting key-sequence $ks$, a validation error is reported if $n.keyTabs[KR.refer]$ does not contain a mapping for $ks$ (i.e., if there is no entry $n.keyTabs[KR.refer][ks]$), where $KR.refer$ is the key constraint that $KR$ refers to. The above (simplified) description of XSV’s flow of execution is illustrated in Figure 3.1.

3.1.2 Modifications to XSV

We have created a modified version of XSV, called XSV+. The modifications are as follows.

- Added a module for structural validation only, with no validation of identity constraints. This module, called StructuralValidation, contains a modified version of the validateElement function (and a validate function that calls it), that does not call the validateKeys function. This module is useful for checking the performance improvement of incremental validation, by validating structural constraints from scratch using this module and validating key and keyref constraints incrementally. The results can be compared to validation from scratch of both
structural constraints and identity constraints. Furthermore, the difference between the times of full validation from scratch and of structural validation from scratch may be used as an estimate of the time it takes XSV to validate identity constraints from scratch.

- Changed the keyTab structure so that $n.keyTab[ks]$ contains not only a node but also an indication of whether it is a selector-identified node of $n$, i.e., $n.keyTab[ks] = (n, isSelectorIdentified)$.

- Added data structures similar to the ones described in Section 2.3.

- Modified the code of the validateKeys function, so that it adds information to the new data structures. As a result, after an initial validation of the document from scratch using the XSV+, the data structures contain all the needed information (which is maintained by the incremental algorithms).

- Added functions that implement the incremental validation algorithms.

### 3.2 Implementation Details

#### 3.2.1 Data Structures

The data structures used in the implementation are similar to the ones described in Section 2.3. In order to enable fast lookups, most data structures use Python dictionaries (which are similar to hash tables, and are very easy to use in Python).

- **KeySelIdent.** A dictionary that maps a key constraint to another dictionary, which maps selector-identified nodes to their key-sequences. That is, for a key $K_1$ and a selector-identified node $n$ of $K_1$, whose key-sequence is $ks$, $KeySelIdent[K_1][n] = ks$ ($KeySelIdent$ does not support access according to a key-sequence, nor is such access needed).

- **KeyrefSelIdent.** A structure similar to $KeySelIdent$, used for keyref constraints. $KeyrefSelIdent[KR_1][n] = ks$.

- **KeyFieldInfo.** For each key constraint, maps a field node to a list of the node’s occurrences as a field, where an occurrence is identified by a selector-identified node and the index of the field in this node’s key-sequence. $KeyFieldInfo[K_1][field] = [(node1, index1), (node2, index2), ...]$
(note that it is possible for a node \( n \) to appear twice in the list of occurrences, if two different field expressions evaluate to the same field node in the context of \( n \)).

- **KeyrefFieldInfo.** Similar to **KeyFieldInfo**. \( \text{KeyrefFieldInfo}[KR_1][field] = [(\text{node}1, \text{index}1), (\text{node}2, \text{index}2), ...] \).

- **KeyScopingNodes.** Holds the scoping nodes of every key constraint. \( \text{KeyScopingNodes}[K_1] \) is a dictionary that holds all scoping nodes of \( K_1 \) as keys (the values are irrelevant).

- **KeyrefScopingNodes.** Similar to **KeyScopingNodes**.

- \( n.keyTabs \) holds key information for a node \( n \). \( n.keyTabs[K_1][ks] = (s, \text{isSelectorIdentified}) \), where \( K_1 \) is a key constraint, \( ks \) is a key-sequence, \( s \) is a node (a selector-identified node of \( n \) or one of \( n \)'s descendants) and \( \text{isSelectorIdentified} \) is a boolean value.

- \( n.KeyrefInfo \) holds keyref information for a node \( n \). If \( n \) is a scoping node of a keyref \( KR_1 \), \( s \) is a selector-identified node of \( n \), the key-sequence of \( s \) is \( ks \) and the key-sequence \( ks \) appears in counter selector-identified nodes of \( n \), then \( n.KeyrefInfo[KR_1][ks] = \text{counter} \) and \( n.KeyrefInfo[KR_1][s] = ks \).

The following data structures hold information that is used for performing a rollback of the changes made to the other data structures by an incremental validation algorithm.

- **keyTabsRollbackInfo.** Holds information used to roll back changes made to the **keyTabs** structures. If \( n.keyTabs[K][ks] = (s, b) \) and then the entry for \( n.keyTabs[K][ks] \) changes, the old entry is saved in **keyTabsRollbackInfo**, i.e., \( \text{keyTabsRollbackInfo}[n][K][ks] = (s, b) \). In the case where an entry is added by the validation algorithm, but no entry existed beforehand, \( s \) and \( b \) are null.

- **KeyrefInfoValueChangeRollbackInfo.** Holds information used to roll back changes made to the **KeyrefInfo** structures, specifically changes to the key-sequences of selector-identified nodes (these changes are made by the algorithms that validate value changes). If \( n.KeyrefInfo[KR][s] = ks \) and then the key-sequence of \( s \) changes, this information is saved in **KeyrefInfoValueChangeRollbackInfo**, i.e., \( \text{KeyrefInfoValueChangeRollbackInfo}[n][KR][s] = ks \).
• **KeyrefInfoAdditionRollbackInfo.** This data structure is similar to **KeyrefInfoValueChangeRollbackInfo**, but saves information regarding selector-identified nodes that were added. That is, if $s$ is a new selector-identified node of $n$ (added by the algorithm for adding a subtree) for keyref $KR$, with key-sequence $ks$, then $KeyrefInfoAdditionRollbackInfo[n][KR][s] = ks$.

• **KeyrefInfoDeletionRollbackInfo.** This data structure is similar to **KeyrefInfoValueChangeRollbackInfo**, but saves information regarding selector-identified nodes that were deleted. That is, if $s$ is a deleted selector-identified node of $n$ (i.e., $s$ is a part of a subtree that is deleted by the algorithm for deleting a subtree) for keyref $KR$, with key-sequence $ks$, then $KeyrefInfoDeletionRollbackInfo[n][KR][s] = ks$.

### 3.2.2 Populating the Data Structures

In XSV+, as part of the execution of the validateKeys function on a node $n$, we add relevant information to the data structures. As we build the keyTabs for $n$ (as in XSV), we add the additional boolean $isSelectorIdentified$ to each entry. If $n$ is a scoping node of a key $K$, the selector expression of $K$ is evaluated. For each resulting selector-identified node $s$, the field expressions are evaluated to produce field nodes $f_i$ (where $i$ is the index of a field of the key), whose values combine to create a key-sequence $ks$. As these evaluations are done, we add the following information to the data structures:

- $KeyScopingNodes[K][n] = True$
- $KeySelIdent[K][s] = ks$
- $KeyFieldInfo[K][f_i] += (s, i)$

Similar information is added for keyrefs. If $n$ is a scoping node of a keyref $KR$, the selector expression is evaluated. For each resulting selector-identified node $s$, the field expressions are evaluated to produce field nodes $f_i$, whose values combine to create a key-sequence $ks$. As these evaluations are done, we add the following information to the data structures:

- $KeyrefScopingNodes[KR][n] = True$
- $KeyrefSelIdent[KR][s] = ks$
- $KeyrefFieldInfo[KR][f_i] += (s, i)$
3.2.3 Incremental Validation Functions

- \textit{validateFieldChange}(document, schema, field, newVal). Receives a document, a schema, a field (an object which is an instance of the XMLInfoset.Element class) and a new value. The function changes the value of the field and updates the data structures according to the algorithm of Section 2.6.1. If, during the execution of the algorithm, a violation of constraints is discovered, all updates to the data structures are rolled back and the field is changed back to its original value.

- \textit{validateFieldChanges}(document, schema, changeList). Implements the algorithm of Section 2.6.2. \textit{changeList} is a list of tuples, each tuple of the form \((field, newVal)\).

- \textit{validateAddition}(document, schema, parent, child, index). Implements the algorithm of Section 2.6.3. \textit{parent} is an existing node of the document and \textit{child} is a new node (the root of a subtree), to be added as the \textit{i}'th child of \textit{parent}. The \textit{child} object may be created by loading an XML file into memory, and using the document element of the resulting Document object as \textit{child}.

- \textit{validateDeletion}(document, schema, t). Implements the algorithm of Section 2.6.4. \textit{t} is an existing node of the document, that needs to be deleted (along with its subtree).

3.3 Designing a Fully Incremental Validator

Incremental structural validation is mostly local to the changed node and possibly its parent:

- When changing the value of a simple-type node, we need to check that the new value conforms to the type of the node.

- When adding a subtree as the child of a node \(p\), we need to first structurally validate the new subtree, and then check that, after the addition, the string created by concatenating the labels of \(p\)'s children belongs to the regular language defined by \(p\)'s type.

- When deleting a subtree which is the child of a node \(p\), we need to check that, after the deletion, the string created by concatenating the labels of \(p\)'s children belongs to the regular language defined by \(p\)'s type.
Also, the validity of ID and IDREF attributes needs to be checked. This can be done by using a data structure that holds all ID values in the document and allows logarithmic access time, as described in [36].

Algorithms for adding and deleting a subtree are presented in [36], whereas [35] presents algorithms for adding and deleting a leaf node. Both present techniques for efficiently checking whether, after adding a node/subtree as a child of a node \( p \) (respectively, deleting a node/subtree which is a child of a node \( p \)), the string created by concatenating the labels of \( p \)'s children belongs to the regular language defined by \( p \)'s type.

Algorithms for incremental structural validation can easily be combined with our algorithms for incremental validation of key and keyref constraints, by simply performing the structural validation and then, if successful, validating key and keyref constraints. It seems that there is no point in coupling the structural validation and the key and keyref validation more tightly, since they check significantly different aspects of the document, and since structural validation does not need to traverse the path from the point of change to the root.

When adding a subtree, a structural algorithm needs to traverse the new subtree in order to structurally validate it. This can be used to collect information needed for validation of key and keyref constraints, thus improving performance. Note, however, that this does not improve the time complexity of the combined algorithm.

### 3.4 Chapter Summary

In this chapter we presented an overview of validation in the open-source validator XSV. Then we presented an implementation, based on XSV, of the incremental validation algorithms presented in Chapter 2. In this implementation, the document and the schema are first loaded into memory, where they are represented by objects. In the process, data structures that represent the state of the document with respect to key and keyref constraints are populated. The incremental algorithms operate on the memory representation of the document, and update the data structures.
Chapter 4
Experimentation

4.1 Specification of Experiments

In order to determine how much time is saved by checking key and keyref constraints incrementally, the experiments consist of performing a series of changes of the same type (i.e., changing the value of a node, changing several values, adding a subtree or deleting a subtree) on the in-memory representation of the document, and calculating the average validation time per change. We calculate the average validation time using two different validation methods:

1. Re-validate the entire document, using the validate function of XSV.

2. Re-validate the entire document with respect to structural constraints only (using the StructuralValidation module of XSV+), and then call the appropriate incremental algorithm for validating key and keyref constraints.

In this way, we compare full validation (structural constraints and identity constraints) using two different methods. One performs full validation from scratch, while the other performs only structural validation from scratch, and checks key and keyref constraints incrementally. The time difference between the two methods is clearly due to the incremental checking of key and keyref constraints.

Our experiments show that running the incremental algorithms often takes about 0.01 of the time it takes to run structural validation from scratch. Thus, the comparison described here does not give a clear picture of the improvement gained by using incremental validation of key and keyref constraints. It would be better to compare the time it takes to incrementally check key and keyref constraints with the time it takes to validate only these
constraints from scratch. Denote the time it takes to re-validate the document (both structural constraints and identity constraints) by $T_{\text{scratch}}$. Denote the time it takes to re-validate the document with respect to structural constraints by $T_{\text{struct}}$. Denote the time it takes to validate key and keyref constraints incrementally by $T_{\text{inc}}$. $T_{\text{scratch}} - T_{\text{struct}}$ is a good estimate of the time it takes to validate key and keyref constraints from scratch. Comparing this to $T_{\text{inc}}$ gives a good estimate of the improvement gained by incrementally validating key and keyref constraints.

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 Figure 4.1. The schema is also illustrated in Figure 4.2 that shows the structure of elements as a tree, where elements that may appear in a sequence of one or more instances are marked with $1..\infty$. 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 $1..2 \times size$. For every 'c' element, the number of 'c' child elements was chosen uniformly from the range $1..2 \times size$. For every 'a' element, $4 \times size$ key sequences were chosen, and $size$ 'e' elements were generated for each one. The documents mentioned above were generated with $size$ 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 $KeyInfo$ structures of siblings.

4.1.1 Single-value Change

For each document, we generated a sequence of single value changes. 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 by the following algorithm.
  - With 80% probability: Choose a key field uniformly from all key fields in the document. 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
Figure 4.1: The schema used for experiments.

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 selector-identified 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 f1 field. Let 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 $f_2$ value as $p$ but a different $f_1$ value. If such nodes exist, choose the new $f_1$ value randomly from their $f_1$ 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.

### 4.1.2 Multiple-value Change

We ran tests on the 346KB document. Each test is a sequence of transactional changes, and each transactional change consists of $\text{TRANSACTION\_SIZE}$ random simple-type node value changes, where $\text{TRANSACTION\_SIZE} \in \{100, 150, 200, 300, 700, 1000\}$ varies from test to test. We performed the tests both with XSV (i.e., validation from scratch) and with XSV+ (i.e., validation
We also performed the tests using XSV with both full validation from scratch and structural validation from scratch (in order to calculate the difference between the two), and using XSV+ with incremental validation only (i.e., no structural validation).

