RC23733 (W0510-004) October 3, 2005
Computer Science

IBM Research Report

Conflicting XML Updates

Mukund Raghavachari
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Oded Shmueli
Technion - Israel Institute of Technology

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

Conflicting XML Updates

Mukund Raghavachari
IBM T.J. Watson Research Center
raghavac@us.ibm.com

Oded Shmueli
Technion – Israel Institute of Technology
oshmu@cs.technion.ac.il

September 28, 2005

Abstract

The importance of XML as a universal data representation format has led to several efforts to integrate XML as a construct in a programming language. There has been growing interest in the addition of update operations in these languages, for example, to languages such as XQuery [20] and XJ [8]. These update operations (whether the semantics are mutating or value-based) support concise and declarative specification of transformations of XML data. The presence of update operations raises the question of detecting data dependencies between reads and updates of XML documents. In this paper, we formalize the notions of updates on XML data and conflicts between update operations. We show that conflict detection is NP-complete when the update operations are specified using XPath expressions that support the use of the child and descendant axis, wildcard symbols, and branching. We also provide polynomial time algorithms for update conflict detection when the patterns do not use branching.

1 Introduction

The rising importance of XML as a standard for data representation has led to several efforts aimed at reducing the burden of developing applications that process XML data. The strategy has been to incorporate XML as a first-class construct in a programming language, whether it be an imperative language, such as Java and C# (XJ [8], Xtatic [6], Cω [5]), a functional language (XDuce [11], CDuce [4]) or a domain-specific language (XQuery [20]). Despite evident differences in the design of these languages, all of them, in some sense, have a data model that represents XML as trees, and support deconstruction of XML data (using XPath expressions [19] or pattern matching [10]).

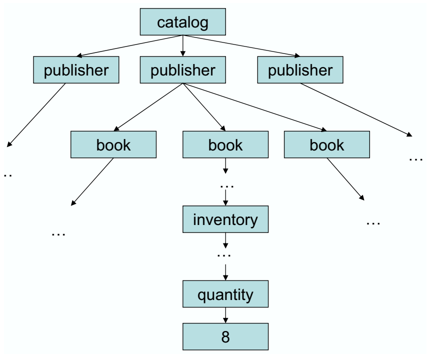


Figure 1: Tree representation t of an XML document.

In most languages that integrate XML as a first-class construct, programmers construct a new XML document t' from an existing XML document t by deconstructing t to determine the locations that are to be modified, and reconstructing the new t'. We argue that supporting *update* operations in these languages (with either mutating semantics or value-based semantics) is important, because it supports the writing of shorter, more declarative programs.

Consider, for example, the XML document t shown in Figure 1 (depicted as a tree). Assume we wish to add the XML subtree "`<restock/>`" to all nodes that satisfy the constraint C, where C is " all 'books' whose 'quantity' descendant contains a value less than 10." The constraint C is represented concisely by the XPath expression $//book[.//quantity < 10]$.[1] Consider the statement below, where the interpretation of an "insert" is equivalent to constructing a copy of t, evaluating the XPath expression on the copy, inserting a copy of "`<restock/>`" as a child of each node selected by the XPath expression, and returning the modified copy (the XPath expressions considered in this paper cannot observe order between siblings; assume the tree is inserted as an arbitrary child):

`insert` $\$t//book[.//quantity < 10]$`, <restock/>`

This insertion operation is more concise and declarative than the corresponding (recursive) program that would perform the same task in existing XML-based languages. In imperative languages such as XJ, such update statements on XML data are natural (and expected). XQuery, as well, is exploring the addition of update operations such as insertions, deletions, etc. [16, 21] (with imperative semantics — trees are mutated in-place).

When a language supports update operations on XML data, the question arises of how to detect conflicts between update operations. We consider the

[1]Here the "//" is the descendant operator, and the [...] delimit *predicates*, which are constraints that must be true of any subtree of the result. We consider a subset of predicates corresponding to conjunctions over paths (*branching*) in this paper. For an introduction to XPath and its semantics, see [18].

case of mutating semantics for update operations first, since the problem is more immediate in that case. A standard problem in most imperative languages is that of data dependence analysis, that is, determining when a read of some data structure may conflict with a write to that data structure. The ability to determine that reads do not conflict with writes allows compilers to use techniques such as code motion and common subexpression elimination to improve the performance of program execution.

Consider the following program fragment written in a pidgin language that supports imperative updates:

```
1 x = ...
2 y = read $x//A
3 insert $x/B, <C/>
4 z = read $x//C
```

The first assignment to `y` returns all `A` descendants of the tree referred to by `x`. A "read" operation returns references to nodes in `x` that are in the result of the evaluation of the XPath expression on `x`. The "insert" operation adds a `C` child to all nodes labeled `B` that are children of the root of the tree referred to by `x` (if there are no `B` children in the tree, then no nodes are added). `x` is mutated in place. Clearly, the read of Line 4 cannot be interchanged with the insertion of Line 3. If they were to be interchanged, then if `x` has a `B` child, the `read` $\$x//C$ would not "see" the `C` children added by the insertion operation. Suppose, however, the read operation of Line 4 were `z = read` $\$x//D$. Then, it could safely be interchanged with the insertion of Line 3. This could enable many optimizations. For example, in searching the tree referred to by `x` for `A` descendants in Line 2, a compiler could generate code to extract the `D` descendants as well.

If updates were to have non-mutating (functional) semantics, the ability to detect update conflicts can be beneficial as well. For example, consider the fragment below:

```
1 let x = ...  in
2    let y = read $x/*/A in
3       let z = insert $x/B, <C/> in
4          let u = read $z/*/A
```

We can interpret a read operation as evaluating the XPath expression on `x` and returning the set of trees consisting of the subtrees rooted at the nodes in the result of the evaluation. The $*$ symbol used in the XPath expression is a wildcard symbol — it matches any node in the tree. So, the read operation of Line 2 selects all nodes labeled `A` in document such that it is the grandchild of the root node of the document. The insertion operation is interpreted as constructing a new copy of `$x` that has a `C` node inserted into all `B` children of `$x`; the XPath expression of the insertion statement extracts all `B` children of the root of the document. Observe that `let u = read` $\$z/*/A$ will return the same result as `let y = read` $\$x/*/A$ — the insertion cannot add any nodes that affect the result of the XPath expressions. The knowledge that the insertion operation does not change the result of Line 4 would allow a compiler to replace Line 4 with the simpler assignment `let u = y`.

The subject of this paper is detecting when update operations on XML data conflict. We consider three operations — read, insertion, and deletion — the semantics of which are formalized in Section 3. A read R and an insert operation I conflict if the result of executing $R(I(t))$ is different from that of executing $R(t)$ for some XML tree t. We focus on a reference-based semantics, based on that proposed by the XQuery update standard [16, 21] and XJ [8]. We discuss alternate semantics as well and how results can be applied in a straightforward manner to these semantics. The operations we define use a restricted subset of XPath expressions — only the child and descendant axes will be allowed (along with wildcards and branching). We show that even for this simple subset, all conflict detection problems are NP-complete. We provide polynomial-time algorithms for the subset that does not allow the use of branching. The contributions of this paper are the following:

- A formalization of the update conflict problem for XML data. We present formalizations of two different reference-based semantics (*node conflicts* and *tree conflicts*) and a value-based semantics (*value conflicts*). We focus on node conflict semantics, but discuss how to modify results to apply them to the other semantics.
- We show that the read-insert conflict problem and the read-delete conflict problem are NP-complete for XPath expressions that use only the child and descendant axes, and have branching and wildcard symbols.
- We provide polynomial-time algorithms for the read-insert conflict problem and the read-delete conflict problem for XPath expressions that use only the child and descendant axes, and wildcard symbols (no branching).

Section 2 formalizes the abstractions we use for XML and XPath expressions. Section 3 defines the semantics of reads, insertions, and deletions and provides the formal statement of update conflict detection. Section 4 provides polynomial time algorithms for read-insert and read-delete detection, when the XPath expressions used do not contain branching. Section 5 demonstrates that the detection problem is NP-complete for the general XPath expressions we consider in the paper. In Section 6, we discuss extending our results to other domains (for example, schema-based conflict detection). In Section 7, we review related work, and we conclude in Section 8.

2 Preliminaries

We present mostly conventional abstractions for XML documents and XPath expressions (adapted from Miklau and Suciu [12]). In the next section, we define the syntax and semantics of *reads*, *insertions*, and *deletions* which will lead to the formal statement of the *conflict* problem.

2.1 XML Trees

An XML document is modeled as a labeled tree, where each node of a tree is labeled with a symbol from an infinite alphabet Σ. The set of all trees over Σ will be denoted T_Σ. Since the fragment of XPath expressions we consider in this paper does not depend on the order between children, the trees in T_Σ are unordered, and as is standard with XML, unranked (that is, the symbols in Σ do not specify an arity). The subset of Σ that is used as labels of nodes of a tree $t \in T_\Sigma$ will be denoted Σ_t.

For a tree $t \in T_\Sigma$, we will use $\textsc{nodes}_t$ and $\textsc{edges}_t$ to refer to the sets of nodes and edges of the tree, respectively; $\textsc{root}(t)$ will refer to the root node of the tree t, and for a node $n \in \textsc{nodes}_t$, $\textsc{label}_t(n)$ will refer to the label on n ($\textsc{label}_t(n) \in \Sigma$). The size of a tree, $|t|$, is the number of nodes in $\textsc{nodes}_t$. We assume relations $\textsc{child}(t) \subseteq \textsc{nodes}_t \times \textsc{nodes}_t$ and $\textsc{desc}(t) \subseteq \textsc{nodes}_t \times \textsc{nodes}_t$ that are defined in the obvious manner.

A *path* from n_1 to n_k in a tree is a sequence of edges (n_1, n_2), (n_2, n_3), $\ldots$, (n_{k-1}, n_k) such that $(n_i, n_{i+1}) \in \textsc{edges}_p, 1 \leq i \leq k-1$.

A subtree t' of a tree t rooted at a node $n \in \textsc{nodes}_t$, $\textsc{Subtree}_n(t)$, is defined as the tree where $\textsc{nodes}_{t'}$ is the set of nodes consisting of n and all descendants of n in t. The edges of t' are all edges $(u, v) \in \textsc{edges}_t$ such that u and v are both in $\textsc{nodes}_{t'}$.

2.2 Tree Patterns

Rather than working with XPath expressions directly, we use a more convenient formalism, *tree pattern* [12], that corresponds (roughly) to the XPath grammar below. The symbol $* \notin \Sigma$ denotes a wildcard label (it will match any label), and $\sigma \in \Sigma$:

$$e \rightarrow e/e \mid e//e \mid e[e] \mid e[.//e] \mid \sigma \mid *$$

A tree pattern p is a tree over the alphabet $\Sigma \cup \{*\}$. The set of edges of p is partitioned into two disjoint sets, $\textsc{edges}_/(p)$ and $\textsc{edges}_{//}(p)$, which represent *child constraints* and *descendant constraints*, respectively. We will depict descendant edges with double lines, and child edges with single lines in figures. Each pattern contains a distinguished node $\mathscr{O}(p) \in \textsc{nodes}_p$. We depict output nodes by using a thicker border for them than for other nodes. The size of a pattern, $|p|$, is defined as the number of nodes in $\textsc{nodes}_p$. The subset of Σ that is used as labels of nodes of a pattern p will be denoted Σ_p.

