Extension d'un algorithme de Diff & Merge au Merge Interactif

Xuan Truong Vu [*], Pierre Morizet-Mahoudeaux [*], Joost Geurts [*], Stéphane Crozat [**]
* UMR-CNRS 6599, Heudiasyc, Université de Technologie de Compiègne, France
** Unité Ingénierie des Contenus et Savoirs, Université de Technologie de Compiègne, France

Résumé. De nombreuses recherches visant à gérer automatiquement la fusion de différentes versions de documents textuels structurés ont été menées et communément regroupées dans le théme "*Diff & Merge* de documents XML". Nous proposons dans cet article, une alternative appelée *merge interactif*. Cette approche consiste à ne pas appliquer systématiquement la fusion automatique mais à rendre la transformation séquentielle et interactive. L'objectif est de proposer à l'utilisateur une liste d'opérations associées au document original, qu'il/elle pourra confirmer ou non, selon l'objectif de la fusion et la détection de conflits et d'incohérences.
Mots-clés. Édition collaborative, documents structurés, documents fragmentés, *diff & merge*

Abstract. Numerous works to manage automatically the *merge* of the various versions of structured textual documents have been developed and correspond to the domain of "XML documents *Diff & Merge*". We propose in this paper, an alternative named *interactive merge*. This approach consists in not applying systematically the automatic merge but in keeping the transformation sequential and interactive. The objective is to propose to the user a list of operations associated with the original document, which he/she can confirm or not, according to the objective of the fusion and the detection of conflicts and inconsistencies.

1. Introduction

L'écriture collaborative est une forme d'écriture renouvelée par le numérique. Elle est devenue une pratique importante dans le monde académique, les organisations, les entreprises, et au sein des communautés en ligne. Aujourd'hui, de nombreux documents sont issus de travaux collaboratifs : journaux, manuels techniques, présentations, articles scientifiques, cours, etc. Dans un environnement collaboratif, chaque contributeur peut ajouter, modifier et supprimer des contenus.
Si cela facilite la réalisation de documents qui exploitent au mieux les compétences de chacun, cela impose une charge supplémentaire pour maintenir la cohérence de l'ensemble et coordonner les efforts individuels. La nature de la collaboration est très variée selon la dimension et la stratégie du groupe et différentes plateformes d'édition documentaire collaborative existent sur le marché pour y répondre, dont MediaWiki ou Google Docs sont des exemples populaires. Le travail présenté dans cet article s'inscrit dans le cadre du projet ANR C2M[1] dont l'objectif est de répondre aux besoins d'écriture collaborative dans le cas de documents structurés et fragmentés : On appelle document structuré un document dont la structure logique est décrite plutôt que la mise en forme physique (André et al. 1989) ; et document fragmenté un document composé par intégration de plusieurs fragments, chacun pouvant être mobilisé pour plusieurs usages au sein de différents documents (Crozat, 2007).
En mode collaboratif et fragmenté, chaque modification d'un fragment appelle des décisions délicates en terme de répercussion pour chaque utilisateur (auteurs, lecteurs, relecteurs, co-auteurs, etc.) et pour chaque document (version avancée, version simplifiée, version papier, version écran, version de relecture, version officielle, version adaptée, etc.). La question est alors de savoir quelles décisions prendre automatiquement et/ou comment aider les utilisateurs à les prendre. Une partie de la réponse porte sur la capacité à rendre intelligibles toutes les modifications aux utilisateurs. Dans cet article nous nous penchons sur les solutions permettant de comparer a posteriori deux (ou plusieurs) versions d'un même document pour en évaluer les proximités et différences.
Dans la seconde section, nous exposerons brièvement les différentes méthodes et outils existants de *differencing* et de *merging* pour les documents XML. Dans la section suivante, nous présenterons une nouvelle approche appelée "*merge* interactif" permettant à un utilisateur de visualiser, sans

[1] http://scenari.utc.fr/c2m

ambiguïté, les différences entre deux versions d'un document et de faire des choix pour procéder à leur fusion (par l'acceptation de certaines modifications et le refus d'autres). Nous donnerons en conclusion les premiers résultats obtenus, les fonctionnalités à étudier et implémenter et enfin, les perspectives associées à ce travail.

2. Méthodes et outils de différentiel de documents XML

Lorsqu'un document numérique textuel est partagé par plusieurs auteurs, il est nécessaire de pouvoir identifier les différences qui peuvent exister entre les sources pour pouvoir les synchroniser et fusionner correctement. Les travaux relatifs à ces problèmes forment le domaine du *diff* & *merge* pour lequel il existe actuellement de multiples solutions, libres et commerciales, pour divers cas d'usage, chacune apportant sa propre approche et donc ses propres propriétés et optimisations. Certaines d'entre elles sont orientées documents textuels (e.g le format MS Doc), d'autres orientées documents structurés (e.g XML). Nous donnons ci-dessous un bref panorama de ce domaine.

2.1. Edits history

Edits history est une technique consistant à capturer toutes les actions (*edit*) de l'utilisateur sur l'éditeur et les mémoriser dans un fichier appelé *edit log*. Chaque *edit log* est donc dupliqué et transféré à d'autres utilisateurs afin de le comparer avec leurs propres *edit logs*. Comparer des documents revient à comparer des *edit logs*. Pour synchroniser des documents, il suffit de rejouer sur un document les actions transférées depuis d'autres documents. Cette méthode doit résoudre deux principaux de problèmes : premièrement, il faut capturer toutes les actions de l'utilisateur : deuxièmement, il faut s'assurer de la cohérence et la complétude du fichier. En effet, chaque *edit* va changer la position des *edits* dépendants. Un exemple d'un tel outil est Microsoft Office Groove.

2.2. Change detection

Au contraire de l'*Edit history*, *Change detection* ne requière aucune connaissance de l'histoire de l'édition du fichier. Elle cherche à déterminer, à partir des seuls fichiers courants, les changements qui ont été réalisés dans chacun d'eux. Différents algorithmes existent actuellement.

2.2.1 Les algorithmes *Line oriented* traitent tous les documents comme une série linéaire de lignes. UNIX *diff*[2] est un exemple typique et le plus connu.. Il cherche la séquence la plus longue de lignes communes entre deux fichiers. Les lignes uniques dans l'un des deux documents seront supprimées ou insérées pour passer d'un fichier à l'autre. Une variante de *diff* est *diff*3 implémentant le three-way *merge*, qui examine dynamiquement des mots et même des caractères au lieu de lignes (par exemple, google-diff-match-patch[3]). Ces outils sont très adaptés et efficaces pour traiter des documents textuels mais ils ne sont pas directement applicables à des documents structurés tel que XML ou XHTML, car *diff* n'est pas en mesure de distinguer les informations structurelles.