4.1.3 Addition

In order to check the addition algorithm, we changed the schema by allowing a 'c' element to have a 'cp' child (in addition to its 'c_' children), where a 'cp' element is defined as follows.

```xml
<xs:element name="cp">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="c" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="c_" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="e" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

We also changed the selector expression of the key from $c_ -$ to $c_ - | cp/c_-$ and changed the selector of the keyref from $e$ to $./e$. Thus, by adding a subtree whose root is a 'cp' node, we can simultaneously add a new key scoping node (a 'c' child of the new 'cp' node), new key selector-identified nodes to an existing scoping node (the parent 'c' node to which we add a 'cp' child), and new keyref selector-identified nodes to an existing scoping node (the ancestor 'a' node).

We created several XML files, of different sizes, each contains a subtree rooted at a 'cp' node: a 7.5KB subtree, a 24KB subtree, and two 71KB subtrees, one without 'e' nodes and one with 'e' nodes. We tried adding each one of the subtrees to each one of the three XML documents. For each combination of document and subtree, we loaded the document and subtree to memory, and ran a series of tests. In each test, we added the subtree as a child of the first 'c' node (in document order), and randomized the field values of the subtree (by replacing some of them with new values).

Measurements were taken in both ways described above: (1) [Full validation from scratch] vs. ([structural validation from scratch] + [incremental validation of key and keyref constraints]). (2) ([Full validation from scratch] vs. [incremental validation of key and keyref constraints]).

74
- [structural validation from scratch]) vs. [incremental validation of key and keyref constraints].

In order to be able to perform a sequence of addition operations, of structurally equivalent subtrees (with different field values), to the same location in the tree, and without violating structural constraints, we removed the added subtree from the document after every operation, and rolled back the changes made to the data structures.

### 4.1.4 Deletion

In order to check the deletion algorithm, we changed the schema by making all elements optional (i.e., minOccurs=0), so that deletion does not violate structural constraints (which would make checking key and keyref constraints pointless). For each one of the three XML documents, we performed a series of deletion operations. For each deletion, we chose the node to be deleted (along with its subtree) as follows. With a 5% chance, we chose randomly from the field nodes (i.e., nodes that serve as a field of some selector-identified node). Such a deletion is bound to fail, since it causes a field expression to evaluate to an empty set for some selector-identified node. With a 95% chance, we chose randomly from the non-field nodes. In order to be able to perform a sequence of deletion operations without drastically reducing the size of the document, we re-attached the deleted subtree to the document after every operation, and rolled back the changes made to the data structures.

### 4.2 Results and Analysis

#### 4.2.1 Setup

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 (see Figure 4.3). 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).

Now we show, for each type of test, the following measurements:

1. The average time of checking structural constraints from scratch and
identity constraints incrementally, compared with the average time for full validation from scratch.

2. The average incremental validation time \(T_{\text{inc}}\), compared with the average time of validation of key and keyref constraints from scratch (calculated by \(T_{\text{scratch}} - T_{\text{struct}}\)).

### 4.2.2 Single Value Change

Figure 4.4 shows the results using comparison (1) above (the average time of checking structural constraints from scratch and identity constraints incrementally, compared with the average time for full validation from scratch). The incremental validation time is significantly shorter, even if we take into account the larger setup time of XSV+ (an improvement of over 30%). Figure 4.5 shows the results using comparison (2) (the average incremental validation time \(T_{\text{inc}}\), compared with the average time of validation of key and keyref constraints from scratch (calculated by \(T_{\text{scratch}} - T_{\text{struct}}\))). These results allow us to see the dramatic speedup of key and keyref validation, gained by using the incremental algorithm.

### 4.2.3 Several Value Changes

Figure 4.6 shows the results using comparison (1). Figure 4.7 shows the results using comparison (2). Results are shown for different \(\text{TRANSACTION\_SIZE}\) values. The time to validate from scratch is independent of \(\text{TRANSACTION\_SIZE}\).
Figure 4.4: Average time per single value change (seconds), comparison 1.

Figure 4.5: Average time per single value change (seconds), comparison 2.
Figure 4.6: Average time per transactional change (seconds), for a 346KB document, comparison 1.

Incremental validation, on the other hand, becomes slower as \textit{TRANSACTION\_SIZE} increases.

### 4.2.4 Addition

Figure 4.8 shows the results using comparison (1), for the 346KB document. Figure 4.9 shows the results using comparison (2), for the 346KB document. Figure 4.10 shows the results using comparison (1), for the 682KB document. Figure 4.11 shows the results using comparison (2), for the 682KB document. Results for the 1.41MB document are similar.

The results (for all documents) show a speedup of roughly two orders of magnitude in comparison 2, and roughly 30\% in comparison 1. Naturally, validation times, both incremental and from scratch, become longer as the subtree becomes larger.

### 4.2.5 Deletion

Figure 4.12 shows the results using comparison (1). Figure 4.13 shows the results using comparison (2). The results show a 30\% speedup in comparison 1, and a speedup of two orders of magnitude in comparison 2.
Figure 4.7: Average time per transactional change (seconds), for a 346KB document, comparison 2.

Figure 4.8: Average time per addition operation (seconds), comparison 1, 346KB document.
Figure 4.9: Average time per addition operation (seconds), comparison 2, 346KB document.

Figure 4.10: Average time per addition operation (seconds), comparison 1, 682KB document.
Figure 4.11: Average time per addition operation (seconds), comparison 2, 682KB document.

Figure 4.12: Average time per deletion operation (seconds), comparison 1.
Figure 4.13: Average time per deletion operation (seconds), comparison 2.

4.3 Space Consumption

We measured the space (i.e., amount of memory) consumed by the Python application before and after loading the document into memory (and populating the data structures in the process). We measured this for XSV and for XSV+, in order to see how much more memory our data structures require. Before loading the document, the Python process takes up 7MB of memory.

- When loading documents with XSV, the memory consumption is as follows.
  - After loading the 346KB document, the memory consumption is 53MB.
  - 682KB document \(\Rightarrow\) 96MB of memory.
  - 1.41MB document \(\Rightarrow\) 193.5MB of memory.

- When loading documents with XSV+, the memory consumption is as follows.
  - After loading the 346KB document, the memory consumption is 59MB (an 11.3% increase compared to XSV).
  - 682KB document \(\Rightarrow\) 107MB of memory (an 11.46% increase compared to XSV).
  - 1.41MB document \(\Rightarrow\) 216.7MB of memory (an 11.99% increase compared to XSV).
The memory consumption of XSV+ is quite high, but that is the case also for XSV. The added data structures of XSV+ increase the memory consumption by less than 12%.

4.4 Chapter Summary

In this chapter we presented experiments, done with the implementation which was described in Chapter 3. The experiments compared the time it takes to validate various change operations from scratch with the time it takes to validate the operations when key and keyref constraints are checked incrementally. Measurements were taken using two methods of comparison. Test runs were performed on various XML documents. Results showed that incremental validation of key and keyref constraints is much faster than validation of these constraints from scratch. The speedup for single-value changes, addition and deletion is about 30% in comparison 1 and two to three orders of magnitude in comparison 2. For transactional changes, the speedup becomes smaller as the transaction size becomes larger.
Chapter 5

Foreign Key Navigation

5.1 Motivation

In this chapter we consider the problem of navigating, from a selector-identified node of a key $K$, to its referring "children" according to a keyref $KR$, and, conversely, from a selector-identified node of $KR$, to the "parent" it references. Consider the schema depicted (informally) in Figure 5.1. A Sale element describes a sale made by a specific sales person (identified by SalesPersonID) to a specific customer (identified by CustNumber) on a specific date. We assume that at most one sale is made on a specific date to a specific customer by a specific sales person. For every such sale there is an invoice. The status of an invoice may be 'in preparation', 'issued', 'in progress', 'paid' or 'deal complete'. Suppose the schema contains a key definition, within the scope of the Sales element, with a selector expression that selects Sale elements, and with fields (SalesPersonID, CustNum, Date). Suppose further that the schema contains a keyref definition within the same scope, with a selector expression that selects the Invoice elements, and with fields (SalesPersonID, CustNum, Date).

Suppose we wish to find all Sale elements whose corresponding Invoice has a 'paid' status. There is no easy way (and possibly no way at all!) in XPath to navigate from Sale elements to their corresponding Invoice elements and vice versa. We introduce foreign-key-navigation axes to facilitate such navigation. With these axes, we can simply navigate from elements specified in $//Invoice[Status = "paid"]$ directly to the Sale elements that those Invoice elements reference. Suppose the keyref is called $KR$, then we

\footnote{Had XML Schema (4) allowed using the parent axis in the definition of key fields then the Sale.SalesPersonID element would not be needed and the key would be (parent :: SalesPerson/ID, CustNum, Date).}
simply write //Invoice[Status = "paid"]/KR_Parent. Similarly, to navigate from Sale elements to their associated paid invoices, we can write //Sale/KR_Children[Status = "paid"].

One might question the necessity of foreign-key axes. In some cases the need for such navigation may be eliminated by adding 'artificial' id fields (attributes or elements) that uniquely identify the selector-identified nodes of the key, thus essentially creating a single-field key. In the sales example above, we might add a new SaleID element (or attribute) as a child element (or attribute) of the Sale element, and of the Invoice element. However, this solution, even in cases where it is applicable, is very cumbersome. It is also redundant to add such elements for each key and foreign key defined in the schema. Furthermore, the values of these artificial elements or attributes need to be carefully maintained so that they remain unique and correspond to the actual key and keyref fields to which they relate. Maintenance is particularly difficult if XML data files are independently maintained, and especially in a distributed environment. Hence, the solution of adding new axes seems more practical.

Instead of adding foreign-key axes to XPath 1.0, one might use XQuery. In the example of Figure 5.1, we can use the following XQuery expression to retrieve the Sale elements of paid invoices, instead of writing //Invoice[Status = "paid"]/KR_Parent.

```xquery
for $s in document("my_doc.xml")//Sale
for $i in document("my_doc.xml")//
```

Figure 5.1: The need for foreign key navigation.
This query is also a legal XPath 2.0 query (XPath 2.0 includes a limited form of the XQuery FLWOR expressions, containing only For clauses and Return).

There are several advantages to using foreign-key-navigation axes and not XQuery in such scenarios:

- Foreign-key-navigation axes enable information retrieval in a simple navigational language that does not use variables.
- The XQuery expression may become very cumbersome. Accommodating the semantic complexities of foreign key references, described in Section 1.2, is non-trivial. This example is rather simple, but in more complicated examples (i.e., when the key and foreign key are defined in different scoping elements, and the scoping elements may have more than one instance in the document), the XQuery expression becomes more complicated. The expression might get even more complicated because it must identify nodes that are not selector-identified nodes but have the same tags as such nodes, and avoid performing navigation from these nodes.
- The syntax of foreign-key-navigation axes is more intuitive, and makes it clear that we are retrieving information according to a key and foreign key defined in the schema and not simply according to the values of arbitrary fields.
- The use of foreign-key-navigation axes can help a 'smart' query processor to execute the query more efficiently, using the constraints imposed on the document by the key and foreign key definitions.
- Without foreign-key-navigation axes, each XQuery programmer would need to write code to perform such tasks, instead of all programmers using the same, built-in functionality.

5.2 Foreign-key-navigation Axes Defined

Foreign-key-navigation axes are defined as follows. Suppose the schema contains definitions for a key $K$, and a keyref $KR$ that refers to $K$. Two new
axes, \textit{KR\_Children} and \textit{KR\_Parent}, are implicitly defined. Let \( y \) be a selector-identified node of \( KR \), and let \( n \) be the scoping node of \( y \) which is closest to the root (i.e., there is no scoping node of \( y \) which is closer to the root than \( n \)). In the context of \( y \), the \textit{KR\_Parent} axis navigates to the node that \( y \) references, relative to \( n \). In terms of the KeyInfo structures, defined in Section 2.3, this is the node that appears in \( n.KeyInfo[K] \) and has the same key-sequence as \( y \). Let \( x \) be a selector-identified node of \( K \). In the context of \( x \), the \textit{KR\_Children} axis navigates to the nodes \( y_i \) such that (1) \( y_i \) is a selector-identified node of \( KR \); (2) \( y_i \) has the same key-sequence as \( x \); (3) \( x \) appears in \( n_i.KeyInfo[K] \), where \( n_i \) is the scoping node of \( y_i \) which is closest to the root. In other words, \textit{KR\_Children} is the reverse axis of \textit{KR\_Parent}.

Note that since foreign key references are relative to a keyref scoping node, foreign-key-navigation may be ambiguous if a keyref-scoping node is not specified. In the definition above, we define which is the relevant scoping node. When we check whether a node \( y \) references a node \( x \), we do this relative to the top-most (i.e., closest to the root) \( KR \) scoping node of \( y \). Note that \( y \) may have several scoping nodes, all on the path from the root to \( y \). A different option is to check references relative to the bottom-most scoping node of \( y \). We could also check references according to all scoping nodes of \( y \), and then the \textit{KR\_Parent} axis would possibly navigate to more than one node.

Another option is to define axes \textit{KR\_Children(P)} and \textit{KR\_Parent(P)}, where \( P \) is a path expression that, when evaluated in the context of the document root, navigates to a single scoping node of the keyref \( KR \). Then, navigation would be performed relative to the specified scoping node. From here on, we consider only the first option (i.e., we consider references within the top-most scoping node of a keyref selector-identified node). The results we present can be adapted to comply with the other options.

Foreign-key-navigation axes may appear wherever an XPath axis may appear. These axes adhere to the semantics of keyref references, described in Section 1.2.

It is legal to write \( P/KR\_Children \) where \( P \) is an expression that does not necessarily navigate to a selector-identified node of \( K \). When \textit{KR\_Children} is applied to a context set, it navigates from selector-identified nodes of \( K \) that appear in the context set to the corresponding selector-identified nodes of \( KR \). For other nodes, the application of the axis returns an empty set of nodes.

Consider the XML Schema \textit{SchemaCD}, depicted in Figure 5.2. This schema defines a key \( K \) and a keyref \( KR \), both within the scope of the root \( R \). In an XML document that conforms to this schema, the combination of
Figure 5.2: SchemaCD.

f1 and f2 must be unique for a C node, and for each D node there must exist a C node with the same f1 and f2 values. Figure 5.3 shows BaseXML, a specific XML document that conforms to this schema.

Foreign-key-navigation axes navigate between the elements defined by the selector expression of the key definition and the elements defined by the selector expression of the keyref definition. In this example these are /R/A/C and /R/B/D, respectively. For instance, the query //C[Name = "b"]/*[f1 = "3"]/KR_Children selects the D nodes which reference a C node with Name = "b" and f1 = 3. In BaseXML this is the node D1. The query //D[f1 = "3"]/*[f2 = "2"]/KR_Parent/Name/text() returns "a".

Note that if the domain of field values is finite then basic foreign-key-navigation can be performed by traversing all possibilities of field values. For example, suppose we constrain the f1 values to be only 1 or 3 and the f2 values to be 2 or 4. Then instead of writing //C[Name = "b"]/*/KR_Children we can write the following query.
//D[(f1 = "1" and f2 = "2" and //C[Name = "b"]/*[f1 = "1" and f2 = "2"]) or (f1 = "3" and f2 = "4" and //C[Name = "b"]/*[f1 = "3" and f2 = "4"]) or ...].

In other words, we explicitly check, for every possible combination of field values in a D node, whether there exists a //C[Name = "b"] node with the same field values. Note, however, that to fully emulate foreign-key-
Figure 5.3: BaseXML (a conforming document to SchemaCD)

navigation axes, we need to accommodate the complex semantics described in Section 1.2.

5.3 Evaluation of Foreign-key-navigation Axes

We examine the problem of evaluating a query $Q$ on a document $D$ that conforms to an XML Schema $S$. We denote the number of symbols in the string representation of the document $D$ (respectively, the query $Q$ and the schema $S$) by $|D|$ (respectively, $|Q|$ and $|S|$). A simple polynomial-time algorithm for evaluating an XPath query is presented as algorithm 6.3 in [15]. In this section we briefly describe the algorithm and explain how it can be adapted to evaluate queries that may also contain foreign-key-navigation axes. We show that for such queries the algorithm still runs in polynomial time. We assume that the queries are evaluated on documents that are valid with respect to a schema (in which keys and foreign keys are defined), otherwise foreign-key-navigation is meaningless.

The algorithm uses the concept of context-value tables. A context-value
Figure 5.4: Example parse tree

Table for an expression $e$ contains a row for every possible context on which $e$ may be evaluated. A context is a tuple $(x, k, n)$, where $x$ is a context node, $k$ is its position in the (ordered) context set and $n$ is the size of the context set. For every such context, the table contains the result of evaluating $e$ on the context (the result may be a node-set, a number, a string or a boolean value). A query is represented as a parse tree of sub-expressions: the root represents the query and each node has children representing its sub-expressions. For example, Figure 5.4 shows the parse tree for the query $\text{descendant::b/following-sibling::*[position()!}=\text{last}()]$: the root node (that represents the whole query) has a child node that represents the sub-expression $\text{descendant::b}$, and another child node that represents the sub-expression $\text{following-sibling::*[position()!}=\text{last}()]$. The latter has two child nodes, one for the sub-expression $\text{following-sibling::*}$ and another for the sub-expression $\text{position()!}=\text{last}()$. The latter has two child nodes that represent the sub-expressions $\text{position()}$ and $\text{last()}$.

The algorithm traverses the parse tree bottom-up. First, it computes context-value tables for the leaves. Then, it moves up the tree - for every node whose children’s tables have been computed, it computes the node’s table based on those of its children. For example, if we have context value tables for the expressions $E_1$ and $E_2$, the table for $E_3 = E_1/E_2$ is constructed as follows. For each possibility of $(x, k, n)$ to be installed in table $E_3$, we fetch a row $((x, k, n), S)$ from the table of $E_1$, where $S$ is a node-set. For every node $y$ whose position in $S$ is $k^y$, we fetch a row $((y, k^y, |S|), S^y)$ from the table of $E_2$. The value of the row for $(x, k, n)$ in the table for $E_3$ is the union of these $S^y$ node-sets. In order to construct a table for the expression $E_1 \text{ RelOp } E_2$, where $\text{RelOp} \in \{<, >, =, \neq, \leq, \geq\}$, for each possible $(x, k, n)$ we fetch a row $((x, k, n), S1)$ from the table of $E_1$ and a row $((x, k, n), S2)$ from the table of $E_2$. The value of the row for $(x, k, n)$ in the table of $E_3$ is $\text{True}$ if and only if there is a node $s1$ in $S1$ and a node $s2$ in $S2$ such that $s1 \text{ RelOp } s2$. 

90
The algorithm continues until it reaches the root of the parse tree. For a query $Q$ and a document $D$ it runs in time $O(|D|^5 \ast |Q|^2)$.

In order for the algorithm to evaluate queries with foreign-key-navigation axes, we need to provide a method of constructing context-value tables for the new foreign-key axes, i.e., for expressions of the form $KR\_Parent::t$ and $KR\_Children::t$, where $KR$ is a keyref constraint and $t$ is either * or a label. Once we provide such a method, the algorithm works as described above, since the only change is in the computation of tables for the leaves of the query parse tree.

In order to construct the context-value table for $KR\_Parent::t$ (respectively, $KR\_Children::t$), we need to evaluate $KR\_Parent::t$ (respectively, $KR\_Children::t$) for every node of the document. In order to do this efficiently, we first create, in a pre-processing stage, the KeySN, KeyrefSN, KeySelIdent, KeyrefSelIdent, KeyInfo and KeyrefInfo structures, defined in Section 2.3. Creating the structures takes time $O(|S|^4 + |S|^2|D|^2 log|D|)$ in the worst case.

Figure 5.3 shows an example of the data structures created during the pre-processing stage. In this example $a$ and $a'$ are scoping nodes of a foreign key $KR$ and $e_1, e_2, e_1'$ and $e_2'$ are their selector-identified nodes. $c_3, c_1, c_2, c_1'$ and $c_2'$ are the scoping nodes of a key $K$ and $d_3, d_1, ..., d_22'$ are their selector-identified nodes. $c_3$ is a scoping node that has other scoping nodes as children. Some of its selector-identified nodes are also selector-identified nodes of its child scoping nodes. Note that the key selector-identified nodes with key-sequence $(1,2)$ cannot be referenced since this key-sequence appears both in $c_1$ and $c_2$ and also both in $c_1'$ and $c_2'$. This is why this sequence does not appear in $a.KeyInfo[K]$ and $a'.KeyInfo[K]$.

Following the pre-processing stage, we proceed to constructing the context-value tables. In order to do this, we need to evaluate the foreign-key-navigation axes.

For a context node $x$, we evaluate $KR\_Children$ as follows. First we search for $x$ in $KeySelIdent[K]$. If $x$ does not appear there, we return $\emptyset$. Otherwise, let $ks$ be the key-sequence of $x$ (that appears in $x$’s record in $KeySelIdent[K]$). For each ancestor $n$ of $x$, if $n$ is a scoping node of $KR$ (which we check by looking for $n$ in $KeyrefSN[KR]$) and $x$ appears in $n.KeyInfo[K]$ then we search for $ks$ in $n.KeyrefInfo[KR]$. For each record $(y, ks) \in n.KeyrefInfo[KR]$, we fetch a record $(y, ks, SN)$ from $KeyrefSelIdent[KR]$. If $n$ is the scoping node of $y$ which is closest to the root (recall that $SN$ is ordered by distance from the root) then $y$ is added to the result set.

Now we consider the time complexity of evaluating $KR\_Children$ for a context node $x$. Searching for $x$ in $KeySelIdent[K]$ takes time $O(\log|D|)$.
Checking if a node $n$ is a scoping node of $KR$ takes time $O(\log |D|)$. So does looking for $x$ in $n.KeyInfo[K]$. Searching for $ks$ in $n.KeyrefInfo[KR]$ takes time $O(|S|\log |D|)$. Fetching records from $KeyrefSelIdent[KR]$ for the resulting selector-identified nodes takes time $O(|D|\log |D|)$. Since we go over all ancestors of $x$, the time complexity is $O(|D|^3 \log |D| + |S||D|^2\log |D|)$.

The context-value table for $KR_Children$ contains $O(|D|^3)$ rows, but we only need to evaluate the axis for the $O(|D|)$ possible context nodes, which takes time $O(|D|^3 \log |D| + |S||D|^2\log |D|)$ (we keep the results in a search tree, and then for each of the $O(|D|^3)$ rows we fetch the result according to the context node). Therefore, the construction of the context-value table for $KR_Children$ takes time $O(|D|^3 \log |D| + |S||D|^2\log |D|)$.

For a context node $y$, we evaluate $KR_Parent$ as follows. First we search for a node in $KeyrefSelIdent[K]$. If $y$ does not appear there, we return $\emptyset$. Otherwise, let $(s, ks, SN)$ be the found record. Let $n$ be the first node in $SN$ (i.e., the closest to the root). We search for $ks$ in $n.KeyInfo[K]$. If a record $(x, ks, b)$ appears there, we return $\{x\}$. Otherwise, we return $\emptyset$.

Now we consider the time complexity of evaluating $KR_Parent$ for a context node $y$. We search for a node in $KeyrefSelIdent[K]$, and then search for a key-sequence in a $KeyInfo[K]$ structure. Therefore the time complexity is $O(|S|\log |D|)$. The construction of the context-value table (of $O(|D|^3)$ rows)
takes $O(|D|^3 \log |D| + |S||D|\log |D|)$. 

### 5.3.1 Time Complexity

**Theorem 5.1** In the presence of foreign-key-navigation axes, the complexity of evaluating a query $Q$ on a document $D$ that conforms to an XML Schema $S$ is $O(|S|^4 + |S|^2|D|\log |D| + |Q||D|^3\log |D| + |Q||S||D|^2\log |D| + |D|^5|Q|^2)$. For a fixed schema this complexity is $O(|D|^5 * |Q|^2)$.

**Proof** When evaluating a query $Q$, we construct at most $O(|Q|)$ context-value tables for the foreign-key-navigation axes, which takes time $O(|Q||D|^3\log |D| + |Q||S||D|^2\log |D|)$. To this we add the complexity of the pre-processing stage and the complexity of the algorithm for query evaluation presented in [15].

**Improved Complexity**

- A top-down query evaluation algorithm is also presented (briefly) in [15]. It has the same worst case time complexity. Evaluation of foreign-key-navigation axes can be incorporated into it in a similar manner to the bottom-up algorithm.

- In [17], the time complexity of the top-down algorithm is improved to $O(|D|^4|Q|^2)$, by keeping the size of the context-value tables at most $O(|D|^2)$ (instead of $O(|D|^3)$). Since the complexity of computing the context-value tables for the foreign-key-navigation axes was already less than $O(|D|^4|Q|^2)$ (for a fixed schema, it takes $O(|Q||D|^3\log |D| + |Q||D|^2\log |D|)$ time to create these tables, even if they are of size $O(|D|^3)$), the improved complexity result is applicable to queries with foreign-key-navigation axes as well.

- The $O(|D|^4|Q|^2)$ complexity result is also presented in [18]. Therefore, queries with foreign-key-navigation axes can be evaluated in polynomial time, with the same time complexity (for a fixed schema) as in [15], [17] and [18]. That is, the time complexity is $O(|D|^4|Q|^2)$.

### 5.3.2 Space Complexity

The top-down algorithm, MinContext, presented in [18], achieves space complexity of $O(|D|^2|Q|^2)$. Our modifications add the space complexity of the pre-processing data structures. We now consider this space complexity:

- **KeySN**: For each key constraint, KeySN contains an entry for each scoping node. Thus the space complexity is $O(|S||D|)$. 

93
• **KeyrefSN**: Same as **KeySN** - $O(|S||D|)$.

• **KeySelIdent**: Contains an entry for each selector-identified node. Since the entry contains a list of scoping nodes, its size is $O(|D|)$. Thus, the size of this data structure (for all key constraints) is $O(|S||D|^2)$.

• **KeyrefSelIdent**: Same as **KeySelIdent** - $O(|S||D|^2)$.

• **KeyInfo**: Each selector-identified node may have an entry in the **KeyInfo** structure of each of its ancestors. Thus the size of these structures is $O(|S||D|^2)$.

• **KeyrefInfo**: $O(|S||D|^2)$.

Thus queries with foreign-key-navigation axes can be evaluated in space complexity $O(|D|^2(|Q|^2+S))$. For a fixed schema, the complexity is $O(|D|^2|Q|^2)$, which is the same as the space complexity of evaluating standard XPath queries.

### 5.4 Expressive Power

**Definition 5.1** A query is a function that maps XML data trees to node sets. If $t$ is an XML data tree and $Q$ is a query, then $Q(t)$ is a set of nodes extracted from $t$.

**Definition 5.2** The expressible set of an XPath fragment is the set of queries that can be expressed via expressions in that fragment. A fragment $f$ is said to have a larger expressive power than another fragment $g$ if the expressible set of $f$ properly contains the expressible set of $g$.

In Section 5.5, we show that for a large class of schemas in which only single-field keys are allowed, one can, in many cases, rewrite expressions containing foreign-key-navigation axes into equivalent expressions that use ordinary XPath. We conjecture that for multi-field keys, augmenting XPath with foreign-key-navigation axes increases its expressive power. We prove this for $XPath'$, a substantial fragment of XPath.

**Definition 5.3** $XPath'$ is the fragment of XPath defined via the following grammar:

```
locpath ::= '/locpath | locpath'/locpath | locpath']['locpath | locstep.
locstep ::= axis::'ntst'['bexpr'] . . . '|'['bexpr']
```

94
\[ bexpr ::= bexpr \text{ 'and' } bexpr | bexpr \text{ 'or' } bexpr | \text{ 'not'('bexpr') } | \text{ locpath } | \text{ comparison}. \]

\[ \text{comparison ::= compoperand '==' compoperand.} \]

\[ \text{compoperand ::= locpath | value.} \]

\[ \text{value ::= number | string.} \]

\[ \text{axis ::= 'self' | 'child' | 'parent' | 'descendant' | 'descendant-or-self' | 'ancestor' | 'ancestor-or-self'.} \]

In other words, the following restrictions are imposed on full XPath:

- Use of the 'following', 'following-sibling', 'preceding' and 'preceding-sibling' axes is not allowed.
- Expressions contain no functions except for the Boolean function `not()`, and no arithmetic operations are allowed.
- Inequality (`! =`) is not allowed (whereas equality (`=`) is allowed). The `<`, `≤`, `>` and `≥` operators are also not allowed.

The \( \text{XPath}' \) fragment is quite similar to the Core XPath fragment, defined in [16]. However, there are two important differences.

- Core XPath includes all XPath axes, whereas \( \text{XPath}' \) does not include the 'following', 'following-sibling', 'preceding' and 'preceding-sibling' axes.
- Core XPath is only navigational, in the sense that it does not include comparisons, whereas \( \text{XPath}' \) includes comparisons via the '=' operator. This is an important issue when considering the expressive power of an XPath fragment, since comparisons have an existential semantics.

**Definition 5.4** \( \text{XPath}'_fk \) is an extension of \( \text{XPath}' \) that includes foreign-key-navigation axes. It is defined via a very similar grammar. The difference is in the definition of the 'axis' grammar element:

\[ \text{axis ::= 'self' | 'child' | 'parent' | 'descendant' | 'descendant-or-self' | 'ancestor' | 'ancestor-or-self' | 'FK_Children' | 'FK_Parent', for every possible string 'FK'.} \]

Note that the axes `FK_Children` and `FK_Parent` have meaning only for a query that is evaluated on a document that conforms to a schema in which a keyref named 'FK' is defined (otherwise, these axes always return an empty set).

**Theorem 5.2** The expressive power (see definition [5.2]) of \( \text{XPath}'_fk \) is greater than that of \( \text{XPath}' \).
Overview of the proof: We show a query that can be expressed in $XPath'_{fk}$ but not in $XPath'$. Given an XML data tree that conforms to $SchemaCD$ (see Figure 5.2), the query selects the $D$ nodes that reference $C$ nodes whose $Name$ is "a". In $XPath'_{fk}$ this is written as $//C[Name = "a"]/KR_Children$. Intuitively, this cannot be done using an $XPath'$ expression, because in order to obtain the correct $D$ nodes, the expression must check, for a $D$ node $d$, whether there exists a $C$ node $c$ whose $Name$ is "a" and that has both the same $f_1$ value and the same $f_2$ value as $d$. Since there are no variables in XPath 1.0, the expression cannot ensure that the $f_1$ value of $d$ and the $f_2$ value of $d$ are compared to the $f_1$ and $f_2$ values of the same $C$ node.

We prove the result using two claims:

1. We restrict the discussion to expressions that do not use constant values (i.e., the expression can compare values from the document to other values from the document but not to constant values). We show that such an expression cannot perform the required task (i.e., obtain the correct $D$ nodes) on a specific conforming document, $BaseXML$, depicted in Figure 5.3. Intuitively, if an expression can perform the required task for all documents that conform to $SchemaCD$ then it should be able to do so without using constants, since there are infinitely many values that may appear in the conforming documents (and so a fixed set of constants in the expression will not help).

2. Based on our proof for $BaseXML$, we define a set of similar documents and show that there is no expression that works correctly for all documents in the set, even if constants are allowed. We define a set of XML data trees $\{T_0, T_1, \ldots\}$ with the same structure as $BaseXML$ but with different values in each tree. The value of an $f_1$ or $f_2$ node in $T_i$ is $10^i \ast$ (the value of the corresponding node in $BaseXML$). $T_0$ is $BaseXML$. For example, the value of $D_1.f_1$ in $T_4$ is $3 \ast 10^4 = 30000$. For each document in this set, there is no expression that works correctly without using constants. Therefore, if there is an expression that works correctly for some document in the set, it does so by using the values that appear in the document (i.e., these values appear as constants in the expression). Since there is no upper bound on the values that appear in the documents of the set, there is no one expression that works correctly for all these documents. We prove claim (1) above in the following manner. We denote the fragment of $XPath'$ that disallows constants by $XPath'_{NoConst}$. We prove that every predicate in $XPath'_{NoConst}$ gives the same result for
the nodes $D_1$ and $D_2$ of the document $BaseXML$. Therefore, there is no $XPath_{NoConst}$ expression that selects the $D$ nodes of $BaseXML$ that reference $C$ nodes whose $Name$ is "a". If there was such an expression $e$, it would select $D_2$ but not $D_1$. This means $e$ could be used to construct a predicate that distinguishes between $D_1$ and $D_2$, which is a contradiction. Given a query $e$ that, regardless of the context, evaluates to a node set that contains $D_2$ but not $D_1$, a predicate that distinguishes between $D_1$ and $D_2$ can be written as $[self = e/D]$.

Intuitively, a predicate that uses no constants can only use comparisons of $f_1$ and $f_2$ values of $D$ nodes to $f_1$ and $f_2$ values of $C$ nodes in order to distinguish between $D_1$ and $D_2$. However, $BaseXML$ is constructed in a way that "confuses" predicates and prevents them from distinguishing between different $D$ nodes. The following properties hold for $BaseXML$:

- $\text{distinct_values}(//C/f_1) = \text{distinct_values}(//C[Name = "a"]/f_1) = \text{distinct_values}(//C[Name = "b"]/f_1) = \text{distinct_values}(//D/f_1) = \{1, 3\}$

- $\text{distinct_values}(//C/f_2) = \text{distinct_values}(//C[Name = "a"]/f_2) = \text{distinct_values}(//C[Name = "b"]/f_2) = \text{distinct_values}(//D/f_2) = \{2, 4\}$

Let $p$ be a location path, $\text{comp}$ a comparison (of the form $p_1 = p_2$ where $p_1$ and $p_2$ are location paths) and $n$ a node. We denote the set of nodes obtained from evaluating $p$ in the context of $n$ by $p(n)$ and the Boolean result of evaluating $\text{comp}$ in the context of $n$ by $\text{comp}(n)$. In order to prove that an $XPath_{NoConst}$ predicate cannot distinguish between $D_1$ and $D_2$, we prove that (a) For every location path $p$, $p(D_1) = \emptyset$ if and only if $p(D_2) = \emptyset$; and (b) For every comparison $\text{comp}$, $\text{comp}(D_1) = \text{comp}(D_2)$. Therefore, the predicate gives the same result for $D_1$ and for $D_2$. This proves claim (1) and completes the proof.

A detailed proof of Theorem 5.2 is presented in appendix A.

### 5.5 Single-field Navigation

Optimization of standard XPath queries, that do not contain foreign-key-navigation axes, is widespread. Therefore, it is useful to transform queries with foreign-key-navigation axes into equivalent standard XPath queries. In Section 5.4 we showed that foreign-key-navigation axes cannot be expressed in $XPath'$. Now we analyze the case where the key and foreign key have only one field. Intuitively, foreign-key-navigation is "easier" in such cases.
We show that in many cases expressions that use these axes (i.e., expressions in XPath\textsuperscript{fk}) can be rewritten into XPath\textsuperscript{'} expressions (that do not).

Let $K$ be a single-field key, defined in some schema $S$. Let $K\_Sel$ be $K$’s selector expression and let $f\_exp$ be $K$’s (only) field expression. Let $KR$ be a foreign key that refers to $K$. Let $KR\_Sel$ be $KR$’s selector expression and let $g\_exp$ be $KR$’s (only) field expression. For XPath\textsuperscript{fk} expressions that use the foreign-key-navigation axes $KR\_Parent$ and $KR\_Children$, we show equivalent expressions that do not use these axes, under the following assumptions:

1. $K$ and $KR$ are defined in the same scoping element, and a scoping node cannot appear as a descendant of another scoping node (thus, a selector-identified node can only have one scoping node).

2. There is an XPath\textsuperscript{'} query $ScopeExp$ that returns the instances of this scoping element (i.e., the scoping nodes). $ScopeExp$ is independent of context. It starts with $/$.

3. There is an expression $K\_Sel_{\text{exact}}^{-1}$ such that if $n'$ is a selector-identified node of a scoping node $n$ of the key $K$, $K\_Sel_{\text{exact}}^{-1}(n') = \{n\}$. In other words, $K\_Sel_{\text{exact}}^{-1}$ is guaranteed to return only the correct scoping node.

4. There is an expression $KR\_Sel_{\text{exact}}^{-1}$ such that if $n'$ is a selector-identified node of a scoping node $n$ of the keyref $KR$, $KR\_Sel_{\text{exact}}^{-1}(n') = \{n\}$.

**Definition 5.5** The reverse expression of an XPath expression $e$, where $e$ does not start with ‘/’, is an expression $e^{-1}$ such that for every two nodes $n$ and $n'$, $n' \in e(n)$ if and only if $n \in e^{-1}(n')$. The reverse expression of an XPath expression $/e$, where $e$ does not start with ‘/’, is an expression $(/e)^{-1}$ such that for every node $n'$, if $n' \in e(root)$ then $(/e)^{-1}(n') = \{root\}$, otherwise $(/e)^{-1}(n') = \emptyset$, where root is the root node of the document. Note that $(/e)^{-1}$ is equivalent to $e^{-1}[\neg(parent :: *)]$.

Note that $(/e)^{-1}$ can be used to check whether a node is reachable from the root via $e$, since $(/e)^{-1}(n')$ is nonempty if and only if $n' \in e(root)$.

**Lemma 5.1** Let $e$ be the XPath\textsuperscript{'} expression $axis_1 :: ntest_1[bexp_1]/.../axis_{n-1} :: ntest_{n-1}[bexp_{n-1}]/axis_n :: ntest_n[bexp_n]$, and let $e'$ be the XPath\textsuperscript{'} expression $self :: ntest_n[bexp_n]/axis_n^{-1} :: ntest_{n-1}[bexp_{n-1}]/.../axis_2^{-1} :: ntest_1[bexp_1]/axis_1^{-1} :: *$. Then $e'$ is a reverse expression of $e$.

$\overset{2}{axis_i^{-1}}$ is the opposite of $axis_i$. For example, $child^{-1}$ is $parent$. 

98
**Proof** We need to prove that for every two nodes $x, y \in e(x) \iff x \in e'(y)$.

1. Suppose $y \in e(x)$ and prove $x \in e'(y)$: $y \in e(x)$ implies the existence of nodes $v_1, v_2, \ldots, v_{n-1}$ such that $v_1 \in axis_1 \colon ntest_1[bexp_1](x)$, $v_2 \in axis_2 \colon ntest_2[bexp_2](v_1)$, ..., $v_{n-1} \in axis_{n-1} \colon ntest_{n-1}[bexp_{n-1}](v_{n-2})$, $y \in axis_n \colon ntest_n[bexp_n](v_{n-1})$. This implies that:
   - $v_{n-1} \in axis_n^{-1}(y)$, $v_{n-2} \in axis_{n-1}^{-1}(v_{n-1})$, ..., $v_1 \in axis_2^{-1}(v_2)$, $x \in axis_1^{-1}(v_1)$.
   - $y$ satisfies the node test $ntest_n$, and for all $i \in 1..n-1$, $v_i$ satisfies $ntest_i$.
   - $bexp_n(y) = True$, and for all $i \in 1..n-1$, $bexp_i(v_i) = True$. Note that XPath does not include functions, specifically the "context-dependent" functions position() and last(). Therefore, the value $bexp_i(v)$ for a node $v$ does not depend on the location of $v$ within the set of nodes for which the predicate $[exp]$ is evaluated.

Therefore, the following hold:
$$y \in self \colon ntest_n[bexp_n](y),
\quad v_{n-1} \in axis_n^{-1} \colon ntest_{n-1}[bexp_{n-1}](y), \ldots, v_1 \in axis_2^{-1} \colon ntest_1[bexp_1](v_2),
\quad x \in axis_1^{-1}(v_1).$$
Thus, $x \in self \colon ntest_n[bexp_n]/axis_n^{-1} \colon ntest_{n-1}[bexp_{n-1}]/\ldots,axis_2^{-1} \colon ntest_1[bexp_1]/axis_1^{-1} \colon *y$, i.e., $x \in e'(y)$.

2. Suppose $x \in e'(y)$ and prove $y \in e(x)$: $x \in e'(y)$ implies that $self \colon ntest_n[bexp_n](y) = \{y\}$ (otherwise $e'(y) = \emptyset$). Furthermore, it implies the existence of nodes $v_{n-1}, v_{n-2}, \ldots, v_2, v_1$ such that $v_{n-1} \in axis_n^{-1} \colon ntest_{n-1}[bexp_{n-1}](y), \ldots, v_1 \in axis_2^{-1} \colon ntest_1[bexp_1](v_2), x \in axis_1^{-1}(v_1)$. This implies that:
   - $v_1 \in axis_1(x)$, $v_2 \in axis_2(v_1)$, ..., $v_{n-1} \in axis_{n-1}(v_{n-2})$, $y \in axis_n(v_{n-1})$.
   - $y$ satisfies the node test $ntest_n$, and for all $i \in 1..n-1$, $v_i$ satisfies $ntest_i$.
   - $bexp_n(y) = True$, and for all $i \in 1..n-1$, $bexp_i(v_i) = True$.

Therefore, the following hold:
$$v_1 \in axis_1 \colon ntest_1[bexp_1](x),
\quad v_2 \in axis_2 \colon ntest_2[bexp_2](v_1), \ldots, v_{n-1} \in axis_{n-1} \colon ntest_{n-1}[bexp_{n-1}](v_{n-2}),
\quad y \in axis_n \colon ntest_n[bexp_n](v_{n-1}).$$
Thus \( y \in axis_1 \cdot ntest_1[bexp_1]/.../axis_{n-1} \cdot ntest_{n-1}[bexp_{n-1}]/axis_n \cdot ntest_n[bexp_n](x) \), i.e., \( y \in e(x) \).

**Lemma 5.2** Let \( e \) be the expression \( /axis_1 \cdot ntest_1[bexp_1]/.../axis_{n-1} \cdot ntest_{n-1}[bexp_{n-1}]/axis_n \cdot ntest_n[bexp_n] \), and let \( e' \) be the expression \( self \cdot ntest_n[bexp_n]/axis_n^{-1} \cdot ntest_{n-1}[bexp_{n-1}]/.../axis_2^{-1} \cdot ntest_1[bexp_1]/axis_1^{-1} \cdot *[not(parent :: *)] \). Then \( e' \) is a reverse expression to \( e \).

**Proof** Let \( e_1 \) be the expression \( axis_1 \cdot ntest_1[bexp_1]/.../axis_{n-1} \cdot ntest_{n-1}[bexp_{n-1}]/axis_n \cdot ntest_n[bexp_n] \). We need to prove that for every node \( x \), if \( x \in e_1(root) \) then \( e'(x) = \{root\} \), otherwise \( e'(x) = \emptyset \). According to Lemma 5.1, \( e' = e_1^{-1}[not(parent :: *)] \), where \( e_1^{-1} \) is a reverse expression of \( e_1 \).

1. If \( x \in e_1(root) \) then \( root \in e_1^{-1}(x) \). The expression \( not(parent :: *) \) is True for the root node and False for any other node. Thus, \( e'(x) = \{root\} \).

2. If \( x \notin e_1(root) \) then \( root \notin e_1^{-1}(x) \). The expression \( not(parent :: *) \) is True for the root node and False for any other node. Thus, \( e'(x) = \emptyset \).

**Lemma 5.3** Every XPath' expression \( e \) has a reverse expression.

**Proof** Follows from Lemma 5.1 and Lemma 5.2. Note that these lemmas remain correct if we allow multiple predicates in a single location step, i.e., replace \( bexp_i \) with \( [bexp_i]..[bexp_k] \). The same goes for nested predicates. This is because the predicates do not need to be reversed (the same predicates are applied in \( e \) and in \( e^{-1} \), only the axes are reversed), and thus predicate nesting and multiple predicates do not effect the transformation of the expression.

Examples: If \( e \ = \ l_1/l_2[l_3]/\text{descendant-or-self} \ = \ l_4[l_5] \) then \( e^{-1} \ = \ self \cdot l_4[l_5]/\text{ancestor-or-self} \ = \ l_2[l_3]/parent \ = \ l_1/parent :: * \).

If \( e \ = \ \text{descendant} :: \ l_1/\text{descendant} :: l_2 \mid \text{descendant} :: l_3/\text{descendant} :: l_4 \) then \( e^{-1} \ = \ self :: l_2/\text{ancestor} :: l_1/\text{ancestor :: *} \mid self :: l_4/\text{ancestor :: *} \). Here, the \( l_i \)'s are labels.

Note that this formulation would not be correct if XPath' allowed the functions \( last() \) and \( position() \). When a predicate that contains such functions is applied to a node-set, whether or not a node satisfies the predicate depends on the node’s position within the node-set. Therefore, if such a predicate
appears in an expression, we cannot simply use the predicate in the reverse expression, since it would be applied to a different node-set.

**Theorem 5.3** Let $P$ and $Q$ be some XPath' expressions. Then the following hold.

- Let $e_1$ be the expression $/P/KR\_Children/Q$. Then $e_1$ is equivalent to the expression $e_2$, where $e_2 = \text{ScopeExp}/KR\_{Sel}[g\_exp = KR\_{Sel}^{-1}/KR\_{Sel}[(/P)^{-1}]/f\_exp]/Q$.

- Let $e_3$ be the expression $/P/KR\_Parent/Q$. Then $e_3$ is equivalent to the expression $e_4$, where $e_4 = \text{ScopeExp}/K\_Sel[f\_exp = K\_Sel^{-1}/K\_Sel[(/P)^{-1}]/g\_exp]/Q$.

Note that $e_2$ and $e_4$ may contain the union operator $|$ in non-top level positions. However, as explained in Section 5.6, there is an equivalent XPath' expression that uses the union operator only at the top level.

