Université de Technologie de Compiègne
Unité Ingénierie des Contenus et Savoirs
Laboratoire Heudiasyc, UMR CNRS 6599

RAPPORT de MASTER

Merging Interactif des Documents XML

Xuan Truong VU

Compiègne, 1er juillet 2011

Sous la direction de

Prof. Pierre MORIZET-MAHOUDEAUX
Prof. Stephane CROZAT

XML Document Interactive Merging

Xuan Truong VU

1er juillet 2011

Abstract

eXtensible Markup Language (XML) has become the standard representation format for structured documents in different fields. When several authors are expected to work collaboratively, it is very important for one to know what others have done and to merge, integrate consistently their effort into a sigle work. This task can be done manually for relatively small contributions, but quickly becomes time consuming and error prone if the contributions are comprehensive. Therefore, an automatic tool is desired. Substantial research to this end have been carried out and commonly grouped by the theme "XML Document Diff & Merge". Although time and memory efficient implementations exist, most of them are generic and data oriented. As a result they are not necessarily appropriate for text-based documents, or fragmented documents, such as used by SCENARI, as they lack the logic of presentation, usage and decision for end-users, authors.

After a comparative study of various existing implementations, we chose one of them, "XML 3-way Differencing and Merging Tool". On the latter, we made improvements and developed what we called "interactive merge" which objective is to make intelligible and interactive merging for text-based XML documents. The experiments were performed on real Scenari documents. Based on the results of our experiments, we propose some optimizations to increase the performance and the quality of structured & fragmented document diff & merge.

Keyworks : Tree, HTML, XML, Document, Version, Tree-matching, Differencing, Merging, Visualization.

Merging Interactif des Documents XML

Xuan Truong VU

1er juillet 2011

Résumé

L'eXtensible Markup Language (XML) est devenu le format standard de représentation des documents structurés en différents domaines. Quand plusieurs auteurs sont censés travailler collaborativement, il est très important pour l'un de savoir ce qu'ont fait les autres ainsi que de fusionner, d'intégrer leurs efforts dans un seul travail de manière consistante. Cette tâche peut être manuellement accomplie, mais elle devient rapidement source d'erreurs et de perte de temps dès lorsque les contributions sont complètes. Il fallait donc posséder un outil automatique. De nombreuses recherches visant à cette fin ont été menées et communément regroupées par le thématique "Diff & Merge documents XML". Bien que les implémentations optimisées pour le temps et la mémoire existent, la plupart d'elles restent généralistes ou orientés données, très peu sont destinées à documents basés texte ou des documents fragmentés. En plus, elles manquent des logiques de présentation, manipulation et décision pour et par des utilisateurs finaux, les auteurs.

Après une étude comparative de différentes implémentations existantes, nous avons choisi l'une d'entre elles, "XML 3-way Merging and Differencing Tool". C'est sur ce dernier que nous avons réalisé des améliorations et développé une approche appelée "Merge interactif" consistant à rendre intelligible et interactif la fusion des documents basés texte. Les expérimentations ont été effectuées sur les vrais documents issus du modèle OPTIM (Scenari). A partir de nos premières expérimentations, nous proposons des optimisations permettant d'augmenter la performance et la qualité de la comparaison et la fusion des documents structurés et fragmentés.

Mots clés : Arbre, Document, Version, HTML, XML, Tree-matching, Comparaison, Fusion, Visualisation

Remerciement

Ce travail n'aurait pas été possible sans le soutien des personnes de l'Unité Ingénierie des Contenus et Savoirs à l'Université de Technologie de Compiègne. Je tiens à remercier particulièrement à mes responsables de stage, professeur Pierre Morizet Mahoudeau et professeur Stéphane Crozat pour leur encadrement idéal, leurs conseils précieux et surtout leur confiance. Je remercie également Joost Geurt, Lydie Edward et d'autres membres du projet C2M pour tous les échanges enrichissantes et bénéfiques tout le long de la période du stage.

Et enfin, je voudrais exprimer mes reconnaissances sincères à toutes les enseignants et toutes les amis qui m'ont beaucoup aidé pour mes études à l'UTC (Compiègne, France).

Table des matières

Liste des notations

$R, a, a', a_1, b, ..., m, n, ...$	des noeuds de l'arbre
(m, n)	un arc relie le noeud m au noeud n
$T, T_0, T_1, ...$	des arbres, des versions du document XML
$T(n)$	un sous arbre enraciné au noeud n
$a, b, ...$	un ensemble
$\emptyset$	un ensemble vide
$fonction(.)$	une fonction
$\omega, \omega_i, \omega_j, ...$	des opérations d'édition
H	une hiérarchie des opérations d'éditions
$N(.)$	le nombre de
.	représente une variable, un paramètre, une valeur

Liste des acronymes

XML	eXtensible Markup Language
HTML	HyperText Markup Language
XHTML	reformulation de HTML dans XML
W3C	World Wide Web Consortium
3DM	3-way Differencing & Merging Tool
GUI	Graphical User Interface
IHM	Interface Humain Machine
DTD	Document Type Definition
API	Application Programming Interface

Chapitre 1

Introduction générale

1.1 Écriture Collaborative

L'écriture collaborative est une forme d'écriture collective par plusieurs personnes qui décident de se mettre ensemble afin de produire un document plus complexe et plus riche. L'écriture collaborative est devenue un pratique dominant dans l'académie, dans les organisation, les entreprises, au sein des communautés en ligne et en d'autres domaines. Aujourd'hui, la plupart des documents sont produits des travaux collaboratifs. Les journaux, les manuels techniques, les présentations, les articles scientifiques, les cours ne sont que quelques exemples de l'écriture collaborative.

Dans un environnement réellement collaboratif, chaque contributeur a une capacité presque égale à ajouter, à modifier et à supprimer des contenus. L'écriture collaborative est plus facile si le groupe a un objectif précis en tête et plus difficile si le but est absent ou tellement vague. Elle permet un meilleur résultat en terme de document final, car elle permet de réduire le temps d'exécution des tâches, de réduire des erreurs, de varier des points de vues et compétences. Mais de l'autre côté, elle impose une charge supplémentaire non négligeable pour maintenir la cohérence et la coordination des efforts individuels compte tenu de la différence culturelle et de la distance des interlocuteurs.

La nature de la collaboration est très variée selon la dimension et la stratégie du groupe, le rôle de chaque contributeur et la relation entre les contributeurs. Certains projets d'écriture collaborative peuvent être supervisés par un rédacteur ou un groupe de rédacteurs, d'autres se développent sans aucun contrôle. De nombreux différents plateformes documentaires collaboratifs existent sur le marché. Chacune a ses propres aspects techniques et est dédié à des communautés spécifiques. Prenons nous deux exemples incontournables, parmi d'autres, Wiki et Google Docs.

Wiki
Un wiki est un site web dont les pages sont ouvertes et modifiables par tout ou une partie des visiteurs du site. Wiki est l'un des premiers systèmes permettant l'écriture collaborative et la contribution des connaissances via l'Internet. On accède à un wiki, en lecture comme en écriture, avec un navigateur classique. On peut visualiser les pages dans deux modes différentes : le mode lecture, qui est les mode par défaut, et le mode écriture, qui présente la page sous une forme modifiable. En mode écriture, le texte de la page, affiché dans un formulaire web, s'enrichit d'un certain nombre de caractères supplémentaires, suivant les règles d'une syntaxe informatique particulière : le wikitexte, qui permet d'indiquer la mise en forme du texte, de créer des liens, de disposer des images, etc. Certains wikis permettent à un contributeur enregistré de suivre l'évolution d'une page, ou les contributions d'une personne en particulier, ou toutes les créations de page par exemple. Ces

suivis permettent de réagir rapidement à des actes de vandalismes, ou de spam. Un historique des révisions est mis à disposition en ligne pour que l'utilisateur puisse comparer deux versions consécutives, consulter et reculer à une version antérieure. La communauté wiki est très large. Quelques wiki sont complètement ouvert au public, mais la plupart est privé. En particulier au sein des entreprises, ils servent la documentation interne. Le plus consulté de tous les wikis est l'encyclopédie libre Wikipédia [1].

Google Docs
Google Docs est une suite d'applications en ligne de Google. Des documents, des feuilles de calcul et des présentations peuvent être créés en ligne, importés via l'interface web ou envoyés par e-mail. Les documents sont automatiquement sauvegardés sur les serveurs de Google afin de prévenir la perte de données, et un historique de révisions est automatiquement maintenu. Les documents peuvent être partagés, ouverts et édités par plusieurs utilisateurs de Google en temps réel. Les changements effectués sont immédiatement visualisés par d'autres utilisateurs. L'utilisateur hors-ligne est informé des changements du document par e-mail. L'utilisateur peut créer une discussion (chat) sur une partie de contenu du document. Google Cloud Connect est un plug-in pour Microsoft Office (2003, 2007 et 2010) qui peut stocker automatiquement et synchroniser un document Microsoft Word, une présentation PowerPoint ou un feuille de calcul Excel via une copie sur Google Docs. La copie sur Google Docs est automatiquement mise à jour chaque fois que le document Microsoft Office est enregistré. Google Sync Cloud maintient les versions précédentes de Microsoft Office document et permet à plusieurs utilisateurs de collaborer en travaillant sur le même document en même temps. La popularité de Google Docs, parmi les entreprises, est de plus en plus due au renforcement des fonctions de partage et d'accessibilité. En outre, Google Docs a connu une augmentation rapide de la popularité parmi les étudiants et les établissements d'enseignement.

1.2 Contexte du stage

Le travail présenté dans ce rapport s'inscrit dans le cadre du projet ANR C2M qui cherche à répondre aux besoins de l'écriture collaborative des documents structurés et fragmentés. *"Le projet se positionne à l'intersection de deux révolutions - technologiques et d'usages - centrales dans le contexte des mutations documentaires liées à l'avènement du numérique.*

La première de ces révolutions est celle de la chaîne éditoriale, initiée au début des années 80 au sein des entreprises ayant de forts enjeux de documentations techniques (aéronautique par exemple), et démocratisée par le standard XML d'une part et des outils qui permettent un accès facilité aux technologies d'autre part. Cette révolution permet de pousser la puissance éditoriale du document numérique et de dépasser les pratiques bureautiques : multi-supports, réutilisation sans recopie, rééditorialisation, intégration multimédia, etc.

La seconde de ces révolutions est celle du documentaire collaboratif, développé dès les années 80 à travers la GED, prolongé dans les années 90 par les CMS Web, et consacré au milieu des années 2000 par l'ECM en entreprise d'une part, et les usages grand public dit "Web 2.0" d'autre part. La force de ces nouveaux usages est avant tout la démocratisation des outils de création et de diffusion : chacun peut facilement écrire numérique et mettre en ligne. Elle est également vecteur d'une interrogation intéressante sur les acteurs et les pratiques : les frontières classiques entre auteurs, lecteurs, éditeurs, rédacteurs ou contributeurs tendent à se reconfigurer.

1. http://fr.wikipedia.org/wiki/Wikipedia

L'objectif du projet est celui d'une technologie permettant à la fois la réalisation d'une documentation très qualitative, respectant les exigences de contextes professionnels pour lesquels le document a une valeur essentielle (documentation technique, formation, etc.) ; et des usages collaboratifs permettant à des communautés de s'organiser autour de cycles innovants de gestion de l'information : production, maintenance, qualification, diffusion, etc." selon [33]

1.3 Problématique de fragmentation en contexte collaboratif

Selon [33], l'enjeu du projet repose sur le fait qu'il cherche à cohabiter deux paradigmes totalement différents :

1. *"Le modèle dominant mobilisé pour représenter un document numérique est de type "**1 document = 1 fichier**". Les systèmes de GED ou ECM s'appuient massivement sur ce principe.*
2. *Les chaînes éditoriales annule cette identité entre document et fichier. Elles consacrent le principe de fragmentation en faisant la règle "**1 document = N fragments**". Un document est une agrégation de plusieurs fragments dont sa forme lisible est obtenue par une reconstruction dynamique à partir des liens entre les fragments.*

La complexité naît alors de deux axes orthogonaux :

1. *Dans une logique collaborative, chaque fragment peut être modifié par plusieurs auteurs (complexité habituellement gérée par des mécaniques de gestion que connaissent les systèmes d'ECM : check-in/check-out, versionning, historisation, etc.) ;*
2. *Mais chaque document est également impacté par la modification de chacun de ses fragments. La rupture de l'identité fichier-document introduit une complexité nouvelle qui entre en résonance avec la première.*

Chaque modification appelle donc des décisions délicates : répercussion des modifications sur quelles catégories d'utilisateurs (auteurs, lecteurs, relecteurs, co-auteurs, etc.) ? Sur quels documents (version avancée, version simplifiée, version papier, version écran, version de relecture, version officielle, version adaptée, etc.) ? 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.