The set of all tree patterns will be denoted $P^{//,[],*}$. We will also be interested in the class of *linear patterns*, $P^{//,*}$, which is defined as the subset of $P^{//,[],*}$ where each node has a single outgoing edge, and the output node is the leaf node of the tree.

The translation of XPath expressions into tree patterns is straightforward, and is omitted. In Figure 2, the tree pattern p is derived from the XPath expression $a[.//c]/b[d][*//f]$.

The subpattern p' of a pattern p rooted at a node $n \in \textsc{nodes}_p$, $\textsc{Subpattern}_n(p)$, is defined as the subtree of p rooted at n. For our purposes, it will be sufficient

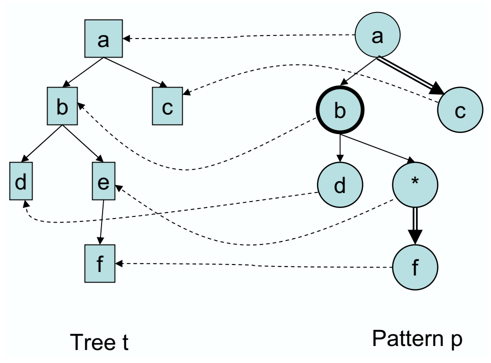


Figure 2: Example of an XML tree t, a tree pattern p, and an embedding from p into t. Double-lined edges in the figure depict edges in $\textsc{edges}_{//}(p)$. The node corresponding to $\mathscr{O}(p)$ is marked with a thick border.

to assume that an arbitrary node in p' is marked as the output node.

Given a pattern p and $n, n' \in \textsc{nodes}_p, \textsc{seq}_n^{n'}$ is the linear pattern p' derived from p, where $\textsc{nodes}_{p'} = \{\omega \in \textsc{nodes}_p | \omega$ is in the path from n to $n'\}$, and $\textsc{edges}_{p'}$ consists of the edges used in the path from n to n'.

A *chain* in a pattern p is a sequence of nodes $n_1, n_2, \ldots, n_k \in \textsc{nodes}_p$, such that $(n_i, n_{i+1}) \in \textsc{edges}_/(t), 1 \leq i < k$. We will also say that the sequence is a *chain from* n_1 *to* n_k. $\textsc{star-length}(p)$ of a pattern p is the number of nodes in the longest chain in p, where each node in the chain is labeled with $*$.

2.3 Embeddings

The semantics of the evaluation of an XPath expression is given in terms of embeddings [12] of a tree pattern p into a tree t. An *embedding* is a function $\mathcal{E} : \textsc{nodes}_p \rightarrow \textsc{nodes}_t$ such that all of the following conditions are satisfied:

- ($\textsc{root-preserving}$) $\mathcal{E}(\textsc{root}(p)) = \textsc{root}(t)$
- ($\textsc{label-preserving}$) $\forall n \in \textsc{nodes}_p, \textsc{label}_p(n) = * \vee \textsc{label}_p(n) = \textsc{label}_t(\mathcal{E}(n))$
- ($\textsc{edges}_/(p)$-$\textsc{satisfied}$) $\forall (u, v) \in \textsc{edges}_/(p), (\mathcal{E}(u), \mathcal{E}(v)) \in \textsc{child}(t)$
- ($\textsc{edges}_{//}(p)$-$\textsc{satisfied}$) $\forall (u, v) \in \textsc{edges}_{//}(p), (\mathcal{E}(u), \mathcal{E}(v)) \in \textsc{desc}(t)$

For example, Figure 2 depicts the embedding of a tree pattern p into a tree t.

Given the definition of embeddings, the *evaluation* of a tree pattern p on a tree t, $[\![p]\!](t)$, is defined as the subset of $\textsc{nodes}_t$ such that:

$$[\![p]\!](t) = \{\mathcal{E}(\mathscr{O}(p)) | \mathcal{E} \text{ is an embedding from } p \text{ into } t\}$$

We will sometimes use an alternative evaluation function, $[\![p]\!]_T(t)$, that returns a set of trees rather than a set of nodes:

$$[\![p]\!]_T(t) = \{t' \in T_\Sigma | t' = \textsc{Subtree}_n(t), n \in [\![p]\!](t)\}$$

The tree patterns in $P^{//,[],*}$ are always *satisfiable*, that is, there is always at least one tree $t \in T_\Sigma$ such that $[\![p]\!](t) \neq \emptyset$. For any pattern p, consider the tree W, where $\textsc{nodes}_W = \textsc{nodes}_p$ and $\textsc{edges}_W = \textsc{edges}_p$. If $\textsc{label}_p(n) \neq *$, $\textsc{label}_W(n) = \textsc{label}_p(n)$; otherwise, $\textsc{label}_W(n) = \sigma$, for some arbitrary $\sigma \in \Sigma$. It is straightforward to see that there is an embedding of p into W. We shall refer to W as a *model* for p, denoted $\mathscr{M}_p$. For example, the tree t in Figure 2 is a model for the tree pattern p in the figure.

3 Defining Conflicts

We present the semantics for read, insertion, and deletion operations and three different semantics for determining when conflicts occur. One semantics is value-based in that equality of operations is based on tree isomorphisms:

Definition 1 *Trees t, t' are* isomorphic*, denoted $t \cong t'$ if:*

- $\textsc{label}_t(\textsc{root}(t)) = \textsc{label}_{t'}(\textsc{root}(t'))$*, and*
- *Let C_t be the children of $\textsc{root}(t)$ in t, and let $C_{t'}$ be the children of $\textsc{root}(t')$ in t'. If C_t is empty, $C_{t'}$ is empty. Otherwise, there is a bijection $f : C_t \to C_{t'}$ such that for each $c_i \in C_t$, $\textsc{Subtree}_{c_i}(t) \cong \textsc{Subtree}_{f(c_i)}(t)$.*

A set of trees T is isomorphic to a set of trees T', $T \cong T'$, if there is a function f mapping trees in T to trees in T' such that for each tree $t \in T$, $t \cong f(t)$, and a function f' mapping trees in T' to trees in T such that for each tree $t' \in T'$, $t \cong f'(t)$.

The other semantics will be reference-based in that equality is based on node equality.

Definition 2 *Trees t, t' are* equivalent *if $\textsc{nodes}_t = \textsc{nodes}_{t'}$ and $\textsc{edges}_t = \textsc{edges}_{t'}$. Equivalence of sets of trees is based on this notion of equivalence.*

We now define the operations supported on trees:

- $\textsc{read}_p(t)$ where $p \in P^{//,[],*}$ and $t \in T_\Sigma$ *projects* a set of nodes from a tree. It is defined as $[\![p]\!](t)$.
- $\textsc{insert}_{p,X}(t)$, where $p \in P^{//,[],*}$ and $t, X \in T_\Sigma$: The insertion operation evaluates p on t and inserts a fresh copy of X as a subtree of each node in the result of the evaluation.

  Let $\mathcal{R} = [\![p]\!](t)$. Let $X_1, X_2, \ldots, X_{|\mathcal{R}|}$ be a set of trees such that $X_i \cong X$ and $\textsc{nodes}_{X_i} \cap \textsc{nodes}_{X_j} = \emptyset, 1 \leq i, j \leq |\mathcal{R}|$. Furthermore, the set of nodes of each X_i is disjoint from $\textsc{nodes}_t$. For each $n_i \in \mathcal{R}, 1 \leq i \leq |\mathcal{R}|$ add X_i as a child of n_i. In other words, construct a tree t', such that $\textsc{nodes}_{t'} = \textsc{nodes}_t \cup \bigcup_{i=1}^{|\mathcal{R}|} \textsc{nodes}_{X_i}$ and $\textsc{edges}_{t'} = \textsc{edges}_t \cup \bigcup_{i=1}^{|\mathcal{R}|} (\textsc{edges}_{X_i} \cup \{(n_i, \textsc{root}(X_i))\})$.

We will refer to the nodes in $\mathcal{R}$ as *insertion points*. Observe that if the result of evaluation of p on t is the empty set, t is unchanged.

- $\textsc{delete}_p(t)$, where $p \in P^{//,[],*}, t \in T_\Sigma$: The delete operation evaluates p on t and deletes the subtree rooted at any node in the result of the evaluation. Let $\mathcal{R} = [\![p]\!](t)$. Let $\mathcal{D}$ be the set consisting of n and all descendants of n in t. The result of the delete operation is a tree t', where $\textsc{nodes}_{t'} = \textsc{nodes}_t - \mathcal{D}$, and $\textsc{edges}_{t'}$ consists of the edges (u, v) in $\textsc{edges}_t$ where both u and v are in $\textsc{nodes}_{t'}$.

  We will refer to the nodes in $\mathcal{R}$ as *deletion points*. We require that $\mathscr{O}(p) \neq \textsc{root}(p)$, which ensures that the result of the deletion is a tree.

For notational convenience, we will often conflate the tree pattern associated with an operation with the tree pattern itself. For example, $\mathscr{O}(R)$ will stand for $\mathscr{O}(p)$ in the read operation $R = \textsc{read}_p(t)$.

The read, insertion, and deletion operations can be executed on a tree t in time polynomial in the size of their inputs (that is, $|t|$,$|p|$, and $|X|$). The patterns we consider are a subset of *Core XPath*, which can be evaluated in time linear in the size of the tree and the pattern [7]. Given the result of the evaluation of a pattern, the insertion and deletion operations can be executed easily in time linear in the size of t in standard tree representations. We now define what it means for two operations to *conflict*. We first provide reference-based semantics of conflicts.

Definition 3 (read-insert conflict) *A read $R = \textsc{read}_p(t)$ has a* node conflict *with an insertion $I = \textsc{insert}_{p',X}(t)$ if there exists $t \in T_\Sigma$, $R(I(t)) \neq R(t)$. If such a t exists, we call t a* witness *to the conflict.*

A read $R = \textsc{read}_p(t)$ has a tree conflict *with an insertion $I = \textsc{insert}_{p',X}(t)$ if there exists $t \in T_\Sigma$, $[\![p]\!]_T(I(t)) \neq [\![p]\!]_T(t)$.*

Intuitively, the difference between node and tree conflicts is that the node conflict definition only verifies that the nodes returned by $R(t)$ and $R(I(t))$ are the same. The tree conflict definition verifies that no node conflict exists *and* none of the trees rooted at a node in $R(I(t))$ contain a modified subtree. For example, consider the read operation R that returns the root node of a tree, and an insertion operation I that adds a subtree X to child labeled `B` of the root node. According to the node conflict definition, the two operations do not conflict — $R(t)$ and $R(I(t))$ both return the root node of the document. The tree conflict definition, however, would signal a conflict since the subtree of $I(t)$ rooted at $\textsc{root}(I(t))$ is not the same as the subtree of t rooted at $\textsc{root}(t)$.

