RECONCILIATION OF CHANGES ON XML LISTS

Eunjung Lee [a]

Department of Computer Science, Kyonggi University, San 94 Yi-ui Dong, Young-Tong Gu
Suwon,South Korea
ejlee@kyonggi.ac.kr

Abstract

In this research, we developed an efficient reconciliation method for mobile environments by using edit scripts of XML data sent from each mobile device. To obtain a simple model for mobile devices, we use the XML list data-sharing model, which allows the insertion/deletion of subtrees only for repetitive parts of the tree based on the document type. Furthermore, for subtrees of repetitive parts, we use keys that are unique between nodes with the same parent. This model not only guarantees that the edit action always results in a valid tree but also allows for a key-based linear time-reconciliation algorithm. Assuming that there are no insertion key conflicts, the proposed algorithm proposed requires a time of O(m) where m is the size of the edit script.

***Keywords*:** XML; edit conflict; reconciliation.

1. Introduction

Owing to the rapid proliferation of XML in many different application areas, it has become possible for users to work concurrently on XML documents to share data. Isolating concurrent accesses is an important issue in XML database systems or distributed applications based on shared XML documents [Cabri (2000)]. Previous research has focused on data reconciliation rather than differences in data structure [Fontain (2002)], [Kermarrec (2001)], [Lindholm (2003)]. Furthermore, in studies on structural reconciliation, researchers have not considered the validity of updated trees. Therefore, a desirable goal is to find a way to guarantee validity and develop a structural three-way merging algorithm [Kermarrec (2001)] at the same time.

As is often mentioned in the literature, the full capability of semi-structuredness is more than adequate for shared data formats of distributed applications. We have found that structural updates on shared trees are often allowed only for the repetitive parts of the tree, for reasons of complexity. Using this property, we can obtain a much simpler and more efficient data-sharing platform.

A new reconciliation method is presented for the proposed model. The method reconciles lists of repetitive parts based on their key values. We also propose syntax based resolution rules for structural conflicts. Using this resolution approach, we show that if two edit scripts do not have a key conflict, then the two edit actions can be reconciled in an amount of time that is linearly related to the length of the to-be-merged edit scripts.

This paper is organized as follows. In section 2, we present our data-sharing model. In section 3, we introduce the concept of structural compatibility, which formalizes the three changes in our data-sharing model. We describe our algorithm of reconciling two edit scripts in section 4, and then summarize the results in section 5.

2. Sharing Model for XML List Data

In this section, we examine sharing patterns in XML-based collaboration applications. As shown below, structural update actions such as subtree insertion/deletions are often applied to repetitive parts in XML, denoted by * in data type definitions (DTDs).

2.1. *Rationale*

In Figure 1, list nodes with repetitive parts are represented by filled circles. The possible shared actions for this type of data are as follows: reading/writing data values, traversing tree nodes, and inserting/deleting subtrees. For the example in Figure 1, the shared actions include adding new items to <items-ordered>, adding a new order to <order-list>, deleting an item from an order, or deleting an order from an order-list. We found the same pattern in other examples in the literature, such as auction data and shopping baskets. For instance, in XML documents including auction status, the actions include adding a new bid to the bid-list and deleting an auction item from the list. In a shared shopping basket, adding/deleting a new item to/from a list is also applied to the repetitive parts. As these examples demonstrate, most shared actions that update tree structures are applied to the repetitive parts of the tree.

Motivated by these observations, we introduce an XML data-sharing interface that allows structural update actions only for the repetitive parts.

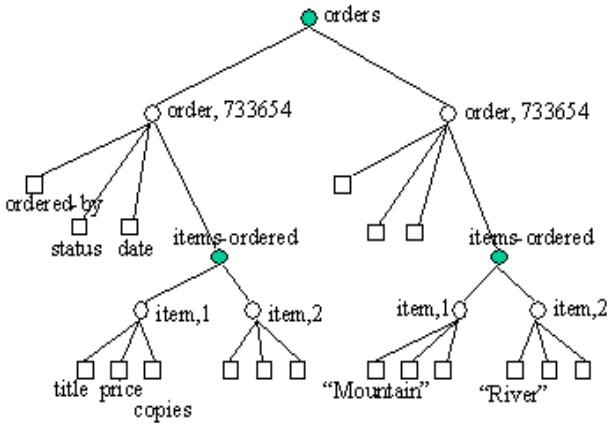

Figure 1. Example of XML tree of <orders>.

2.2. *Valid actions for list structure updates*

The following are actions for sharing XML data. Read(w) and Write(w, $value_{new}$) actions read and update, respectively, the data value of the terminal node. The action Traverse(v) reads the list of children for the internal node v. Insert(v, T_{new}) inserts T_{new} as a subtree of v and Delete(v, $A[k]$) deletes a child tree with a root labeled A and key k.