1.4 Objectif du stage & Organisation du rapport

Dans le cadre de mon stage de Master, j'ai travaillé sur les solutions permettant de comparer a posteriori deux (ou N) versions d'un même document pour en évaluer les proximités et différences :

- Étudier les algorithmes de différentiels existantes (e.g les algorithmes de differencing et de merging)
- Chercher les logiques d'optimisation fonctionnelle en s'appuyant sur la mécanique des primitives Scenari (méta-modèle documentaire)
- Proposer des logiques de présentation, manipulation et décision pour et par les utilisateurs finaux, les auteurs

Ce rapport est divisé en quatre parties. Dans un premier temps, je présenterai la terminologie utilisée à travers ce rapport. Dans un second temps, j'exposerai une étude globale des différents méthodes et outils existantes de differencing et de merging pour les documents XML. Dans la partie suivant, je présenterai mes contributions, en particulier une nouvelle approche appelée "merge interactif" permettant à l'utilisateur de visualiser, sans ambiguïté, des différences et de faire des bons choix de fusion (par l'acceptation de certaines modifications et le refus d'autres). Pour finir, je présenterai les conclusions et les perspectives associées à ce travail.

Chapitre 2

Préliminaires

Avant aborder les différents outils et méthodes existants de differencing et de merging pour les documents XML, il est nécessaire de comprendre des concepts importants qui seront beaucoup utilisés à travers ce mémoire. Nous allons donner ci-dessous les définitions des différentes notions que nous allons utiliser.

2.1 Document Versioning

Dans les systèmes de fichiers, un fichier passe toujours par plusieurs étapes d'avancement avant qu'il prenne sa forme finale. Chaque étape est appelée donc une *version* ou une *révision* selon le système de gestion. Il est nécessaire et important de conserver toutes les différentes versions afin de vérifier l'élaboration du fichier final. Elles sont uniquement numérotées et archivées soit dans un seul dépôt centralisé (e.g CVS, SVN) , soit dans différents dépôts répartis (e.g Mercurial, Darcs, ...).

Il y a deux stratégies pour maintenir les versions dans la base. Premièrement, on archive toutes les versions en tant que telles. Deuxièmement, on ne garde que la version de travail modifiable (*live document*) et les fichiers *delta* permettant, au moyen d'un programme spécialisé, de revenir à une version plus ancienne. Chacune des deux approches a ses avantages mais aussi ses inconvénients. L'archivage des deltas économise la mémoire mais il faudrait avoir un convertisseur précis et efficace pour retrouver une version quelconque. L'archivage de toutes les versions demande certes plus d'espace de stockage mais l'accès aux anciennes versions est désormais direct.

Un document est susceptible d'être concurremment édité par plusieurs personnes. En pratique, le fichier original est dupliqué en deux ou plusieurs copies. Chaque copie évolue indépendamment des autres. Cependant, au bout d'un certain nombre de versions, des branches devraient être fusionnées avec d'autres pour former une seule version unifiée (fig. 2.1).

2.2 Arbre & Structure d'arbre

En théorie de graphes, un *arbre* (ou une *arborescence*) enraciné T est un *graphe acyclique orienté* possédant une unique racine, et tel que tous les noeuds sauf la racine ont un seul parent et sont accessibles depuis la racine. La structure d'arbre est une structure de données récursive et elle est

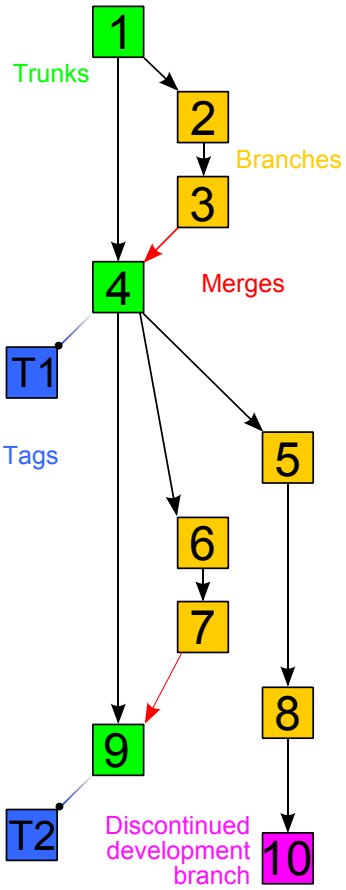


FIGURE 2.1 – Contrôle de révisions d'un projet [wikipedia]

largement utilisée en informatique pour représenter et stocker des données (e.g. Base de données, Indexation, ...)

Les *enfants* d'un noeud u sont des noeuds dont le *parent* est u. Les *descendants* de u sont ses enfants ainsi que les descendants de ses enfants. Si le noeud v est un descendant de u, alors u est son *ancêtre*. Les noeuds n'ayant pas d'enfants sont appelés des *feuilles* ou des *noeuds externes*. La racine est le seul noeud qui n'a pas de parent. Les noeuds qui ne sont ni des feuilles, ni la racine sont appelés *noeuds internes*. Un noeud peut avoir plusieurs enfants, les enfants d'un commun noeud sont des frères.

Un parent et un enfant sont reliés par une *arête*, par exemple l'arête (u, v) connecte le noeud u au noeud v. Le *chemin* allant d'un noeud u_1 à un autre noeud u_n est une suite enchaînée d'arêtes (u_1, u_2) (u_2, u_3) ... (u_{n-1}, u_n). Le nombre d'arêtes est donc la longueur du chemin. La *profondeur* (ou le *niveau*) d'un noeud est la distance de la racine à ce même noeud. La racine est de profondeur 0. La *hauteur* d'un arbre est la longueur du plus long chemin trouvé et incrémenté de 1.

Un arbre peut être ordonné ou non-ordonné. Un arbre est dit ordonné lorsque les enfants de gauche à droite du noeud u sont uniquement numérotés de 1 à k. Pour un arbre non-ordonné, l'ordre des enfant d'un noeud n'est pas important.

Un *sous-arbre* (ou une *branche*) T_u de T est un arbre enraciné au noeud u. Un arbre est étiqueté (ou labellisé), si chacun de ses noeuds possède une étiquette. Deux noeuds ont une même étiquette

quand ils ont de même contenu qui comprend un type, un nom, un texte et des paires attribut-valeur.

Les types de parcours à travers d'un arbre sont le parcours en profondeur et le parcours en largeur. Le *parcours en largeur* (BFS - breath first search) correspond à un parcours par niveau de noeuds. Un niveau est un ensemble de noeuds se situant à la même profondeur. Tous les noeuds du niveau i sont visités de gauche à droite avant les noeud du niveau $i+1$. Le *parcours en profondeur* (DFS - depth first search) est un parcours récursif dans lequel un noeud est visité si et seulement si tous les descendants de son frère de gauche sont visités.

Par exemple, considérons l'arbre illustré dans la figure 2.2. Les noeuds de l'arbre sont labellisés par R, a, b, c; d. R est la racine ayant deux enfants a et d. b et c ont le même parent a. R et a sont les ancêtres de b. Le sous-arbre $T(a)$ contient trois noeuds a, b et c avec a en tant que racine. Le noeud a est un noeud interne tandis que b, c et d sont des feuilles. La hauteur de cette arbre est de 3 et la profondeur de a et d est de 1.

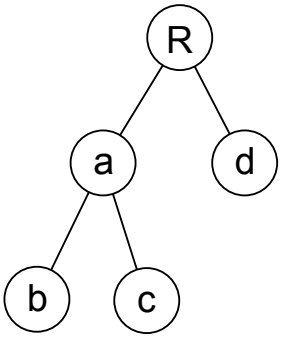


FIGURE 2.2 – Structure d'arbre

2.3 XML & XML Document

L'eXtensible Markup Language - XML est un méta-langage informatique largement connu et utilisé aujourd'hui. XML est standardisé et mis en place par le World Wide Web Consortisum (W3C [1]). Des descriptions complètes de XML et des applications peuvent être trouvées dans les recommandations de W3C [1]. Voici, quelques définitions données sur le site du W3C :

> The Extensible Markup Language (XML) is a simple text-based format for representing structured information : documents, data, configuration, books, transactions, invoices, and much more. It was derived from an older standard format called SGML (ISO 8879), in order to be more suitable for Web use.
>
> XML is one of the most widely-used formats for sharing structured information today : between programs, between people, between computers and people, both locally and across networks.

1. http://www.w3.org/

XML hérite historiquement du méta-langage Standard Generalized Markup Language - SGML, développé au début des années 1980 et beaucoup utilisé dans les systèmes documentaires. Le SGML restait insuffisamment explicite et trop complexe ce qui le rendait difficile et couteux à l'implémentation, notamment à large échelle. XML respecte non seulement la syntaxe de SGML mais ajoute des contraintes supplémentaires afin d'en lever les ambiguïtés.

Il s'agit d'un *méta-langage* car XML permet à l'utilisateur de définir différents langages à balises, qui qualifie XML d'*extensible*. Les langages à balises (ou de balisage) sont des langages permettant d'associer à un contenu (généralement du texte) des balises explicites rendant compte de la structure. Il existe de nombreuses instances de XML pour différents problèmes : mise en forme de documents (e.g XHTML, XSL-FO), formats numériques (e.g SVG), structuration logique du document (e.g DocBook, DITA), Echange de données (e.g SOAP), programmation (e.g XSL, ANT), ainsi que tous les langages localement définis.

Les unités logiques du document XML sont appelés *éléments XML*. Un élément est identifié par une balise (ou tag) de début $<$ *balise* $>$ et une balise de fin $<$ /*balise* $>$. Un élément peut contenir des attributs qui sont spécialisés dans la balise de début en utilisant la syntaxe *nom* = "*valeur*". Entre deux balises de début et de fin d'un élément, il y a en général du texte ou d'autres éléments. Mais il est aussi possible de mélanger le texte et les éléments. Dans ce cas, les éléments sont qualifiés *inline*. Deux éléments ne peuvent pas se chevaucher.

Listing 2.1 – Un exemple de XML

```
<bookstore>
  <book category="cookbook">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  ...
</bookstore>
```

Considérons l'exemple 2.1 qui représente un document XML sous sa forme textuelle :
- $<$ *book* $>$, $<$ *author* $>$ sont des balises où *book*, *author* sont des noms de balises (ou tagname) ;
- *category* = "*cookbook*" est un attribut *category* ayant la valeur "*cookbook*" ;
- La balise $<$ *book* $>$ contient d'autres balises $<$ *title* $>$, $<$ *author* $>$, ...
- Les balises $<$ *title* $>$, $<$ *author* $>$ contiennent des parties purement textuelles *EverydayItalian*, *GiadaDeLaurentiis* qui sont appelées des *éléments textuels*.
- Les balises de début et de fin, ainsi que leur contenu (éléments textuels, autres éléments insérés entre deux balises) constituent un élément.

Le modèle sous-jacent de XML est un modèle d'arbre (fig. 2.3). Un document XML peut être considéré comme la linéarisation d'un arbre. Un élément XML est un noeud ou un sous-arbre de l'arbre. Un élément textuel correspond à une feuille de l'arbre. Pour éviter l'ambiguïté, on les appelle respectivement *noeud d'élément* et *noeud de texte*.

Enfin, il se peut qu'un document XML valide une ou plusieurs DTD [2] ou XML schema [3]. Il s'agit un ensemble de déclarations qui précisent quel type d'élément peut apparaître à quel endroit dans le document, ce qui peut être contenu ou attribut, etc.