2.2.2 Tree oriented

Les méthodes "orientées arbre" (*tree oriented)* prennent en considération la structure d'arborescence du document. Les nœuds et les sous-arbres seront comparés et mis en correspondance à la place des lignes. Les nœuds et les sous-arbres qui ne se correspondent pas, forment les différences entre les documents. On trouvera des études comparatives et détaillées des algorithmes différentiels orientés arbre dans (Cobéna et al. 2002 ; Coneba et al., 2002 ; La Fontaine, 2003 ; Marian et al. 2001 ; Peters, 2005 ; Rönnau, 2008 ; Wang et al. 2003). Ces algorithmes sont majoritairement généralistes ou orientés données XML. Ils sont optimisés en temps d'exécution et utilisation de la mémoire. Certains algorithmes sont spécialisés pour *diff* et *merge* à la fois tandis que d'autres ne traitent que *diff*.. Après une étude exhaustive des algorithmes les plus utilisés (Vu, 2011) nous avons retenu 3DM de Tancred Lindholm (Lindholm, 2003 & 2004), qui est le plus efficace en termes de qualité et clarté des résultats obtenus et dont les sources sont directement accessibles.

---

2 http://www.gnu.org/software/diffutils/diffutils.html

3 http://code.google.com/p/google-diff-match-patch

2.2.3. Unique ID oriented

Tous les algorithmes ou outils mentionnés ci-dessus, sont essentiellement basés sur une valeur de *hash* et le contenu de chacun des nœuds pour les mettre en correspondance par un calcul de similarité (ou dis-similarité). Thao (Thao et al., 2010) a proposé une alternative au three-way *merging* consistant en *l'utilisation d'identifiants uniques*. Si chaque élément XML possède un identifiant unique, la mise en correspondance devient triviale.

2.2.4. Tree-Based textual documents

XML est utilisé non seulement pour transporter des données mais aussi pour encoder des documents textuels. Selon Angelo Di Iorio et al. (Di Iorio et al., 2009), il y a une différence entre le *diffing* d'un XML orienté document littéraire et le *diffing* d'un XML orienté données. Ils ont introduit un nouvel indicateur, *naturalness* qui reflète la capacité de l'algorithme à identifier automatiquement les changements qui pourraient être identifiés par une approche manuelle. Cependant aucune expérimentation complète ne semble avoir été réalisée avec cet algorithme.

2.3 Visualisation

Un moteur différentiel détecte des changements entre deux documents et les enregistre dans une sortie. Quel que soit le format de la sortie, il est toujours difficile pour l'utilisateur d'interpréter un changement dans le document. Il a donc besoin d'une interface de visualisation qui va permettre de mettre en évidence les changements dans leur contexte et faciliter leur manipulation.. En général, il y a deux modes d'affichage : *Side by Side* et *All In One*. Le premier mode consiste à ouvrir deux fichiers dans deux éditeurs identiques, l'un à côté de l'autre. Les différences seront surlignées respectivement dans le premier et le deuxième éditeur. Le second mode ouvre une seule vue mais y représente tous les changements. Nous avons étudié douze outils de visualisation (Vu, 2011), qui nous ont permis de proposer un outil adapté à notre approche en prenant certaines des meilleures caractéristiques dans chacun d'eux.

2.4. Approche retenue

Notre corpus documentaire est encodé en XML et valide des modèles dédiés. Nous avons donc besoin d'un outil de *diff & merge* orienté XML document. L'outil 3DM semble être le meilleur candidat, car son *tree-matcher* est efficient pour le XML généraliste et peut être encore amélioré. Il exploite toutes les opérations d'édition (e.g *update*, *insert*, *delete*, *move* et *copy*) et propose une représentation de bonne qualité des différences entre les documents. Enfin, le résultat du *three-way merge* par 3DM est en général meilleur que d'autres outils équivalents. De plus son code source est librement accessible, permettant de modifier ses modules pour réaliser nos propres optimisations.

Nous avons donc utilisé 3DM comme un framework de travail auquel nous avons ajouté des extensions spécialisées et adaptées à nos documents de façon à obtenir :

- Un *merge* interactif pour choisir, éditer des différences et résoudre des conflits.
- Une amélioration du *matching* heuristique de 3DM
- L'utilisation d'un algorithme différentiel basé sur le texte pour avoir des différences au niveau du contenu des nœuds XML.
- La visualisation des différences

3. Merge interactif

La plupart des outils de *differencing* (*diff*) et de *merging* (*merge*) fonctionnent en deux temps. L'outil de *differencing* sert à montrer en quoi deux versions sont différentes alors que l'outil de *merging* utilise ce résultat pour fusionner automatiquement les changements afin de créer une nouvelle version. Nous proposons, ici, une alternative appelée "*merge* interactif". Cette approche consiste à ne pas appliquer systématiquement la fusion automatique mais à rendre la transformation séquentielle et interactive. L'objectif est de proposer à l'utilisateur une liste d'opérations associées au document original, qu'il pourra confirmer ou non, les unes après les autres. Il doit pouvoir pré-visualiser le résultat d'une opération sur le document avant de décider de l'appliquer réellement. Le document sera modifié après chaque confirmation.

Le *merge* interactif répond à deux motivations principales : la première est qu'il permet à l'utilisateur de ne sélectionner que les changements jugés utiles pour sa propre version ; la seconde est qu'il permet de résoudre manuellement et convenablement les conflits d'un *three-way merging*.
Une raison supplémentaire concerne la visualisation des différences. Actuellement, les outils de *diff* affichent toutes les différences identifiées entre deux versions du document en même temps, ce qui permet d'avoir une vision globale de celles-ci, mais reste limitée aux trois types d'opération basiques *insert*, *delete* et *update*. D'autre part, plus le document a été changé, plus il y a des différences et plus il est difficile d'en donner une image lisible. Le *merge* interactif permet de rejouer en séquence toutes les opérations, ce qui fait perdre la vue globale mais est plus avantageux en termes d'opérations possibles (e.g. *move*, *copy*) et en termes de surcharge visuelle.
Le *merge* interactif ne produit pas lui-même la liste des opérations mais utilise celles fournies par un outil spécialisé. Selon les outils, deux sortes de listes sont disponibles : un ensemble d'opérations non-ordonnées expliquant ce qu'il se passe pendant la fusion mais qui n'est pas destiné à être exécuté ; un script d'opérations ordonnées permettant d'effectuer la fusion automatique. Cependant aucun de ces scripts ne peut être exploité tel quel, car, soit ils ne sont pas destinés à être manipulés, soit l'ordre des opérations est imposé. Avec un ensemble d'opérations non-ordonnées, il est possible de générer une séquence personnalisée d'opérations à condition de pouvoir prendre en considération le fait que certaines d'entre elles sont dépendantes de l'exécution préalable d'autres opérations.
Nous allons présenter dans la section suivante les principes d'élaboration du *merge* interactif, puis nous décrirons son implémentation dans notre prototype.