In order to share XML data, actions on terminal nodes such as read/write data can be controlled concurrently in the same way as in relational databases (RDBs). Therefore, we focus on structural update actions such as subtree insertions and deletions. In this paper, we restrict such actions to repetitive lists in order to guarantee the validity of the actions. A node is called a *list node* if it corresponds to a repeated node in the DTD. A node also is called a *list parent node* if it is a parent node of a list node. The following are definitions for list-update actions.

Definition 2.1. *Let v_L be a list parent node. Then, the list-update actions on v_L are defined as follows.*
(1) ListInsert (v_L, T_{new}): Insert T_{new} as a child of v_L. The key of T_{new} should be unique to siblings.
(2) ListDelete (v_L, $A[k]$): Delete the child subtree $A[k]$ from v_L.

3. Structural Compatibility

In this section, we introduce the concept of structural compatibility, which shows that a condition is reconcilable, in other words, that there is a sequence of valid edit actions.

Definition 3.1. *Two XML trees T_1 and T_2 are said to be structurally compatible (written as $T_1 \approx T_2$) if and only if*
i) label (root (T_1)) = label (root (T_2))
ii) Let $T_{11}, T_{12}, \ldots, T_{1k}$ and $T_{21}, T_{22}, \ldots, T_{2m}$ be the child trees of T_1 and T_2, respectively.
Proof.

(1) If root (T_1) is not a list parent node, then k = m and $T_{1i} \approx T_{2i}$, $1 \le i \le k$.
(2) If root (T_1) and root (T_2) are list parent nodes, then for all i, j such that key (root (T_{1i})) = key (root (T_{2j})), $T_{1i} \approx T_{2j}$. □

Two trees are structurally compatible if they have the same structure for non-repetitive parts. For the repetitive parts, any two corresponding subtrees with the same key should be structurally compatible. This reflects the fact that the two trees differ only in the repetitive parts. Furthermore, the two corresponding subtrees are recursively compatible. Two structurally compatible trees can be merged based on the keys. For the repetitive parts that mutually differ, the two lists are merged. In addition, matching pairs of subtrees are merged recursively.

The following algorithm shows the merging of two trees, together with verification of the structural compatibility.

Algorithm 1. Tree-merging algorithm.
Input: input trees T_1, T_2
Output: resulting tree $T_m = T_1 \oplus T_2$, if $T_1 \approx T_2$; fail otherwise.

1. If label(root(T_1)) ≠ label(root(T_2)), fail.
2. Create a node v as a root(T_m) where label(v) = label(root(T_2)).
3. Let $T_{11}, T_{12}, \ldots, T_{1k}$ and $T_{21}, T_{22}, \ldots, T_{2m}$ be the child trees of T_1 and T_2 respectively.
3-1. If root(T_1) is not a list parent node,
3-1-1. If $n_1 = n_2$ and $T_{1i} \approx T_{2i}$ for all $1 \le i \le n_1$, add $T_{1i} \oplus T_{2i}$ to T_m as a child subtree.
3-1-2. Otherwise, fail.
3-2. If root(T_1) is a list node,
3-2-1. For T_{1i} and T_{2j} such that key(T_{1i}) = key(T_{2j}), if $T_{1i} \approx T_{2j}$, add $T_{1i} \oplus T_{2j}$ to T_m as a child subtree. Otherwise, fail.
3-2-2. For T_{1i}, if there is no subtree of T_2 with the same key, add T_{1i} to T_m.
3-2-3. For T_{2j}, if there is no subtree of T_1 with the same key, add T_{2j} to T_m.

The time requirement for this merging algorithm is linearly related to the size of the tree.

4. Reconciliation for List-Structure Updates

We assume that edit scripts are provided for the updated trees. An edit script can be found by comparing the updated tree with the original tree. In the first subsection, we will discuss the reconciliation of two edit actions. This can be applied to refine edit script or to reconcile two edit scripts. Examples for each are presented in the ensuing subsections.

4.1. *Reconciliation of two edit actions*

There are three edit actions: ListInsert(, T_{new}[k]), ListDelete(, k), and Update(, $value_{new}$). For two different edit actions, a temporal relation is defined to indicate whether they belong to a sequence of edit actions (an edit script) or two different edit scripts. In addition, each edit action has a path denoting a node to apply the action. We find the relation between two paths, i.e., whether they are identical or independent, or whether one includes the other. The following relations express these three properties of reconciling two edit actions.