2. http://www.w3schools.com/dtd/dtd_intro.asp
3. http://www.w3schools.com/schema/schema_intro.asp

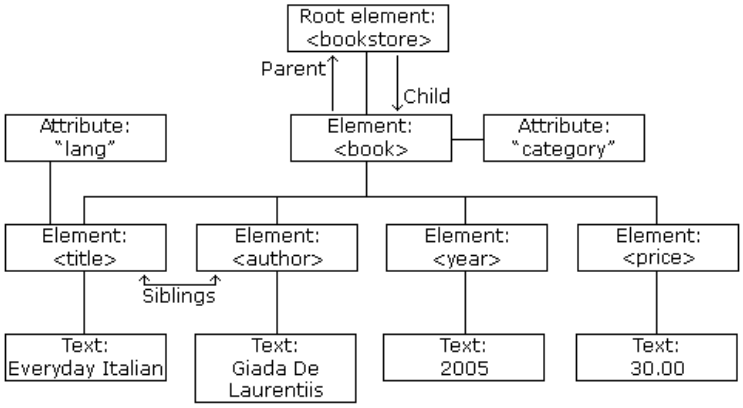


FIGURE 2.3 – Modèle d'arbre du XML

Il est à préciser, comme indiqué par Fuhr dans [27], qu'il est possible de distinguer deux approches de structuration de XML correspondant à deux besoins différents : *orientée données* (*data-centric*) et *orientée document*(*document-centric*). Les langages XML orientés données permettent d'enregistrer et de transporter des données informatiques structurées (comme par exemple des données gérées par des bases de données) selon des formats ouverts (c'est à dire dont on connait la syntaxe) et faciles à manipuler. L'approche orientée document regarde le document XML sous la forme documentaire traditionnelle. XML permet la structuration logique des documents. En effet, le balisage sert au repérage des éléments constitutifs du texte (chapitre, paragraphe, mots importants, etc.) et des relations de hiérarchie entre ces éléments (un paragraphe appartient à un chapitre). Dans un fichier XML orienté document, le texte reste le composant principal. En général, la granularité élémentaire de XML orienté document est plus grande que celle de XML orienté donnée (e.g paragraphe contre quelques mots). XML orienté document est toujours ordonné, c'est à dire l'ordre des enfants est significatif.

Listing 2.2 – XML orienté données

```
<collection>
  <title>Astérix le Gaulois</title>
    <livre>
        <title>Astérix chez les Belges</title>
        <creator>René Goscinny</creator>
        <creator>Albert Uderzo</creator>
        <type>Text</type>
        <description> Astérix chez les Belges est un album de bande
            dessinée de la série Astérix le Gaulois créée par René
            Goscinny et Albert Uderzo. Cet album pubyylié en 1979 est le
             dernier de la série écrit par René Goscinny.
        </description>
    </livre>
</collection>
```

Listing 2.3 – XML orienté documents

```
<roman>
  <title>Le tour du monde en 80 jours</title>
```

```
<chapter id='I'>
  <title> Dans lequel Phileas Fogg et Passepartout s'acceptent
     réciproquement, l'un comme maître, l'autre comme domestique
  </title>
  <content> En l'année 1872, la maison portant le numéro 7 de Saville
     -row, Burlington Gardens - maison dans laquelle Sheridan mourut
     en 1814 - était habitée par Philéas Fogg, esq., l'un des membres
      les plus singuliers et les plus remarqués du Reform-Club de
     Londres ...
    </content>
  </chapter>
  <chapter id='...'> ... </chapter>
</roman>
```

2.4 Modification & Opérations

Les versions d'un document sont nécessairement liées à travers des *modifications*. L'utilisateur est souvent intéressé par les différences entre les versions plutôt qu'à examiner des versions individuelles. Une modification constitue l'évolution entre deux versions et peut être exprimée par des opérations d'édition, ou simplement appelées opérations. Les types d'opérations les plus courantes sont *insert* et *delete*. La plupart des algorithmes précisent l'opération *update* qui peut également être spécifié comme une combinaison de delete et insert. Peu courante est l'opération *move* qui, comme update, peut également être précisée comme une combinaison de delete et insert. Enfin, encore moins courante l'opération *copy* peut être considérée comme un insert d'une chose déjà existante. L'avantage principal de l'utilisation des trois derniers types d'opération, c'est qu'ils diminuent la taille du fichier delta. Une autre raison en en faveur, c'est que la modification est rendue plus naturelle et intuitive pour l'utilisateur final.

Il y a plusieurs interprétations possibles de ces opérations. Dans le modèle introduit par Kuo-Chung Tai [3], la fait de supprimer un noeud implique que ses enfants deviennent les enfants de son parent. Le modèle de Selkow [2] précise que les opérations ne s'appliquent que sur des feuilles ou des sous-arbres entiers. Dans ce cas, lors qu'un noeud u est supprimé, tout le sous-arbre $T(u)$ est supprimé. Ce deuxième modèle semble plus adapté aux documents XML, notamment lorsqu'ils sont définis par une DTD ou une XML schema.

2.5 Tree Matching/Mapping

Les *Tree-matching problems* sont des problèmes informatiques fondamentaux [6], [7]. Ils ont un rôle important et décisif dans la définition et l'implémentation des algorithmes de *differencing* et *merging* des structures d'arborescences. Un tree-matching montre la correspondance entre deux arbres. Il représente tous les node-matching possibles. Un node-matching est une paire de noeuds appariés dans deux arbres. En général, node-matching est une relation *one-to-one*, mais elle peut être aussi de type *one-to-many*. Le tree matching problem n'est pas limité à des node-matching séparés mais peut être généralisé pour matcher les patterns, en l'occurrence des sous-arbres. Il peut y avoir différents types de matching. Le matching par contenu signifie que deux noeuds ont le même

ou presque même contenu. Le matching par structure intervient quand deux noeuds ont des enfants matchés par paire. Le full matching est un matching par contenu et un matching par structure à la fois. Enfin, le matching par contexte est quand deux noeuds, quoique différents, ont deux pères matchés et des frères(précédent et suivant) matchés.

En utilisant le tree-matching, il est possible d'identifier des différences entre deux arbres. En effet, il y aura des noeuds matchés et des noeuds non-matchés. Ceux qui ne sont pas matchés sont les noeuds uniques d'un arbre et donc différents de l'autre.

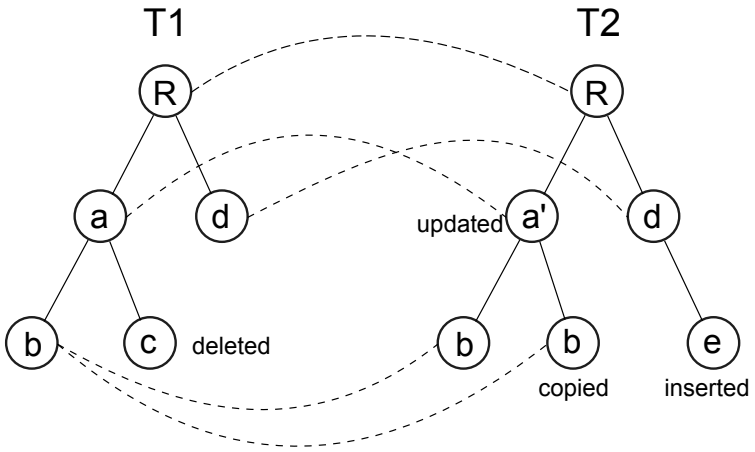


FIGURE 2.4 – Matching entre les noeuds dans deux arbres

2.6 Differencing & Patching

Supposons que nous avons deux versions T_0 et T_1 et que T_1 est dérivé de T_0 par une transformation inconnue. Le problème de *differencing* consiste à identifier un ensemble de changements entre T_0 et T_1. Les changements relevés ne sont pas nécessairement ceux qui ont été réellement faits. Par contre, ils devront permettre d'obtenir T_1 à partir de T_0.

Le processus est divisé en deux étapes. Il doit d'abord identifier d'une manière efficiente les différences, ensuite enregistrer ces différences dans un format utilisable. Typiquement, les différences sont exprimées par des opérations. Cet *output* est la matière essentielle pour la transformation, le *patching*. Le patching est donc très dépendant du format de l'output.

2.7 Merging

Merging correspond à la fusion de deux différentes versions afin de produire une nouvelle version dans laquelle s'intègrent les changements, pas nécessairement tous, venant des deux versions. Cette opération se différencie du patching qui cherche à transformer une version en une autre version. Bien que le but soit différent, merging et patching utilisent un même output généré par le moteur différentiel.

Il y a deux variantes du problème de merging : *two-way merging* et *three-way merging*. La différence est que dans le premier cas, la fusion doit être réalisée à partir des deux seuls fichiers T_1 et T_2, alors que dans le second, on dispose aussi du fichier origine T_0.

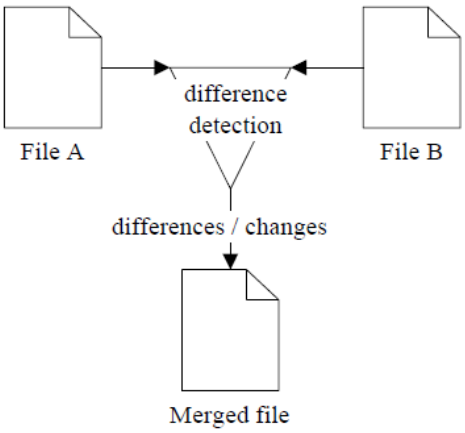


FIGURE 2.5 – Two-way merge

Le two-way merging contient toujours des ambiguïtés, par exemple, une partie peut être considérée à la fois comme une insertion dans le premier document et comme une suppression dans le deuxième. Il y aura donc deux options à choisir : soit intégrer, soit ne pas intégrer cette partie dans le document fusionné. L'intervention de l'utilisateur avant appliquer une différence est donc nécessaire.

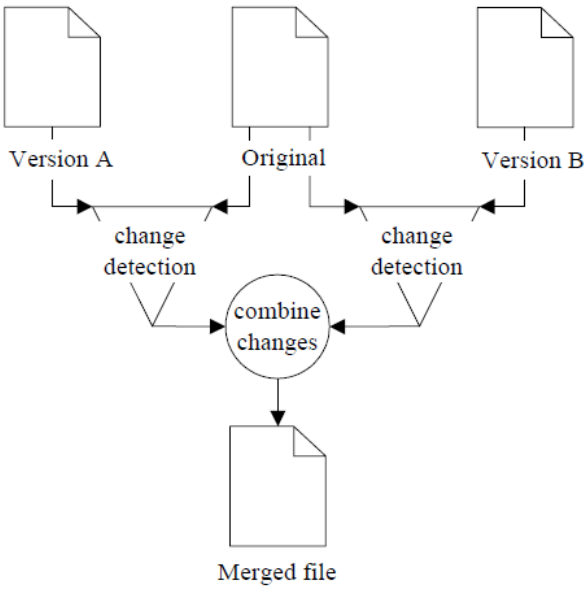


FIGURE 2.6 – Tree-way merge

Dans le three-way merging, T_0 est respectivement comparé avec T_1 et T_2. Les différences de (T_0,T_1) et de (T_0,T_2) sont combinées et enregistrées dans un *delta*. Ce dernier est ensuite appliqué au fichier original afin de produire un fichier fusionné comprenant tous les changements intervenus dans T_1 ainsi que dans T_2. Le three-way merge est largement utilisé, notamment dans les outils de managements de logiciels tels que CVS, Subversion, ...

Le three-way merge est potentiellement plus précis, plus puissant mais aussi plus compliqué que two-way merge. Considérons l'exemple suivant :

T0.xml	T1.xml	T2.xml
<R> <s1> <p1 /> <p2 /> </s1> <s2> <p3 /> <p4 /> </s2> </R>	<R> <s1> <p1 /> <p2 /> </s1> </R>	<R> <s1> <p1 /> </s1> <s2> <p3 /> <p4 /> </s2> </R>
	l'élément <s2> et ses descendants sont supprimés	l'élément <p2> est supprimé

TABLE 2.1 – Un exemple montre la différence entre 2-way et 3-way merge

L'élément <p2> de T_1 n'est pas dans T_2. L'élément <s2> et ses descendants de T_2 ne sont pas dans T_1. Selon two-way merge (T_1,T_2), <p2>, <s2> et ses descendants devront être ajoutés dans T_3. Par conséquence, nous retrouvons exactement T_0. Le tree-way merge (T_0,T_1,T_2) fonctionne différemment. En connaissant T_0, on sait que <s2> et ses descendants sont supprimés dans T_1 et que <p2> est supprimé dans T_2. Alors, au lieu de rajouter ces éléments, le three-way merge les supprimera de T_3 et nous obtiendrons :

```
<R>
  <s1>
    <p2 />
  </s1>
</R>
```

Pendant la fusion, des conflits peuvent se produire, par exemple une même partie est modifiée dans T_1 et dans T_2. Le processus a donc besoin d'une option pour pouvoir continuer. Certains outils en s'appuyant sur leur propre heuristique vont choisir une option privilégiée tandis que d'autres décident de garder quand même toutes les options.