3.1. Génération des séquences d'opération

Cette section présente les aspects principaux du *merge* interactif. En particulier, elle démontre qu'il existe des relations d'ordre entre certaines opérations et qu'il est possible de recombiner dynamiquement une séquence des opérations exécutables et correctes en respectant ces relations.

3.1.1 Définitions des opérations

Nous donnons ci-dessous les définitions des opérations appliquées à un document XML qui seront utiles pour la suite. Un document XML est une structure arborescente dont les nœuds (éléments, texte, ...) sont ordonnés : modifier un document XML revient à modifier un arbre ordonné. Soit un document XML dont la structure est représentée par l'arbre T ordonné, dont les nœuds sont notés *m, n, ...*, nous définissons les opérations:

- *insert(m,k,n)* insère le nouveau nœud *n* en tant que k-ème enfant du nœud *m* ($m \in T$).
- *delete(m)* supprime totalement le sous-arbre enraciné au nœud *m* ($m \in T$).
- *update(m,n)* change la valeur initiale du nœud *m* par la nouvelle valeur *n*.
- *move(m,k,n)* enlève tout le sous-arbre enraciné au nœud *m* de sa place initiale et le déplace au dessous du nœud *n* en tant que k-ème enfant de ce dernier ($n \in T$).

Il est à noter que ces opérations ne se comportent pas toujours de la même façon. En effet, elles dépendent du type de l'objet sur lequel elles portent : un nœud de texte ou un nœud d'élément. La valeur d'un nœud d'élément est le nom de la balise et ses attributs alors que celle d'un nœud de texte est une chaine de caractères. De plus, un nœud de texte n'a pas d'enfant. L'opération *move* peut être spécialisée par une combinaison d'*insert* et *delete*, mais l'objet sur lequel elle porte n'est ni supprimé ni inséré. On pourrait ajouter *copy* à cette liste l'opération mais l'avons exclue car elle est rarement présente dans les outils, génératrice d'erreurs, difficile à gérer lorsque les nœuds ont des identifiants, et s'applique mal à certains types de documents textuels.

3.1.2. Opérations delta

Les quatre opérations *insert*, *delete*, *update* et *move* permettent d'exprimer toutes les différences entre deux versions du document. Cependant l'expression de ces différences n'est pas unique. Dans l'exemple de la Figure 1, pour passer d'un arbre T_0 à un arbre T_1, il est possible que des versions intermédiaires aient donné un arbre tel que T_{01} . Les opérations qui font passer de l'arbre T_0 à l'arbre T_{01} sont la modification du texte « a » en « aa » et l'insertion des nœuds *c* et *d* avec les textes

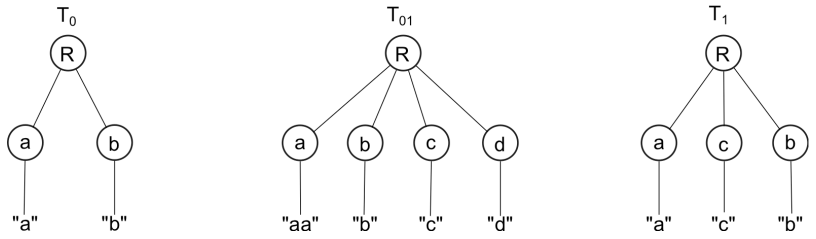

Figure 1. Etapes intermédiaire de transformation

« c » et « d ». Ensuite le nœud *c* est déplacé entre *a* et *b*, et le nœud *d* est supprimé. L'ensemble des opérations nécessaires pour passer de T_0 à T_1 forment la suite *insert*(*R*,*3*,*c*), *insert*(*c*,*1*, « c »), *insert*(*R*,*4*,*d*), *insert*(*d*,*1*, « d»), *update*(*a*, « aa »), *delete*(*d*), *move*(*R*,*2*c*)*. Supposons que seuls T_0 et T_1 soient enregistrés, pour passer de T_0 à T_1 il suffit d'insérer un nœud *c* avec le contenu « c » entre *a* et *b*, ce qui se résume aux trois opérations *insert*(*R*,*2*,*c*), *insert*(*c*,*1*, « a ») et *update*(*a*, « aa »), toutes les autres étant devenues inutiles. Ces trois opérations n'ont rien à voir avec les vraies opérations. Cela s'explique par le fait que certaines opérations de la première phase $T_0 \rightarrow T_{01}$ et certaines autres de la deuxième phase $T_{01} \rightarrow T_1$, s'appliquent aux mêmes objets. Ainsi, les résultats des premières sont annulés ou altérés par les résultats des dernières. Le résultat final est donc exprimé par d'autres opérateurs. Par exemple, ici, *insert*(*R*,*3*,*c*), *insert*(*c*,*1*, « c »), sont remplacées par *insert*(*R*,*2*,*c*), *insert*(*c*,*1*, « a ») et *insert*(*R*,*4*,*d*), *insert*(*d*,*1*, « d»), *delete*(*d*) s'annulent et ne donnent rien. L'ensemble des cas que l'on peut rencontrer sont présentés dans le **Tableau 1**.