**Proof** We may use $\text{ScopeExp}$ in $e_2$ and $e_4$ due to assumption 2. We may use $K\_Sel^{-1}$ due to assumption 3. We may use $KR\_Sel^{-1}$ due to assumption 4. We explain the equivalence of the expressions.

- Equivalence of $e_1$ and $e_2$. The expression $/P/KR\_Children$ navigates to selector-identified nodes of $KR$, that reference selector-identified nodes of $K$ which are reachable via $/P$. Therefore, we need to navigate to $\text{ScopeExp}/KR\_Sel$ and then use a predicate to choose the correct $KR$ selector-identified nodes. We need to compare $g\_exp$ to the $f\_exp$ of the appropriate $K$ selector-identified nodes. The 'appropriate $K$ selector-identified nodes' are those that appear in the same scoping node as the current $KR$ selector-identified node (this is due to assumption 1), and are reachable via $/P$. To obtain these nodes, we navigate to the scoping node via $KR\_Sel^{-1}$ and then execute the selector expression $K\_Sel$. We use $(/P)^{-1}$ to make sure that we obtain only $K$ selector-identified nodes that are reachable via $/P$, i.e., reachable from the root via $P$. Therefore, the expression $/P/KR\_Children$ is equivalent to the expression $\text{ScopeExp}/KR\_Sel[g\_exp = KR\_Sel^{-1}/K\_Sel[(/P)^{-1}]/f\_exp]$. Adding $/Q$, of course, maintains the equivalence.

- Equivalence of $e_3$ and $e_4$ can be argued in a similar manner.
Input: An XPath’fk expression \( e = /axis_1[bexp_1]/.../axis_n[bexp_n] \), such that 
\( bexp_i \), for all \( 1 \leq i \leq n \), is an XPath’fk expression.

Output: An XPath’ expression \( e' \) which is equivalent to \( e \).

\[
\text{transform}(/axis_k[bexp_k]/.../axis_n[bexp_n]) \{ \\
\text{if for all } 1 \leq i \leq n, \text{axis}_i \text{ is not a foreign-key-navigation axis} \\
\quad \text{}\text{// } e \text{ contains no foreign-key-navigation axes, so we leave it as is.} \\
\text{return } /axis_1[bexp_1]/.../axis_n[bexp_n]; \\
\text{Let } i \text{ be the smallest index such that } \text{axis}_i \text{ is a foreign-key-navigation axis; } \\
\text{if } (i==1) \{ \\
\text{Let } Q \text{ be the expression } axis_2[bexp_2]/.../axis_n[bexp_n]; \\
\text{if } (axis_1 == KR\_Children) \text{ for some keyref } KR \\
\text{return } \text{transform}(ScopeExp/KR\_Sel[bexp_1][g.exp = KR\_Sel^{-1}_{exact}/KR\_Sel(not(parent :: *)][f.exp]/Q); \\
\text{if } (axis_1 == KR\_Parent) \text{ for some keyref } KR \\
\text{return } \text{transform}(ScopeExp/K\_Sel[bexp_1][f.exp = KR\_Sel^{-1}_{exact}/KR\_Sel(not(parent :: *)][g.exp]/Q); \\
\} \text{ else } \{ \text{ if } i > 1; \}
\text{Let } P \text{ be the expression } axis_1[bexp_1]/.../axis_{i-1}[bexp_{i-1}]; \\
\text{Let } Q \text{ be the expression } axis_{i+1}[bexp_{i+1}]/.../axis_n[bexp_n]; \\
\text{if } (axis_i == KR\_Children) \text{ for some keyref } KR \\
\text{return } \text{transform}(ScopeExp/KR\_Sel[bexp_i][g.exp = KR\_Sel^{-1}_{exact}/KR\_Sel([P]^{-1})[f.exp]/Q); \\
\text{if } (axis_i == KR\_Parent) \text{ for some keyref } KR \\
\text{return } \text{transform}(ScopeExp/K\_Sel[bexp_i][f.exp = KR\_Sel^{-1}_{exact}/KR\_Sel([P]^{-1})[g.exp]/Q); \\
\}
\}
\]

Figure 5.6: Transformation Algorithm.

The predicate \([not(parent :: *)]\) (instead of \([(/P)^{-1}]\)) allows only the root node. Similarly, the expression \(/KR\_Parent/Q\) is equivalent to the expression \(ScopeExp/K\_Sel[f.exp = K\_Sel^{-1}_{exact}/KR\_Sel[not(parent :: *)][g.exp]/Q\).

The algorithm depicted in Figure 5.6 transforms an XPath’fk expression with possibly several foreign-key-navigation axes (all single-field, not within predicates, and under the assumptions specified above) into an equivalent XPath’ expression. Its correctness follows from Theorem 5.3. Note that for brevity of notation, we handle expressions of the form \( e = /axis_1[bexp_1]/.../axis_n[bexp_n] \). However, \( bexp_i \) can be replaced by a series of predicates \( bexp_{i_1}[bexp_{i_2}][bexp_{i_3}]...[bexp_{i_k}] \) without changing the algorithm.
5.6 Related Operators

5.6.1 A Generalized Navigation Operator

Foreign-key-navigation axes enable 'static' navigation according to key and foreign key constraints as defined in an XML schema. They enable 'jumping' from a node which is one of the selector-identified nodes of a key to selector-identified nodes of a keyref and vice versa. A natural generalization of this kind of navigation is a 'dynamic navigation' operator:

\[ \text{dyn\_nav}([f_1, f_2, \ldots, f_k], \text{SelectorPath}, [g_1, g_2, \ldots, g_k]), \]

where \( \text{SelectorPath} \), \( f_i \) and \( g_i \) (\( 1 \leq i \leq k \)) are XPath expressions. Suppose the operator is executed in some context set \( S_1 \). Each \( f_i \), \( 1 \leq i \leq k \), is executed with respect to \( S_1 \), and must navigate, for each element of \( S_1 \), to a single element with a simple content, or to an attribute. \( \text{SelectorPath} \) is evaluated relative to the root, resulting in some context set \( S_2 \). Each \( g_i \), \( 1 \leq i \leq k \), is evaluated with respect to \( S_2 \) and must navigate to a single element with a simple content, or to an attribute, when executed on an element of \( S_2 \). The operator returns the elements of \( S_2 \) for which there is an element of \( S_1 \) such that for all \( i = 1..k \), the value of the element selected by \( g_i \) is equal to the value of the element selected by \( f_i \).

At first glance, the dynamic navigation operator seems closely related to foreign-key-navigation axes. It seems that one can use this operator to simulate the foreign-key-navigation axes. Suppose we have an XML schema with definitions for a key \( K \) and a keyref \( KR \). The instances of \( K \)'s scoping element are reachable via the XPath expression \( K\text{Scope} \) and the instances of \( KR \)'s scoping element are reachable via the XPath expression \( KR\text{Scope} \). \( K \)'s selector expression is \( K\text{Selector} \) and its field expressions are \( f_1, \ldots, f_k \). \( KR \)'s selector expression is \( KR\text{Selector} \) and its field expressions are \( g_1, \ldots, g_k \). Suppose there is only one scoping node (an instance of the scoping element) for \( K \) and one scoping node for \( KR \). Then, when executed in the context of \( K \)'s selector-identified nodes (i.e., \( K\text{Scope}/K\text{Selector} \)), the expression \( \text{dyn\_nav}([f_1, f_k], KR\text{Scope}/KR\text{Selector}, [g_1, g_k]) \) is equivalent to \( KR\text{Children} \). When executed in the context of \( KR \)'s selector-identified nodes (i.e., \( KR\text{Scope}/KR\text{Selector} \)), the expression \( \text{dyn\_nav}([g_1, g_k], K\text{Scope}/K\text{Selector}, [f_1, f_k]) \) is equivalent to \( KR\text{Parent} \).

However, in most circumstances this equivalence does not hold, since the dynamic navigation operator is not aware of the semantic complexities of keyref references (described in Section 1.2). Also, when the \( KR\text{Children} \) axis is applied to a node which is not a \( K \) selector-identified node, or the \( KR\text{Parent} \) axis is applied to a node which is not a \( KR \) selector-identified node, an empty set must be returned. A dynamic navigation operator, that
has no knowledge of the key and foreign key definitions, does not check this.

The query that, given an XML document that conforms to SchemaCD, returns the $D$ nodes that reference a $C$ node whose $Name$ is "a" (see Section 5.4), can be written as $//C[Name = "a"]/$dyn-nav(\([f_1, f_2]\), $D$, \([f_1, f_2]\))

Since this query cannot be expressed in $XPath'$ (as explained in Section 5.4), the dynamic-navigation operator adds expressive power to $XPath'$, although it cannot capture the semantics of foreign-key-navigation. Also, this operator may be very useful, as it allows navigation according to relations that are not defined at all by the schema’s foreign-key constraints.

5.6.2 The Union Operator

We suggest incorporating the union operator inline as a sub-expression of other expressions. For example: $//a/b/(e_1 | e_2)/c/d$, where $a$, $b$, $c$ and $d$ are some QNames (see [4]) and $e_1$ and $e_2$ are XPath expressions. If $S$ is a set of nodes, $S_1$ is the context set after executing $e_1$ from context set $S$ and $S_2$ is the context set after executing $e_2$ from context set $S$, then the context set after executing $e_1 | e_2$ from context set $S$ is $S_1 \cup S_2$. It is easy to see that in $XPath'$ this can be rewritten as $//a/b/e_1/c/d | //a/b/e_2/c/d$. Clearly, this variation does not add expressive power, though it makes for more succinct expressions.

5.6.3 An Intersection Operator

We define an intersection operator $\cap$. Suppose that $e_1$ and $e_2$ are $XPath'$ expressions, $S$ is a set of nodes, $S_1$ is the context set after executing $e_1$ from context set $S$, and $S_2$ is the context set after executing $e_2$ from context set $S$. Then, the context set after executing $e_1 \cap e_2$, from context set $S$, is $S_1 \cap S_2$. The following lemma shows that adding an intersection operator to $XPath'$ does not increase its expressive power. Note that [22] proves a similar claim (closure under intersection), but it is done for a different XPath fragment, which is not equivalent to $XPath'$ since it does not include comparisons (string values of nodes are disregarded in [22]).

Lemma 5.4 The intersection of two $XPath'$ (see def. 5.3) expressions is equivalent to the union of some $XPath'$ expressions that do not use intersection.

---

3The equivalence holds because $XPath'$ does not include functions such as $position()$, and therefore it does not matter if we navigate to /c once after applying $e_1$ and once after applying $e_2$ and then merge the results, or if we navigate to /c after merging the results of $e_1$ and of $e_2$. 

104
**Proof** Given two XPath' expressions \( e_1 \) and \( e_2 \), we present an algorithm for creating an expression which is equivalent to their intersection and is a union of XPath' expressions. Note that if an intersection appears inside a predicate then this translation will create an expression with union inside a predicate, which is not allowed in standard XPath. However, such an expression can be rewritten so as to include union only at the top level.

The expressions \( e_1 \) and \( e_2 \) may be simplified as follows.

1. According to [25], every XPath expression can be rewritten so as not to include reverse axes (preceding, parent, ancestor).

2. An expression containing 'self' can be rewritten so as not to include it: \( /P/self :: x[pred] \), where \( P \) is some expression can be rewritten as \( /P' :: z[pred] \), where \( P' \) is the same as \( P \) but without the closing node test \( y \) (if \( P \) contained one) and \( z \) is the unification of \( x \) and \( y \), i.e., if \( x \) is \( * \) then \( z \) is \( y \) (which can still be \( * \)), if \( y \) is \( * \) then \( z \) is \( x \), otherwise \( z \) may be either one of \( x \) and \( y \) (and unless they are the same, the whole expression will return an empty set).

3. \( /P/descendant-or-self :: x \) can be written as \( /P/descendant :: x | /P/self :: x \), and once an expression \( e_1 \) is rewritten as \( e_{11} | e_{22} \), its intersection with an expression \( e_2 \) is simply \( (e_{11} \cap e_2) | (e_{12} \cap e_2) \).

Therefore, without loss of generality, the only axes included in \( e_1 \) and \( e_2 \) are 'child' and 'descendant'.

\( e_1 \) and \( e_2 \) each describe a set of paths. The intersection expression must describe the common paths of these sets. This will be done by traversing both expressions and applying the constraints they impose. We’ll denote the number of location steps in \( e_1 \) (respectively, \( e_2 \)) by \( e_1.length \) (respectively, \( e_2.length \)) and the location steps that \( e_1 \) (respectively, \( e_2 \)) is comprised of by \( e_1.step[i], i = 1..e_1.length \) (respectively, \( e_2.step[j], j = 1..e_2.length \)), where each step is comprised of an axis (child or descendant), a node-test and an optional predicate. At each stage of the algorithm, \( position_1 \) and \( position_2 \) will hold the positions we reached so far in \( e_1 \) and \( e_2 \). If \( position_1 = i \) then this means we are before \( e_1.step[i] \) and if \( position_2 = j \) then this means we are before \( e_2.step[j] \).

The intersection algorithm is recursive. At each stage it tries all possibilities of combining the current steps of \( e_1 \) and \( e_2 \). It finds a set of possible expressions and returns an expression which is the union of these expressions.

---

\(^4\)For simplicity we assume only one predicate. The extension to more predicates is straightforward.
The recursive function receives as its arguments the two expressions, the current positions in them and the current intersection expression. It adds a step to the current intersection expression based on the current steps of the two expressions. We denote the addition of this step by \( \text{addStep}(\text{expression}, \text{step}) \). This function returns the new expression created after adding the location step. An empty expression is denoted by \( \text{EMPTY} \).

We denote the unification of two location steps by \( \text{unify}(\text{step}1, \text{step}2) \). The unification is a location step that satisfies the constraints imposed by both steps. Suppose that \( \text{step}1 = \text{axis}1 :: \text{nodeTest}1[\text{pred}1] \) (if there is no predicate then \( \text{pred}1 = \text{True} \)) and \( \text{step}2 = \text{axis}2 :: \text{nodeTest}2[\text{pred}2] \) (if there is no predicate then \( \text{pred}2 = \text{True} \)). We’ll denote the unified step by \( \text{axis} :: \text{nodeTest}[\text{pred}] \). If \( \text{axis}1 \) or \( \text{axis}2 \) are \( \text{child} \) then so is \( \text{axis} \). Otherwise (both are \( \text{descendant} \)) \( \text{axis} \) is \( \text{descendant} \). If \( \text{nodeTest}1 \) is \( * \) then \( \text{nodeTest} = \text{nodeTest}2 \). If \( \text{nodeTest}2 \) is \( * \) then \( \text{nodeTest} = \text{nodeTest}1 \). If neither \( \text{nodeTest}1 \) nor \( \text{nodeTest}2 \) is \( * \) then they must be the same, and we get \( \text{nodeTest} = \text{nodeTest}1 = \text{nodeTest}2 \). Otherwise there is no unified expression. \( \text{pred} = \text{pred}1 \) and \( \text{pred}2 \), since we must enforce the constraints of both steps. The algorithm is shown in Figure 5.7.

Some simple examples:

- \( e1 = /\text{child} :: x/\text{child} :: a, e2 = \text{child} :: */\text{descendant} :: b : \) We first unify \( \text{child} :: x \) and \( \text{child} :: * \) into \( \text{child} :: x \) and add it to the current intersection expression. Then we are at the last steps of both \( e1 \) and \( e2 \), which means we are at the last stage of the algorithm, so we try to unify \( \text{child} :: a \) and \( \text{descendant} :: b \). We can’t, and therefore there is no intersection.

- \( e1 = \text{descendant} :: a/\text{descendant} :: c, e2 = \text{descendant} :: b/\text{descendant} :: c : \) We get \( \text{descendant} :: a/\text{descendant} :: b/\text{descendant} :: c | \text{descendant} :: b/\text{descendant} :: a/\text{descendant} :: c. \)

## 5.7 Chapter Summary

In this chapter we considered enriching XPath with XML-Schema oriented axes. Specifically, we considered axes that correspond to key and keyref constraints. We defined foreign-key-navigation axes and showed that for \( \text{XPath}' \), a substantial fragment of XPath, foreign-key-navigation axes add expressive power. We showed how queries using foreign-key-navigation axes can be evaluated efficiently. In addition, we showed that foreign-key navigation with single-field keys is simpler, and in many common cases equivalent \( \text{XPath}' \)
expressions (with no foreign-key navigation axes) can be written. In Section 5.6.1 we introduced a generalization of foreign-key navigation axes and showed that the generalized operator also adds expressive power to $\text{XPath}'$, but fails to capture the full semantics of foreign-key navigation axes. We then examined adding union and intersection operators to the $\text{XPath}'$ fragment and showed that this does not increase the expressive power.
findIntersection(e1, e2) {
A := recursiveFunc(e1, e2, 1, 1, EMPTY)
} returns A

recursiveFunc(e1, e2, position1, position2, currentIntersection) {
if (position1 = e1.length + 1) or (position2 = e2.length + 1) then
return ∅
if (position1 = e1.length) and (position2 = e2.length) then
  return [addStep(currentIntersection, unify(e1.step[position1], e2.step[position2]))]
if the axis of e1.step[position1] and the axis of e2.step[position2] are both child then