Chapitre 3

Etat de l'art : les différents méthodes et outils de différentiel des documents XML

Dans différents domaines, les documents textuels numériques sont de plus en plus facilement accessibles et partagées par nombreuses personnes. Il est très important, pour un utilisateur, d'identifier rapidement les différences qui peuvent exister entre les sources pour pouvoir les synchroniser et fusionner correctement. Les travaux relatifs à ces problèmes forment le domaine diff & merge. Ce dernier a débuté depuis plusieurs années et il existe actuellement de multiples solutions, libres et commerciales. Chaque solution a sa propre approche et donc ses propres propriétés et optimisations. Certaines d'entre elles sont orientées documents textuels (e.g Doc) , d'autres orientées documents structurants (e.g XML). Elles ont chacune leurs propres cas d'usage. Ce chapitre fournit un panorama global de ce domaine diff & merge.

3.1 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 *edits log*. Une action est aussi appelée edit. Edit log est donc dupliqué et transféré à autres utilisateurs afin de comparer leurs propres edit log. Comparer des documents revient à comparer des edit log. Pour synchroniser des documents, il suffit de rejouer sur un document les actions transférées depuis d'autres documents.

Cette approche permet d'obtenir une vraie collaboration distribuée et en temps réel. En effet, les edit log sont régulièrement échangés via les réseaux. Edit log doit se confronter à deux challenges principaux. Premièrement, il faut absolument capturer toutes les actions de l'utilisateur y compris taper, copier, couper, coller, remplacer, auto-corriger, changer de style, ... Les environnements d'édition de plus en plus modernes et évolués vont rendre cette tâche très compliquée. Le deuxième challenge est d'assurer que tout est envoyé et tout est reçu. Car chaque edit va changer la position des edit dépendantes, un edit perdu va causer l'application incorrecte des edits dépendantes.

Exemple : Microsoft Office Groove.

3.2 Change detection

Change detection est une approche contraire à Edit history. Cette arroche n'a besoin d'aucune connaissance à propos de l'histoire de l'édition du fichier. En possédant seulement des fichiers courants, elle cherche à déterminer les changements faits dans chacun des fichiers. Nous donnons, ici, une vision globale de différents algorithmes de détection des changements.

3.2.1 Line oriented

Cette catégorie d'outils traite tous les documents comme une série linéaire de linges. UNIX diff [1] est un exemple typique et le plus connu. Diff est capable de produire les différences *line-by-line* entre deux fichiers. 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 un autre fichier. Une variante de diff est diff3 implémentant le three-way merge.

Une variante raffinée consiste à examiner dynamiquement des mots et même des caractères au lieu de lignes. Cela permet une comparaison plus détaillée et plus précise, mais elle peut engendrer des inconvénients. Par exemple, deux phrases parlent des choses différentes, mais par coïncidence il y a des petits mots communs. Le fait de montrer ces petits communs entourés par le reste différent va diminuer la visibilité et la clarté de la différence. Certains outils tels que *google-diff-match-patch* [2] sont suffisamment intelligents pour éviter cet inconvénient en indiquant que toute la phrase a changé.

Ces outils sont très adaptés et efficaces vis-à-vis des documents textuels mais ils ne sont pas directement applicables pour des documents structurés tel que XML ou XHTML, car ces derniers contiennent non seulement du texte mais aussi des informations structurelles et diff n'est pas en mesure de les distinguer.

3.2.2 Tree oriented

Il y a d'autres méthodes appelées "orienté arbre" qui prennent en considération la structure d'arborescence du document. Les noeuds et les sous-arbres seront comparés et matchés à la place des lignes. Les noeuds et les sous-arbres qui n'ont pas été matchés, forment les différences entre deux documents. Deux études comparatives et détaillées des algorithmes différentiels orientés arbre ont été menées et rapportées dans [8] et [9]. Les algorithmes sont multiples :

- **LaDiff**, 1996 [10]
- **MHDiff**, 1997 [11]
- **XMLTreeDiff**, 1998 [12]
- **MMDiff & XMDiff**, 1999 [13]
- **IBM's XML Diff and Merge Tool**, 2001 [14]
- **Microsoft's XML Diff and Patch**, 2002 [15]
- **DiffXML**, 2002 [16]
- **XyDiff**, 2002 [17]
- **X-Diff**, 2003 [18]
- **DeltaXML**, 2003 [19], [20], [21]

1. http://www.gnu.org/software/diffutils/diffutils.html
2. http://code.google.com/p/google-diff-match-patch/

– **3DM**, 2004 [22], [23], [24]
– **DaisyDiff**, 2007 [3]
– **XCC**, 2008 [25]
– ...

Ces algorithmes sont majoritairement généralistes ou orientés données XML. Ils sont optimisés pour le temps d'exécution et la mémoire utilisée. Certains algorithmes sont spécialisés pour diff et merge à la fois tandis que d'autres ne traitent que diff (voir Annexe A). Beaucoup de ces algorithmes, mais pas tous, ont des implémentations *open-source* disponibles. Par contre, la plupart n'est plus maintenu depuis 2006. Nous avons étudié plus spécifiquement trois algorithmes, parmi autres : 3DM de Tancred Lindholm, DeltaXML de Robin La Fontaine et DaisyDiff, car ces algorithmes semblent être toujours maintenus.

3DM

3DM a été développé suite au travail de thèse de Tancred Lindolm. Il s'agit d'un outil particulièrement spécialisé pour le three-way merge des fichiers XML. En substituant le fichier T_0 par l'un des deux fichier T_1, T_2, merge(T_1,T_1,T_2) ou merge(T_2,T_1,T_2) cet algorithme devient diff(T_1,T_2). 3DM utilise un algorithme de tree-matching qui peut être considéré comme un algorithme différentiel. Dans 3DM, la construction du matching débute par la mise en correspondance des noeuds one-by-one. Le matching est ensuite étendu pour satisfaire quelques heuristiques. Par exemple, même si deux noeuds n'ont pas un contenu identique, ils sont matchés "par contexte" car leurs pères et leurs frères sont matchés.

3DM traite seulement des arbres ordonnés. Il ne nécessite pas d'avoir des identifiants uniques pour les noeuds, mais si c'est le cas le résultat sera plus fiable. 3DM n'est pas limité aux opérations update/insert/delete, il manipule également l'opération move et l'opération copy des sous-arbres entiers.

Pour effectuer le three-way merging, 3DM qui est un algorithme un peu particulier, n'utilise pas l'edit log. Il se base complètement sur le matching (T_0,T_1) et sur le matching (T_0,T_2). Les noeuds matchés et les nouveaux noeuds sont insérés au fur et à mesure pour construire l'arbre final (voir l'annexe B). Les noeuds supprimés ne sont pas ajoutés. Un ajout de noeud est éventuellement justifié par l'une des opérations supportées par 3DM. Les opérations tracées, contenant des informations très compréhensible pour l'utilisateur, sont inscrites dans un edit log. Le processus est totalement automatisé même si des conflits ont lieu. En effet, une option privilégiée est choisie pour que le processus puisse continuer. Les situations conflictuelles et les options retenues sont également enregistrées dans un *conflict log*.

3DM est disponible en code source Java libre [4]. L'architecture globale de l'outil 3DM est illustrée dans la figure 3.1. L'outil prend deux ou trois fichiers XML à l'entrée, puis parse les fichiers en structures d'arborescence. Les arbres sont ensuite matchés grâce à un *tree matcher*. En utilisant le tree-matching, le three-way merge ou le diff entre deux arbres sont exécutés.

DeltaXML

DeltaXML est un produit commercial [5] créé par Robin La Fontaine. DeltaXML est capable de comparer, de combiner et de synchroniser des documents XML et d'autres types de documents structurés. DeltaXML utilise l'algorithme de Wu [5] pour calculer le LCS (Longest Common Subsequence) afin d'établir le matching initial entre les noeuds à chaque niveau des deux arbres. Les

3. http://code.google.com/p/daisydiff/
4. http://tdm.berlios.de/3dm/doc/index.html
5. http://www.deltaxml.com/index.html

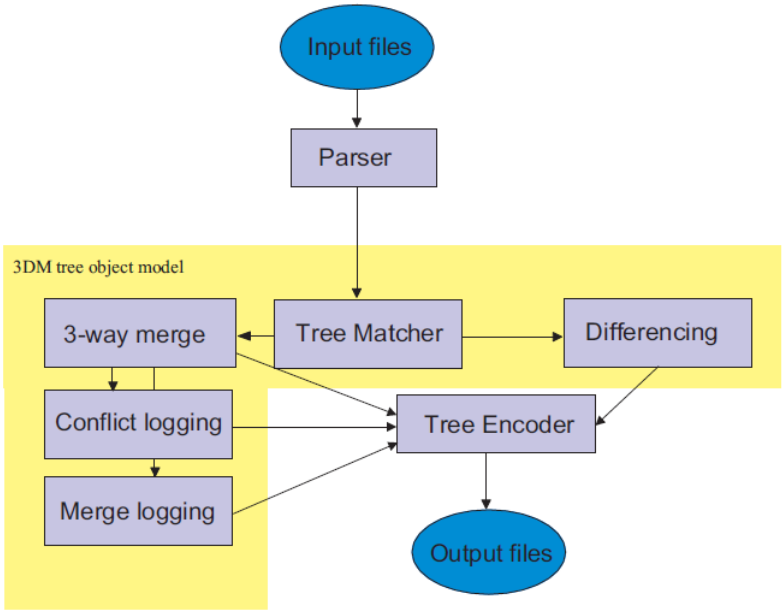


FIGURE 3.1 – Architecture de 3DM

noeuds sont matchés s'ils sont identifiés par une même clé ou s'ils ont le même type d'éléments et leurs enfants sont exactement matchés.

DeltaXML peut traiter des arbres ordonnés ou non-ordonnés dont la taille peut atteindre jusqu'à 100MB. DeltaXML offre two-way merge et three-way merge. Par contre, les opérations supportées sont limitées à update, inserte et delete. L'opération move est remplacée par un delete et un insert. DeltaXML peut détecter même les changements à l'intérieur d'un node de texte qui est souvent considéré insécable par d'autres outils orientés XML. Il est possible avec DeltaXML de définir le niveau de granularité de comparaison, par exemple un paragraphe.

L'un des points intéressants de DeltaXML est son format delta de représentation des changements. Ce dernier est dit *full-context* [20], c'est à dire qu'il contient non seulement les différences mais aussi les données communes aux deux documents. Les deux documents orignaux pourraient être régénérés à partir d'un tel fichier delta.

Listing 3.1 – Un exemple du fichier delta de DeltaXML

```
<para deltaxml:deltaV2="T1!=T2"
deltaxml:version="2.0"
deltaxml:content-type="full-context">
  The
  <deltaxml:textGroup deltaxml:deltaV2="T1">
    <deltaxml:text deltaxml:deltaV2="T1">
    very
    </deltaxml:text>
  </deltaxml:textGroup>
  quick brown fox jumped over
  <deltaxml:textGroup deltaxml:deltaV2="T2">
    <deltaxml:text deltaxml:deltaV2="T2">
    the
```

```
    </deltaxml:text>
  </deltaxml:textGroup>
  lazy dog.
</para>
```

Dans cet exemple, toutes les données de T_1 et T_2, sont présentes. Des balises et attributs spécifiques sont ajoutés pour indiquer où les documents sont différents. T_1 et T_2 peuvent être facilement extraits à partir de cette représentation, en utilisant XSLT ou XQuery par exemple.

DaisyDiff
DaisyDiff est une libraire java open-source de diff qui est plus spécialisée pour les fichiers HTML que pour les fichiers XML. DaisyDiff "comprend" la sémantique des balises HTML. Il examine le texte pour décider si deux noeuds sont égaux ou pas. Le fait de changer seulement quelque mots dans un grand paragraphe ne causera pas que tout le paragraphe entier soit changé. Par conséquent, seuls ces mots supprimés ou ajoutés seront présentés à l'utilisateur. DaisyDiff est aussi capable de détecter un changement de style du texte, par exemple mise en italique.

DaisyDiff a deux mode d'opération : HTML mode et Tag mode. Le tag mode traite le fichier HTML comme du texte mais sait quand même distinguer les balises des textes contenus. HTML mode est le mode par défaut de DaisyDiff. Le fichier HTML est parsé en stucture d'arbre avant être traité. Le HTML mode produit un nouveau fichier HTML qui est en réalité le fichier original avec des annotations indiquant ce qui a changé. Le GUI de DaisyDiff permet de naviguer entre et interagir avec les changements via des raccourcis de clavier et des liens hypertextes. Il est possible de changer le style de présentation des changements par modifier ou remplacer le fichier CSS utilisé par défaut.