Opérations intermédiaires annulées	**Opération identifiée à la fin**
update(m,v) puis update(m,v')	update(m,v')
update(m,v) puis delete(n), n est m ou un ancêtre de m	delete(n)
insert(n,k,m) puis delete(o), o est un ancêtre de n	delete(o)
insert(n,k,m) puis delete(m)	aucune
insert(n,k,m) puis update(m,v)	insert(n,k,m), m vaut v
insert(n,k,m) puis move(o,l,m)	insert(o,l,m)
delete(m) puis insert(n,k,m)	delete tous les enfants de n
delete(m) puis delete(o), o est un ancêtre de m	delete(o)
move(n,k,m) puis delete(m)	delete(m)
move(n,k,m) puis delete(o), o est n ou un ancêtre de n	delete(o) et delete(m)
move(n,k,m) puis move(o,l,m)	move(o,l,m)

Tableau 1. Opérations intermédiaires annulées par d'autres opérations

Les opérations *insert*(*R*,*2*,*c*), *insert*(*c*,*1*, « a ») et *update*(*a*, « aa ») ci-dessus sont appelées *opérations deltas*. Les opérations deltas ne sont pas forcément les opérations qui ont été réellement effectuées, elles utilisent des positions référencées dans l'arbre original et/ou l'arbre final pour exprimer les différences entre deux arbres. Leurs résultats sont visibles dans au moins un arbre. D'une façon générale on définit les opérations delta de la façon suivante :

- *delete*(*m*) est une opération delta si on trouve *m* dans T_0 mais non dans T_1.
- *insert*(*m*,*k*,*n*) est une opération delta si on trouve *n* en tant que k-ème enfant de *m* dans T_1 mais non dans T_0.
- *update*(*m*,v) est une opération delta si on trouve *m* dans T_0 et dans T_1 mais avec différentes valeurs (v dans T_1)
- *move*(*m*,*k*,*n*) est une opération delta si on trouve *m* dans T_0 et T_1 mais à des positions différentes.

Un *delta* Δ de T_0 à T_1 est un ensemble d'opérations *delete*, *insert*, update et *move* satisfaisant les conditions ci-dessus. Δ ne précise aucun ordre entre les opérations, elles sont suffisantes pour passer de T_0 à T_1, cependant, il faut les appliquer une par une et dans un certain ordre pour obtenir le résultat voulu. On dira que Δ de T_0 à T_1 est optimal s'il n'existe aucun Δ', sous-ensemble de Δ permettant d'aller de T_0 à T_1.

3.1.2. Relation d'ordre

La présentation des opérations delta permettant le passage d'une version à une autre n'étant que la mise en évidence de ce qui les différentie, rien n'est dit sur la possibilité de les exécuter dans un ordre quelconque pour passer effectivement d'une version à l'autre. Prenons par exemple (Figure 2.) le cas d'un article intitulé "this is the source file" qui possède initialement deux chapitres. Chacun des chapitres contient son propre titre et des blocs constitués de paragraphes possédant éventuellement un sous-titre. On a supprimé le deuxième chapitre (*delete*) tout en conservant le seul bloc qu'il contient. Ce bloc est donc déplacé (*move*) à l'intérieur du premier chapitre en tant que 3ème enfant. Ensuite, on a changé (*update*) le titre du papier qui est maintenant "this is the cible file".

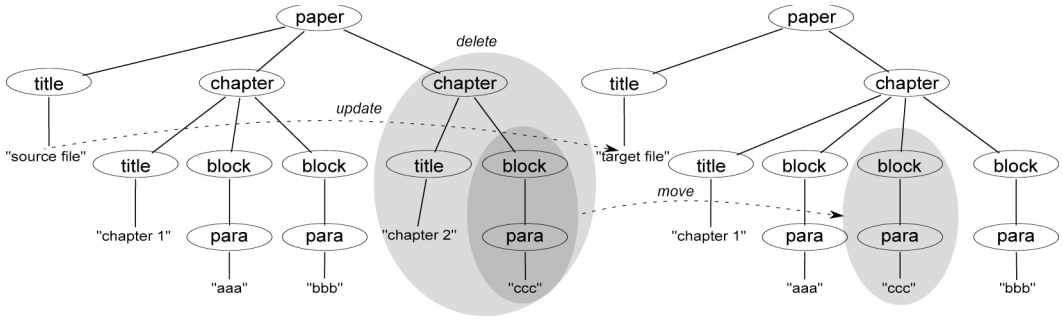

Figure 2. Modifications de "source file" à "target file"

Les opérations qui ont été effectivement réalisées sont des opérations deltas car elles sont toutes repérables sur l'arbre original et l'arbre final. Ces opérations étant a priori indépendantes, il est possible de les exécuter dans n'importe quel ordre. Cependant, en examinant le contexte de l'opération *move*, on s'aperçoit que le bloc à déplacer se trouve dans le chapitre censé être supprimé totalement. Si l'opération *delete* est exécutée avant l'opération *move*, alors tout le chapitre est supprimé, y compris le bloc. En l'absence de son objet, l'opération *move* devient non-exécutable. Ce problème ne se présente pas si l'ordre d'exécution est inversé (*move* avant *delete*). Cet exemple montre l'existence d'une relation d'ordre sur l'exécution des opérations que nous définissons ainsi :

Définition : Deux opérations sont liées par une relation d'ordre, notée ">", lorsque l'exécution de l'une nécessite l'exécution préalable de l'autre pour assurer la faisabilité et l'exactitude des deux.

Soit ω_1 et ω_2 deux opérations, alors $\omega_1 > \omega_2$ signifie que ω_1 est dépendante de ω_2 et que ω_2 est précédente de ω_1. Dans une telle relation, l'opération précédente doit s'exécuter avant la dépendante. Ceci est nécessaire mais non suffisant pour que l'opération dépendante devienne exécutable. En effet, une opération peut dépendre de plusieurs opérations précédentes. Elle n'est exécutable qu'une fois que toutes ses précédentes ont été effectuées. Il faut aussi préciser qu'une opération est susceptible d'être à la fois précédente et dépendante d'autres opérations. Par exemple, soient ω_1, ω_2, ω_3, ω_4, ω_5, des opérations telles que : $\omega_1 > \omega_2$; $\omega_1 > \omega_3$; $\omega_2 > \omega_4$; $\omega_3 > \omega_4$; $\omega_3 > \omega_5$: ω_1 est précédente de ω_2 et ω_3 , ω_2 est précédente de ω_4, ω_3 est précédente de ω_4 et ω_5, ω_4 est directement dépendant de ω_2 et ω_3, ω_4 et ω_5 sont dépendantes par transitivité de ω_1. Ainsi, ω_1 est une précédente indirecte de ω_4 et ω_5, ω_2 et ω_3 ne sont pas en relation, de même que ω_4 et ω_5 . Pour pouvoir exécuter toutes ces opérations, il faut les exécuter dans un ordre valide. Cet ordre n'est pas unique et doit prendre en compte les trois conditions suivantes :

ω_2 et ω_3 doivent être exécutées après ω_1;
ω_4 doit être exécutée après ω_2 et ω_3 ;
ω_5 doit être exécutée après ω_3.

Les opérations ω_1, ω_2, ω_3, ω_4, ω_5, forment un ensemble appelé *hiérarchie* d'opérations.