Both definitions are useful in practice. Suppose one had a program of the form `I;...; R` (where I is executed before R), and I and R have a node conflict.

A compiler could choose to perform the read R before the insert I as long it ensures that any operation that depends on the result of R, and that observes the modification made to the B child by I, executes after I. This could lead to optimizations such as common subexpression elimination. The tree semantics of conflict is useful to determine when R and *any* operation that depends solely on the results of R could be executed before I. We now present a parallel definition for read-delete conflicts.

Definition 4 (read-delete conflict) *A read* $R = \text{READ}_p(t)$ *has a* node conflict *with a deletion* $D = \text{DELETE}_{p'}(t)$ *if there exists* $t \in T_\Sigma$*,* $R(D(t)) \neq R(t)$*.*

A read $R = \text{READ}_p(t)$ *has a* tree conflict *with a deletion* $D = \text{DELETE}_{p'}(t)$ *if there exists* $t \in T_\Sigma$*,* $[\![p]\!]_T(D(t)) \neq [\![p]\!]_T(t)$*.*

Evidently, other kinds of conflicts can arise, for example, delete-insert conflicts. In this paper, we mostly focus on read-insert and read-delete conflicts and defer discussion of other update conflicts to Section 6. We now present value-based semantics of read-insert and read-delete conflicts. They are similar to those of tree conflicts in the reference-based semantics, except that tree isomorphism rather than equality is used.

Definition 5 *A read* $R = \text{READ}_p(t)$ *has a* value conflict *with an insertion* $I = \text{INSERT}_{p',X}(t)$ *if there exists* $t \in T_\Sigma$*,* $[\![p]\!]_T(I(t)) \ncong [\![p]\!]_T(t)$*.*

Definition 6 *A read* $R = \text{READ}_p(t)$ *has a* value conflict *with a deletion* $D = \text{DELETE}_{p'}(t)$ *if there exists* $t \in T_\Sigma$*,* $[\![p]\!]_T(D(t)) \ncong [\![p]\!]_T(t)$*.*

Consider the example of Figure 3. The figure depicts a delete operation on a read that deletes all children of the root labeled δ. The read operation returns all descendants of the root node that are labeled γ. Consider the node n which is deleted by D. According to the reference-based semantics, a node conflict would occur between the read and the delete operation because the node n is not present in $R(D(W))$ but is present in $R(W)$. According to the value-based semantics, however, there is no conflict. The subtree rooted at the node n' is present in both $R(W)$ and $R(D(W))$ and is isomorphic to the subtree rooted at n that is deleted.

Given a tree t, whether t is a witness to a read-insert or read-delete conflict can be decided in polynomial time for all three semantics of conflicts.

Lemma 1 *Given a tree* $t \in T_\Sigma$*, a read* R *and an insertion* I*, it can be determined in polynomial time whether* t *is a witness to a node conflict, a tree conflict, or a value conflict between* R *and* I*.*

Given a tree $t \in T_\Sigma$*, a read* R *and a deletion* D*, it can be determined in polynomial time whether* t *is a witness to a node conflict, a tree conflict, or a value conflict between* R *and* D*.*

Proof: The case for node conflict is trivial since it involves evaluating $R(t)$ and $R(I(t))$ (which can be performed in polynomial time, as stated previously) and verifying that the resulting sets are identical.

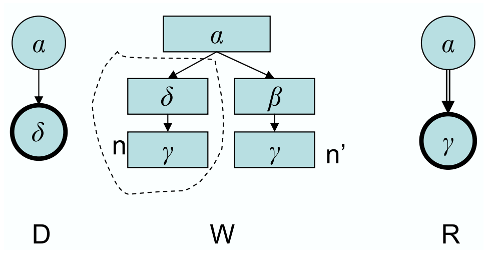


Figure 3: Example of a delete operation on a tree that raises a conflict with reference-based semantics, but not with value-based semantics.

For tree conflicts, one can associate with each node in t a flag marking whether the subtree under it has been modified. In an appropriate tree representation, an insertion or deletion operation can update this information in time linear in the size of t. Checking for a conflict requires verifying set equality of the results of $R(t)$ and $R(I(t))$ and ensuring that none of the nodes in $R(I(t))$ have been marked.

For value conflicts, observe that tree isomorphisms can be decided in time linear in the size of the trees. A slight modification to the algorithm in Aho *et al.* [1] supports labeled tree isomorphism detection. Since the sizes of $R(t)$ and $R(I(t))$ are bounded by $|t|$, verifying that the sets of trees are isomorphic to each other can be performed in polynomial time.

The proof for read-delete conflicts is similar. ■

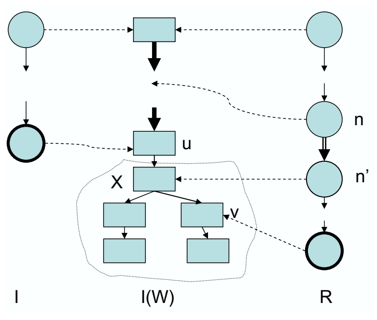
Consider the set of patterns in $P^{//,*}$. For linear patterns, we show that the reference-based semantics are equivalent to the value-based semantics.

Lemma 2 *Given a read* $R = \text{READ}_p$ *and an insertion* $I = \text{INSERT}_{p',X}$*, where* $p, p' \in P^{//,*}$*, there is a tree conflict between* R *and* I *if and only if there is a value conflict between* R *and* I*.*

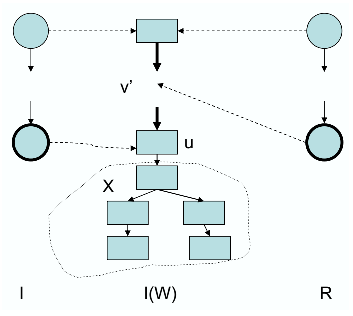
Proof: The "If" case is trivial — if there is a value conflict, certainly there is a tree in $[\![p]\!]_T(I(t))$ that is not equivalent to one in $[\![p]\!]_T(t)$.

Suppose there is a tree conflict in the reference-based semantics and not in the value-based semantics. There are two cases to consider. Either there is also a node conflict between I and R, or a subtree of one of trees in $[\![p]\!]_T(I(t))$ is a modified tree. In the first case, consider the structure of a node insertion conflict as shown in Figure 4a. Since there is a node conflict, there is a tree W such that there is a node v in $[\![p]\!](I(W))$ that is not there in $[\![p]\!](W)$. This node v must have been one of the nodes in a copy of X that was inserted into t by I. If not, any embedding of R into $I(W)$ would also be an embedding of R into W.

Consider the tree W' defined by the path in W from $\text{ROOT}(W)$ to u (shown in thick lines in Figure 4a). Observe that W' is also a witness to a node conflict between I and R. We construct W'' from W' by adding for each node in $R(W')$,



(a) Node Conflict



(b) Tree Conflict

Figure 4: Structure of a read-insert conflict. (a) Node conflict, and (b) Tree conflict.

a child node labeled α, where α is a label not used in X. Again, note that W'' is a witness to a node conflict between R and I (since it does not contain v). Observe that all subtrees rooted at a node in $R(W'')$ would have a child labeled α. W'' will be a witness to a read-insert value conflict because the subtree rooted at v in $I(W'')$ will not be isomorphic to any tree in $[\![p]\!]_T(W'')$. As a result, if there is a node conflict in the reference based semantics, there is a read-insert value conflict.

In the other case, there is a tree conflict in the reference-based semantics because there is a node v' in $R(I(W))$ and $R(W)$ whose subtree has been modified by the insert operation. Consider the structure of such a conflict, shown in Figure 4b. The node v' must be the ancestor of an insertion point u. Consider the tree W' derived from W by adding a node labeled α as a child of v', such that α is not used in W or X. W' is still a witness to a tree conflict between R and I. Moreover, it is a witness to a conflict in the value-based semantics as well. Since v' is the only node in both W' and $I(W')$ with a child labeled α, and the subtree rooted at v' is modified by the insertion, there can be no tree in $[\![p]\!]_T(W')$ that is isomorphic to the subtree of $I(W')$ rooted at v'.

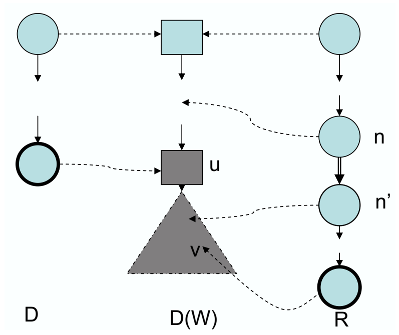


Figure 5: Structure of a read-delete node conflict.

Therefore, the existence of a tree conflict between a read and an insert in the reference-based semantics implies that there is a conflict in the value-based semantics when the patterns used are in $P^{//,*}$. ■

A similar proof technique can be used to show that for linear patterns, tree conflicts and value conflicts are equivalent in the read-delete case.

In this paper, based on the proposed semantics for XQuery and XJ, we focus mainly on *node conflicts* with reference-based semantics (all future references to "conflict" should be interpreted as such, unless stated otherwise explicitly). All results can be extended to other kinds of conflicts as well, and where appropriate, we will discuss the modifications necessary.

4 ptime Algorithms for $P^{//,*}$

We provide polynomial-time algorithms for detecting read-delete and read-insert conflicts when the patterns are linear. What is perhaps surprising is that only the read pattern need be linear — the pattern for the insert and delete can be any pattern in $P^{//,[],*}$. This is surprising because as we show in the next section, when both patterns are in $P^{//,[],*}$, read-insert and read-delete conflict detection is NP-complete.

4.1 Read-Delete Node Conflicts

We start with examining the read-delete case, since it is more straightforward than the read-insert case. Consider an example of a conflict when the deletion and the read are linear patterns. Figure 5 shows the structure that any conflict must have. The figure depicts a deletion D and a read R and the tree that is the result of applying D to a tree W (the shaded subtree rooted at u is deleted from W).

The existence of a node conflict implies that there is an embedding of R into W that maps $\mathcal{O}(R)$ to some node v that is deleted in $D(W)$. There is a node

u in $\textsc{nodes}_W$ that is either an ancestor of v or v itself (there must be at least one for v to have been deleted), where u is a node in $[\![D]\!](W)$.

Consider the edge (n, n') in Figure 5. The nodes in the path from $\textsc{root}(R)$ to n are mapped to nodes in W in the path from $\textsc{root}(W)$ to node u. The nodes in D from $\textsc{root}(D)$ to $\mathscr{O}(D)$ can be embedded into nodes in W in the path from $\textsc{root}(W)$ to u as well (since $u \in [\![D]\!](W)$). Since portions of both R and D are mapped to the nodes in the path $\textsc{root}(W)$ to u, the nodes in R from $\textsc{root}(R)$ to the node n must "match" the nodes in D from $\textsc{root}(D)$ to $\mathscr{O}(D)$. In other words, the sequence of nodes in W from $\textsc{root}(W)$ to u supports embeddings $\mathcal{E}_1$ from D, and $\mathcal{E}_2$ from $\textsc{seq}^n_{\textsc{root}(R)}$. We formalize this notion of matching and show how we can use it to detect read-delete node conflicts.