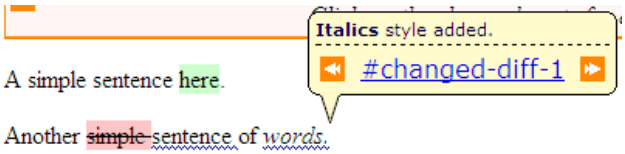


FIGURE 3.2 – GUI de DaisyDiff

3.2.3 Utilisation des IDs uniques

Tous les algorithmes ou outils mentionnés ci-dessus, sont essentiellement basés sur la valeur *hash* et le contenu de chacun des noeuds pour les mettre en correspondance. Ainsi, un calcul de similarité (ou dis-similarité) est effectué. Si le résultat dépasse un seuil fixé, les noeuds sont matchés. Il est alors très probable qu'un noeud ayant subi de grandes transformations ne sera pas matché.

Dans son papier [30], Cheng Thao a proposé un alternative pour le three-way merging consistant en l'utilisation d'identifiants uniques. Chaque élément XML possède un identifiant unique, le matching devient donc direct sans avoir recours à des calculs compliqués. Les noeuds qui ont été beaucoup modifiés ou beaucoup déplacés, sont toujours identifiables. Le matching et la détection des changements sont accomplis pendant le parsing. Il suppose que tous les éléments de T_0 contiennent un attribut d'identité *UID*. Chacun des éléments de T_1 ou de T_2 est examiné s'il contient un UID. Dans le cas positif, l'élément est comparé avec le correspondant dans T_0. Si l'élément a changé, alors l'opération update est ajoutée dans le delta. En cas d'absence d'UID, l'élément est considéré comme un nouvel élément et sera ajouté. Un UID est aussi généré et affecté à cet élément.

Cette approche est avantageuse en termes d'exactitude et de performance. Cependant, il est difficile et compliqué de l'appliquer à grande échelle, car la plupart des éditeurs XML existants ne génèrent pas automatiquement d'UIDs lors de création d'un nouvel élément. En plus, un UID devrait être universellement unique. Dans tous les cas, il ne devrait pas être dupliqué dans différents documents, si non il causera le merging incorrect.

3.2.4 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. [31], il y a une différence entre le diffing d'un XML orienté document littéraire et le diffing d'un XML orienté données. Pour supporter cette idée, ils s'appuient sur deux observations. Premièrement, la sortie d'un diff sur des documents textuels doit être autant que possible fidèle à la sortie d'un diff "manuel" car il est susceptible d'être lu par l'utilisateur. Le deuxième point, plus important, est à propos du modèle d'édition des documents littéraires : les documents sont généralement modifiées en fonction de certains modèles et règles.

La plupart des algorithmes et outils de diff sont généralistes ou orienté données. Ils s'intéressent plutôt à déterminer les changements sur l'arborescence représentant le document que déterminer les changements sur le document même. Ils sont optimisés pour la complexité (le temps d'exécution, la mémoire utilisée) mais peu pour la qualité de la sortie en termes de lisibilité, clarté et précision pour l'utilisateur humain.

Angelo Di Iorio et al. ont introduit un nouvel indicateur, *naturalness* (naturel). Il s'agit de la capacité de l'algorithme à identifier les changements qui pourraient identifiés par une approche manuelle. Ils ont aussi défini de nouvelles opérations significatives et "naturelles" pour exprimer les changements sur un document littéraire.

- **Downgrade** quand un élément est ajouté à un niveau interne de l'arbre, puis un ou plusieurs sous-arbres précédemment situés à même niveau seront attachés à ce nouvel élément.
- **Upgrade** opposé de downgrade, quand un noeud intermédiaire est supprimé, ses enfants seront attachés à son père.
- **Refactoring** quand une sous-structure est scindée en plusieurs sous-structures. Par exemple, le fait de diviser un paragraphe en deux paragraphes est un refactoring.

Ils ont implémenté un algorithme différentiel, appelé JNDiff, permettant de détecter directement des changements naturels en utilisant des structures de données et règles spécifiques. JNDiff identifie dans un premier temps, un ensemble de delete et insert. En suite, il cherche à raffiner ce dernier itérativement en en inférant des opérations naturelles. JNDiff n'est pas encore complété, néanmoins une implémentation en java est disponible à [6]. Il semble que les opérations citées ci-dessus ne sont pas encore supportées.

3.3 Visualisation

Le moteur différentiel a détecté des changements entre deux documents et les a enregistré dans la sortie. Quelle que soit la qualité de la sortie, il est toujours difficile pour l'utilisateur de référencer un changement avec le document. L'utilisateur a donc besoin d'une interface de visualisation des changements. La visualisation va permettre de mettre en évidence les changements dans leur contexte et

6. http://jndiff.sourceforge.net/

de faciliter la manipulation des données par l'utilisateur. En général, il y a deux modes d'affichage : *Side by Side* et *All In One*. Le premier mode consite 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. Dans la suite, nous allons voir quelques interfaces utilisées dans les outils de diff & merge.

3.3.1 Oxygen Diff & Merge

Oxygen Diff[7] est une solution propriétaire permettant de comparer et synchroniser des fichiers et des dossiers. Il compare deux fichiers à la fois en les ouvrant dans deux éditeurs Side By Side (Figure 3.3). Les différences sont surlignées. Entre deux fichiers, une zone de guide colorée permet de localiser facilement les différences correspondantes.

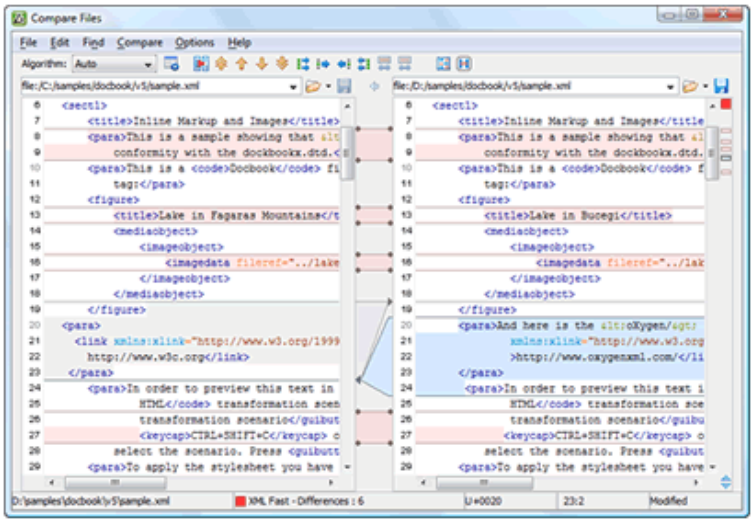


FIGURE 3.3 – Interface d'Oxygen Diff

L'utilisateur peut synchroniser manuellement les deux fichiers en ajoutant ou en enlevant une différence dans un des deux éditeurs. Les différences sont actualisées après chaque enregistrement. L'utilisateur est même capable d'éditer des différences.

3.3.2 Kompare

Kompare[8] est un outil Open-Source de comparaison de fichiers de texte. Il a la même logique de visualisation que OxygenXML Diff & Merge (Figure 3.4). Par contre, il y a un point de plus chez Kompare : le panneau Report de différences. Après avoir visualisé les différences, l'utilisateur peut choisir d'appliquer ou d'annuler quelques différences. Le panneau Report indique l'état de toutes les différences. L'utilisateur peut alors retracer ses actions même si certaines différences ne se représente plus dans les éditeurs à cause de l'actualisation.

7. http://www.oxygenxml.com/xml_diff_and_merge.html
8. http://www.caffeinated.me.uk/kompare/

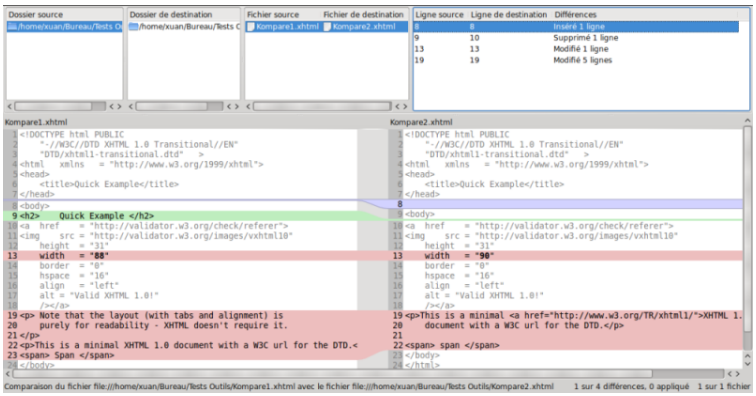


FIGURE 3.4 – Interface de Kompare

3.3.3 Meld, KDiff3

Meld [9] et KDiff3 [10] sont d'autres outils de comparaison des fichiers textuels, ils peuvent comparer jusqu'à trois fichier et les visualiser en même temps (Figure 3.5 3.6).

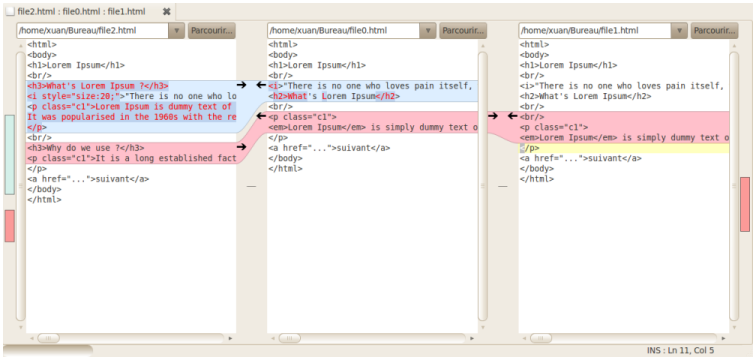


FIGURE 3.5 – Interface de Meld

3.3.4 Diff Doc

Diff Doc [11] est un comparateur graphique de texte (Figure 3.7) qui permet de comparer plusieurs différents types de documents (word, excel, powerpoint, pdf, text, html, xml, ...)

L'interface de Diff Doc permet de changer entre deux modes Side By Side et All In One. En particulier, Diff Doc permet de comparer une sélection dans un fichier et une sélection dans l'autre fichier sans comparer tout le fichier. Cette possibilité est intéressante quand l'utilisateur veut se focaliser sur une partie spéciale d'un très grand document.

9. http://meld.sourceforge.net/index.html
10. http://kdiff3.sourceforge.net/
11. http://www.softinterface.com/

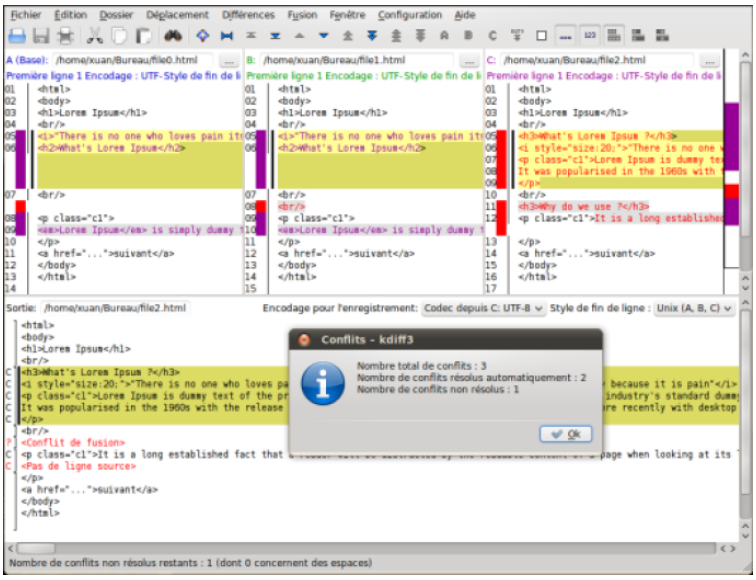


FIGURE 3.6 – Interface de KDiff3

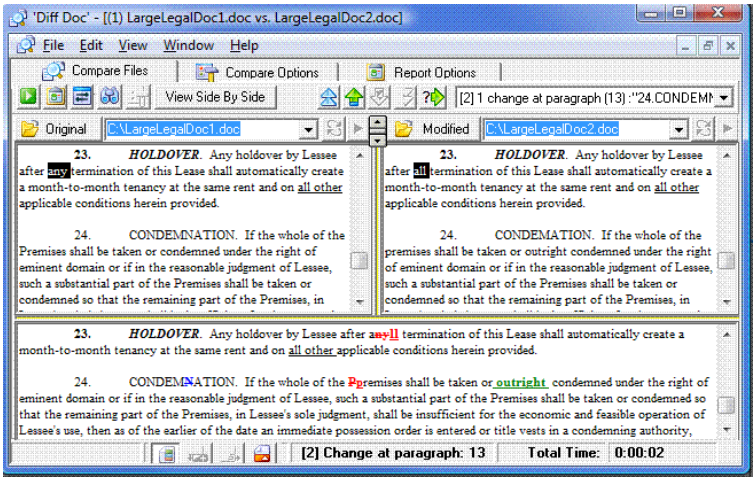


FIGURE 3.7 – Interface de DiffDoc

3.3.5 WinMerge 3.x

WinMerge [12] est un outil différentiel Open Source, sous Window, pour des fichiers binaires ou textuels.