Définition : Une hiérarchie est un ensemble d'opérations dans lequel chaque opération doit être en relation d'ordre avec au moins une autre opération de cette hiérarchie. Si une opération appartient à une hiérarchie, toutes ses précédentes et ses dépendantes y appartiennent également.

Une hiérarchie peut être représentée par un graphe orienté. Une opération est représentée par un nœud qui peut avoir plusieurs prédécesseurs et plusieurs successeurs. Un arc correspond à une relation d'ordre dont le nœud sortant est la précédente et le nœud entrant est la dépendante. Une hiérarchie devient une arborescence à condition que chaque opération ait une seule précédente directe ou n'en ait pas.
Une opération qui n'est ni dépendante ni précédente d'autres opération, est appelée *indépendante*, elle n'appartient à aucune hiérarchie.
Un *delta* Δ constitué des opérations $\{\omega_1, \omega_2, \dots, \omega_n)$ peut être réécrit sous la forme $\{H_1, H_2, \dots, \omega_1, \omega_2, \dots\}$ dans laquelle H_1, H_2, sont des hiérarchies et ω_1, ω_2 sont des opérations indépendantes.
Il faut alors répondre à deux questions : une hiérarchie peut-elle posséder un cycle ? deux hiérarchies sont-elles disjointes ?
La réponse à la seconde question est triviale par la définition d'une hiérarchie. En effet, si une opération appartient à la fois à H_m et H_n, alors toutes ses précédentes et ses dépendantes aussi. Par conséquent, H_m n'est rien autre que H_n.
Pour répondre à la première question, une solution consiste à explorer toutes les relations possibles entre les opérations, puis chercher à les enchainer afin de détecter l'existence de cycles.
Une étude exhaustive nous a permis de trouver les relations de dépendance suivantes :

1. *insert(.,.,n) > insert(n,k,m)*
2. *insert(n,l,.) > insert(n,k,m) si l < k*
3. *move(n,l,.) > insert(n,k,m) si l < k*
4. *delete(o) > insert(n,k,m) si parent(o)= n : et : position(o) <= k*
5. *move(p,l,q) > insert(n,k,m) si parent(q)= n : et : position(q) <= k*
6. *insert(.,.,n) > move(n,k,m)*
7. *insert(n,l,.) > move(n,k,m) si l < k*
8. *move(n,l,.) > move(n,k,m) si l > k*
9. *delete(o) > move(n,k,m) si parent(o)= n : et : position(o) <= k*
10. *move(p,l,q) > move(n,k,m) si parent(q)= n : et : position(q) <= k*
11. *move(n,k,o) > delete(m) si o in T(m)*

Nous pouvons distinguer deux groupes : le premier comprend les relations 1, 6 et 11. Ces relations sont des conditions consistantes sans lesquelles l'exécution des opérations concernées n'est pas possible. Les relations du deuxième groupe ne conditionnent pas l'exécution des opérations mais assurent leur exactitude en termes de résultat final, c'est donc dans ce deuxième groupe que nous cherchons à annuler l'existence de cycles. En fait, ils apparaissent dans les relations 4, 5, 9 et 10 et cela est dû au fait d'utiliser la position exacte, dans l'arbre final, du nœud inséré ou déplacé.
Une première solution consiste à "relativiser" le paramètre *k* dans la définition des opérations *insert* et *move*. La valeur de *k* n'indique pas la position dans l'arbre final, du nœud inséré (ou déplacé) mais indique la position dans l'arbre actuel où le nœud est inséré (ou déplacé). Cela permet d'exécuter correctement les *inserts* et *moves* sans dépendre des autres opérations. Pour chaque *insert* et *move*, il faut donc recalculer *k* en fonction du nombre de *deletes* et *moves* sortants ainsi que du nombre d'*inserts* et *moves* entrants pour les nœuds du même parent et situés à gauche du nœud *m*. Chaque *insert* ou *delete* ou *move* effectué change la liste des enfants. Il faut donc également changer la valeur du *k* des *insert* ou *move* entrant. Concrètement, après l'insertion ou le déplacement d'un nœud dans la liste des enfants, les positions des nœuds les plus à droite à insérer ou à déplacer, doivent être incrémentées de 1; après la suppression et le déplacement d'un nœud en dehors de la liste des enfants, les mêmes positions précédentes doivent être décrémentées de 1 (Figure 3).
Cette première solution permettra d'annuler toutes les relations d'ordre du deuxième groupe. Il nous reste donc les relations du premier groupe 1, 6 et 11. En enchaînant ces trois relations, on peut

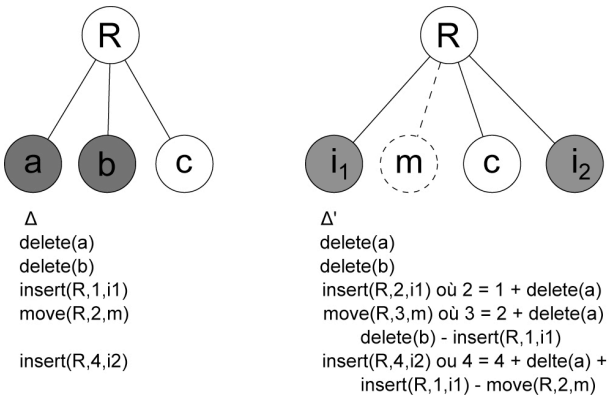

Figure 3. k est recalculé pour assurer l'exécution correcte des opérations

uniquement obtenir une séquence telle que : *insert* supérieur > *insert* inférieur > *move* ou encore inférieur > *delete* (contenant l'objet du *move*) qui ne forment jamais de cycle.
La deuxième solution, plus radicale, consiste à redéfinir *insert* et *move* : au lieu d'insérer ou déplacer un nœud à une position précise *k*, on peut l'insérer ou le déplacer après un nœud *left* :

- *insertAfter(n,left,m)* insère le nœud *m* après le nœud *left* qui est un enfant du nœud *n*
- *moveAfter(n,left,m)* déplace le nœud *m* après le nœud *left* qui est un enfant du nœud *n*

Le nœud *n* est nécessaire car si *left* est égal à null, le nœud *m* est inséré ou déplacé au dessous du nœud *n* en tant que premier enfant. L'*insertAfter(n,null,m)* et le *moveAfter(n,null,m)* sont donc possibles si et seulement si le nœud *n* est présent au moment de l'exécution. Le nœud *n* peut être lui-même l'objet d'un *insertAfter* supérieur. Il faut donc que ce dernier soit préalablement exécuté.