Definition 7 *Linear patterns l and l'* match weakly *if there exists a tree $t \in T_\Sigma$ such that there is an embedding $\mathcal{E}_1$ from l into t and an embedding $\mathcal{E}_2$ from l' into t, and $\mathcal{E}_1(\mathscr{O}(l)) = \mathcal{E}_2(\mathscr{O}(l'))$ or $\mathcal{E}_1(\mathscr{O}(l))$ is a descendant of $\mathcal{E}_2(\mathscr{O}(l'))$.*

Two linear patterns l and l' match strongly *if they match weakly and $\mathcal{E}_1(\mathscr{O}(l)) = \mathcal{E}_2(\mathscr{O}(l'))$.*

Lemma 3 *Consider a read $R = \textsc{read}_p$ and a deletion $D = \textsc{delete}_{p'}$. There is a read-delete node conflict between R and D if and only if there exists an edge (n, n') in $\textsc{edges}_R$ such that:*

- *$(n, n') \in \textsc{edges}_{//}(p)$, D and $\textsc{seq}^n_{\textsc{root}(R)}$ match weakly, or*
- *$(n, n') \in \textsc{edges}_{/}(p)$, D and $\textsc{seq}^{n'}_{\textsc{root}(R)}$ match strongly.*

Proof: (Only if) If there is a read-delete conflict between R and D, there must a witness tree W such that $R(W) \neq R(D(W))$. Consider a node $v \in [\![R]\!](W)$ such that $v \notin [\![R]\!](D(W))$. Let $\mathcal{E}$ be any embedding of R into W that maps $\mathscr{O}(R)$ to v. We assert that $v \notin \textsc{nodes}_{D(W)}$. If $v \in \textsc{nodes}_{D(W)}$, it implies that all ancestors of v in W are present in $\textsc{nodes}_{D(W)}$. $\mathcal{E}$ uses only ancestors of v in W (because, by the definition of linear patterns, $\mathscr{O}(R)$ is the leaf node of R), and therefore, $\mathcal{E}$ is a valid embedding of R into $D(W)$. This contradicts the assumption that $v \notin [\![R]\!](D(W))$. Therefore, v must be a deleted node.

Since v is a deleted node, let (n, n') be the edge in R such that $\mathcal{E}$ maps n to a node that exists in $\textsc{nodes}_{D(W)}$ and n' to a node that has been deleted in $D(W)$. Since $\mathcal{E}$ maps $\textsc{root}(R)$ to $\textsc{root}(W)$ which exists in $D(W)$ and $\mathscr{O}(R)$ to v, which is deleted in $D(W)$, there must be such an edge. Let u be the ancestor of v in W such that $u \notin \textsc{nodes}_{D(W)}$ and all ancestors u' of u are in $\textsc{nodes}_{D(W)}$. u must be a deletion point. Therefore, there must be an embedding from D to the sequence of nodes in the path from $\textsc{root}(W)$ to u. Furthermore, since $\mathcal{E}(n) \in \textsc{nodes}_{D(W)}$, the sequence of nodes in the path from $\textsc{root}(R)$ to n must be mapped to the sequence of nodes in the path from $\textsc{root}(W)$ to u. As a result, D and $\textsc{seq}^n_{\textsc{root}(R)}$ match weakly. If (n, n') is a child edge, $\mathcal{E}(n')$ must map n' to u. Since n is mapped to a node in $\textsc{nodes}_{D(W)}$ and u is the only node that is an ancestor of v that has a parent node in $\textsc{nodes}_{D(W)}$, n' must be mapped to u. As a result, we can conclude that in this case, D and $\textsc{seq}^{n'}_{\textsc{root}(R)}$ match strongly.

(If) The fact that D and $R' = \textsc{seq}^{n}_{\textsc{root}(R)}$ match (weakly or strongly) implies that there is a tree W such that D and R' can be embedded into W. Let $\mathcal{E}$ be an embedding of D into W, and let $u = \mathcal{E}(\mathscr{O}(D))$. Let $R'' = \textsc{seq}^{\mathscr{O}(R)}_{n'}$.

Suppose (n, n') is a descendant edge in $\textsc{edges}_R$. We can construct a witness W' to a read-delete conflict by inserting $\mathscr{M}_{R''}$ as a child of u in W. Let $\mathcal{E}_1$ be an embedding of R' into W and let $\mathcal{E}_2$ be an embedding of R'' into $\mathscr{M}_{R''}$. We can compose the two embeddings to derive an embedding for R into W'. Since by the definition of weak matching, $\mathcal{E}_1$ maps n to an ancestor of u or u itself, and $\mathscr{M}_{R''}$ is inserted as a descendant of u, the edge constraint between n and n' would be satisfied. Since R can be embedded in W' and returns a result in the subtree rooted at u (which will be deleted by D), $R(W') \neq D(R(W'))$.

Suppose (n, n') is a child edge in $\textsc{edges}_R$. If $n' = \mathscr{O}(R)$, by the definition of strong matching, there is an embedding of R into W that maps n' to u. Since n' is the output node of R, u will be in $R(W)$, but will be deleted in $D(R(W))$. If n' is not the output node of R, let n'' be the child of n' in R. We construct W' from W by adding a model for the subpattern rooted at n'' as a child of u. It is straightforward to show that we can construct an embedding from R into W' that selects a node in the subtree rooted at u. This subtree would be deleted by D, and therefore, $R(W') \neq D(R(W'))$. ■

Lemma 3 suggests a mechanism for detecting read-delete conflicts — find an edge $(n, n') \in \textsc{edges}_R$ that matches D strongly or weakly as appropriate. We sketch an algorithm for determining whether two patterns match.

Given linear patterns l and l', we construct regular expressions from the patterns and use language intersection to check whether l and l' match strongly or weakly. First, some technicalities. Since Σ is infinite, we assert that we can restrict the alphabet to the symbols used in l and l', that is $\Sigma_l \cup \Sigma_{l'}$. Let $\Sigma_{l,l'} = \Sigma_l \cup \Sigma_{l'}$. If there is a witness tree W to a matching that uses symbols other than those in $\Sigma_{l,l'}$, observe that we can replace those symbols with ones from $\Sigma_{l,l'}$; only nodes labeled $*$ in l and l' could have mapped to them. Secondly, note that the size of $\Sigma_{l,l'}$ depends solely on the sizes of l and l'.

We now describe the construction of regular expressions from linear patterns. Let $(.)$ be stand for any symbol in $\Sigma_{l,l'}$, that is, it is equivalent to $\sigma_1|\sigma_2|\ldots$ for each $\sigma_i \in \Sigma_{l,l'}$. For a node n in a pattern l, let $sym(n)$ be defined as $\textsc{label}_l(n)$ if $\textsc{label}_l(n) \neq *$, and $(.)$, otherwise.

We define a function $\mathscr{R} : \textsc{nodes} \rightarrow regexp$ as follows (for a pattern l):

- $\mathscr{R}(\textsc{root}(l)) = sym(n)$,
- $\mathscr{R}(n \neq \textsc{root}(l))$: Let n' be the parent of n in l. If (n', n) is a descendant edge, $\mathscr{R}(n) = \mathscr{R}(n') \cdot (.)^* \cdot sym(n)$. If (n', n) is a child edge, $\mathscr{R}(n) = \mathscr{R}(n') \cdot sym(n)$.

Let $r_1 = \mathscr{R}(\mathscr{O}(l))$ and $r_2 = \mathscr{R}(\mathscr{O}(l'))$. We state (the proof is omitted for space) that the linear patterns l and l' match strongly if and only if $L(r_1) \cap L(r_2) \neq \emptyset$, where $L(r_1)$ and $L(r_2)$ are the languages denoted by r_1 and r_2, respectively. l and l' match weakly if and only if $L(r_1) \cap L(r_2 \cdot (.)^*) \neq \emptyset$. As is customary, we can construct non-deterministic finite state automata from the regular expressions, and verify in time polynomial in the size of l and l' whether the intersection is non-empty.

Since we can detect whether a linear pattern matches one another (weakly or strongly) in polynomial time, the following theorem is immediate — for each edge (n, n') in $\textsc{edges}_R$ in a read R, we can verify whether a deletion D matches $\textsc{seq}^{n}_{\textsc{root}(R)}$ or $\textsc{seq}^{n'}_{\textsc{root}(R)}$ (as appropriate).

Theorem 1 *For a read $R = \textsc{read}_p$ and a deletion $D = \textsc{delete}_{p'}$, where $p, p' \in P^{//,*}$, a read-delete node conflict can be detected in polynomial time.*

REMARKS: In practice, rather than verifying whether each edge in R matches D separately, one can use an algorithm based on dynamic programming to determine whether a match exists. With respect to alternate semantics of updates, observe that a tree conflict occurs if and only if either there is a node conflict or D is weakly matched by R. Recall that for linear patterns, the value-based and reference-based semantics (for tree conflicts) are equivalent. Therefore, the theorem above applies to all discussed kinds of conflicts.

We show now that the deletion operation need not be linear. As long as the read operation is in $P^{//,*}$, we can detect read-delete conflicts in polynomial time.

Lemma 4 *A read $R = \textsc{read}_p$ and a deletion $D = \textsc{delete}_{p'}$, where p is a linear pattern and $p' \in P^{//,[],*}$, have a node conflict if and only if R and $D' = \textsc{seq}^{\mathscr{O}(D)}_{\textsc{root}(D)}$ have a node conflict. Note D is not necessarily a linear pattern.*

Proof: If R and D have a conflict, there is a tree W that is a witness to the conflict. W is also a witness to a read-delete conflict between D' and R. $[\![D]\!](W) \subseteq [\![D']\!](W)$, because any embedding of D into W defines an embedding of D' into W. Since the set of deletion points in the evaluation of D' on W contains all the deletion points of that of D on W, any node whose deletion is necessary to show the conflict in $R(D(W))$ will also be deleted in $R(D'(W))$.

(If) If D' and R have a conflict, let W be a witness to the conflict. We construct a tree W' from W as follows. Consider any node $n \in \textsc{nodes}_{D'}$. Let c be a child of n in D that is not in D', and let $\mathscr{M}_c$ be a model for $\textsc{Subpattern}_c(D)$. Add $\mathscr{M}_c$ as a child of each node in W. This will ensure that any embedding of D' into W can be extended easily into an embedding of D into W'.

Given an embedding $\mathcal{E}$ of D' into W, let n be a node in $\textsc{nodes}_{D'}$ and let $c \notin \textsc{nodes}_{D'}$ be a child of n in D. The extension of $\mathcal{E}$ maps the subpattern of D rooted at c to the child of $\mathcal{E}(n)$ in W' that corresponds to $\mathscr{M}_c$. Since any embedding of D' into W can be extended into one of D into W', $[\![D']\!](W) \subseteq [\![D]\!](W')$. Moreover, since W' only adds nodes to W, $[\![R]\!](W) \subseteq [\![R]\!](W')$. Let v be a node in $R(W)$ that is deleted in $R(D'(W))$. v will be a node in $R(W')$ and the deletion point that causes v to be deleted will be present in $[\![D]\!](W')$. Therefore, there is a read-delete conflict between R and D. ■

Corollary 1 *For a read $R = \textsc{read}_p$ and a deletion $D = \textsc{delete}_{p'} X$, where $p \in P^{//,*}$ and $p' \in P^{//,[],*}$, a read-delete node conflict can be detected in polynomial time.*

4.2 Read-Insert Node Conflicts

Consider an example of a node conflict when the insert I and the read R are linear patterns. Figure 4a shows the structure that any such node conflict must have. The figure depicts I and R and the tree that is the result of applying I to a tree W (a subtree X is inserted as a child of a node u).