L'interface de WinMerge est de type Side by Side (Figure 3.8). Cette interface offre quelques points intéressants. Premièrement, les paragraphes appariées sont mis à même niveau dans deux éditeurs. Deuxièmement, le panneau d'emplacement à gauche représente une carte de position des différences. Ce panneau est pratique pour l'utilisateur pour avoir une vision globale et la possibilité de localisation rapide des différences. L'utilisateur connait immédiatement quelle partie a beaucoup changée.

12. http://winmerge.org/about/?lang=fr

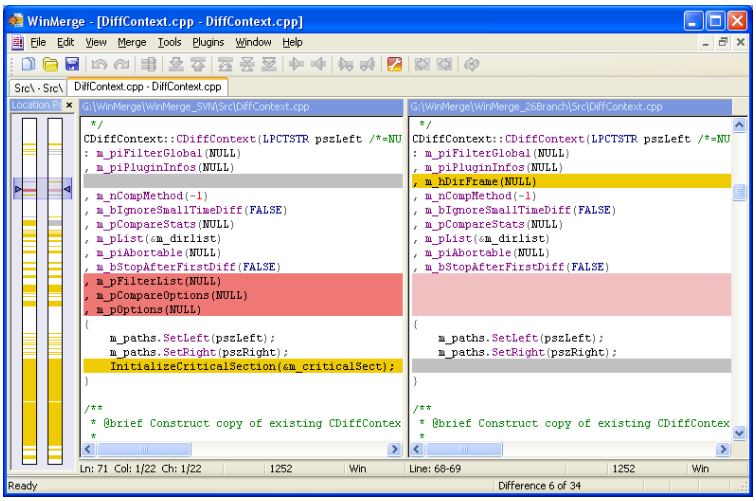


FIGURE 3.8 – Interface de WinMerge

3.3.6 ExamXML

ExamXML [13] est un outil de diff dédié spécifiquement aux documents XML. L'interface est de type side-by-side. Par contre, il ne s'étend pas à tout le document mais le représente sous forme d'arbre interactif (Figure 3.9). Il y a trois modes d'affichage différents : tout le document ou seulement les différences ou seulement les parties identiques. ExamXML est une produit commercial de A7soft et ne fonctionne que sous Window. Il existe aussi un API en Java nommé JExamXML pour les développeurs.

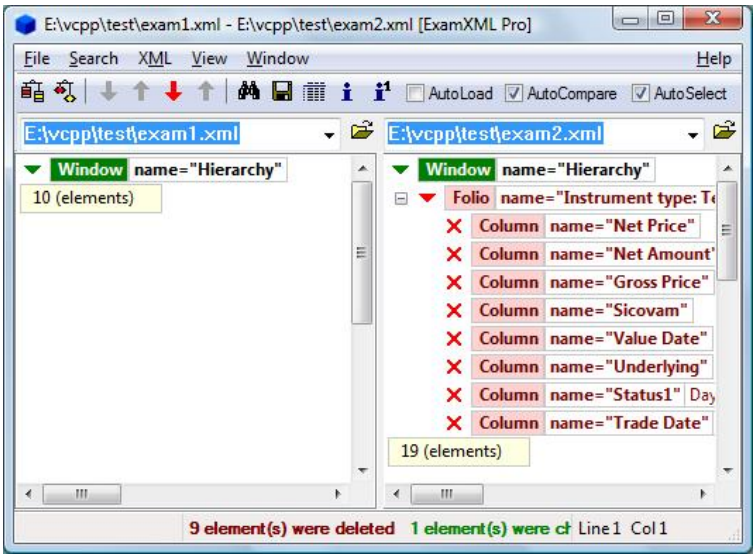


FIGURE 3.9 – Interface d'Exam XML

13. http://www.a7soft.com/

3.3.7 HTML Match

HTML Match [14] est un utilitaire visuel de comparaison de pages web. Il a son propre navigateur (Fig. 3.10). Avec HTML Match, il est possible de comparer des codes sources HTML ou de comparer des contenus textuels.

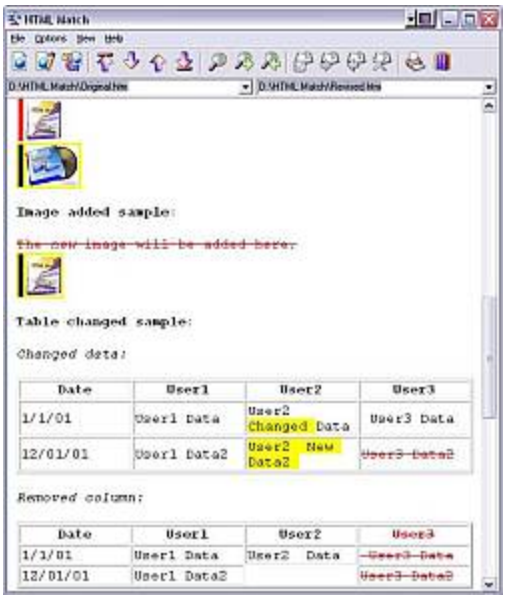


FIGURE 3.10 – Interface navigateur de HTML Match

3.3.8 SiteDelta

SiteDelta [15] est une extension pour Firefox qui scanne la page web visitée et détecte si des changements ont eu lieu depuis la dernière visite. Il est paramétrable pour scanner toute la page web ou seulement une zone de la page web. SiteDelta détecte et met en évidence les nouveaux textes et les textes supprimés. SiteDelta est très utile si l'utilisateur suit un site web qui évolue fréquemment et qui n'offre pas de flux RSS. Les différences sont visibles dans la même page web (Fig. 3.11).

3.3.9 Daisy Diff

Il s'agit d'une librairie Java [16] de diff. Comme SiteDelta, Daisy Diff est dédié spécifiquement aux fichiers HTML. Daisy Diff prend à l'entrée deux fichiers HTML et produit un nouveau fichier avec les annotations pour les différences (Fig. 3.12). En particulier, Daisy Diff reconnait également les changements dans le style des éléments. Il est possible de naviguer entre les différences et de cliquer sur les différences pour obtenir des annotations.

14. http://www.htmlmatch.com/
15. http://sitedelta.schierla.de/index.en.php
16. http://code.google.com/p/daisydiff/

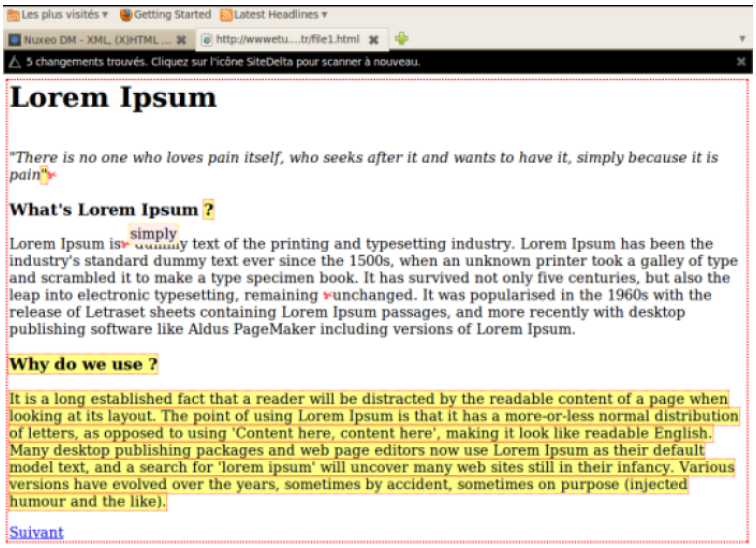


FIGURE 3.11 – Page web scanné et annoté par SiteDelta

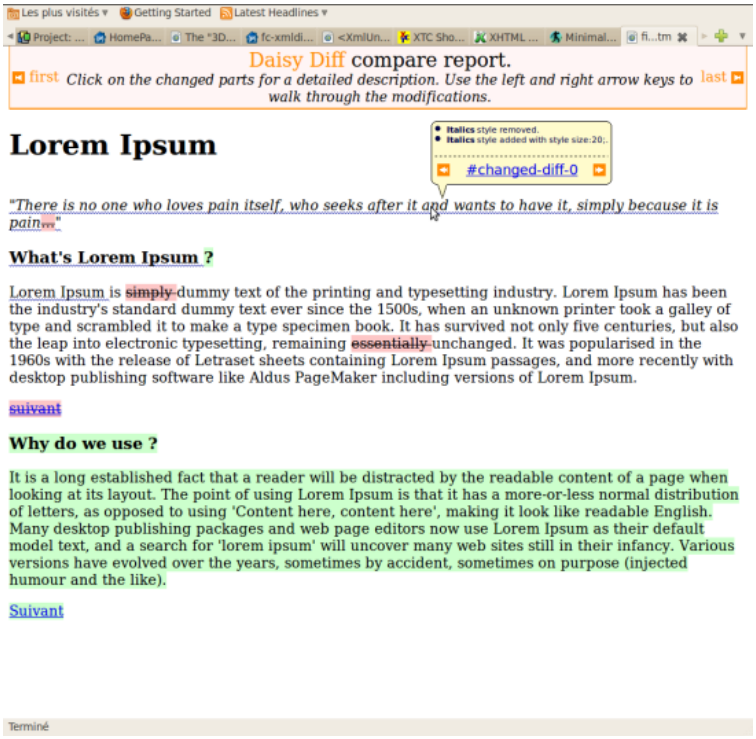


FIGURE 3.12 – Output HTML de DaisyDiff

3.3.10 DeltaXML

DeltaXML [17] offre aussi la fonctionnalité de rapport en format HTML (Fig. 3.13). Le fichier HTML contient le contenu du fichier XML sous forme d'une arborescence interactive ainsi que tous les changements qui ont eu lieu.

17. http://www.deltaxml.com/

FIGURE 3.13 – HTML report de DeltaXML

3.3.11 Visual Comparaison of Hierarchically Organized Data

Dans [28], Danny et Jarke ont introduit un modèle original de visualisation en vue de la comparaison des structures de données hiérarchiquement organisées telle que la structure d'arbre.

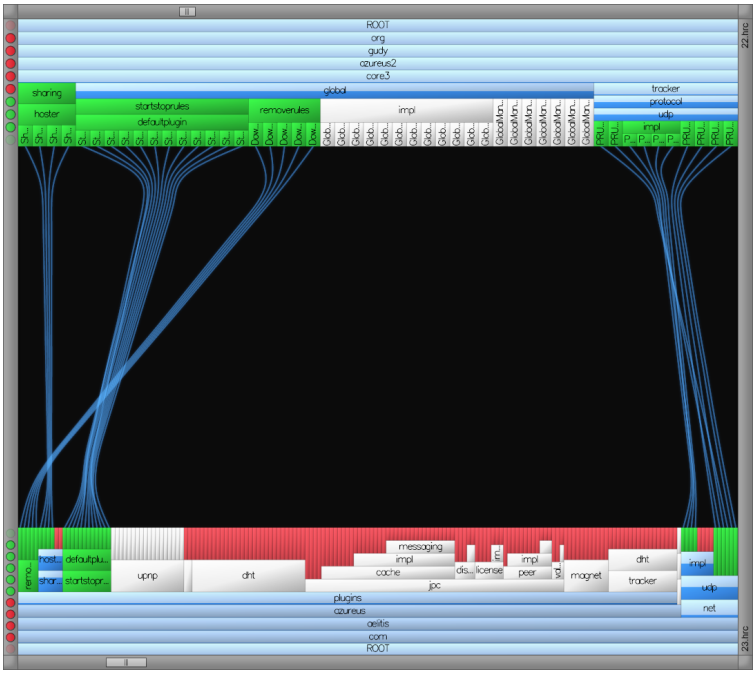


FIGURE 3.14 – Visual Comparaison

Leur technique consiste à placer une paire de hiérarchies, l'une opposé de l'autre, et à décrire comment ces hiérarchies sont reliées en explicitant les liens entre sub-hiérarchies (Fig. 3.14). L'utilisateur peut interagir avec les liens pour retrouver les sous-hiérarchies appariées. Cette méthode a été conçue initialement pour la comparaison des versions de logiciels.