- *insertAfter(.,.,n) > insertAfter(n,null,m)*
- *insertAfter(.,.,n) > moveAfter(n,null,m)*

Dans le cas contraire, l'*insertAfter(n,left,m)* et le *moveAfter(n,left,m)* nécessitent que le nœud *left* soit présent au moment de l'exécution. Il se peut que ce dernier soit également l'objet d'une autre opération *insertAfter(n,left.left,left)* ou *moveAfter(n,left.left,left)* sachant que le nœud *left.left* peut être égal à *null*. Il faut donc exécuter les *insertAfter* et *moveAfter* l'un après l'autre en commençant par les nœuds plus à gauche.

- *insertAfter(n,left.left,left) > insertAfter(n,left,m), left ≠ null*
- *moveAfter(n,left.left,left) > insertAfter(n,left,m), left ≠ null*
- *insertAfter(n,left.left,left) > moveAfter(n,left,m), left ≠ null*
- *moveAfter(n,left.left,left) > moveAfter(n,left,m), left ≠ null*

Il reste à démontrer l'inexistence de cycles. Parmi les trois opérations, seules *insertAfter* et *moveAfter* peuvent être à la fois dépendantes et précédentes d'autres opérations alors que le *delete* est uniquement dépendante, ce qui implique qu'un cycle, s'il existe, contiendrait seulement des opérations *insertAfter* et *moveAfter*. L'insertAfter et le moveAfter sont pratiquement identiques en termes de relations avec d'autres opérations, il suffit donc d'examiner l'une des deux. L'insertAfter

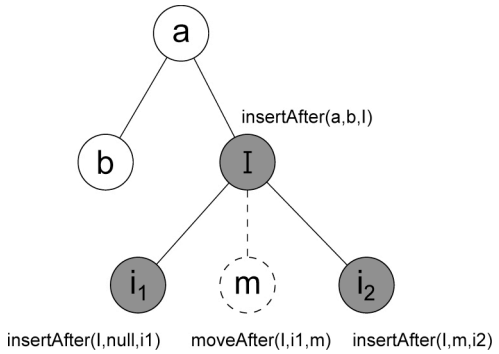

Figure 4. InsertAfer dépend d'InsertAfter à gauche et d'InsertAfter supérieur :
insertAfter(a,b,I) > insertAfter(I,null,i1) > moveAfter(I,i1,m) > insertAfter(I,m,i2)

peut dépendre de l'insertAfter (ou moveAfter) à gauche qui peut également dépendre de l'insertAfter (ou moveAfter) encore plus à gauche. Si le nœud inséré (ou déplacé) de ce dernier est le premier enfant de son parent, l'insertAfter (ou moveAfter) dépend de l'insertAfter supérieur qui à son tour peut dépendre d'un insertAfter encore supérieur ou d'un insertAfter (ou moveAfter) à gauche. Le processus peut continuer ainsi de suite, mais dans une seule direction, vers le haut de l'arborescence (Figure 4). Il n'y aura jamais d'opération insertAfter (ou moveAfter) qui dépende du tout premier insertAfter, il n'y aura donc pas de cycle.
Ces deux solutions permettront d'éliminer les cycles et d'assurer l'exécution correcte des opérations. La première solution ne change pas la définition des opérations mais elle est compliquée à cause des calculs qu'elle génère. La deuxième solution n'exige pas de calcul, mais elle nécessite d'enregistrer aussi l'information relative au nœud à gauche pour toutes les opérations insertAfter et moveAfter.

3.1.3. Principe d'acceptation et de refus

Une opération est exécutable quand toutes ses conditions d'exécution sont satisfaites et ne l'est pas si au moins une condition n'est pas satisfaite. Accepter une opération ne rend pas tout de suite exécutable ses opérations directement dépendantes mais permet de satisfaire l'une des conditions d'exécution de celles-ci. Au contraire, le fait de refuser une seule opération aura éventuellement un effet en cascade sur plusieurs autres opérations. Par simplicité, on peut dire que refuser une opération signifie refuser toutes ses opérations dépendantes (directement ou par transitivité). Ce n'est pas tout à fait correct car il faut prendre en considération la nature de chaque condition. Comme indiqué ci-dessus, il y a deux groupes de relations. Le premier est lié à la possibilité de l'opération (est-elle possible ?), alors que le deuxième concerne l'exactitude de l'opération (est-elle correcte ?). Si la condition insatisfaite est du premier groupe, le refus en cascade est inévitable. Par contre, s'il s'agit d'une condition du deuxième groupe, l'opération reste exécutable mais sera incorrecte. Cela demande donc de modifier les paramètres des opérations dépendantes :

- Refuser un *insert* ou *move* d'un nœud implique dés-incrémenter la position des *inserts* et *moves* plus à droite,
- Refuser un *insertAfter* ou *moveAfter* d'un nœud implique changer le paramètre 'nœud à gauche' de l'*insertAfter* ou *moveAfter* à droite par le nœud à gauche de l'opération refusée.

3.1.4 Opération réversible

Durant l'exécution des opérations, il est parfois nécessaire d'inverser (ou annuler) des opérations acceptées. Les opérations, telles que définies, ne sont pas inversibles parce que dans leur définition, il manque des données complémentaires. Par exemple, l'opération *delete*(*m*) supprime un sous-arbre enraciné au nœud *m*. L'inverser nécessite une opération *insert*, mais on ne sait pas où exactement réinsérer le sous-arbre supprimé. Marian et al., (Marian et al., 2001) ont défini des *opérations complètes* (*completed delta*) permettant non seulement d'exécuter mais aussi d'inverser l'opération. Il est également à noter qu'inverser une opération implique rend insatisfaites des conditions relatives dans lesquelles l'opération joue le rôle de précédent.

3.1.5. Algorithme de mise en ordre des opérations

La remise en un ordre valide des opérations pour un *merge* interactif n'est pas strictement nécessaire. En effet, durant le *merge*, l'utilisateur peut ne sélectionner que les opérations exécutables quel que soit leur ordre. En revanche, si l'opération fait partie d'une hiérarchie, il est pratique d'en percevoir immédiatement les opérations dépendantes et d'enchaîner toute la hiérarchie. Ces actions ne sont plus triviales quand les opérations relatives sont dispersées, il faut donc pouvoir les mettre ensemble. En particulier, si nous voulons être en mesure de les rejouer en séquence, il faut trouver un ordre valide pour en assurer le bon fonctionnement. Un ensemble d'opérations non-ordonnées contient éventuellement plusieurs hiérarchies distinctes et mélangées avec des opérations indépendantes. Notre algorithme de mise en ordre des opérations cherche itérativement à remettre celles appartenant à une même hiérarchie ensemble puis à placer une opération après ses opérations précédentes. Dans un premier temps, l'opération précédente est remontée devant l'opération

dépendante si ce n'était pas déjà le cas. Une fois que toutes les opérations précédentes sont placées devant les opérations dépendantes, il met cette dernière juste après sa précédente dans la liste. La terminaison de l'algorithme est assurée grâce à la caractéristique acyclique des hiérarchies.