- $_1$, $_2$ ∈ {D, I, U}
- Relations between time ($_1$) and time ($_2$) ∈ {<, >, ~}
- Relations between path ($_1$) and path ($_2$) ∈ {<<, >>, =, ~}

Definition 4.1. *Let two actions be denoted as $_1$, $_2$. $_1$ is said to include $_2$ (denoted as $_1 \lhd$ $_2$) iff $_1$ $_2$(T) = $_1$(T) for all T (XML trees). In addition, $_1$ is compatible with $_2$ (denoted as $_1 \sim$ $_2$) iff $_1$ $_2$(T) = $_2$ $_1$(T) for all T and $_1 \lhd$ $_1$ $_2$ and $_2 \lhd$ $_1$ $_2$. Moreover, we define conflicts and annuls as follows:*

- *$_1$ conflicts with $_2$ iff the two relevant actions cannot be serialized.*
- *$_2$ annuls $_1$ iff $_1$ $_2$(T) = T.*

The path properties are used to identify the relations between two actions. The paths could be related in one of four ways: path ($_2$) << path ($_1$), path ($_1$) << path ($_2$), path ($_1$) = path ($_2$), or path ($_1$) ~ path ($_2$). If the two paths are independent, then the actions are compatible. In all other cases, the two actions can be in

conflict. For example, let $_1$ be an insert action and $_2$ be a delete action. If path ($_1$) << path ($_2$), then a descendant of the inserted subtree is deleted by $_2$. The result of the two actions is identical to the single action of inserting a subtree in which the descendant is deleted

The following table shows the relationships between two actions.

Table 1. Reconciliation Table of edit actions.

		In the same edit script			In different edit scripts		
		$\pi_1=\pi_2$	$\pi_1<\pi_2$	$\pi_2<\pi_1$	$\pi_1=\pi_2$	$\pi_1<\pi_2$	$\pi_2<\pi_1$
$D(\pi_1, k_1)$	$D(\pi_2, k_2)$	-	-	σ_2	*O*	*C*	*C*
$D(\pi_1, k_1)$	$I(\pi_2, T_{new2}[k_2])$	*Conflict if* $k_1 = k_2$	-	σ_2/σ_1	*C if* $k_1 = k_2$	*C*	-
$I(\pi_1, T_{new1}[k_1])$	$I(\pi_2, T_{new2}[k_2])$	-	σ_1/σ_2	σ_2/σ_1	*C if* $k_1 = k_2$	-	-
$D(\pi_1, k_1)$	$U(\pi_2, x)$	-	-	-	-	*C*	-
$I(\pi_1, T_{new1}[k_1])$	$U(\pi_2, x)$	-	σ_1/σ_2	-	-	-	-
$U(\pi_1, k_1)$	$U(\pi_2, x_2)$	σ_2	-	-	*C*	-	-
$I(\pi_1, T_{new1}[k_1])$	$D(\pi_2, k_2)$	*O*	σ_1/σ_2	σ_2	*Relative order is not meaningful*		
$U(\pi_1, x_1)$	$D(\pi_2, k_2)$	-	-	σ_2			
$U(\pi_1, x_1)$	$I(\pi_2, T_{new2}[k_2])$	-	-	-			

O: compatible, C: conflict, -: not applicable

4.2. *Refining edit script*

This section will present the refinement process for an edit script by using the relations described above. The result is a refined edit script in which there is no redundant action and no two actions conflict with one another.

Algorithm 2. Refining algorithm of an edit script.
Input: List edit script $\Sigma = (\sigma_1\sigma_2...\sigma_n)$.
Output: Refined edit script Σ or fail when Σ is not refinable.
1. For all pairs of edit actions σ_i, $\sigma_j \in \Sigma$ such that $i < j$,
1-1. if $\sigma_j \lhd \sigma_i$, remove σ_i from Σ.
1-2. if $\sigma_i \lhd \sigma_j$, remove σ_j and substitute σ_i with $\sigma_i' = \sigma_i \sigma_j$.
1-3. if σ_i, σ_j have a pair of delete/insert actions for the same subtree with the same key, then for deleted subtree T_1 and inserted subtree T_2,
1-3-1. if $T_1 \approx T_2$,,
1-3-1-1. find edit script Ω s.t. $\Omega(T_1) = T_2$
1-3-1-2. let W' be the edit script in which π_1/π_2 is extended to each action in Ω,
1-3-1-3. remove σ_j from Σ and substitute σ_i with Ω',
1-3-2. return fail otherwise.

Theorem 4.2. *For every there exists a unique refined edit script equivalent to '.*

In the next section, we assume that the edit scripts are refined with the above algorithm.

4.3. *Reconciliation of two edit script*

This section will demonstrate a reconciliation algorithm for two refined edit scripts. The result is a merged edit script. We use the findings described above on a pair of edit actions and syntactic resolution rules on two conflicting actions.

In our model, we use following three resolution rules:
(1) ListInsert > ListDelete
(2) Update > ListDelete
(3) ListDelete > ListDelete if <

where > represents a precedence rule between actions. For data updates, we assume some appropriate rules given by users.