  if there is a unification to e1.step[position1] and e2.step[position2] then
    return recursiveFunc(e1, e2, position1 + 1, position2 + 1, addStep(currentIntersection, unify(e1.step[position1], e2.step[position2])))
  else return ∅.
else return S1 ∪ S2 ∪ S3, where S1, S2 and S3 are defined as follows:
//S1 holds the expressions we get if we proceed by unifying the current steps
//of e1 and e2.
If there is a unification to e1.step[position1] and e2.step[position2] then
  S1 := recursiveFunc(e1, e2, position1 + 1, position2 + 1, addStep(currentIntersection, unify(e1.step[position1], e2.step[position2])))
else S1 := ∅.
//S2 holds the expressions we get if we proceed by adding e1.step[position1]
//and deferring e2.step[position2].
If the axis of e2.step[position2] is descendant then
  S2 := recursiveFunc(e1, e2, position1 + 1, position2, addStep(currentIntersection, e1.step[position1]))
else S2 := ∅.
//S3 holds the expressions we get if we proceed by adding e2.step[position2]
//and deferring e1.step[position1].
If the axis of e1.step[position1] is descendant then
  S3 := recursiveFunc(e1, e2, position1, position2 + 1, addStep(currentIntersection, e2.step[position2]))
else S3 := ∅.
}

Figure 5.7: The intersection algorithm.
Chapter 6

Conclusion

In this thesis, we dealt with various aspects of XML Schema identity constraints. First, we explained what these constraints are, and showed how they can be defined in XML Schema. We also showed an example of the usefulness of such constraints. Then, we introduced some terminology: scoping nodes, selector-identified nodes, fields and key-sequences. We continued to explain the non-trivial semantics of the constraints, specifically, the complex semantics of keyref references (that is, when a node is considered to be referencing another node).

Next, we dealt with incremental validation of identity constraints. We defined operations that change the values of document nodes, add nodes or delete nodes. We defined data structures that, given a document and a schema, capture the state of the document with respect to the key and keyref constraints defined in the schema. First, we explained how identity constraints can be validated from scratch. Then, for each operation, we showed an algorithm that incrementally validates the operation. That is, the algorithm updates the data structures affected by the operation, and checks if performing the operation violates the constraints. If the constraints are violated, this is detected and the data structures remain unchanged. Otherwise, the data structures are updated to reflect the changes made to the document. Validation is done without re-validating the whole document, but rather traversing only small parts of the document, which are affected by the change.

We presented an implementation of the algorithms, based on an existing open-source validator. We explained how validation is done in XSV, and reviewed the changes made to XSV in order to support incremental validation of key and keyref constraints. We experimented with the implementation, showing the significant speedup gained by validating identity constraints incrementally.
Next, we presented foreign-key-navigation axes. These are new XPath-like axes, that enable navigation from selector-identified nodes of a key to their referencing (keyref selector-identified) nodes, and vice versa. We showed how queries that use these new axes can be efficiently evaluated. We also showed how, in some cases, queries with single-field foreign-key-navigation axes can be transformed into standard XPath queries.
Bibliography


[38] XSV (XML Schema Validator).
http://www.ltg.ed.ac.uk/ ht/xsv-status.html

113
Appendix A

A Detailed Proof of Theorem 5.2

Definition A.1 \( \text{XPath'}_{\text{NoConst}} \) is a fragment of \( \text{XPath} \) which is the same as \( \text{XPath'} \), except that its expressions contain no constants except the constants "a" and "b".

In the following definitions and lemmas, up to and including Lemma A.9, all executions of expressions are with respect to the XML data tree \( \text{BaseXML} \) (Figure 5.3) and all expressions are in \( \text{XPath'}_{\text{NoConst}} \). In the following lemmas we prove that an \( \text{XPath'}_{\text{NoConst}} \) expression cannot "distinguish" between the tree nodes \( D_1 \) and \( D_2 \).

Definition A.2 A set \( S \) is D-symmetric if the following conditions hold:

1. If \( S \) contains some \( D \) node then it contains all \( D \) nodes.
2. If \( S \) contains some \( D.f1 \) node then it contains all \( D.f1 \) nodes.
3. If \( S \) contains some \( D.f2 \) node then it contains all \( D.f2 \) nodes.
4. If \( S \) contains some \( D.f1.text() \) node then it contains all \( D.f1.text() \) nodes.
5. If \( S \) contains some \( D.f2.text() \) node then it contains all \( D.f2.text() \) nodes.

Examples:

- \( \{C1\} \) is D-symmetric because it does not contain any \( D \) related nodes.
• \{C1, D1.f1, D2.f1, D3.f1, D4.f1\} is D-symmetric because it contains all \(D.f1\) nodes.

• \{D1\} is not D-symmetric because it contains \(D1\) but not \(D2, D3\) and \(D4\).

**Lemma A.1** If \(P\) and \(Q\) are D-symmetric sets then \(P \cup Q\) is D-symmetric.

**Proof** Suppose that \(R = P \cup Q\). \(R\) contains some \(D\) node \(\implies\) \(P\) or \(Q\) contain some \(D\) node \(\implies\) \(P\) or \(Q\) contain all \(D\) nodes \(\implies\) \(R\) contains all \(D\) nodes. Similarly for \(D.f1, D.f2, D.f1.text()\) and \(D.f2.text()\).

Next, we define the concept of C-symmetry. Intuitively, a set of nodes is C-symmetric if it maintains symmetry between the C nodes of the same Name value, i.e., for every Name value (i.e., \(a\) and \(b\)), if it contains a component of one C node with that Name value (i.e., the node itself or one of its descendants) then it must contain the corresponding components of all the other C nodes with the same Name value.

**Definition A.3** A set \(S\) is called C-symmetric if for all \(n \in \{a, b\}\), the following conditions are satisfied:

1. If \(S\) contains a C node with Name = \(n\) then \(S\) contains all C nodes with Name = \(n\).

2. If \(S\) contains a C.f1 node of a C node with Name = \(n\) then \(S\) contains all C.f1 nodes of C nodes with Name = \(n\).

3. If \(S\) contains a C.f2 node of a C node with Name = \(n\) then \(S\) contains all C.f2 nodes of C nodes with Name = \(n\).

4. If \(S\) contains a C.Name node of value \(n\) then \(S\) contains all C.Name nodes of value \(n\).

5. If \(S\) contains a C.f1.text() node of a C node with Name = \(n\) then \(S\) contains all C.f1.text() nodes of C nodes with Name = \(n\).

6. If \(S\) contains a C.f2.text() node of a C node with Name = \(n\) then \(S\) contains all C.f2.text() nodes of C nodes with Name = \(n\).

7. If \(S\) contains a C.Name.text() node of value \(n\) then \(S\) contains all C.Name.text() nodes of value \(n\).

**Examples:**

• \{D1\} is C-symmetric because it does not contain any C related nodes.
• \{D1, C1.f1, C4.f1\} is C-symmetric because it contains both C1.f1 and C4.f1.

• \{C3\} is not C-symmetric because it contains C3 but not C2 (that has the same Name as C3).

**Lemma A.2** If \(P\) and \(Q\) are C-symmetric sets then \(P \cup Q\) is C-symmetric.

Next, we define the concept of D-identicalness. Intuitively, two sets of nodes \(S1\) and \(S2\) are D-identical if their intersection is D-symmetric, and there are two \(D\) nodes \(D_i\) and \(D_j\) such that the non-identical parts of the sets consist of \(D_i\)-related nodes (\(D_i\)‘s child nodes (\(f1\) and \(f2\)) or its grandchild text nodes) in \(S1\) and corresponding \(D_j\)-related nodes in \(S2\). In other words, the sets may differ only with respect to \(D_i\) and \(D_j\). A special case of this is when \(S1\) and \(S2\) are identical, so that their intersection equals both sets, and therefore the sets are required to be D-symmetric.

Define \(W_{ij} = \{(P, Q) | P\) consists of some of the \(D_i\)-related nodes \(D_i.f1.text()\), \(D_i.f2.D_i.f2.text()\) and \(Q\) consists of the \(D_j\)-related nodes that correspond to the nodes in \(P\)\}. For example, if \(P\) contains \(D_i.f1.text()\) then \(Q\) contains \(D_j.f1.text()\). Some of the members of \(W_{ij}\) are \(\{(D_i, D_j)\}\); \(\{(D_i, D_i.f1), (D_j, D_j.f1)\}\); \(\{(D_i, D_i.f2), (D_j, D_j.f2)\}\); \(\{(D_i, D_i.f1.D_i.f2), (D_j, D_j.f1.D_j.f2)\}\) etc.

**Definition A.4** Two sets of nodes \(S1\) and \(S2\) are called D-identical if one of the following holds:

1. \(S1\) and \(S2\) are identical and \(S1\) is D-symmetric.

2. There are two distinct \(D\) nodes \(D_i\) and \(D_j\) and two sets of nodes \(T1\) and \(T2\) such that \(S1\) contains \(T1\), \(S2\) contains \(T2\), \(S1\setminus T1\) is identical to \(S2\setminus T2\) and \(S1\setminus T1\) is D-symmetric. \(T1\) and \(T2\) may contain only nodes related to \(D_i\) and \(D_j\), respectively, i.e., \((T1, T2) \in W_{ij}\). \((D_i, D_j)\) are called the diff-base of \((S1, S2)\).

Examples:

• \(\{D1\}, \{D2\}\) are D-identical, with diff-base=(\(D1,D2\)).

• \(\{D1, D2, D3, D4, D2.f1, D2.f2.text()\}\), \(\{D1, D2, D3, D4, D3.f1, D3.f2.text()\}\) are D-identical, with diff-base=(\(D2,D3\)).

• \(\{D1, D1.f1\}, \{D1, D2.f1\}\) are not D-identical because \(\{D1\}\) is not D-symmetric.
• \{D1\}, \{D1\} are not D-identical because they are not D-symmetric.

The \(D_i\) and \(D_j\) nodes in this definition are some two \(D\) nodes that appear in BaseXML (see Figure 5.3).

**Lemma A.3** If \(P1, Q1\) are D-identical, \(P2, Q2\) are D-identical and \(\text{diff-base}(P1, Q1) = \text{diff-base}(P2, Q2)\), then \(P1 \cup P2, Q1 \cup Q2\) are D-identical and have the same diff-base.

Next, we define the concept of C-identicalness which is similar to D-identicalness, but here the nodes with respect to which the two sets may differ are two arbitrary \(C\) nodes, \(C_i\) and \(C_j\), that have the same \(\text{Name}\) value. The intersection of the sets must be \(C\)-symmetric.

Define \(W_{ij} = \{(P, Q) \mid P\) consists of some of the \(C_i\)-related nodes \(C_i, C_1, f1, C_i.f1.text(), C_i.f2, C_i.f2.text(), C_i.Name, C_i.Name.text()\) and \(Q\) consists of the \(C_j\)-related nodes that correspond to the nodes in \(P\}\) For example, if \(P\) contains \(C_i.f1.text()\) then \(Q\) contains \(C_j.f1.text()\). Some of the members of \(W_{ij}\) are \([(C_i, \{C_j\}); (\{C_i, f1\}, \{C_j, f1\}); (\{C_i, f2\}, \{C_j, f2\}); (\{C_i.f1, C_i.f2\}, \{C_j.f1, C_j.f2\})\).

**Definition A.5** Two sets of nodes \(S1\) and \(S2\) are called C-identical if one of the following holds.

1. \(S1\) and \(S2\) are identical and \(S1\) is \(C\)-symmetric.

2. There are two distinct \(C\) nodes \(C_i\) and \(C_j\) that have the same \(\text{Name}\) value and two sets of nodes \(T1\) and \(T2\) such that \(S1\) contains \(T1\), \(S2\) contains \(T2\), \(S1\backslash T1\) is identical to \(S2\backslash T2\) and \(S1\backslash T1\) is \(C\)-symmetric. \(T1\) and \(T2\) may contain only nodes related to \(C_i\) and \(C_j\), respectively, i.e., \((T1, T2) \in W_{ij}\).

\((C_i, C_j)\) are called the diff-base of \((S1, S2)\).

Examples:

• \(\{C3, C3.\text{Name}\}, \{C2, C2.\text{Name}\}\) are C-identical, with \(\text{diff-base} = (C3, C2)\).

• \(\{C1, C2.\text{f1}\}, \{C1, C3.\text{f1}\}\) are not C-identical, because \(\{C1\}\) is not \(C\)-symmetric (since it contains only one of the two \(C\) nodes whose \(\text{Name}\) is "b").

---

1If \(P\) is D-identical to \(Q\) with a diff-base of \((D_i, D_j)\) then \(Q\) is D-identical to \(P\) with a diff-base of \((D_j, D_i)\).

2In this definition, the nodes \(C_i\) and \(C_j\) are some two \(C\) nodes that appear in the XML data tree BaseXML. Note that if \(P\) is C-identical to \(Q\) with a diff-base of \((C_i, C_j)\) then \(Q\) is C-identical to \(P\) with a diff-base of \((C_j, C_i)\)
Lemma A.4 If $P_1, Q_1$ are $C$-identical, $P_2, Q_2$ are $C$-identical and $\text{diff-base}(P_1, Q_1) = \text{diff-base}(P_2, Q_2)$, then $P_1 \cup P_2, Q_1 \cup Q_2$ are $C$-identical and have the same diff-base.

Definition A.6 We use the following notation for XPath expressions. A navigation step is of the form $\text{Axis::NodeTest}$, i.e., it is similar to a location-step but without predicates. A navigational-expression is a sequence of navigation steps and predicates. A comparison is an expression of the form $e_1 = e_2$ where $e_1$ and $e_2$ are navigational-expressions or constants. A predicate is written as $\lbrack \text{predExpr} \rbrack$, where predExpr is an expression (a navigational-expression, a comparison or a combination of such expressions joined by logical connectors) that is evaluated to true or false.

The following sequence of definitions is useful in defining preservation properties of expressions.

Definition A.7 Let $\text{exp}$ be a navigational-expression. Let $S_1'$ and $S_2'$ be the context sets after executing $\text{exp}$ from initial context sets $S_1$ and $S_2$, respectively. We say that $\text{exp}$ preserves $D$-identicalness if for every such sets $S_1, S_2$, if $S_1$ and $S_2$ are $D$-identical then $S_1'$ and $S_2'$ are $D$-identical, and also if $S_1'$ and $S_2'$ are not identical (which of course can only happen if $S_1$ and $S_2$ are not identical) then $S_1'$ and $S_2'$ have the same diff-base as $S_1$ and $S_2$.

Definition A.8 Let $\text{exp}$ be a navigational-expression. Let $S'$ be the context set after executing $\text{exp}$ from initial context set $S$. We say that $\text{exp}$ preserves $D$-symmetry if for every such set $S$, if $S$ is $D$-symmetric then $D$-symmetry is preserved after every step of the execution (i.e., after every navigational step or predicate the context set is $D$-symmetric). In particular, $S'$ is $D$-symmetric.

Definition A.9 Let $\text{exp}$ be a navigational-expression. Let $S_1'$ and $S_2'$ be the context sets after executing $\text{exp}$ from initial context sets $S_1$ and $S_2$, respectively. We say that $\text{exp}$ preserves $C$-identicalness if for every such sets $S_1, S_2$, if $S_1$ and $S_2$ are $C$-identical then $S_1'$ and $S_2'$ are $C$-identical, and also if $S_1'$ and $S_2'$ are not identical (which of course can only happen if $S_1$ and $S_2$ are not identical) then $S_1'$ and $S_2'$ have the same diff-base as $S_1$ and $S_2$.

\[\text{Our navigational expressions do not include the union operator |. This is because unless it is used at the top level of an expression, i.e., to return nodes, its effect is the same as that of the OR operator (which we do consider). Our concern here, as will become clear later, is with predicates (inside which the navigational expressions appear), and therefore we do not need the | operator.}\]
Definition A.10 Let \( \text{exp} \) be a navigational-expression. Let \( S' \) be the context set after executing \( \text{exp} \) from initial context set \( S \). We say that \( \text{exp} \) preserves C-symmetry if for every such set \( S \), if \( S \) is C-symmetric then C-symmetry is preserved after every step of the execution (i.e., after every navigational step or predicate the context set is C-symmetric). In particular, \( S' \) is C-symmetric.

Lemma A.5 If \( \text{exp} \) is an XPath'\_NoConst navigational-expression with no predicates then the following claims hold: (1) \( \text{exp} \) preserves C-symmetry, (2) \( \text{exp} \) preserves D-symmetry, (3) \( \text{exp} \) preserves C-identicalness, (4) \( \text{exp} \) preserves D-identicalness.

Proof 1. (C-symmetry) By induction on the number of navigation steps \( n \). \( S \) is the context set before executing the expression and \( S' \) is the context set after it. \( S \) is C-symmetric and we need to prove that so is \( S' \).

Induction base: \( n = 0 \implies S' = S \) and therefore \( S' \) is C-symmetric.

Induction step: After \( k \) steps we have a context set \( S'' \) which is C-symmetric and we need to prove that after an additional step we get \( S' \) which is C-symmetric.

- Section 1 of the C-symmetry definition: We need to show that if \( S' \) contains some \( C \) node then it also contains all \( C \) nodes with the same \( \text{Name} \).

If \( S' \) contains some \( C \) node then there are three possibilities.

(a) The \( C \) node was already in \( S'' \) and it stayed there by using \( \text{self} \). This means all the other \( C \) nodes in \( S'' \) also appear in \( S' \).