While the structure is similar to the read-delete case, the read-insert case is somewhat more complicated. Unlike the read-delete case, where the subtree rooted at u can be any tree, in the read-insert case, the nodes in R from n' to $\mathscr{O}(R)$ should be mappable to X for a conflict to occur. The existence of a conflict implies that there is an embedding that maps $\mathscr{O}(R)$ to some node v in X and maps the nodes in R in the path from $\textsc{root}(R)$ to $\mathscr{O}(R)$ to nodes in the path from $\textsc{root}(W)$ to v in $I(W)$. Since v is in X, it must be the descendant of an insertion point $u \in \textsc{nodes}_W$ (the insertion point where X is inserted). For u to have been selected as an insertion point, there must have been an embedding of I to the nodes in the path from $\textsc{root}(W)$ to u.

Consider the edge (n, n') in Figure 4a. It is the edge in R in the path from $\textsc{root}(R)$ to $\mathscr{O}(R)$ that straddles W and X in the sense that n is mapped to a node in W and n' is mapped to a node in X. In any witness to a conflict, there must always be such an edge because $\textsc{root}(R)$ is always mapped to $\textsc{root}(W)$ and $\mathscr{O}(R)$ is mapped to v which is a node in X.

Definition 8 *Given a read R and an insert I, let (n, n') be an edge in $\textsc{edges}_R$. (n, n') is the* cut edge *for R and I if there exists a tree $W \in T_\Sigma$ and an embedding $\mathcal{E}$ of R into $I(W)$ such that $\mathcal{E}(n) \in \textsc{nodes}_W$ and $\mathcal{E}(n') \notin \textsc{nodes}_W$.*

Lemma 5 *There is a read-insert conflict between R and I if and only if there is a cut edge for R and I.*

Proof: If there is a read-insert conflict, let $W \in T_\Sigma$ be a tree such that $R(I(W)) \neq R(W)$. Any node v in $R(I(W))$ that is not in $R(W)$ must not be in $\textsc{nodes}_W$. Otherwise, suppose v is in $\textsc{nodes}_W$. In a linear pattern, the output node is the leaf node of the pattern; all other nodes in the pattern are mapped to ancestors of v. Therefore, the embedding of R into $I(W)$ also defines an embedding of R into W, and $v \in R(W)$, contradicting our earlier assumption. Let $\mathcal{E}$ be an embedding of R in $I(W)$ that maps $\mathscr{O}(R)$ to v. Since all embeddings map $\textsc{root}(R)$ to $\textsc{root}(W)$, which is in $\textsc{nodes}_W$, there must be some edge (n, n') in $\textsc{edges}_R$ such that $\mathcal{E}(n) \in \textsc{nodes}_W$ and $\mathcal{E}(n') \notin \textsc{nodes}_W$. (n, n') is a cut edge for R and I.

If there is a cut edge for R and I, by definition, there is a witness tree W and embedding $\mathcal{E}$ of R into $I(W)$ such that $\mathcal{E}(n) \in \textsc{nodes}_W$ and $\mathcal{E}(n') \notin \textsc{nodes}_W$. Since $\mathscr{O}(R)$ is the leaf node of R, it must either be n' or be mapped to a descendant of $\mathcal{E}(n')$. In either case, $\mathcal{E}(\mathscr{O}(R)) \notin \textsc{nodes}_W$. Therefore, $R(I(W))$

contains at least one node not in $R(W)$ implying that a read-insert conflict exists. ■

Since any embedding of R into $I(W)$ that causes a conflict must have a cut edge (n, n'), we can use a similar strategy as for the read-delete case to detect conflicts:

- Choose an edge (n, n') in EDGES_R.
- Construct a witness tree W such that (n, n') is a cut edge for an embedding of R into $I(W)$.
- If no such tree can be constructed, choose another edge until all edges have been tried.
- If no edge can be found, R and I do not have a read-insert conflict.

From Figure 4a, it should be clear that for an edge $(n, n') \in \text{EDGES}_R$ to be a cut edge, it must satisfy two constraints. The insert I must match $\text{SEQ}^n_{\text{ROOT}(R)}$, and there should be an embedding of $\text{SEQ}^{\mathscr{O}(R)}_{\text{ROOT}(n')}$ into X. We formalize these requirements below.

Lemma 6 *Consider a read $R = \text{READ}_p$ and an insert $I = \text{INSERT}_{p',X}$. Let (n, n') be an edge in EDGES_p. (n, n') is a cut edge for R and I if and only if:*

- *If $(n, n') \in \text{EDGES}_/(p)$, I and $\text{SEQ}^n_{\text{ROOT}(R)}$ match strongly. If $(n, n') \in \text{EDGES}_{//}(p)$, I and $\text{SEQ}^n_{\text{ROOT}(R)}$ match weakly, and*
- *If $(n, n') \in \text{EDGES}_/(p)$, there is an embedding from $\text{SEQ}^{\mathscr{O}(R)}_{n'}$ to X. If $(n, n') \in \text{EDGES}_{//}(p)$, there is an embedding from $\text{SEQ}^{\mathscr{O}(R)}_{n'}$ to X or some subtree of X.*

Proof: (Only if) If (n, n') is a cut edge for R and I, then there must be a witness tree W and an embedding $\mathcal{E}$ of R into $I(W)$ such that $\mathcal{E}(n) \in \text{NODES}_W$ and $\mathcal{E}(n') \notin \text{NODES}_W$. Let $v = \mathcal{E}(n')$. Consider the nearest ancestor of v in $I(W)$, u, that is in NODES_W. u must be an insertion point. As a result, there must be an embedding of I into W using only the nodes from $\text{ROOT}(W)$ to u. Moreover, $\mathcal{E}(n)$ must be mapped to an ancestor of v that is in NODES_W. Therefore, it must be mapped to either u or to an ancestor of u. The restriction of $\mathcal{E}$ to the nodes in the path from $\text{ROOT}(R)$ to n shows that there is an embedding from $\text{SEQ}^n_{\text{ROOT}(R)}$ into W using only the nodes from $\text{ROOT}(W)$ to u. Therefore, one can conclude that $\text{SEQ}^n_{\text{ROOT}(R)}$ and I match weakly (using W as a witness). If (n, n') is a child edge, $\mathcal{E}$ must map n to u because it is the only ancestor of v that has a child not in NODES_W, which implies that $\text{SEQ}^n_{\text{ROOT}(R)}$ and I match strongly.

If (n, n') is a cut edge, the nodes in R in the path from n' to $\mathscr{O}(R)$ must be mapped to nodes in the clone of X inserted at u. Therefore, there must be an embedding from $\text{SEQ}^{\mathscr{O}(R)}_{n'}$ to X or some subtree of X. If (n, n') is a child edge,

n' must be mapped to the root of the subtree corresponding to X because it is the only node that has a parent that is in NODES_W.

(If) Let (n, n') be an edge in EDGES_R. The fact that $R' = \text{SEQ}^{n}_{\text{ROOT}(R)}$ and I match (strongly or weakly), implies that there is a tree W and embeddings $\mathcal{E}_I$ from I into W and $\mathcal{E}_1$ from R' into W such that $\mathcal{E}_1(\mathscr{O}(R'))$ is mapped to an ancestor of or to the same node as $\mathcal{E}_I(\mathscr{O}(I))$. Moreover, there is an embedding $\mathcal{E}_2$ of $R'' = \text{SEQ}^{\mathscr{O}(R)}_{n'}$ into X. Consider the embedding $\mathcal{E}_R$ of R into $I(W)$ defined as $\mathcal{E}_1(w)$ if w is n or an ancestor of n, and $\mathcal{E}_2(w)$, if w is n' or a descendant of n'. If (n, n') is a descendant edge, clearly $\mathcal{E}_R$ satisfies the edge. Since $\mathcal{E}_1$ maps n to $\mathscr{O}(I)$ or to an ancestor of $\mathscr{O}(I)$, and $\mathcal{E}_2$ maps n' to a node not in NODES_W, $\mathcal{E}_R(n')$ must be a descendant of $\mathcal{E}_R(n)$. If (n, n') is a child edge, I and R' match strongly. Therefore, $\mathscr{O}(R')$ must be mapped to $\mathscr{O}(I)$. Moreover, $\text{ROOT}(R'')$, which is n', must be mapped to the root of X. Therefore, $\mathcal{E}_R$ satisfies the edge constraint between n and n'. Clearly, $\mathcal{E}_R$ is an embedding. Since $\mathcal{E}_R$ maps n to a node in NODES_W and n' to a node not in NODES_W, (n, n') is a cut edge. ■

Using the mechanisms described in the previous section for determining whether an edge matches another (weakly or strongly), we can identify cut edges in polynomial time.

Lemma 7 *Given a read R and and insert I, we can verify in polynomial time whether an edge (n, n') in EDGES_R is a cut edge for R and I.*

Proof: Given R and (n, n'), we can verify whether $\text{SEQ}^{n}_{\text{ROOT}(R)}$ and I match weakly or strongly as appropriate in polynomial time. We can also verify whether there is an embedding of $\text{SEQ}^{\mathscr{O}(R)}_{n'}$ into X or a subtree of X as appropriate. By Lemma 6, these facts are sufficient to determine whether (n, n') is a cut edge for R and I. ■

As a result, we can conclude that read-insert conflicts can be detected in polynomial time.

Theorem 2 *For a read R and an insert I, a read-insert node conflict can be detected in polynomial time if R and I use patterns in $P^{//,*}$.*

REMARKS: As in the read-delete case, R and I have a tree conflict if and only if I and R have a node conflict or I and R match weakly. We can conclude, therefore, that the theorem above applies to all discussed types of conflicts between update operations.

As in the read-delete case, we can show that only the read needs to be a linear pattern.

Lemma 8 *A read $R = \text{READ}_p$ and an insert $I = \text{INSERT}_{p',X}$, where p is a linear pattern and $p' \in P^{//,[],*}$, have a node conflict if and only if R and $I' = \text{SEQ}^{\mathscr{O}(I)}_{\text{ROOT}(I)}$ have a node conflict. Note I is not necessarily a linear pattern.*

Proof: Similar to the proof of Lemma 4. ■

Corollary 2 *For a read $R = \text{READ}_p$ and an insert $I = \text{INSERT}_{p',X}$, where $p \in P^{//,*}$ and $p' \in P^{//,[],*}$, a read-insert node conflict can be detected in polynomial time.*

5 NP-completeness of $P^{//,[],*}$

We show that the read-insert and read-delete node conflict problems are NP-complete when the patterns used are from $P^{//,[],*}$.