[Algorithm 3] Reconciliation algorithm for edit scripts.
Input: $\Sigma_1 = \{\sigma_1,\sigma_2, ...,\sigma_k\},\Sigma_2 = \{\rho_1,\rho_2, ...,\rho_n\}$,
RP: a set of conflict resolution rules
Output: merged edit script Λ

1. $i \leftarrow 1, j \leftarrow 1$
2. For all $i < k, j < n$, *do the following for* σ_i, σ_j,
2-1. If $\sigma_i \sim \rho_j$ *and preorder* (σ_i) > *preorder* (ρ_j), *then* $\Lambda = \Lambda + \rho_j, j \leftarrow j+1$
2-2. If $\sigma_i \sim \rho_j$ *and preorder* (σ_i) < *preorder* (ρ_j), *then* $\Lambda = \Lambda + \sigma_i, i \leftarrow i+1$
2-3. If path (σ_i) << *path* (ρ_j), *then* $j' = max_{j' \geq j}$ (*path* (σ_i) < *path* (ρ_j))), $j \leftarrow j'+1$
2-3-1. Compare the precedence between σ_i *and* $\rho_j, \ldots, \rho_j$' *by RP, and add the action with the higher precedence to* Λ.
2-3-2. $i \leftarrow i+1, j \leftarrow j'+1$,
2-3-3. If path (σ_i) > *path* (ρ_j), *then for* $[\sigma_i, \ldots, \sigma_{i'}]$, *do step 2-3.*
2-4. If path (σ_i) = *path* (ρ_j),
2-4-1. If both actions are ListDelete, add σ_i. $\Lambda = \Lambda + \sigma_i, i \leftarrow i+1, j \leftarrow j+1$
2-4-2. If both are ListUpdate, select one to add to Λ *using RP.*
2-4-3. If both are ListInsert,
2-4-3-1. If key $(\sigma_i) \neq$ *key* (ρ_j), *then add both to* Λ.
2-4-3-2. Otherwise, merge them with tree-merging algorithm.
2-4-3-2-1. If Ti ≈ *Tj is not satisfied, then return fail.*
2-4-3-2-2. $\Lambda = \Lambda$ + *ListInsert (path* (σ_i), *Ti* ⊕ *Tj)*
2-4-3-3. $i \leftarrow i+1, j \leftarrow j+1$.

This algorithm runs $O(|\ _1| + |\ _2|)$ in step 2. If there are no key conflicts between the subtree insertions, then each step 2 requires only O (1), and therefore, an overall time of O ($|\ _1| + |\ _2|$). However, if subtree merging is needed as in step 2-4-3-2-2, then the time requirement becomes $O(|T_1|*|T_2|)$.

5. Conclusion

In this research, we focused on developing an efficient reconciliation method for mobile environments by using edit scripts of XML data sent from each mobile device. To obtain a simple model for mobile devices, we used the XML list data-sharing model, which allows insertion/deletion of subtrees only for repetitive parts of the tree based on the document type. For subtrees of repetitive parts, we also used keys, which are unique between nodes with the same parent. This model not only guarantees that the edit action always results in a valid tree, but also allows a linear time-reconciliation algorithm due to key-based list reconciliation.

The time required for the proposed reconciliation algorithm is $O(n)$, where n is the tree size. Furthermore, assuming that there are no insertion key conflicts, we can obtain the result within a time of $O(m)$, where m is the size of the edit script.

In real-world applications, XML data and usage models often agree with our list sharing model because allowing arbitrary insertion/deletion actions might render the resulting data invalid and inconsistent. Therefore, we expect that the proposed method could be used in XML data-sharing applications where efficiency is more important than data flexibility.

References

[1] G. Cabri, L. Leonardi, F. Zambonelli. “XML dataspaces for mobile agent coordination.” In Proceedings of the 2000 ACM symposium on Applied computing, 2000.
[2] S. Chawathe et al. “Change detection in hierarchically structured information.” ACM SIGMOD 1996, pp. 493-504, 1996.
[3] R. Fontaine. “Merging XML files: a new approach providing intelligent merge of XML data sets.” In Proceedings of XML Europe 2002, May 2002.
[4] A.-M. Kermarrec et al. “The IceCube approach to reconciliation of divergent replicas.” 20th Symposium on the Principles of Distributed Computing (PODC), Newport RI (USA), Aug. 2001.
[5] T. Lindholm. “Consistency and replication: XML three-way merge as a reconciliation engine for mobile data.” In Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access, pp. 93-97, 2003.
[6] C. Mascolo, L. Capra, S. Zachariadis, W. Emmerich. “XMIDDLE: A Data-Sharing Middleware for Mobile Computing.” Personal and Wireless Communications, April 2002.