(b) We navigated to the \( C \) node from a \( C.f1, C.f1.text() \), \( C.f2, C.f2.text() \), \( C.Name \) or \( C.Name.text() \) node of \( S' \). Since \( S' \) is C-symmetric it contains the \( f1 \) nodes of all \( C \) nodes with the same \( \text{Name} \) value, and therefore all those \( C \) nodes will appear in \( S' \). Similarly for the \( f2 \) and \( \text{Name} \) nodes and for the \( \text{text()} \) nodes.

(c) We navigated to the \( C \) node from one of its ancestors. Since all \( C \) nodes have the same parent, this means \( S' \) contains all \( C \) nodes.

- Section 2 of the C-symmetry definition: We need to show that if \( S' \) contains some \( C.f1 \) node then it contains the \( f1 \) nodes of all \( C \) nodes with the same \( \text{Name} \). If \( S' \) contains some \( C.f1 \) node then there are four possibilities.
(a) It was already in $S''$ and stayed using \textit{self}. This means all the other \textit{C.f1} nodes also stayed.

(b) We navigated to it from its child \textit{text()} node. Since $S'$ is \textit{C}-symmetric it contains the \textit{C.f1.text()} nodes of all \textit{C} nodes with the same \textit{Name} value, and therefore the \textit{C.f1} nodes of all those \textit{C} nodes will appear in $S'$.

(c) We navigated to it from its parent \textit{C} node. Since $S''$ is \textit{C}-symmetric, it contains all other \textit{C} nodes with the same \textit{Name} and therefore $S'$ contains all the required \textit{C.f1} nodes.

(d) We navigated to it from \textit{A} or \textit{R}. This means that $S'$ contains the \textit{C.f1} nodes of all \textit{C} nodes in the XML tree.

- Section 3 of the \textit{C}-symmetry definition: We need to show that if $S'$ contains some \textit{C.f2} node then it contains the \textit{f2} nodes of all \textit{C} nodes with the same \textit{Name}. This is analogous to section 2.

- Section 4 of the \textit{C}-symmetry definition: We need to show that if $S'$ contains some \textit{C.Name} node then it contains all \textit{C.Name} nodes with the same value. This is analogous to section 2.

- Section 5 of the \textit{C}-symmetry definition: We need to show that if $S'$ contains some \textit{C.f1.text()} node then it contains all \textit{C.f1.text()} nodes of \textit{C} nodes with the same \textit{Name}. If $S'$ contains some \textit{C.f1.text()} node then there are four possibilities.

  (a) It was already in $S''$ and stayed using \textit{self}. This means all the other \textit{C.f1.text()} nodes also stayed.

  (b) We navigated to it from its parent \textit{C.f1} node. Since $S''$ is \textit{C}-symmetric, it contains the \textit{C.f1} nodes of all other \textit{C} nodes with the same \textit{Name} and therefore $S'$ contains all the required \textit{C.f1.text()} nodes.

  (c) We navigated to it from its grandparent \textit{C} node. Since $S''$ is \textit{C}-symmetric, it contains all other \textit{C} nodes with the same \textit{Name} and therefore $S'$ contains all the required \textit{C.f1.text()} nodes.

  (d) We navigated to it from \textit{A} or \textit{R}. This means that $S'$ contains the \textit{C.f1.text()} nodes of all \textit{C} nodes in the XML tree.

- Section 6 of the \textit{C}-symmetry definition: We need to show that if $S'$ contains some \textit{C.f2.text()} node then it contains all \textit{C.f2.text()} nodes of \textit{C} nodes with the same \textit{Name}. This is very similar to section 5.
Section 7 of the definition: We need to show that if \( S' \) contains some \( C.Name() \) node then it contains all \( C.Name() \) nodes with the same value. This is very similar to section 5.

2. (D-symmetry) Very similar to the proof for C-symmetry.

3. (D-identicalness) By induction on the number of navigation steps \( n \).
   \( S1 \) and \( S2 \) are D-identical and \( S1' \) and \( S2' \) are the context sets after executing the expression from \( S1 \) and \( S2 \), respectively.
   We need to prove that \( S1' \) and \( S2' \) are D-identical, and if they are not identical then they have the same diff-base as \((S1, S2)\).
   Induction base: \( n = 0 \). \( \implies S1' = S1 \) and \( S2' = S2 \) and therefore \( S1' \) and \( S2' \) are D-identical.
   Induction step: After \( k \) steps we have \( S1'' \) and \( S2'' \) that are D-identical and we need to prove that after an additional step we get \( S1' \) and \( S2' \) that are D-identical, and the diff-base is also preserved, i.e., they differ in the same \( D \) nodes that \( S1 \) and \( S2 \) differ in (if they are not identical).
   We consider the following cases.

   (a) If \( S1'' \) and \( S2'' \) are identical then clearly so are \( S1' \) and \( S2' \). \( S1'' \) and \( S2'' \) are also D-symmetric. We need to prove that so are \( S1' \) and \( S2' \). This follows from section 2 of this lemma, in which we proved that every navigation step preserves D-symmetry.

   (b) If \( S1'' \) and \( S2'' \) are not identical, then there are two distinct \( D \)-nodes \( D_i \) and \( D_j \) such that there is a D-symmetric common part of these sets, and in addition to that part \( S1'' \) contains some \( D_i \)-related nodes while \( S2'' \) contains the corresponding \( D_j \)-related nodes.
   The navigation from the common, D-symmetric part, leads to identical, D-symmetric sets (since the navigation step preserves D-symmetry).
   Therefore, we only need to prove that the navigation from the different parts of \( S1'' \) and \( S2'' \) preserves D-identicalness.
   To do this we will prove that D-identical context sets are created when a navigation step is executed from the different, \( D_i \)-related and \( D_j \)-related nodes (of \( S1'' \) and \( S2'' \), respectively), and also that the diff-base remains \((D_i, D_j)\). This is enough due to Lemma A.3.
   There are several possibilities.

   - \( S1'' \) contains \( D_i \) and \( S2'' \) contains \( D_j \): If the navigation step is \(/f1 \) or \(/f2 \) or \(/child :: * \) then we get \{\( D_i.f1 \),\( D_j.f1 \)\} or \{\( D_i.f2 \),\( D_j.f2 \)\} or \{\( D_i.f1, D_i.f2 \),\( D_j.f1, D_j.f2 \)\}. In each
of these possibilities we get D-identical sets with a diff-base of \((D_i, D_j)\).

If the navigation step uses the parent or ancestor axes then the sets we get are identical and do not contain any \(D\) nodes, so they are D-identical.

If the ancestor-or-self axis is used then the sets contain \(D_i\) and \(D_j\), respectively, in addition to a common D-symmetric part, and therefore they are D-identical (and the diff-base is preserved).

- \(S_1''\) contains \(D_i.f1\) and \(S_2''\) contains \(D_j.f1\): Navigation from \(D_i.f1\) or \(D_j.f1\) via the child axis leads to empty sets. Navigation to the \text{}() nodes leads to the sets \(\{D_i.f1.text()\}\), \(\{D_j.f1.text()\}\). If the parent axis is used then we get \(\{D_i\}\), \(\{D_j\}\).

- \(S_1''\) contains \(D_i.f2\) and \(S_2''\) contains \(D_j.f2\): similar analysis.
- \(S_1''\) contains \(D_i.f1.text()\) and \(S_2''\) contains \(D_j.f1.text()\): similar analysis.
- \(S_1''\) contains \(D_i.f2.text()\) and \(S_2''\) contains \(D_j.f2.text()\): similar analysis.

4. (C-identicalness) By induction on the number of navigation steps \(n\).

\(S_1\) and \(S_2\) are C-identical and \(S_1'\) and \(S_2'\) are the context sets after executing the expression from \(S_1\) and \(S_2\), respectively.

We need to prove that \(S_1'\) and \(S_2'\) are C-identical, and if they are not identical then they have the same diff-base as \((S_1, S_2)\).

Induction base: \(n = 0\). \(\implies S_1' = S_1\) and \(S_2' = S_2\) and therefore \(S_1'\) and \(S_2'\) are C-identical.

Induction step: After \(k\) steps we have \(S_1''\) and \(S_2''\) that are C-identical and we need to prove that after an additional step we get \(S_1'\) and \(S_2'\) that are C-identical, and the diff-base is also preserved, i.e., they differ in the same \(C\) nodes that \(S_1\) and \(S_2\) differ in (if they are not identical).

We consider the following cases.

(a) If \(S_1''\) and \(S_2''\) are identical then clearly so are \(S_1'\) and \(S_2'\). \(S_1''\)
and $S_2''$ are also C-symmetric. We need to prove that so are $S_1'$ and $S_2'$. This follows from section 1 of this lemma, in which we proved that every navigation step preserves C-symmetry.

(b) If $S_1''$ and $S_2''$ are not identical, then there are two distinct C-nodes $C_i$ and $C_j$ such that there is a C-symmetric common part of these sets, and in addition to that part $S_1''$ contains some $C_i$-related nodes while $S_2''$ contains the corresponding $C_j$-related nodes.

The navigation from the common, C-symmetric part, leads to identical, C-symmetric sets (since the navigation step preserves C-symmetry).

Therefore, we only need to prove that the navigation from the different parts of $S_1''$ and $S_2''$ preserves C-identicalness.

To do this we will prove that C-identical context sets are created when a navigation step is executed from the different, $C_i$-related and $C_j$-related nodes (of $S_1''$ and $S_2''$, respectively), and also that the diff-base remains $(C_i,C_j)$. This is enough due to Lemma A.4.

There are several possibilities.

- $S_1''$ contains $C_i$ and $S_2''$ contains $C_j$ in its stead: If the navigation step is $/f1$ or $/f2$ or $/Name$ or $/child :: *$ then we get \{${C_i}.f1$,${C_j}.f1$\} or \{${C_i}.f2$,${C_j}.f2$\} or \{${C_i}.Name$,{${C_j}.Name$\} or \{${C_i}.f1$,$C_i$.f2$,$C_i$.Name\},\{${C_j}.f1$,$C_j$.f2,$C_j$.Name\}. In each of these possibilities we get C-identical sets with a diff-base of $(C_i,C_j)$.

  If the navigation step uses the parent or ancestor axes then the sets we get are identical and do not contain any C nodes, so they are C-identical.

  If the ancestor-or-self axis is used then the sets contain $C_i$ and $C_j$, respectively, in addition to a common C-symmetric part, and therefore they are C-identical (and the diff-base is preserved).

- $S_1''$ contains $C_i.f1$ and $S_2''$ contains $C_j.f1$: Navigation from $C_i.f1$ or $C_j.f1$ via the child axis leads to empty sets.

  Navigation to the text() nodes leads to the sets \{${C_i}.f1$.text()\}, \{${C_j}.f1$.text()\}.

    If the parent axis is used then we get \{${C_i}$\},\{${C_j}$\}.

    If the ancestor axis is used then we get sets that contain $C_i$ and $C_j$, respectively, in addition to a common C-symmetric part (that does not contain any C or f1 or f2 or Name nodes). If ancestor-or-self is used then the sets contain...
\{C_i,C_i.f1\} and \{C_j,C_j.f1\}, respectively in addition to the common part.
In each of these cases the sets are C-identical and the \textit{diff-base} is preserved.

- $S1''$ contains $C_i.f2$ and $S2''$ contains $C_j.f2$: similar analysis.
- $S1''$ contains $C_i.Name$ and $S2''$ contains $C_j.Name$: similar analysis.
- $S1''$ contains $C_i.f1.text()$ and $S2''$ contains $C_j.f1.text()$: similar analysis.
- $S1''$ contains $C_i.f2.text()$ and $S2''$ contains $C_j.f2.text()$: similar analysis.
- $S1''$ contains $C_i.Name.text()$ and $S2''$ contains $C_j.Name.text()$: similar analysis.

The result of executing a navigational-expression $exp$ on a context node $n$, i.e., the context set after the execution, will be written as $exp(n)$. The Boolean result of a comparison $comp$ (of the form $e1 = e2$) will be written as $comp(n)$. The Boolean result of a predicate expression $predExpr$ will be written as $predExpr(n)$.

**Lemma A.6** Let $comp$ be a comparison of the form $e1 = e2$ where $e1$ and $e2$ are navigational-expressions in XPath’.\textit{NoConst} that preserve D-identicalness, C-identicalness, D-symmetry and C-symmetry, and possibly $e1$ or $e2$ is the constant "a" or the constant "b". Let $D_i$ and $D_j$ be some D nodes and let $C_i$ and $C_j$ be C nodes with the same Name. Then, the following hold:

1. $comp(D_i) = comp(D_j)$, $comp(D_i.f1) = comp(D_j.f1)$, $comp(D_i.f2) = comp(D_j.f2)$, $comp(D_i.f1.text()) = comp(D_j.f1.text())$, $comp(D_i.f2.text()) = comp(D_j.f2.text())$.

2. $comp(C_i) = comp(C_j)$, $comp(C_i.f1) = comp(C_j.f1)$, $comp(C_i.f2) = comp(C_j.f2)$, $comp(C_i.Name) = comp(C_j.Name)$, $comp(C_i.f1.text()) = comp(C_j.f1.text())$, $comp(C_i.f2.text()) = comp(C_j.f2.text())$, $comp(C_i.Name.text()) = comp(C_j.Name.text())$.

**Proof** If $e1$ and $e2$ are both constants then the comparison is always \textit{true} or always \textit{false}, regardless of the context. Therefore, for the rest of the proof we will assume that only one of $e1$ and $e2$ may be a constant.
Without loss of generality, we will consider only the case where $e2$ is a constant (which is analogous to the one where $e1$ is a constant).
When an expression is a constant, we will consider the “context set” after its execution to be a set containing only the constant (formally this is not a context set, since it does not contain nodes).

Also note that if an expression is not a constant then the context set after its execution cannot contain constants, since navigation from a node leads to other nodes.

1. Let $S_1$ and $S_2$ be the context sets before $comp$ is evaluated.

   $S_1 = \{s_1\}$ and $S_2 = \{s_2\}$, and we can have $s_1 = D_i, s_2 = D_j$ or $s_1 = D_i, f_1, s_2 = D_j, f_1$ or $s_1 = D_i, f_2, s_2 = D_j, f_2$ or $s_1 = D_i, f_1.text(), s_2 = D_j, f_1.text()$ or $s_1 = D_i, f_2.text(), s_2 = D_j, f_2.text()$.

   In any one of those cases $S_1$ and $S_2$ are D-identical and C-symmetric.

   Let $S_1'$ and $S_2'$ be the context sets after executing $e_1$ from context sets $S_1$ and $S_2$, respectively. Let $S_1''$ and $S_2''$ be the context sets after executing $e_2$ from context sets $S_1$ and $S_2$, respectively.

   Since $e_1$ and $e_2$ preserve D-identicalness, $S_1'$ and $S_2'$ are D-identical, and also $S_1''$ and $S_2''$ are D-identical. The diff-base is $(D_i, D_j)$.

   Since $e_1$ and $e_2$ preserve C-symmetry, $S_1'$, $S_2'$, $S_1''$ and $S_2''$ are C-symmetric.

   We want to prove that $comp(s_1) = comp(s_2)$, i.e., that $comp(s_1)$ implies $comp(s_2)$ and $comp(s_2)$ implies $comp(s_1)$. It is sufficient to prove that $comp(s_1)$ implies $comp(s_2)$ (the other direction is symmetric).

   $comp(s_1) = true \implies$ there is a node $x$ in $S_1'$ and a node $y$ in $S_1''$ such that the string values of $x$ and $y$ are equal. There are several possibilities.

   - $e_2$ is a constant: $x$ is a $C.Name$ or a $C.Name.text()$ node. Because of D-identicalness, $x$ also belongs to $S_2'$. Since $e_2$ is a constant, $S_2'' = S_1''$. Therefore, $comp(s_2) = true$.

   - Neither $x$ nor $y$ is one of $D_i, D_i, f_1, D_i, f_2, D_i, f_1.text()$ and $D_i, f_2.text()$: $x$ also belongs to $S_2'$ and $y$ belongs to $S_2''$ (because of the D-identicalness of $S_1', S_2'$ and of $S_1'', S_2''$), and therefore we get $comp(s_2) = true$.

   - $x$ is one of $D_i, D_i, f_1, D_i, f_2, D_i, f_1.text()$ and $D_i, f_2.text()$: There is a node $z$ in $S_2'$ such that $z$ is the corresponding ”$D_j$ related” node, i.e., $D_j$ or $D_j, f_1$ or $D_j, f_2$ or $D_j, f_1.text()$ or $D_j, f_2.text()$ (see Figure A.1).

   This is ensured by the D-symmetry requirement of the D-identicalness definition. (Because either $S_1'$ and $S_2'$ are identical and D-symmetric, which means they both contain both the $D_i$-related and the $D_j$-related node, or they are not identical and then the identical part
is D-symmetric, which means that if \( x \) is in the identical part then the \( D_j \)-related node is also in it and if \( x \) is not in the identical part then \( S2' \) contains the \( D_j \)-related node that corresponds to \( x \). Now we will examine the possibilities for \( y \).

- \( y \) is identical to \( x \): There is a node \( w \) in \( S2'' \) that is identical to \( z \) (for the same reason that \( z \) exists in \( S2' \)). Therefore, \( \text{comp}(s2) = \text{true} \).

- \( y \) is not identical to \( x \) but rather a different node with the same string value: We will go over the possibilities for \( x \).
  
  * \( x \) cannot be a \( D \) node because in our XML tree every \( D \) node has a different string value.