3.2. Implémentation

Le *merge* interactif ne produit pas lui-même les opérations permettant la transformation entre les versions du document. Il récupère la liste des opérations enregistrées par 3DM mais ne les exploite pas immédiatement car il faut examiner préalablement les relations éventuelles entre les opérations telles que présentées dans la section précédente. La première implémentation mobilise également un algorithme de comparaison de texte, pour compléter 3DM. Elle a été testée sur les documents au format XML issus de notre base documentaire.

3.2.1. Algorithmes utilisés

En *merge* interactif, nous voulons être en mesure de retracer et visualiser des changements à la fois au niveau de la structure de l'arborescence du document et dans son contenu textuel. Pour ce faire, nous avons eu recours à deux algorithmes complémentaires : 3DM de Lindohlm, destiné à examiner la structure d'arbre de XML et Google-diff-match-patch, pour différencier des textes. Ce dernier va mettre en évidence les mots et même les caractères du contenu d'un nœud texte qui ont été insérés ou supprimés. 3DM enregistre toutes les opérations dans un fichier de format XML appelé *edit log*. L'ordre d'enregistrement des opérations est celui de l'insertion des nœuds en vue de construire l'arbre final. Chaque enregistrement correspond à une opération précise, les propriétés de l'opération sont décrites par des paires "attribut-valeur", qui indiquent la position des nœuds impliqués par l'opérateur dans l'arbre original et final, du parent adoptif d'un nœud déplacé ou inséré. Les informations nécessaires au parcours des arbres pour retrouver les nœuds impliqués par les opérations sont encodées. La version actuelle du *merge* interactif n'implémente que les relations du premier groupe, elle ne traite pas encore des cycles. Le positionnement de certains nœuds insérés ou déplacés peut donc être inexact. Dans la prochaine implémentation, pour éliminer les cycles, l'une des deux solutions mentionnées ci-dessus sera utilisée, la deuxième solution étant préférable, l'information sur le nœud à gauche d'un nœud inséré (ou déplacé) pouvant être enregistrée par 3DM.

3.2.2. Vue globale de l'implémentation

Le premier prototype du *merge* interactif a été réalisé en Java. La classe principale est un JFrame, qui contient un panel de la classe *InteractiveMergePanel*. Ce dernier joue à la fois le rôle de vue et de contrôleur. Elle présente les données (des opérations, la structure et le contenu textuel du document XML). Elle reçoit les actions de l'utilisateur et les traite. Les données sont réellement modifiées par la classe *Merge*.

L'*edit log* est modélisé par la classe *EditLog*, qui contient la méthode *sort*, qui implémente l'algorithme de remise en ordre des opérations. L'opération est stockée dans l'objet *Operation*, qui possède la méthode *isBelongTo* afin d'examiner si une opération est dépendante d'une autre opération. La classe *Path* est utile pour manipuler les paths des nœuds de 3DM.

Les fichiers XML sont parsés et traités par un parser de type DOM qui crée pour chacun des fichiers, un objet d'arbre interne facile à accéder et à modifier. Pour assigner cet objet d'arbre à un Swing JTree destiné à s'afficher sur l'interface, nous avons utilisé les classes *XMLTreeNode* et *XMLTreeModel* de Rob Lybarger[4] La classe *TreeCellCustomRenderer* permet de changer l'affichage de l'arbre.

3.2.3. Interface graphique du merge interactif

L'interface principale du *merge* interactif (Figure 5) est constituée de trois panneaux :

1. Le premier panneau affiche la liste des opérations regroupées en hiérarchies. Une opération est représentée par son type, le nœud concerné et d'autres paramètres. Les opérations activées sont susceptibles de s'exécuter immédiatement. Les opérations désactivées sont

[4] http://www.developer.com/xml/article.php/3731356/Displaying-XML-in-a-Swing-JTree.htm

dépendantes. Elles doivent attendre l'exécution de leurs opérations précédentes afin d'être activées et exécutables. L'utilisateur peut choisir les opérations activées pour les exécuter l'une après l'autre dans n'importe quel ordre.

2. Le deuxième panneau représente la structure d'arborescence interactive du document XML. Cet arbre s'étend à tous les nœuds internes et non aux feuilles textuelles. En cliquant sur un nœud, le contenu textuel de ce nœud est affiché dans le troisième panneau.
3. Le troisième panneau affiche le contenu textuel du document XML dans un format purement textuel. Les titres (du papier, du chapitre, de la session, ...) sont en gras et les paragraphes sont espacés.

Ce premier prototype a été utilisé sur des documents académiques réels de notre base documentaire et a permis d'en tester les différentes fonctionnalités (parcours d'arborescence, sélection d'opérations ou de séquences d'opération, validation de séquences dépendantes ou annulation, retour en arrière, prévisualisation de l'effet d'un opérateur, visualisation de modifications de textes, …) et d'évaluer la qualité de leur usage, tant du point de vue opérationnel que de la convivialité de l'interface. Il a été testé sur des documents dont la structure pouvait atteindre jusqu'à 17 niveaux d'éléments, contenant des balises iline de mise en forme, de référence ou de l'hyperlien et sur lesquels 3DM avait enregistré plus de 60 opérations de modifications