5.1 Read-Insert Node Conflicts

Suppose a read operation $R = \text{READ}_p(t)$ conflicts with an insertion $I = \text{INSERT}_{p',X}(t)$, we show that there is an XML tree W that witnesses the conflict, where the size of W is polynomial in the size of R and I. This fact allows one to present a non-deterministic polynomial time algorithm for deciding whether a read-insert conflict exists. One can guess a tree W of size polynomial in the inputs, and execute $R(W)$ and $R(I(W))$ to verify whether W acts as a witness. For the NP-hardness result, we reduce the problem of containment of XPath expressions [12] to the read-insert conflict problem (actually, we reduce the dual non-containment problem).

5.1.1 Membership in NP

If a witness W to a read-insert conflict exists, there must be at least one node n in $R(I(W))$ that is not in $R(W)$ (since insertion only adds nodes, any embedding in $R(W)$ is valid in $R(I(W))$. We show how to construct a W' from W such that $R(I(W'))$ still contains n, but $R(W')$ does not. This construction consists of two steps:

- *Marking* nodes in W that are essential to the conflict. In trimming a witness tree to polynomial size, we ensure that these nodes and relationships between them are maintained. Marking will guarantee that the witness node n is retained in $R(I(W))$.
- An operation, called *reparenting*, for constructing a smaller tree W' from a tree W. Reparenting guarantees that if $n \notin R(W)$, then $n \notin R(W')$.

We formalize the definitions of these terms and then show how successive reparenting can be used to prune a witness tree to size polynomial in the inputs, while ensuring that the pruned tree is still a witness.

Definition 9 (Marking) *Given a tree W that is a witness to a read-insert conflict:*

1. *Let $n_{witness}$ be a node that is in $R(I(W))$ but not in $R(W)$; if there is a node conflict on W, there must be at least one such node.*

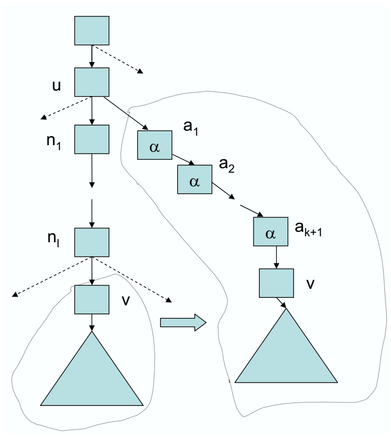


Figure 6: Structure of a reparent operation. The subtree rooted at v is moved and reconnected to u using $k+1$ nodes labeled α, where k is STAR-LENGTH(p) for the pattern p.

2. *Let e_R be an embedding of R into $I(W)$ such that $e_R(\mathscr{O}(R)) = n_{witness}$. Mark nodes in W based on e_R as follows. For all $n \in image(e_R)$:*
   - (a) *Mark n if $n \in$ NODES$_W$ (Observe that ROOT(W) is marked since it is in the range of every embedding by definition),*
   - (b) *If $n \notin$ NODES$_W$, mark the nearest ancestor n' of n in $I(W)$ such that $n' \in$ NODES$_W$. Choose an embedding e_I of I in W such that $e_I(\mathscr{O}(I)) = n'$. Mark all nodes in $image(e_I)$.*

Observe that the total number of nodes marked using this mechanism is at most $|I| \cdot |R|$. Now, the definition of reparenting:

Definition 10 (Reparent) *Given a tree t and a pattern p with STAR-LENGTH$(p) = k$. Let u, v be two nodes in t such that u is an ancestor of v and the number of nodes in the path from u to v is greater than $k+3$. Let v_p be the parent of v in t. The* reparenting *of v with respect to u and p is defined as the tree t' derived from t by:*

1. *Deleting (v_p, v) from EDGES$_t$.*
2. *Adding $\{a_1, a_2, \ldots, a_{k+1}\}$ to NODES$_t$ (where the $a_i, 1 \leq i \leq k+1$ are not in NODES$_t$ originally). Each of the a_i is labeled with a symbol $\alpha \notin \Sigma_p$.*
3. *Adding $(u, a_1), (a_1, a_2), \ldots, (a_{k+1}, v)$ to EDGES$_t$.*

Figure 6 depicts the basic structure of a reparent operation. We show that the reparenting operation does not add any new results in the sense that $[\![p]\!](t')$ contains either new nodes labeled α or a subset of the nodes in $[\![p]\!](t)$, where t' is the result of reparenting t with respect to p.

Lemma 9 *If a node v is reparented with respect to u and p, $u, v \in \textsc{nodes}_t, t \in T_\Sigma$, then $[\![p]\!](t') \cap \textsc{nodes}_t \subseteq [\![p]\!](t)$.*

Proof: We need to show that for $o \in \textsc{nodes}_T, o \in [\![p]\!](t') \implies o \in [\![p]\!](t)$. Suppose there is a node $o \in \textsc{nodes}_t, o \in [\![p]\!](t')$, but $o \notin [\![p]\!](t)$. Let $\mathcal{E}_1$ be an embedding of p into t' that maps $\mathscr{O}(p)$ to o. We show that given $\mathcal{E}_1$, there is an embedding $\mathcal{E}_2$ of p into t that maps $\mathscr{O}(p)$ to o as well, contradicting the assumption that $o \notin [\![p]\!](t)$.

$\mathcal{E}_1$ must map at least one node in $\textsc{nodes}_p$ to one of the newly inserted nodes labeled α. Otherwise, $\mathcal{E}_1$ would also be an embedding of p into t, since the derivation of t' does not change the parent-child or ancestor-descendant relationships or labels of any of the other nodes in t'. If $\mathcal{E}_1$ maps nodes to one or more of the nodes labeled α, any node in p that maps to such a node must be labeled with $*$ (since, by definition, α does not occur in p).

We define two sets:

- $\mathcal{R}_u$ is the empty set if $\mathcal{E}_1$ does not map any node in p to u. Otherwise, let $E = \{\omega \in \textsc{nodes}_p \mid \mathcal{E}_1(\omega) = u\}$. Then, $\mathcal{R}_u = \{\omega' \in \textsc{nodes}_p \mid$ there is a chain from ω to ω' in p for some $\omega \in E$, $\textsc{label}_{t'}(\mathcal{E}_1(\omega')) = \alpha\}$. In other words, $\mathcal{R}_u$ is the set of nodes in p that are mapped to a newly inserted node and are reachable from a node in E using only child edges.

- $\mathcal{R}_v$ is the empty set if $\mathcal{E}_1$ does not map any node in p to v. Otherwise, let $E = \{\omega \in \textsc{nodes}_p \mid \mathcal{E}_1(\omega) = v\}$. $\mathcal{R}_v = \{\omega' \in \textsc{nodes}_p \mid$ there is a chain in p from ω' to ω for some $\omega \in E$, $\textsc{label}_{t'}(\mathcal{E}_1(\omega')) = \alpha\}$. In other words, $\mathcal{R}_v$ is the set of nodes that are mapped to a newly inserted node such that a node in E is reachable using only child edges. We will say that a node ω in $\textsc{nodes}_p$ is *reachable from* $\mathcal{R}_v$ if $\omega \in \mathcal{R}_v$ or ω is the descendant in p of a node in $\mathcal{R}_v$.

We refer to the $k+1$ nodes labeled α in t' as $a_1, a_2, \ldots, a_{k+1}$, and the $l > k+1$ nodes between u and v in t as $n_1, n_2, \ldots, n_l$. The construction of $\mathcal{E}_2$ is straightforward. For each $\omega \in \textsc{nodes}_p$:

1. If $\mathcal{E}_1$ maps ω to a node in $\textsc{nodes}_t$ (that is, it does not assign ω to one of the newly inserted nodes labeled α), $\mathcal{E}_2$ assigns ω to the same node. More precisely, if $\mathcal{E}_1(\omega) = n, n \in \textsc{nodes}_t$, then $\mathcal{E}_2(\omega) = n$.

2. If $\mathcal{E}_1(\omega) = a_i, 1 \leq i \leq k+1$ and $\omega \in \mathcal{R}_u$, then $\mathcal{E}_2(\omega) = n_i$.

3. If $\mathcal{E}_1(\omega) = a_i, 1 \leq i \leq k+1$, and ω is reachable from $\mathcal{R}_v$, then $\mathcal{E}_2(\omega) = n_{l-k-1+i}$.

   Observe that $\mathcal{R}_u$ and $\mathcal{R}_v$ are disjoint. Otherwise, there would be a chain $\omega_1, \omega_2, \ldots, \omega_{k+3}$ in p, where $\mathcal{E}_1(\omega_1) = u$ and $\mathcal{E}_1(\omega_{k+3}) = v$ and the remaining nodes would be mapped to the $k+1$ nodes labeled α. This would contradict the assumption that $\textsc{star-length}(p) = k$. So, this assignment is well defined.

4. If $\mathcal{E}_1(\omega) = a_i, 1 \le i \le k+1$ and ω is in neither $\mathcal{R}_u$ nor reachable from $\mathcal{R}_v$, then $\mathcal{E}_2(\omega) = n_i$.

It remains to be shown that $\mathcal{E}_2$ is a valid embedding. By construction, it should be clear that $\mathcal{E}_2$ is root-preserving and label-preserving since only nodes in p labeled $*$ are mapped to different nodes in $\mathcal{E}_2$ and $\mathcal{E}_1$.

We first consider child edges constraints, where $(\omega, \omega') \in \textsc{edges}_/(p)$. If ω, ω' are both mapped to nodes that are not labeled α in $\mathcal{E}_1$, then $\mathcal{E}_2$ assigns them to the same nodes in W'. Reparenting does not change the relationship between these nodes and the edge is satisfied.

If ω' is mapped to a node labeled α in $\mathcal{E}_1$ and ω is not, then ω must be mapped to u (it is the only node in W that is the parent of a node labeled α). By construction $\mathcal{E}_2$ maps ω to u as well, and ω' must be in $\mathcal{R}_u$. $\mathcal{E}_1(\omega')$ must be a_1, and therefore, $\mathcal{E}_2(\omega') = n_1$, which satisfies the constraint. Similarly, if ω is mapped to a node labeled α in $\mathcal{E}_1$ and ω' is not, $\mathcal{E}_1(\omega')$ must be v. This implies that $\mathcal{E}_2(\omega')$ is v as well, and ω must be in $\mathcal{R}_v$. $\mathcal{E}_1(\omega)$ must be a_{k+1} implying $\mathcal{E}_2(\omega) = n_{l-k-1+(k+1)} = n_l$, which is the parent of v in t.

The remaining case is when ω, ω' are both mapped to nodes labeled α. If both ω and ω' are reachable from $\mathcal{R}_v$, then by construction (Step 3), both will be assigned to nodes that satisfy the parent-child relationship. If $\mathcal{E}_1(\omega) = a_i, \mathcal{E}_1(\omega') = a_{i+1}, 1 \le i \le k$, Step 3 would assign ω and ω' to $n_{l-k-1+i}$ and n_{l-k+i} respectively, which satisfy the parent-child relationship. If both ω and ω' are not reachable from $\mathcal{R}_v$, then again, the construction (Step 2 or Step 4) of $\mathcal{E}_2$ would assign them to nodes that satisfy the parent-child relationship. There can never be a child edge from an ω not in $\mathcal{R}_v$ to a node in $\mathcal{R}_v$. Otherwise, by the definition of $\mathcal{R}_v$, ω would be in $\mathcal{R}_v$ — there would be a chain from ω (via ω') to a node in p that is mapped to v.