  * \( x \) is \( D_i.f1 \): There are several possibilities for \( y \): (1) \( y \) can be \( D_i.f1.text() \). This means \( S2'' \) contains \( D_j.f1.text() \), which has the same value as \( z = D_j.f1 \). (2) \( y \) can be some other \( D.f1 \) node. This means \( e2 \) must navigate (from the starting \( D_i \)-related node \( s1 \)) to \( B \) or \( R \) and then to the other \( D \) node, but since \( \{B\} \) and \( \{R\} \) are D-symmetric sets, and \( e2 \) preserves D-symmetry (after every step), we get that \( S1'' \) contains all \( D.f1 \) nodes, including \( D_i.f1 \). Therefore \( S2'' \), which is D-identical to \( S1'' \), contains \( D_j.f1 \). Since \( S2' \) (which is D-identical to \( S1' \)) also contains \( D_j.f1, \text{comp}(s2) = \text{true} \). (3) \( y \) can be the \( \text{text}() \) node of some other \( D.f1 \) node, in which case the analysis is very similar. (4) \( y \) can be a \( C.f1 \) node (for every possible \( f1 \) value there are two \( C \) nodes). Since \( S1'' \) is C-symmetric, it contains also the \( C.f1 \) node of the other \( C \) node with the same Name. Then we get that \( z \) is \( D_j.f1 \).
and $S_2''$ (which is D-identical to $S_1''$) contains the two $C.f1$ nodes, one of which has the same value as $D_j.f1$, so that $\text{comp}(s_2) = true$. (5) $y$ can be a $C.f1.text()$ node, in which case the analysis is very similar to the previous case.

* $x$ is $D_i.f2$: Similar reasoning shows that $\text{comp}(s_2) = true$.
* $x$ is $D_i.f1.text()$ or $D_i.f2.text()$: The analysis is similar.

- $y$ is one of $D_i$, $D_i.f1$, $D_i.f2$, $D_i.f1.text()$, $D_i.f2.text()$: The analysis is analogous to the case were $x$ is one of these nodes.

2. (The proof for $C$ node comparisons is similar.)

**Definition A.11** The depth of a predicate is the level of predicate nesting in it, i.e., if a predicate $p$ does not contain predicates then its depth is 1 and if it contains predicates and the maximal depth among those predicates is $k$ then the depth of $p$ is $k + 1$.

**Definition A.12** The predicate-depth of an expression is the maximal depth of predicates appearing in the expression, or 0 if the expression contains no predicates.

**Lemma A.7** Let $e$ be a navigational-expression in $\text{XPath}'_{\text{NoConst}}$. Then the following conditions are satisfied: (1) $e$ preserves D-identicalness, (2) $e$ preserves C-identicalness, (3) $e$ preserves D-symmetry, (4) $e$ preserves C-symmetry.

**Proof** The proof is by induction on the predicate-depth of $e$.

Basis: $\text{depth} = 0$. There are no predicates. This case is handled in Lemma A.5.

Induction step: Assume the lemma holds for an expression of depth $k$ and consider depth $k + 1$.

1. (D-identicalness) The proof is by induction on the number of steps, $m$, in $e$, where a step is either a navigation step or a predicate. Let $S_1$ and $S_2$ be D-identical sets with a diff-base of $(D_i,D_j)$ (unless they are identical) and let $S_1'$ and $S_2'$ be the context sets after executing the expression from $S_1$ and $S_2$, respectively. We need to prove that $S_1'$ and $S_2'$ are D-identical and that the diff-base is preserved (if they are not identical).

Basis: zero steps. Here, $S_1' = S_1$, $S_2' = S_2$.

Induction. After $m$ steps, we have $S_1''$ and $S_2''$ that are D-identical. We need to prove for $m + 1$ steps.
If the next step is a navigation step (not a predicate) then D-identicalness is preserved because this is the same case as in Lemma A.5.

Suppose the next step is a predicate. The predicate may contain comparisons and navigational-expressions of predicate-depth smaller than or equal to $k$. First we will show that for every two $D$ nodes $D_r$ and $D_q$, the predicate gives the same result for a $D_r$-related node and for the corresponding $D_q$-related node. To show this, we will go over all the possible $D_r$ and $D_q$ related nodes:

- The result of the predicate is the same for $D_r$ and for $D_q$ because the comparisons in the predicate give the same result for $D_r$ and for $D_q$ (by Lemma A.6), and the navigational-expressions in the predicate preserve D-identicalness (hypothesis of the induction on predicate-depth) and therefore for such a navigational-expression $e_1$, $e_1(D_r)$ is non-empty iff $e_1(D_q)$ is non-empty.

  Note: We can use Lemma A.6 here because the expressions in the comparison are of depth $\leq k$ and therefore preserve D-identicalness, C-identicalness, D-symmetry and C-symmetry.

- Using a similar argument, we obtain: (1) The result of the predicate is the same for $D_r.f_1$ and for $D_q.f_1$. (2) The result of the predicate is the same for $D_r.f_2$ and for $D_q.f_2$. (3) The result of the predicate is the same for $D_r.f_1.text()$ and for $D_q.f_1.text()$. (4) The result of the predicate is the same for $D_r.f_2.text()$ and for $D_q.f_2.text()$.

Therefore we deduce that: (1) The common, D-symmetric part of $S_1''$ and $S_2''$ becomes a common, D-symmetric part of $S_1'$ and $S_2'$ after the predicate is executed. (2) If $S_1''$ contains some $D_i$-related node and $S_2''$ contains the corresponding $D_j$-related node, the predicate gives the same result for the $D_i$-related node and for the $D_j$-related node and therefore $S_1'$ contains the $D_i$-related node iff $S_2'$ contains the $D_j$ related nodes. Therefore, the predicate preserves D-identicalness.

2. (C-identicalness) We will prove by induction on the number of steps in $e$, where a step is either a navigation step or a predicate.

$S_1$ and $S_2$ are C-identical and $S_1'$ and $S_2'$ are the context sets after executing the expression from $S_1$ and $S_2$, respectively.

We need to prove that $S_1'$ and $S_2'$ are C-identical and the diff-base is preserved.

Induction base: Zero steps $\implies S_1' = S_1$, $S_2' = S_2$.

Induction step: After $m$ steps we have $S_1''$ and $S_2''$ that are C-identical.
We need to prove for $m + 1$ steps. If the next step is a navigation step (not a predicate) then C-identicalness is preserved because this is the same case as in Lemma A.5. Suppose the next step is a predicate. The predicate may contain comparisons and navigational-expressions of predicate-depth smaller than or equal to $k$. We will show that the predicate gives the same result for a $C_i$-related node of $S'_{1''}$ and for a $C_j$-related node of $S'_{2''}$ and therefore preserves C-identicalness. We will show this for all possible $C_i$ and $C_j$ related nodes.

- $S'_{1''}$ contains $C_i$ and $S'_{2''}$ contains $C_j$ instead: By definition of C-identicalness, $C_i$ and $C_j$ have the same Name value. Therefore, $S'_{1'}$ contains $C_i.f_1$ iff $S'_{2'}$ contains $C_j.f_1$. If $S'_{1''}$ contains $C_i.f_2$ and $S'_{2''}$ contains $C_j.f_2$ instead, then $S'_{1'}$ contains $C_i.f_2$ iff $S'_{2'}$ contains $C_j.f_2$. If $S'_{1''}$ contains $C_i.Name$ and $S'_{2''}$ contains $C_j.Name$ instead, then $S'_{1'}$ contains $C_i.f_2$ iff $S'_{2'}$ contains $C_j.f_2$. If $S'_{1''}$ contains $C_i.f_1.text()$ and $S'_{2''}$ contains $C_j.f_1.text()$ instead, then $S'_{1'}$ contains $C_i.f_1.text()$ iff $S'_{2'}$ contains $C_j.f_1.text()$. If $S'_{1''}$ contains $C_i.f_2.text()$ and $S'_{2''}$ contains $C_j.f_2.text()$ instead, then $S'_{1'}$ contains $C_i.f_2.text()$ iff $S'_{2'}$ contains $C_j.f_2.text()$. If $S'_{1''}$ contains $C_i.Name.text()$ and $S'_{2''}$ contains $C_j.Name.text()$ instead, then $S'_{1'}$ contains $C_i.f_2.text()$ iff $S'_{2'}$ contains $C_j.f_2.text()$.

Therefore, the predicate preserves C-identicalness.

3. (C-symmetry) We will prove by induction on the number of steps in $e$, where a step is either a navigation step or a predicate.

Induction base: Zero steps $\implies S' = S$.

Induction step: After $m$ steps we have a C-symmetric context set $S''$. We need to prove for $m + 1$ steps.
If the next step is a navigation step (not a predicate) then C-symmetry is preserved because this is the same case as in Lemma A.5.

Suppose the next step is a predicate. The predicate may contain comparisons and navigational-expressions of predicate-depth smaller than or equal to $k$. The predicate will give the same truth value for all $C$ nodes with the same Name (i.e., will filter all or none of them), because the comparisons will give the same value for these $C$ nodes (by Lemma A.6) and the navigational-expressions in the predicate evaluate to the same truth value for all $C$ nodes with the same Name since they preserve C-identicalness (hypothesis of the induction on predicate-depth) and therefore for such a navigational-expression $e_1$, $e_1(C)$ is non-empty for one $C$ node iff it is non-empty for all $C$-nodes with the same Name.

Using a similar argument, we obtain:

- The predicate will give the same truth value for all of the $C.f1$ nodes of $C$ nodes with the same Name.
- The predicate will give the same truth value for all of the $C.f2$ nodes of $C$ nodes with the same Name.
- The predicate will give the same truth value for all $C.Name$ nodes with the same value.
- The predicate will give the same truth value for all of the $C.f1.text()$ nodes of $C$ nodes with the same Name.
- The predicate will give the same truth value for all of the $C.f2.text()$ nodes of $C$ nodes with the same Name.
- The predicate will give the same truth value for all of the $C.Name.text()$ nodes with the same value.

Therefore, the predicate preserves C-symmetry.

4. (D-symmetry) We will prove by induction on the number of steps in $e$, where a step is either a navigation step or a predicate.

Induction base: Zero steps $\implies S' = S$.

Induction step: After $m$ steps we have a D-symmetric context set $S''$. We need to prove for $m + 1$ steps.

If the next step is a navigation step (not a predicate) then D-symmetry is preserved because this is the same case as in Lemma A.5.

Suppose the next step is a predicate. The predicate may contain comparisons and navigational-expressions of predicate-depth smaller than or equal to $k$. The predicate will give the same truth value for all $D$
nodes (i.e., will filter all or none of them), because the comparisons will give the same value for all $D$ nodes (by Lemma [A.6]) and the navigational-expressions in the predicate evaluate to the same truth value for all $D$ nodes since they preserve D-identicalness (hypothesis of the induction on predicate-depth) and therefore for such a navigational-expression $e_1$, $e_1(D)$ is non-empty for one $D$ node iff it is non-empty for all $D$-nodes.

Using a similar argument, we obtain:

- The predicate will give the same truth value for all $D.f_1$ nodes.
- The predicate and will give the same truth value for all $D.f_2$ nodes.
- The predicate will give the same truth value for all $D.f_1.text()$ nodes.
- The predicate and will give the same truth value for all $D.f_2.text()$ nodes.

Therefore, the predicate preserves D-symmetry. ■

**Lemma A.8** Let $comp$ be a comparison of the form $e_1 = e_2$, where $e_1$ and $e_2$ are navigational-expressions in $XPath'_{NoConst}$ or constants. Then $comp(D_1) = comp(D_2)$ ($D_1$ and $D_2$ are two specific nodes in $BaseXML$ (see Figure 5.3)).

**Proof** According to Lemma [A.7] $e_1$ and $e_2$ preserve C-identicalness, D-identicalness, C-symmetry and D-symmetry. Therefore, by Lemma [A.6] we obtain $comp(D_1) = comp(D_2)$. ■

**Definition A.13** A predicate $pred$ is a D12Predicate if $pred$ gives a different result for $D_1$ and for $D_2$, i.e., if the context set before executing the predicate contains both $D_1$ and $D_2$ then the context set after executing the predicate will contain exactly one of the two nodes.

**Lemma A.9** There exist no D12Predicates in $XPath'_{NoConst}$.

**Proof** Suppose, for the sake of deriving a contradiction, that there exists a D12Predicate, of the form $[predExpr]$. $predExpr$ may contain comparisons and navigational-expressions. We claim that $predExpr(D_1) = predExpr(D_2)$. In proof, at the ”top level” of $predExpr$ there are comparisons and navigational-expressions, connected with Boolean operators ($and$, $or$), and possibly also with the Boolean function $not()$. According
to Lemma A.8 for every comparison \( \text{comp} \), \( \text{comp}(D_1) = \text{comp}(D_2) \). According to Lemma A.7 for every navigational-expression \( \text{exp} \), \( \text{exp}(D_1) \) and \( \text{exp}(D_2) \) are D-identical, and therefore \( \text{exp}(D_1) \) is non-empty iff \( \text{exp}(D_2) \) is non-empty, which means \( \text{exp} \) evaluates to the same truth value when operating in the context of \( D_1 \) and also in the context of \( D_2 \). So, \( \text{predExpr} \) evaluates to the same truth value when operating in the context of \( D_1 \) and in the context of \( D_2 \). The claim is now established, which implies that \( \text{pred} \) is not a D12Predicate. This is a contradiction and thus \( \text{pred} \) can not exist.

**Definition A.14** \( D12XMLTrees \) is the set of XML trees \( \{T_0, T_1, \ldots \} \) with the same structure as \( \text{BaseXML} \) (see Figure 5.3) but with different values in each tree. The value of an \( f_1 \) or \( f_2 \) node in \( T_i \) is \( 10^i \times (\text{the value of the corresponding node in } \text{BaseXML}) \). \( T_0 \) is \( \text{BaseXML} \). Note that Lemma A.9 applies for all \( T_i \), since increasing the \( f_1 \) and \( f_2 \) values proportionally does not affect our proof.

For example, the value of \( D_1.f_1 \) in \( T_4 \) is \( 3 \times 10^4 = 30000 \).

**Lemma A.10** There is no predicate in \( XPath' \) (i.e., with constants) that, for every XML tree in \( D12XMLTrees \), gives a result for \( D_1 \) that is different from the result for \( D_2 \).

**Proof** According to Lemma A.9 there is no such predicate in \( XPath'_\text{NoConst} \), because there is no such predicate for \( \text{BaseXML} \). So, if there is such a predicate in \( XPath' \), it must use constants other than "a" and "b". In order to derive a contradiction, suppose that such a predicate exists. Since constants can only be used in comparisons with expressions, in order for the predicate to give a different result for \( D_1 \) and for \( D_2 \), it must contain some constants (other than "a" and "b") that have the same string value as nodes that appear in the tree (otherwise all comparisons to these constants will evaluate to \( \text{false} \)). Since the predicate is (syntactically) finite, it contains a finite number of constants. Let us denote the maximal number of characters in a constant that appears in the predicate by \( M \). Since the string value of every \( f_1 \) and \( f_2 \) node in \( T_i \) has \( i + 1 \) characters and the string values of \( D \) and \( C \) nodes contain even more characters, for all \( i \geq M, T_i \) contains no nodes whose string values appear in the predicate (and are not "a" or "b"). Therefore, the predicate will give the same answer for \( D_1 \) and for \( D_2 \) in every such tree (since the constants other than "a" and "b" are irrelevant and thus the predicate operates as it would with no constants other than "a" or "b"). Thus we derive a contradiction.

133
Lemma A.11 There is no expression $e$ in XPath$'$ such that for every XML data tree in $D_{12}$XMLTrees, $e$ returns the $D$ nodes that reference $C$ nodes whose Name is "a" (with an initial context set that contains only the root node).

Proof Suppose, for the sake of deriving a contradiction, that such an expression $e$ exists. Then, for every tree in $D_{12}$XMLTrees, $e$ returns the set $\{D_2, D_3\}$. Therefore, for every tree in $D_{12}$XMLTrees, the following predicate gives a different result for $D_1$ and for $D_2$: $\cdot = /e$. This evaluates to true on $D_2$ but not on $D_1$. This contradicts Lemma A.10.

Theorem A.1 There is no expression $e$ in XPath$'$ such that for every XML data tree that conforms to SchemaCD, $e$ returns exactly the $D$ nodes that reference $C$ nodes whose Name is "a".

Proof Suppose, for the sake of deriving a contradiction, that such an expression exists. Then, this expression also works for the XML trees of $D_{12}$XMLTrees, since they all conform to SchemaCD. But this contradicts Lemma A.11. ■

And now the proof for Theorem 5.2.

Proof We show a query that can be written in XPath$'_{fk}$ but not in XPath$'$. Given an XML data tree that conforms to SchemaCD, the query selects the $D$ nodes that reference $C$ nodes whose Name is "a". In XPath$'_{fk}$ this is written as $//C[\text{Name} = "a"]/\text{KR}\_\text{Children}$. According to Theorem A.1 such a query cannot be written in XPath$'$. ■
Appendix B

Incremental Validation Implementation: XSV+

We present code samples from the implementation.

B.1 Single-value Change

B.1.1 validateFieldChange()

This is the main function that implements the algorithm. Parameters: doc is an XML document, s is a schema, field is a node in the document, newVal is a value and outFile is a file object, to which validation result and time measurement are written.