3.3.12 History flow visualizations

Tous les interfaces précédents permettent de représenter deux ou même trois versions du document avec leurs différences. Si le document est un objet partagé entre plusieurs personnes et est en constante évolution, cette fonctionnalité n'est plus suffisante. Par exemple, chaque page dans un

wiki est activement éditée par nombreux utilisateurs et passe par plusieurs versions. Un utilisateur veut étudier, à travers l'histoire des versions, le dynamisme collaborative des participants : qui a contribué et ce qu'il a apporté au fils du temps. Dans cette optique, Fernanda et al. ont introduit dans [29] le concept *History flow Vizualisation*. Graphiquement, chaque révision est représentée par une pile segmentée par différentes couleurs. Chaque couleur correspond à un contributeur et le segment indique sa contribution dans la version. Deux segments correspondants dans deux piles contiguës seront reliés pour mettre en évidence ce qui survit, ce qui change et ce qui s'élimine (figure 3.15).

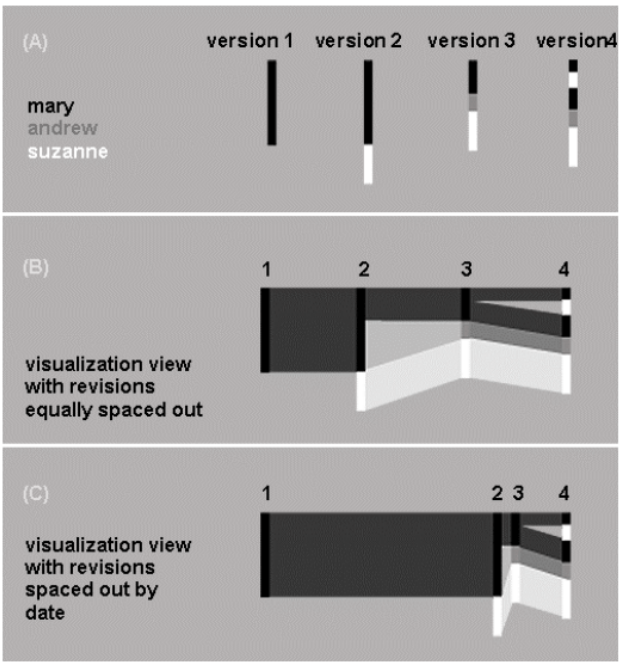


FIGURE 3.15 – History flow visualisations mecanism

3.4 Conclusion

Notre corpus documentaire (e.g Scenari) est encodé en XML et valide des modèles spécialisés. Nous avons donc besoin d'un outil de diff & merge orienté XML document. L'outil 3DM semble être le meilleur candidat, car son code source est librement accessible et il paraît assez aisé de modifier ses modules pour réaliser les optimisations. Son tree-matcher qui effectue le tree-matching, est efficient pour le XML généraliste et peut être encore amélioré. 3DM, en utilisant toutes les opérations (e.g update, insert, delete, move et copy), a une représentation de bonne qualité des différences entre les documents. En fin, le résultat du three-way merge par 3DM est en général meilleur que d'autres outils équivalents.

Nous utiliserons donc 3DM comme un framework de travail dans le quel nous apportons des extensions spécialisées et adaptées à nos documents. Nous avons définis quelques pistes privilégiées :

1. Merging interactif pour choisir, éditer des différences et plus important, pour résoudre des conflits.
2. Amélioration du matching heuristique de 3DM

3. Utilisation d'un algorithme différentiel basé sur le texte pour avoir des différences au niveau du contenu des noeuds.
4. Visualisation des différences

Nous avons commencé par approfondir le premier piste. Il est également l'objet présenté dans le chapitre suivant, "Contribution".

Chapitre 4

Contribution : Merge interactif

La plupart des outils de differencing (diff) et de merging (merge) produisent dans un premier temps, une liste enregistrant des changements identifiés dans une version par rapport à une autre version. Enfin, un outil de *differencing* sert à montrer comment ces deux versions sont différentes alors qu'un outil de *merging* utilise ce résultat pour fusionner automatiquement les changements afin de créer une nouvelle version.

Nous proposons, ici, un alternative appelé *merge interactif*. Cette approche consiste à ne pas appliquer systématiquement la fusion automatique mais à rendre la transformation séquentielle et interactive vis-à-vis de l'utilisateur. L'utilisateur va voir au début s'afficher le document original et une liste associée d'opérations. Ensuite, il va confirmer les opérations les unes après les autres, en les acceptant ou en les refusant. Par contre, il n'y aura pas un ordre d'exécution imposé. C'est à l'utilisateur de choisir les opérations à confirmer. Il peut pré-visualiser le résultat de l'opération sur le document avant de décider de l'appliquer réellement. Le document est modifié après chaque confirmation.

Il y a deux motivations principales pour le merge interactif. La première de ces deux motivations est qu'il permet à l'utilisateur de sélectionner les changements utiles et d'ignorer ceux non-utiles pour sa propre version. L'utilisateur est donc sûr de la consistance de son document. Par exemple, l'utilisateur dispose une version dont un paragraphe est unique et il est intéressé par un certain paragraphe trouvé dans une autre version. Pour ajouter ce nouveau paragraphe et conserver également son paragraphe unique, l'utilisateur devra accepter l'opération insert et refuser l'opération delete. La deuxième motivation est que le merge interactif permet à l'utilisateur de résoudre manuellement et convenablement les changements conflictuels survenus lors du three-way merging. Par exemple, deux contributeurs ont différemment modifié le titre d'un même document. L'utilisateur, une troisième personne, qui voudrait fusionner ces deux contributions, sera donc informé de cette double modification et sera en mesure de choisir l'un des deux nouveaux titre. Ces deux motivations représentent deux possibilités qui sont très utiles pour les auteurs notamment en contexte collaboratif, car ils ont toujours besoin de s'assurer la consistance de leurs documents partagés.

Un usage additionnel du merge interactif est de l'utiliser pour visualiser les 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. Par contre, ils sont limités aux trois types d'opération basiques insert, delete et update. En plus, plus le document a été changé, plus il y a des différences et plus il est difficile pour l'utilisateur de suivre l'ensemble. Avec le merge interactif, on peut rejouer en séquence toutes les opérations l'une après l'autre, ce qui peut être un

inconvénient en perdant la vue globale mais plus avantageux en termes d'opérations possibles (e.g. move, copy) et en termes de charge de visualisation pour l'utilisateur.

Notre merge interactif ne produit pas lui-même la liste des opérations mais utilise celle donnée par un outil spécialisé. Nous avons testé plusieurs outils et constatons deux tendances principales de création de cette liste : un ensemble d'opérations non-ordonnées expliquant ce qui se passe pendant la fusion ; un script d'opérations ordonnées permettant d'effectuer la fusion automatique. Le script n'est pas conforme au principe du merge interactif car simplement il n'est pas destiné à être manipulé ou que l'ordre des opérations soit modifié. Avec un ensemble d'opérations non-ordonnées, il est possible d'en recombiner une séquence personnalisée d'opérations. Cette dernière devrait cependant prendre en considération le fait que certaines opérations ne peuvent pas être exécutées avant certaines autres opérations.

Ce chapitre est divisé en deux grandes sections. La première section présente les aspects principaux du merge interactif alors que la deuxième section présente notre premier prototype.

4.1 Séquence d'opérations

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

4.1.1 Opérations

Tout d'abord, il est crucial de définir formellement les opérations appliquées sur un document XML, car tous les raisonnements ultérieurs vont s'appuyer sur elles. Un document XML est une structure d'arborescence dont les éléments sont les noeuds. Modifier un document XML revient à modifier un arbre ordonné. Soit un document XML dont la structure est représentée par l'arbre T ordonné avec des noeuds $m, n, ...$, nous définissons les opérations suivantes :

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

Il est pourtant à noter que ces opérations ne se comportent pas toujours de même manière En effet, elles dépendent du type de l'objet en question : s'il s'agit d'un noeud de texte ou s'il s'agit d'un noeud d'élément (voir "XML & XML Document"). La valeur d'un noeud d'élément est son nom de balise et ses attributs tandis que pour un noeud de texte, c'est une chaine de caractères. En plus, un noeud de texte n'a pas d'enfants.

La définition de l'opération *insert* est, ici, un peu différente de celle trouvée dans certaines littératures qui la définissent souvent comme une insertion d'un sous-arbre au dessous d'un noeud courant dans l'arbre. Cette façon de définir réduit certainement beaucoup le nombre d'opérations insert. Nous préférons quand même notre définition, car elle s'adapte mieux au principe du merge interactif. Par

exemple, quand on ajoute un bloc dans lequel on ajoute ensuite successivement deux paragraphes. L'utilisateur aura donc un insert de l'élément bloc et deux inserts pour deux paragraphes distincts à confirmer. L'un des deux paragraphes insérés pourra être accepté tandis que l'autre pourra être refusé, ce qui ne serait pas possible s'il y avait une seule insertion du bloc en entier.

L'opération move peut être spécialisée par une combinaison d'insert et delete. Elle a donc deux contextes d'application : le contexte de sortie et le contexte d'entrée. Le move est appelé *move sortant* dans le contexte de sortie et *move sortant* dans le contexte d'entrée. L'objet sur lequel elle porte n'est ni supprimé ni inséré.

La liste ci-dessus présente les opérations les plus courantes, elle n'est pas exhaustive. On peut y ajouter également l'opération copy et d'autres. En théorie, l'opération copy duplique exactement un sous-arbre et mettre la copie à un autre endroit dans l'arbre. Nous avons exclus, ici, l'opération copy à cause de plusieurs raisons :

- Très peu d'outils et de méthodes implémentent l'opération copy. Même s'ils le supportent, ils ont aussi l'option pour l'ignorer.
- Dans la pratique, le fait d'avoir deux ou plusieurs parties identiques dans une version, en général du à une erreur de duplication, n'est pas fréquent.
- Si les éléments possèdent des identifiants, l'opération copie va créer deux éléments de même identifiant ce qui n'est pas valable pour la bonne syntaxe du document XML.
- Le sous-arbre, l'objet de l'opération copy, subit aussi d'autres opérations, par exemple la modification d'une feuille. Il n'y a pas de moyen de déterminer si le copy doit être appliqué avant ou s'il doit être exécuté après ces opérations. Les résultats obtenus dans les deux cas ne sont pas le même.
- Le copy n'est pas vraiment adapté aux documents basés texte dans lesquels les structures internes (e.g. chapitre, section, sous-section, bloc, paragraphe, etc.) sont souvent répétées et seuls les textes situés aux feuilles sont distingués. De fait, si l'on inclut l'opération copy, plusieurs fausses copies seront détectées. En réalité, avec un éditeur XML moderne tel que celui trouvé dans les chaînes éditoriales, souvent l'utilisateur insère un contenu et l'éditeur génère automatiquement la structure interne. Dans ce cas, pour l'utilisateur, il est plus naturel et logique d'exprimer le changement par l'opération insert que par l'opération copy.