Now let us consider descendant edge constraints. If ω, ω' are not mapped to nodes labeled α by $\mathcal{E}_1$, $\mathcal{E}_2$ maps to the same nodes as $\mathcal{E}_1$. Observe that reparenting preserves descendant constraints between these nodes. If ω is mapped to a node not labeled α by $\mathcal{E}_1$, and ω' is, again $\mathcal{E}_2$ preserves this relationship. All of the nodes in the chain n_1 to n_l would be descendants of $\mathcal{E}_2(\omega) = \mathcal{E}_1(\omega)$. Similarly, if ω is mapped to a node labeled α by $\mathcal{E}_1$, and ω' is not, again $\mathcal{E}_2$ preserves this relationship. All of the nodes in the chain n_1 to n_l are ancestors of $\mathcal{E}_2(\omega') = \mathcal{E}_1(\omega')$.

Finally, we verify the case where ω, ω' are mapped to nodes labeled α by $\mathcal{E}_1$. if ω is reachable from $\mathcal{R}_v$, then ω' must be too. The construction preserves the appropriate relationship between $\mathcal{E}_2(\omega)$ and $\mathcal{E}_2(\omega')$. The same is true if both ω and ω' are not reachable from $\mathcal{R}_v$. Finally, if ω' is reachable from $\mathcal{R}_v$, and ω is not, then observe that if $\mathcal{E}_2(\omega) = n_i$ and $\mathcal{E}_2(\omega') = n_j$, then $\mathcal{E}_1(\omega) = a_i$ and $\mathcal{E}_1(\omega') = a_m$, where $j = l - k - 1 + m$. Since $l > k + 1 \implies j > m$, the descendant relationship is maintained.

Since we can recreate any embedding that results in an output node in $\textsc{nodes}_t$ in $[\![p]\!](t')$, $[\![p]\!](t') \cap \textsc{nodes}_t \subseteq [\![p]\!](t)$. ■

Given a witness of a read-insert conflict, we use *reparenting* to whittle it down to a size polynomial in the size of the inputs. While doing so, we have to ensure that the pared-down tree still causes a conflict. We assert that if W is a witness to a read-insert conflict, appropriate reparenting will result in a tree that is still a witness to the conflict.

Lemma 10 *Let n and n' be nodes in W that are marked such that n is the nearest ancestor of n' that is marked, that is, $\forall n'', (n, n'') \in \textsc{desc}(t) \wedge (n'', n') \in \textsc{desc}(t) \implies n''$ is unmarked. Moreover, let the number of nodes in the path from n to n' be greater than $k + 3$, where $k = \textsc{star-length}(R)$. Consider the tree W' derived from W by reparenting n' with respect to n and R. W' is a witness to the read-insert conflict.*

Proof: W' contains all nodes that are marked in W and retains the relationships between the marked nodes and node labels on these nodes. As a result, it can be shown that the embedding e_I of Definition 9 is preserved. Since e_I is preserved all insertions necessary to preserve e_R (of Definition 9) will be performed and e_R will be preserved in $I(W')$. Therefore, $n_{witness} \in [\![R]\!](I(W'))$. Furthermore, by Lemma 9, $n_{witness} \notin [\![R]\!](W) \implies n_{witness} \notin [\![R]\!](W')$. Therefore, W' is still a witness to the read-insert conflict. ■

We can now conclude that a there is a witness tree polynomial in the size of the inputs.

Lemma 11 *If a read R conflicts with an insertion I, there exists a tree W of size $|R| \cdot |I| \cdot (k + 1)$ such that $R(I(W)) \neq R(W)$, where $k = \textsc{star-length}(R)$.*

Proof:[Sketch] Given any witness tree W', we mark all nodes as described in Definition 9, and iteratively, reparent nodes n for which the number of nodes in the path from the nearest marked ancestor n' to n is greater than $k + 3$. Once this process ends, we discard all nodes that are unmarked and do not contain a marked node as a descendant (observe that at the end of this stage, all leaves of the tree will be marked). There are at the most $|R| \cdot |I|$ marked nodes and the distance of each node to its nearest marked ancestor is at the most $k + 1$. The total number of nodes in the tree W is bounded by $|R| \cdot |I| \cdot (k + 1)$.

Observe that the alphabet of the labels in the witness tree can be limited to labels from $\Sigma_R \cup \Sigma_I \cup \alpha$, where α is a symbol not in Σ_R. Therefore, the total size of the tree is polynomial in the size of the inputs. ■

Theorem 3 *Read-insert node conflict detection for $P^{//,[],*}$ is in NP.*

Proof: Given Lemma 11, we can non-deterministically guess a tree of size less than $|R| \cdot |I| \cdot (k + 1)$ with labels from $\Sigma_R \cup \Sigma_I \cup \alpha$, where α is a symbol not in Σ_R. By Lemma 1, we can verify in polynomial time whether a tree t is a witness to the read-insert conflict. ■

REMARKS: Extending the results to tree and value conflicts mainly involves modifying which nodes are marked. In the tree conflict case, if one has a witness W, let n be the root of a tree in $R(I(W))$ that is not equivalent to any in $R(W)$. If n is not present in $R(W)$ then we can mark the tree as in the node conflict case and prove that there exists a tree polynomial in the size of the inputs such that the same tree conflict would occur. If n is present in $R(W)$, then let v be an insertion point in the subtree rooted at n. Choose an embedding of R into W that selects n and mark all nodes in W in the image of the embedding. Then, choose an embedding of I into W that selects v and mark all nodes that participate in that embedding. Reparenting with respect to this set of marked nodes will ensure that the tree conflict exists in a tree polynomial in the size of the inputs.

To show that read-insert conflict detection is in NP for value-based semantics, we pursue the same strategy of marking as for the tree conflict case. When, at the end of the reparenting stage, subtrees consisting of unmarked nodes are discarded (as in Lemma 11), one must ensure that the conflict still occurs in the pruned tree. Let p be a node that is not discarded and that is the parent of such a discarded tree. We add a child node to each such p, where each child node is labeled with a separate distinct symbol that is not already used in the tree (or in R, I, X). This addition will guarantee that any conflict present in W will also be present in the pruned tree.

5.1.2 NP-Hardness

For the NP-hardness result, we reduce the non-containment problem for XPath expressions:

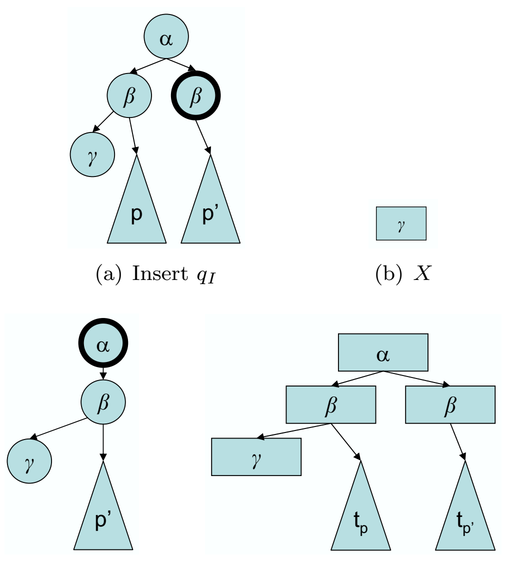
Definition 11 *A pattern p is contained in another pattern p', denoted $p \subseteq p'$, if $\forall t \in T_\Sigma, [\![p]\!](t) \neq \emptyset \implies [\![p']\!](t) \neq \emptyset$. In other words, the existence of an embedding of p into t implies the existence of an embedding of p' into t.*

Let p, p' be two patterns in $P^{//,[],*}$. Miklau and Suciu [12] have shown that the decision problem of whether $p \not\subseteq p'$ is NP-hard. We reduce the non-containment problem to that of determining whether a read-insert conflict exists.

Theorem 4 *Read-insert node conflict detection for $P^{//,[],*}$ is NP-hard.*

Proof: Given an instance of the non-containment problem, that is, two patterns $p, p' \in P^{//,[],*}$, we construct an instance of the read-insert conflict problem. We construct an insertion operation $I = \text{INSERT}_{q_I,X}$ and a read operation, $R = \text{READ}_{q_R}$, where q_I, X, and q_R are constructed from p and p' as depicted in Figure 7a-c; q_I is the pattern equivalent to the XPath expression $\alpha[\beta[p][\gamma]]/\beta[p']$, q_R is equivalent to $\alpha[\beta[p'][\gamma]]$, and X is the tree consisting of a single node labeled γ, where α, β, γ are symbols not used in p and p'. Observe that the construction can be performed easily in polynomial time.

We show that there is a read-insert node conflict between R and I if and only if $p \not\subseteq p'$. If $p \not\subseteq p'$, then we can construct a witness tree W as shown in Figure 7d, where the root node is labeled α and contains two distinct children labeled β. One β child contains a subtree t_p for which there is an embedding of p into t, but no embedding of p' into t_p. Since $p \not\subseteq p'$, the existence of such a t_p is guaranteed. This β child also contains a node labeled γ. The other β



(a) Insert q_I (b) X

(c) Read q_R (d) Witness W

Figure 7: (a) Query q_I used in insertion operation (b) Tree added in insertion operation (c) query q_R used in read operation (d) Structure of witness tree.

child contains a tree $t_{p'}$ for which there is an embedding of p' into $t_{p'}$, but no γ child. R has no embedding in W because the β child of $\textsc{root}(W)$ that matches p' does not have a γ child. R does have an embedding after the execution of the insertion operation, which inserts a γ node into the appropriate point. As a result, $R(W) = \emptyset$, but $R(I(W)) = \{\textsc{root}(W)\}$, which implies a read-insert conflict.

If there is an read-insert conflict, there is a tree W that is a witness to the conflict. Observe that for all $t \in T_\Sigma$, $R(t)$ returns at the most a single node ($\textsc{root}(t)$). Since $R(W) \neq R(I(W))$, $R(W)$ must be empty and $R(I(W))$ must be $\textsc{root}(W)$. Since $R(I(W))$ and $R(W)$ are different, the insert operation modifies the tree W. There must be, therefore, an embedding of I into W. Consider the subtree of W, t_p, to which an embedding maps the subpattern of I corresponding to p. There can be no embedding from p' into this subtree; this would imply that there is an embedding of R into W where the subpattern corresponding to p' is mapped to t_p. As a result, t_p is a tree that has an embedding from p but not from p', proving the assertion that $p \not\subseteq p'$.