```python
def validateFieldChange(doc, s, field, newVal, outFile):
    
    # Structural Validation
    validate_NoKeys(doc.documentElement, s, None)
    
    (keyUpdates, keyrefUpdates) = getKeyAndKeyrefUpdates([(field, newVal)])
    # Update the data structures
    try:
        updatePath(field, keyUpdates, keyrefUpdates)
        commitChanges(keyUpdates, keyrefUpdates)
        res=True
    except:
        
```

135
This function receives a list of field changes of the form \((\text{fieldNode}, \text{newValue})\) and prepares the lists of key-sequence changes \(\text{keyUpdates}\) and \(\text{keyrefUpdates}\). It implements the first stage (‘Finding affected nodes’) of the algorithms of Sections 2.6.1 and 2.6.2.

```python
def getKeyAndKeyrefUpdates(fieldChanges):
    # A dictionary. \(\text{keyUpdates}[n]=(\text{ks}, \text{ks}')\), where \(n\) is
    # a key selector-identified node, whose key-sequence
    # stands to be changed from \(\text{ks}\) to \(\text{ks}'\) due to the field changes.
    keyUpdates = {}
    # A dictionary. \(\text{keyrefUpdates}[n]=(\text{ks}, \text{ks}')\), where \(n\) is
    # a keyref selector-identified node, whose key-sequence
    # stands to be changed from \(\text{ks}\) to \(\text{ks}'\) due to the field changes.
    keyrefUpdates = {}

    # Calculate key updates
    for (key, keyFieldInfo) in KeyFieldInfo.items():
        # updates for this key
        keyUpdates1 = {}
        for (field, newVal) in fieldChanges:
            if keyFieldInfo.has_key(field):
                for (node, index) in keyFieldInfo[field]:
                    oldSeq = KeySelIdent[key][node]
                    seq = oldSeq
                    if keyUpdates1.has_key(node):
                        seq = keyUpdates1[node][1]
                    newSeq = seq[0:index] + tuple([newVal]) + seq[(index+1):len(seq)]
                    keyUpdates1[node] = (oldSeq, newSeq)
        if keyUpdates1 != {}:
            keyUpdates[key] = keyUpdates1

    # Calculate keyrefUpdates

    return (keyUpdates, keyrefUpdates)
```

136
B.1.3 updatePath()

def updatePath(startNode, keyUpdates, keyrefUpdates):
    updatePathToJoinNode(startNode, None, None, {}, keyUpdates, keyrefUpdates)

B.1.4 updatePathToJoinNode()

Following one or more value changes of fields in the document, which cause key-sequences in the document to change, this function updates the data structures associated with nodes along a path, starting from a specified node and moving up the tree until either the root or a join node (if a set of join nodes is specified) is reached. It is used by the algorithms of Sections 2.6.1 and 2.6.2. Note that `keyUpdates` and `keyrefUpdates` are calculated using `getKeyAndKeyrefUpdates()`.

    def updatePathToJoinNode(startNode, startChild, startChangesInChild, joinNodes, keyUpdates, keyrefUpdates):
        currentNode = startNode
        currentChild = startChild
        changesInChild = startChangesInChild
        while not(isinstance(currentNode,Document)) and
             not(joinNodes.has_key(currentNode)):
            changeList=None
            if changesInChild!=None:
                changeList = [(currentChild, changesInChild)]
            changesInChild = updateNodeK(currentNode, changeList, keyUpdates)
            removedSequences=getRemovedSequences(changesInChild)
            updateNodeKR(currentNode, removedSequences, keyrefUpdates)
            currentChild = currentNode
            currentNode = currentNode.parent
        return (currentChild,changesInChild)

B.1.5 updateNodeK()

This function updates the data structures pertaining to key constraints which are associated with a single node (keyTabs, childrenKeyInfo).

    def updateNodeK(node, changesInChildren, keyInfo):
        if len(node.keyTabs)==0 and changesInChildren==None:
            return None
        changes={}

137
# process changes in children

```
if changesInChildren!=None and changesInChildren!=[]:
    # We go over all existing key constraints
    for key in KeySelIdent.keys():
        changesForKey={}

        # key-sequences for which we need to change childrenInfo
        childrenInfoChangedSequences={}

        for (child,changesInChild) in changesInChildren:
            if changesInChild.has_key(key):
                if not(node.childrenInfo.has_key(key)):
                    node.childrenInfo[key]={}
                childrenInfo = node.childrenKeyInfo[key]

                for (ks,n) in changesInChild[key].items():
                    # value for ks is irrelevant, we just want to
                    # know if it’s there
                    childrenInfoChangedSequences[ks]=None
                    if not(childrenInfo.has_key(ks)):
                        childrenInfo[ks] = {}  
                    if (n!=None):
                        childrenInfo[ks][child]=n
                    else:
                        if childrenInfo[ks].has_key(child):
                            del childrenInfo[ks][child]

                if childrenInfoChangedSequences!={}:
                    if not(node.keyTabs.has_key(key)):
                        node.keyTabs[key]={}
                    keyTab = node.keyTabs[key]

                    for ks in childrenInfoChangedSequences.keys():
                        if keyTab.has_key(ks):
                            if keyTab[ks][1]==True:
                                # we don’t touch a selectorIdentified entry
                                continue
                            else:
                                old = keyTab[ks][0]
                                del keyTab[ks]
```
addKeyTabsRollbackInfo(node, key, ks, old, False)
changesForKey[ks] = None
if len(childrenInfo[ks]) == 1:
    # ks appears once in the children’s data structures
    nodeInChild = childrenInfo[ks].values()[0]
    keyTab[ks] = (nodeInChild, False)
    # note: if we removed a row in the previous 'if'
    # then we already added to rollbackinfo and
    # the following line will not do anything
    addKeyTabsRollbackInfo(node, key, ks, None, None)
changesForKey[ks] = nodeInChild

if childrenInfo[ks] == {}:
    del childrenInfo[ks]

if changesForKey != {}:
    changes[key] = changesForKey

#process keyUpdates
for (key, updatesForKey) in keyUpdates.items():
    if not(node.keyTabs.has_key(key)):
        node.keyTabs[key] = {}
    keyTab = node.keyTabs[key]
    if not(node.childrenInfo.has_key(key)):
        node.childrenInfo[key] = {}
    childrenInfo = node.childrenInfo[key]
    if not(changes.has_key(key)):
        changes[key] = {}
    changesForKey = changes[key]

    # updates (n, (oldSeq, newSeq)) in updatesForKey where the node n is
    # a selector-identified node of the 'node' parameter of updateNodeK().
    relevantUpdates = {}

    for (n, (oldSeq, newSeq)) in updatesForKey.items():
        if keyTab.has_key(oldSeq) and keyTab[oldSeq] == (n, True):
            relevantUpdates[n] = (oldSeq, newSeq)
            #dealing with old sequences
        for (n, (oldSeq, newSeq)) in relevantUpdates.items():
            # remove n from keyTab, possibly replacing it with a child record

139
if childrenInfo.has_key(oldSeq) and len(childrenInfo[oldSeq])==1:
    nodeInChild=childrenInfo[oldSeq].values()[0]
    keyTab[oldSeq]=(nodeInChild,False)
    changesForKey[oldSeq]=nodeInChild
else:
    del keyTab[oldSeq]
    changesForKey[oldSeq]=None
    addKeyTabsRollbackInfo(node,key,oldSeq,n,True)
#dealing with new sequences
for (n,(oldSeq,newSeq)) in relevantUpdates.items():
    # Get the current entry for newSeq in keyTab, if one exists.
    oldNode=None
    oldBool=None
    if keyTab.has_key(newSeq):
        oldNode=keyTab[newSeq]
        oldBool=False
        if keyTab[newSeq][1]==True:
            raise "Duplicate key sequence for Key "
            + repr(key.name) + ": " + repr(newSeq)
    keyTab[newSeq]=(n,True)
    addKeyTabsRollbackInfo(node,key,newSeq,oldNode,oldBool)
    changesForKey[newSeq]=n
    if changesForKey=={}:
        del changes[key]
    if (changes=={}):
        return None
    return changes

B.1.6 updateNodeKR()

This function updates the KeyrefInfo structure of a single node, and makes
sure that for every key-sequence of a keyref selector-identified node there is
a corresponding entry in the keyTabs data structure.

def updateNodeKR(node, removedSequences, keyrefUpdates):
    # make sure every new sequence has a match,
    # and update the sequences in KeyrefInfo
    for (keyref,updates) in keyrefUpdates.items():
        if not(node.KeyrefInfo.has_key(keyref)):
continue
keyrefInfo=node.KeyrefInfo[keyref]
keyTab = node.keyTabs[keyref.refer]
for (n,(oldSeq,newSeq)) in updates.items():
    if keyrefInfo.has_key(n):
        # (relevant update)
        if not(keyTab.has_key(newSeq)):
            raise "New keyref sequence for Keyref " +
            repr(keyref.name) +
            " has no matching key: " + repr(newSeq)
        keyrefInfo[n]=newSeq
        keyrefInfo[newSeq]=keyrefInfo.get(newSeq,0)+1
        keyrefInfo[oldSeq]=keyrefInfo[oldSeq]-1
        if (keyrefInfo[oldSeq]<1):
            del keyrefInfo[oldSeq]
        addKeyrefInfoRollbackInfo(node,keyref,n,oldSeq)

#check that there are no references to removed sequences
for (key,removedForKey) in removedSequences.items():
    for (keyref,keyrefInfo) in node.KeyrefInfo.items():
        if keyref.refer==key:
            for ks in removedForKey:
                if keyrefInfo.has_key(ks):
                    raise "Removed key sequence has a referencing node " +
                    "in Keyref " +
                    repr(keyref.name) + ": " + repr(ks)

return
XML

שרון קרישר
(XML) דורות בסקמה

דורא על מחקר

לשם מדליי חלקי של הדרישות לקבלה לתואר
מגיסטר למדעי מידע המחשב

שרון קרישר

הוגש לסמט הטכניון – מכון טכנולוגי לישראל

נולש תשמ"ו – חיפה

אפריל 2006
המחבר נועה בנויהית פרופ' עודד ש삿אי
בפנולש למדעי המוחהש
הענין
תוכן
תקציר
1
1.1.
1.2.
1.3.
1.3.1.
1.3.2.
1.4.
1.4.1.
1.4.2.
1.4.3.
1.4.4.
1.4.5.
1.5.
1.6.
1.7.
2.
2.1.
2.2.
2.2.1.
2.2.2.
2.2.3.
2.2.4.
2.2.5.
2.2.6.
2.3.
2.3.1.
2.3.2.
2.3.3.
2.3.4.
2.4.
2.5.
2.6.
2.6.1.
2.6.2.
2.6.3.
2.6.4.
2.7.
3.
3.1.
3.1.1.
3.1.2.
3.2.
3.2.1.
3.2.2.
3.2.3.
3.3.
3.4.
4.
4.1.
4.1.1.
4.1.2.
4.1.3. חוספת
4.1.4. מחיקה
4.2. תוצאות ו holster
4.2.1. צעדיםראשוניים
4.2.2. סימון ספר
4.2.3. כספים
4.2.4. חוספת
4.2.5. מחיקה
4.2.6. ציר פיקוד
4.4. סיכום הפרק
5. זורן מסהת
5.1. מוטיבציה
5.2. תרשים
5.3. ס颋
5.3.1. סטtement
5.3.2. מסקנות
5.4. חיתוכ
5.5. נגון שדה
5.6. איפור
5.6.1. איפור
5.6.2. איפור
5.6.3. איפור
5.7. סיכום הפרק
6. סיכום
111. ביבליוגרפיה
115. המחשבת מפורצת של משמע
135. XSV+. מוסדות והלידיקインタ
135. .1. a. validateFiledChange()
136. .2. b. getKeyAndKeyrefUpdates()
137. .2. c. updatePath()
137. .3. d. updatePathToJoinNode()
137. .2. e. updateNodeK()
140. .2. f. updateNodeKR()
רשימת ציורים

1.1 צירום

1.2 צירום

2.1 צירום

2.2 צירום

2.3 צירום

2.4 צירום

2.5 צירום

2.6 צירום

2.7 צירום

2.8 צירום

2.9 צירום

3.1 צירום

4.1 צירום

4.2 צירום

4.3 צירום

4.4 צירום

4.5 צירום

4.6 צירום

4.7 צירום

4.8 צירום

4.9 צירום

4.10 צירום

4.11 צירום

4.12 צירום

4.13 צירום

5.1 צירום

5.2 צירום

5.3 צירום

למה הוכחת עבור איור A.6
ה стоимостиים קבליים

ואיתנו

בנויים

זה

אחד

ולכן

אילוץ

הצמתים

מפתח

הצומת

מפתח

לצומת

אותם

לשם

במערכות

),

כדי

המתאים

הגדרת

צומת

מכילים

, לשתתפ

Pooling

בза"מ

וח

routeParams

 índex.

דרק

בז

, או

ב Married

 fists.

וח

, 

ב Married

 fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.

Pooling

Indexed

fists.
המחק המתרחש בשתי דרכים. הדרכה, פיתוח האלגוריתמים וה렬ציה (incremental validation).

האלגוריתמים יקטנים פעולות קבוצתיות של ייצוג שלبناء סטרה בלבית ייצוג של נקודת הציון (של ש♀רי)

וגם בשתי דרכים, הדרכה, פיתוח האלגוריתמים וה렬ציה (incremental validation).
In structural validation, the XSV (XML structural validation) technique is used to check that the XML document is well-formed. The XSV technique is based on the concept of validating documents against a schema. The schema defines the structure of the XML document, and the XSV engine checks that the document adheres to this structure.

The XSV engine uses a set of rules to validate the XML document. These rules are defined in the schema and can include constraints such as element names, attribute values, and the order of elements. The XSV engine checks that the document satisfies these rules.

In our project, we have developed an algorithm that is designed to handle XML data efficiently. The algorithm is based on the XSV technique and is implemented using C++.

To validate an XML document, the algorithm first parses the document and checks that it is well-formed. Then, it uses the XSV technique to validate the document against the schema.

The algorithm is designed to handle large XML documents efficiently. It uses a combination of techniques to achieve this, including caching and batch processing.

In conclusion, our algorithm is a powerful tool for validating XML documents. It is capable of handling large datasets efficiently and can be used in a variety of applications.
The program receives a string of XPath, downloads the template, and parses it with the XQuery processor. The XQuery processor then converts the XPath into XQuery and downloads the template. The XQuery processor then converts the template into an HTML document and sends it to the user.