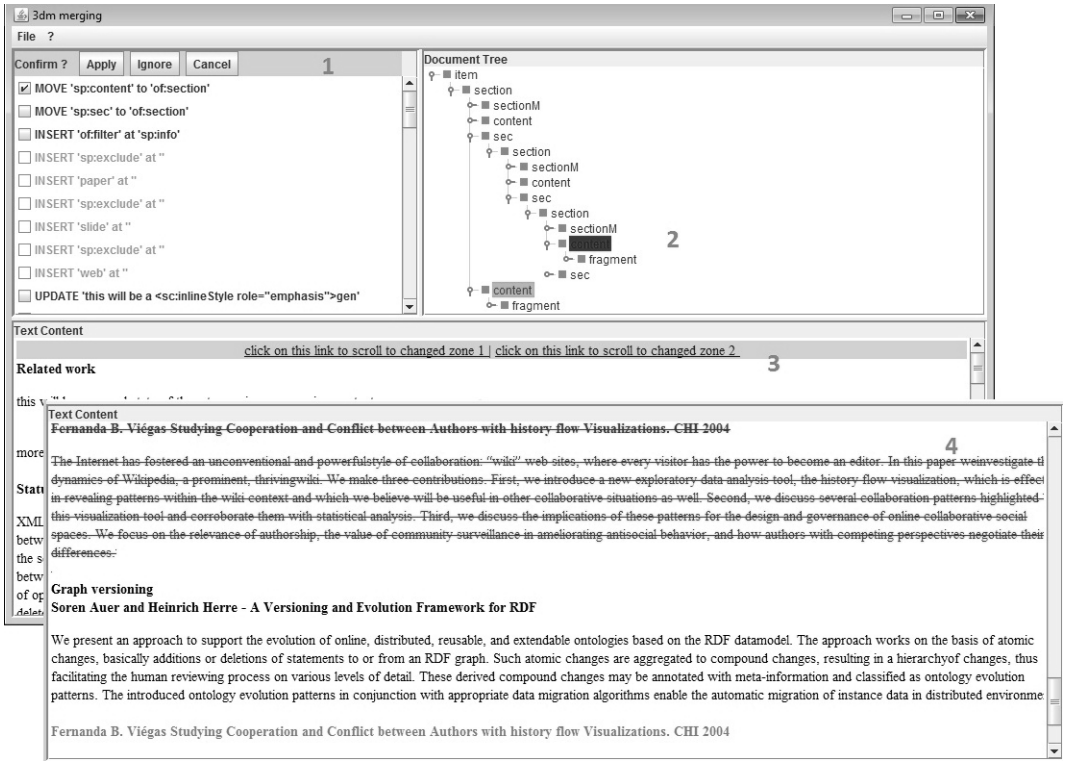

Figure 5. Interface principale du merge interactif : 1 - Liste d'opérations, avec pop-up de prévisualisation ; 2 - Structure d'arbre du document XML et mise en couleur des modifications ; 3 - Contenu textuel du document XML et mise en couleur des modifications.

4. Conclusions et perspectives

Nous avons présenté dans cet article une extension des algorithmes de *diff* & *merge* sur des documents structurés qui permet de réaliser les opérations de *merge* de façon interactive. Après avoir redéfini les opérateurs qui permettent de caractériser les différences entre deux versions d'un document, tant sur leur structure que sur leur contenu, nous avons défini une relation d'ordre qui permet de proposer des séquences d'exécution cohérentes et exécutables de ces opérateurs. Nous avons pu en particulier montrer qu'il était possible, grâce à ces nouvelles définitions d'opérateurs, de s'affranchir en grande partie du risque de boucles et d'incohérence dans l'exécution des

opérations. Nous avons enfin proposé une implémentation de l'algorithme et une interface d'exploitation qui permet à l'utilisateur de sélectionner les opérations valides ou à rejeter et de les exécuter en visualisant leur effet sur les documents. Les choix heuristiques et la gestion de conflits lors de la fusion de plusieurs éditions d'un même document s'en trouvent alors améliorés.

L'implémentation actuelle du prototype est encore limitée dans son applicabilité. Les fonctionnalités à étudier et implémenter incluent :

- Gérer les conflits lors du three-way *merge* : la version actuelle est limitée à un fichier et sa modification, alors qu'il faudrait pouvoir traiter deux versions différentes d'une même source. 3DM fournit les informations nécessaires pour réaliser ce traitement en interactif.
- Exploiter le schéma du document : seules les structures XML ont été prises en compte, or on pourrait intégrer la sémantique issue des modèles documentaires des chaînes éditoriales Scenari (Crozat, 2007) mobilisées dans le projet C2M.

- Comparer des réseaux de fragments : un fragment contient des contenus et des références à d'autres fragments. Le fragment racine est le fragment qui n'a pas de parent et inclut par transitivité tous les fragments du document. Comparer deux documents revient à comparer deux réseaux de fragments, ce qui n'est pas possible directement avec 3DM. Cependant, on peut inclure les contenus de tous les fragments dans le fragment racine en vue de créer un seul fichier, puis comparer ces fichiers et appliquer le *merge* interactif. Lors de l'enregistrement du fichier, il faudra re-fragmenter le document résultant.

Références

André J., FurutaR., Quint V., Structured documents, Cambridge University Press, 1989.

Coneba G., Adbessalem T. , Hinnach Y. A comparative study for XML change detection, Research Report, INRIA, 2002

Cobéna G., Abiteboul S., Marian A., Detecting Changes in XML Documents, Proceedings of the 18th International Conference on Data Engineering, 41-52. Feb. 2002.

Crozat S., Scenari, la chaîne éditoriale libre, Eyrolles, 2007.

Di Iorio A., Schirinzi M., Vitali F., Marchetti C., A Natural and Multi-layered Approach to Detect Changes in Tree-Based Textual Documents, In Proceedings of ICEIS'2009. pp.90-101.

La Fontaine R., DeltaXML, Change Control for XML : Do It Right XML Europe, May 2003.

Lindholm T., XML three-way merge as a reconciliation engine for mobile data, Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access, 93-97, Sept. 2003.

Lindholm T., A three-way merge for XML documents, Proceedings of the 2004 ACM symposium on Document engineering, 1-10, Oct. 2004.

Marian A., Abiteboul S., Cobéna G., Mignet L. Change-Centric Management of Versions in an XML Warehouse Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

Peters L. Change Detection in XML Trees : a Survey In : third Twente Student Conference on IT ; June 2005

Rönnau S., Pauli C., Borghoff U.M., Merging changes in XML documents using reliable context fingerprints, Proceeding of the eighth ACM symposium on Document engineering, September 16-19, 2008, Sao Paulo, Brazil.

Thao C., Ethan V. Munson E.V., Using Versioned Tree Data Structure, Change Detection and Node Identity for Three-Way XML Merging, DocEng2010, September 21-24, 2010, Manchester, United Kingdom.

Vu X.T., Merging Interactif de Documents XML, rapport de Master mention Science et technologies de l'Information et de la Communication, Université de Technologie de Compiègne, juin 2011.

Wang, Y., DeWitt D.J., Cai, J. : X-Diff, An Effective Change Detection Algorithm for XML Documents, 19th International Conference on Data Engineering, 519-530. Mar. 2003.