Since R and I conflict if and only if $p \not\subseteq p'$, the read-insert node conflict problem is NP-hard. ■

Corollary 3 *Read-insert node conflict detection for patterns in* $P^{//,[],*}$*is NP-complete.*

REMARKS: Observe that the reduction shown for node conflicts cannot be used directly for tree conflicts. Every tree that causes an insertion to occur would be a witness to a tree conflict. We modify R slightly, where we add a child node labeled δ to $\textsc{root}(R)$ and mark this child node as the output node. Since the subtree under node matching δ is never modified by I, there will be a tree conflict between the modified R and I if and only if there is a node conflict between the modified R and I.

For the value conflicts, note that in this reduction, the value and tree conflict semantics match (in a sense). A modified R, R', as suggested for the tree conflict case, can be used to show that the problem is NP-hard for value conflicts as well. If $p \not\subseteq p'$, there is a tree W such that $R'(W) = \emptyset$ and $R'(I(W)) \neq \emptyset$ (which would raise a conflict in the value-based semantics). We claim that read-insert detection is NP-complete for all three kinds of conflicts.

5.2 Read-Delete Node Conflicts

The characteristics of the read-delete conflict detection problem for $P^{//,[],*}$ are similar to that of the read-insert problem. We, therefore, only sketch the proofs of membership in NP and NP-hardness.

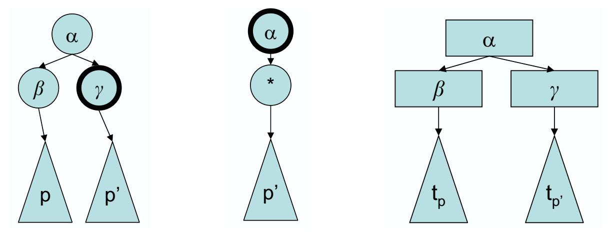
Theorem 5 *Read-delete node conflict detection for $P^{//,[],*}$ is in NP.*

Proof:[Sketch] The idea behind the proof of membership in NP is similar to the read-insert case — demonstrate that any witness W to a read-delete conflict can be pruned into a witness of size polynomial in the inputs. Let R be the read operation and D be the delete operation. Let v be a node in $R(W)$ that is not in $R(D(W))$. We select an embedding of R into W where $\mathscr{O}(R)$ is mapped to v and mark all nodes in W that participate in the embedding. We also consider the ancestor of v in W, u, such that u is not in $R(D(W))$ and the parent of u is in $R(D(W))$. There must be at least one such node since $\textsc{root}(W)$ is in $R(D(W))$. u must be a deletion point — we select an embedding of D in W that selects u as the output node, and mark all nodes that participate in the embedding. As in the read-insert case, we then reparent to reduce W to a new tree W' polynomial in the size of the inputs such that all marked nodes and the relationships between them are preserved. $R(W')$ will contain v, but $R(D(W'))$ will not. As a result, W' will be a witness to the read-delete conflict.

Since any conflict can be found in a tree of size less than k where k is polynomial in the size of D and R, we can non-deterministically guess a tree and verify whether it is a witness to the conflict. As a result, the read-delete problem is in NP. ■

Theorem 6 *Read-delete node conflict detection for $P^{//,[],*}$ is NP-hard.*

Proof: As in the read-insert case, we provide a reduction to the non-containment problem. Given an instance of the non-containment problem, that is, two patterns $p, p' \in P^{//,[],*}$, we construct an instance of the read-delete conflict problem. We construct a deletion operation $D = \textsc{delete}_{q_D}$ and a read operation, $R = \textsc{read}_{q_R}$, where q_D, q_R are constructed from p and p' as depicted



(a) Delete q_D (b) Read q_R (c) Witness W

Figure 8: (a) Query q_D used in deletion operation (b) query q_R used in read operation (c) Witness W to read-delete conflict.

in Figure 8a-b; q is the pattern equivalent (roughly) to the XPath expression $q_D = \alpha[\beta[p]]/\gamma[p']$, and q_R is equivalent to $\alpha[*[p']]$, where α, β, γ are arbitrary symbols. Observe that the construction can be performed easily in polynomial time.

We show that $p \not\subseteq p'$ if and only if D and R have a read-delete conflict. If $p \not\subseteq p'$, there must be a tree t_p such that there is no embedding from p' into t_p. We use the witness tree in Figure 8c to show a read-delete node conflict. In W, $t_{p'}$ is some tree for which there is an embedding of p' into $t_{p'}$. Clearly $[\![R]\!](W)$ contains $\text{ROOT}(W)$ since $\text{ROOT}(W)$ contains a grandchild that has an embedding from p'. The delete operation, D, will remove the subtree of W rooted at the node labeled γ. Therefore, $R(D(W))$ will not contain $\text{ROOT}(W)$ since $D(W)$ does not contain any child of $\text{ROOT}(W)$ into which p' can be embedded (by assumption, p' cannot be embedding into t_p). As a result W is a witness to a read-delete conflict.

If there is a read-delete conflict, let W be a witness to the conflict. Since $R(D(W)) \neq R(W)$ and $R(t)$ for any tree in T_Σ contains at most one node (the root of t), $R(D(W)) = \emptyset$. Suppose $p \subseteq p'$. Since the deletion operation modifies W, there must be an embedding of D into W. Therefore, $\text{ROOT}(W)$ must contain a child u labeled β such that there is an embedding of p into a subtree under u, t_p, where the root of p is mapped to the root of t_p. This tree is not deleted by D — only children of $\text{ROOT}(W)$ labeled γ are affected by the delete operation. As a result, since $p \subseteq p'$, there must be an embedding of R into $D(W)$, where the node labeled $*$ in R is mapped to u and p' in R is mapped to t_p. The existence of an embedding contradicts the fact that $R(D(W)) = \emptyset$. Therefore, $p \not\subseteq p'$.

Since $p \not\subseteq p'$ if and only if D and R have a read-delete conflict, we can conclude that the read-delete problem for patterns in $P^{//,[],*}$ is NP-hard. ■

Corollary 4 *Read-delete conflict detection for patterns in $P^{//,[],*}$ is NP-complete.*

REMARKS: As in the read-insert case, a slight modification to R (adding a child labeled δ of $\text{ROOT}(R)$ as the output node) will ensure that $p \not\subseteq p'$ if and only if the modified R and D do not have a tree or value conflict. The details are omitted for space.

6 Discussion

We now examine extensions to the problems discussed in the paper.

Complex Updates While we have focused on read-delete and read-insert conflicts, the other conflicts (delete-insert, delete-delete, and insert-delete) are of interest as well. Informally, we can define conflicts in these situations as two operations o_1, o_2 conflict if there is a tree $t \in T_\Sigma$ such that $o_1(o_2(t))$ is not equal to $o_2(o_1(t))$, where o_1 and o_2 can be either an insert or a delete.

In the reference-based semantics, the definition of these conflicts is not completely straightforward. Two insertions I_1 and I_2 ought not to have an insert-insert conflict if I_1 and I_2 are identical; in this case, for any $t \in T_\Sigma$, $I_1(I_2(t))$ ought to be considered equivalent to $I_2(I_1(t))$. In the reference semantics, the problem is with the clones of X (in $\text{INSERT}_{p,X}$) that are inserted into any tree — they do not preserve node equality. A suitable reference-based semantics of conflicts would have to distinguish these nodes appropriately. Value-based semantics do not have this problem.

The reductions from XPath containment provided in Section 5 can be modified in a straightforward manner to show that each of these conflicts is NP-hard. We conjecture it possible to extend the techniques of Section 5 to show membership in NP as well.

Fragments of XPath A question we have not considered in this paper is the complexity of tree patterns in $P^{//,[]}$, that is, the subset of $P^{//,[],*}$ that allows branching, but not the use of the wildcard operator. Containment for this subset is in PTIME [2].

For other subsets of XPath, we remark that the satisfiability problem ($P^{//,[],*}$is always satisfiable) can be encoded as a read-delete (or read-insert) problem. A read that selects all nodes in a tree would always conflict with a delete unless the delete pattern were not satisfiable. For subsets of XPath that can result in unsatisfiable tree patterns (for example, those involving parent or ancestor), this reduction may be useful.

Schema Information The complexity of conflicts when schema information (for example, DTDs) is available is an open problem. In general, the addition of DTDs appears to raise the complexity of issues related to XPath. For example, as mentioned before, containment of $P^{//,[]}$ in in PTIME. When the problem of containment is constrained to detect whether for all trees conforming to a DTD, one XPath is contained in another, the problem is coNP-complete [13].

7 Related Work

The subject of updates in XML has only recently started getting attention. Efforts have been focused on the specification of an appropriate set of update operations [16, 17, 21] and incremental validation of XML [3, 14]. There has been a significant amount of work on containment [2, 12] and satisfiability [9] for various fragments of XPath expressions. Update conflict detection has some similarity to these problems in that the properties of fragments of XPath expressions play a role, but the ability to change trees through updates adds an interesting new dimension.

The field of data dependencies in programs has been well-studied — mostly for programs that operate on arbitrary data structures. Some analyses, for example, *shape analysis* [15], use properties of data structures (for example, being a tree). XPath expressions, considered as pointer expressions, are fairly powerful (they support transitive closure with the descendant axis). We are unaware of any static analysis results that apply directly to the update conflict problem.

8 Conclusion

We believe that update operations (with value-based or reference semantics) are useful additions to current XML-based programming languages. With the presence of update operations, a natural question is when modifications to data cause two operations to conflict. We have provided three alternate formulations of semantics for conflicts, two that are reference-based, and one that is value-based. For linear XPath expressions that do not support branching, but allow child and descendant axes, and the wildcard operator, we have shown that polynomial-time algorithms exist for all three semantics. If the branching operator is also allowed, the problem becomes NP-complete. The development of an understanding of updates can lead to more efficient compilers for languages such as XQuery and XJ.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *The VLDB Journal*, 11(4):315–331, 2002.

[3] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems (TODS)*, 29(4):710–751, 2004.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the Eighth International Conference on Functional Programming*, pages 51–63, 2003.

[5] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in Cω. `http://research.microsoft.com/~emeijer`.

[6] V. Gapeyev and B. Pierce. Regular object types. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 151–175, July 2003.

[7] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, 2002.

[8] M. Harren, M. Raghavachari, O. Shmueli, M. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: Facilitating XML processing in Java. In *Proceedings of World Wide Web (WWW)*, pages 278–287, May 2005.

[9] J. Hidders. Satisfiability of XPath expressions. In *Proceedings of Database and Programming Languages*, volume LNCS 2921 of *Lecture Notes in Computer Science*, 2003.

[10] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 2003.

[11] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[12] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[13] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proceedings of the 9th International Conference on Database Theory*, pages 315–329. Springer-Verlag, 2002.

[14] M. Raghavachari and O. Shmueli. Efficient schema-based revalidation of XML. In *Proceedings of Extending Database Technology (EDBT)*, volume 2992 of *LNCS*, March 2004.

[15] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[16] G. Sur, J. Hammer, and J. Siméon. UpdateX - an XQuery-based language for processing updates. In *PLAN-X*, January 2004.

[17] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, June 2001.

[18] P. Wadler. Two semantics of XPath. `http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf`.

[19] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, November 1999.

[20] World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, April 2005. W3C Working draft.

[21] World Wide Web Consortium. *XQuery Update Facility Requirements*, June 2005.