4.1.2 Delta

Nous avons vu précédemment les définitions des opérations insert, delete, update et move. Ces quatre opérations permettent d'exprimer toutes les différences entre deux versions du document. Considérons l'exemple suivant :

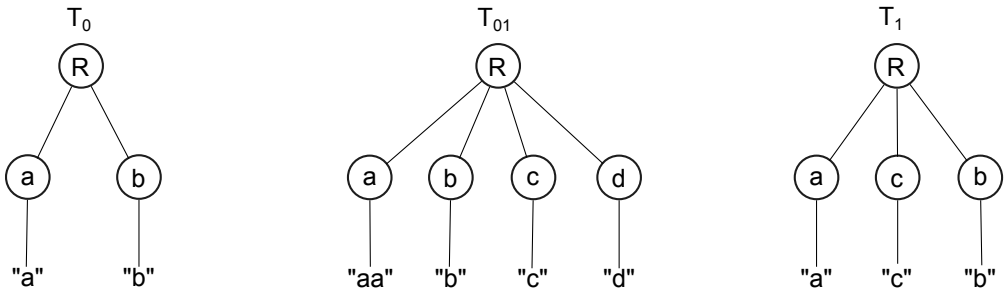


FIGURE 4.1 – Étape intermédiaire de transformation

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)

TABLE 4.1 – Opérations intermédiaires annulées par d'autres opérations

Soit T_0 un document ayant initialement la racine R et deux éléments a et b ayant respectivement des textes "a" et "b". En réalité, T_0 subit une modification en deux temps (fig. 4.1). Dans un premier temps, l'auteur a inséré après a et b deux nouveaux éléments c et d ayant respectivement des textes "c" et "d". Il a aussi modifié le texte initial "a" de l'élément a par le texte "aa". T_0 est donc passé à T_{01}, un statut intermédiaire. Dans un second temps, après une vérification, l'auteur a décidé de garder seulement l'élément c et de déplacer c devant b. T_0 est devenu finalement T_1. T_0 a réellement subit sept opérations effectuées dans l'ordre : $insert(R, 3, c)$, $insert(c, 1, "c")$, $insert(R, 4, d)$, $insert(d, 1, "d")$, $update(a, "aa")$, $delete(d)$, $move(R, 2, c)$.

Supposons que seuls T_0 et T_1 soient enregistrés. Lors de la révision du document, l'utilisateur en possession uniquement T_0 et T_1 veut savoir ce qui a changé dans T_0 par rapport à T_1. Il s'aperçoit facilement que pour passer de T_0 à T_1, il suffit d'insérer l'élément c avec le texte "c" entre a et b et de changer le texte de l'élément a. Il faut donc seulement trois opérations $insert(R, 2, c)$, $insert(c, 1, "c")$, $update(a, "aa")$. Ces trois opérations n'ont rien à voir avec les vraies opérations. Cela s'explique par le fait que certains opérations de la première phase (T_0 - T_{01}) et certains autres opérations de la deuxième phase (T_{01} - 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, $insert(R, 3, c)$, $insert(c, 1, "c")$, $move(R, 2, c)$ sont remplacées par $insert(R, 2, c)$, $insert(c, 1, "c")$ et $delete(d)$, $insert(R, 4, d)$, $insert(d, 1, "d")$ s'annulent et ne donnent rien. Ce ne sont pas des cas isolés, d'autres cas sont présentés dans le tableau 4.1.

Les opérations $insert(R, 2, c)$, $insert(c, 1, "c")$, $update(a, "aa")$ ci-dessus sont appelées *opérations deltas*. Les opérations deltas ne sont pas forcément des vraies opérations effectuées. Elles utilisent des positions référencées à l'arbre original et/ou l'arbre final pour exprimer les différences entre deux arbres. Leurs résultats sont visibles dans au moins un arbre :

- $delete\,(m)$ est une opération delta si on trouve m dans T_0 mais non dans T_1.
- $insert\,(n, k, m)$ est une opération delta si on trouve m en tant que k-ème enfant de n 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* (n, k, m) est une opération delta si on trouve n 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. Les opérations deltas dans Δ de T_0 à T_1 sont suffisantes pour passer de T_0 à T_1. Cependant, il faut les appliquer une par une sur T_0 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 pas un Δ', un sous-ensemble de Δ permettant d'aller de T_0 à T_1. Un Δ contenant des moves redondants (fig. 4.2) n'est pas optimal. Pour qu'il devienne optimal, il suffit de supprimer les moves redondants.

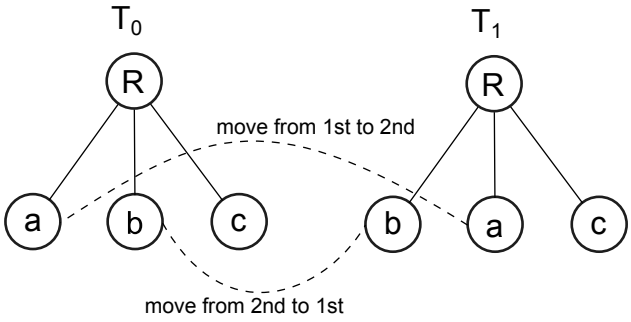


FIGURE 4.2 – Quoique les deux move satisfassent la condition de move delta, l'une seule des deux est suffisante pour transformer T_0 en T_1 et l'autre sera redondante

La liste des opérations utilisées dans le merge interactif est un delta. Les opérations sont des opérations deltas.

4.1.3 Relation d'ordre

Exemple :
Un papier titré "this is the source file" a initialement deux chapitres. Chacun des chapitres contient son propre titre et des blocs qui sont constitués de paragraphes de contenus et possèdent éventuellement un sous-titre. On a voulu supprimer (opération delete) le deuxième chapitre mais garder pourtant le seul bloc qu'il contient. Ce bloc est donc déplacé (opération move) à l'intérieur du premier chapitre en tant que 3-ème enfant. Ensuite, on a changé (opération update) le titre du papier qui est maintenant "this is the cible file". La figure 4.3 illustre les changements entre "source file" et "target file".

Dans ce cas, les vraies opérations sont également des opérations deltas car elles sont toutes repérables sur l'arbre original et l'arbre final. Admettons que notre outil de différentiel a correctement détecté toutes les trois opérations effectuées, il les enregistre sans préciser d'ordre d'exécution. Ces opérations étant a priori indépendantes, il est possible de les exécuter dans n'importe quel ordre (e.g. 1.update, 2.delete, 3.move ou 1.update, 2.move, 3.delete, etc.). Cependant, en examinant de plus près le contexte de l'opération move, on peut s'apercevoir un problème potentiel. En effet, 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 absence de son objet, l'opération move devient immédiatement non-exécutable. Ce problème ne se répète pas si l'ordre d'exécution est inversé (move avant delete).

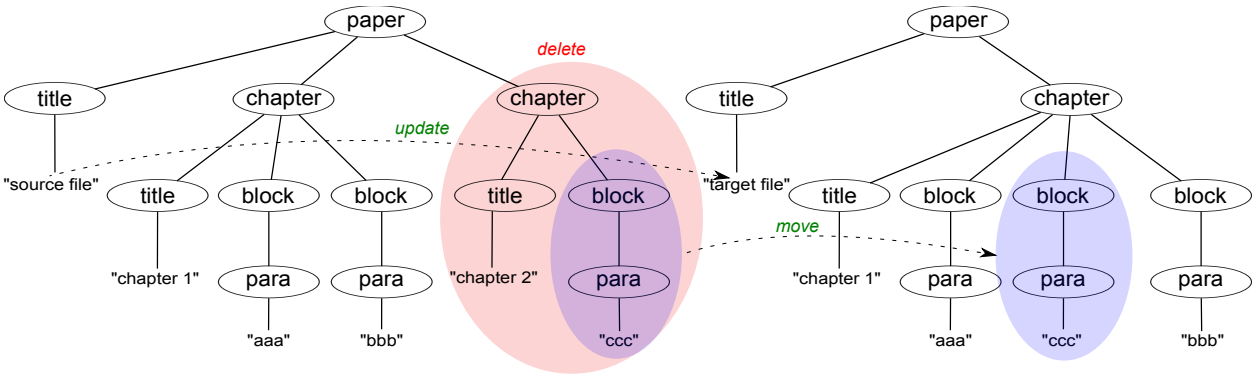


FIGURE 4.3 – Changements entre deux arbres

A partir de cet exemple, nous confirmons qu'il existe, entre certaines opérations, des relations d'ordre :

> **Définition** : Une relation d'ordre entre deux opérations a lieu quand l'exécution d'une opération nécessite l'exécution préalable de l'autre opération pour assurer la faisabilité et l'exactitude de toutes les deux opérations. Cette relation est aussi notée par l'opérateur "$>$".

Soient ω_1, ω_2 deux opérations, alors $\omega_1 > \omega_2$ signifie que ω_1 est une opération dépendante de l'opération ω_1 et que ω_2 est une opération précédente de l'opération ω_1. Dans une telle relation d'ordre, l'opération précédente doit s'exécuter avant l'opération dépendante. Ceci est nécessaire mais non suffisant pour que l'opération dépendante devienne exécutable. En effet, une opération peut dépendre non seulement d'une opération précédente mais de plusieurs. Elle n'est exécutable qu'une fois que toutes ses précédentes ont été effectuées. Il est aussi à 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 où :

$$\omega_1 > \omega_2$$
$$\omega_1 > \omega_3$$
$$\omega_2 > \omega_4$$
$$\omega_3 > \omega_4$$
$$\omega_3 > \omega_5$$

ω_1 est précédente de ω_2 et ω_3. ω_2 est précédente de ω_4. ω_3 est précédente de ω_5 et aussi de ω_4. ω_4 est directement dépendante 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, ω_3 ne sont pas en relation. C'est aussi le cas pour ω_4 et ω_5. Pour pouvoir exécuter toutes ces opérations, il faudrait les exécuter dans un ordre valide. Cet ordre n'est pas unique et doit prendre en compte trois conditions suivant :

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 ;

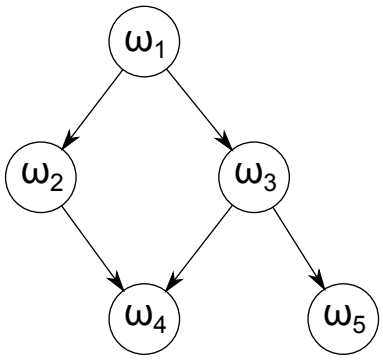


FIGURE 4.4 – Hiérarchie d'opérations

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 (fig. 4.4). Une opération est représenté par un noeud qui peut avoir plusieurs prédécesseurs et plusieurs successeurs. Un arc correspond à une relation d'ordre dont le noeud sortant est la précédente et le noeud entrant est la dépendante. Une hiérarchie devient une arborescence à condition que chaque opération a une seule précédente directe ou n'en a pas.

Une opération qui n'est ni dépendante ni précédente d'autres opération, est appelée *indépendante*. Ainsi, elle n'appartient à aucune hiérarchie. Alors, $\Delta = \{\omega_1, \omega_2, ..., \omega_n\}$ est réécrit :

$$\begin{aligned}&\Delta = \{H_1, H_2, ..., \omega_i, \omega_j\}\\&H_1, H_2, ... \textit{ sont des hierarchies}\\&\omega_i, \omega_j, ... \textit{ sont des operations independantes}\end{aligned}$$

Deux questions se posent. Premièrement, est il possible d'avoir un cycle ?

$$\begin{aligned}&\omega_i \in H, \omega_j \in H : \omega_j > \omega_i \textit{ ou } \omega_j > ... > \omega_i\\&(\omega_i > \omega_j) \ ?\end{aligned} \tag{4.1}$$

Deuxièmement, les hiérarchies sont-elles disjointes ?

$$H_m \cap H_n = \emptyset \ ? \tag{4.2}$$

Preuve
La question 4.2 est d'ors et déjà confirmée par la définition de la hiérarchie. En effet, si une opération appartient à la fois à H_m et à H_n, alors toutes ses précédentes et ses dépendantes le sont aussi. Par conséquent, H_m n'est rien autre que H_n.

Pour répondre à la question 4.1, une solution consiste à explorer toutes les relations possibles entre les opérations, puis chercher à les enchainer afin de détecter si un cycle existe.

Nous utilisons, ici, les notions d'arbre original, d'arbre finale et d'arbre actuel (ou d'arbre en modification). L'arbre original désigne l'arbre initial qui n'a subit aucune opération. L'arbre actuel est en réalité l'arbre original après d'avoir subi des opérations. L'arbre actuel évolue dans le temps et devient l'arbre final quand toutes les opérations de Δ ont été correctement exécutées.

- **update(m,v)**
  L'opération update nécessite deux paramètres, un noeud m et une valeur v. Si le noeud m n'est pas présent dans l'arbre actuel, l'opération n'est pas possible. En plus, ce noeud doit être un noeud de l'arbre original. Parce qu'il ne peut pas y avoir, dans un Δ, simultanément insert(n,k,m) et update(m,v) (voir le tableau 4.1). Nous en déduisons que l'opération update est une opération indépendante, elle ne dépend d'aucune autre opération. Update peut s'exécuter à tout moment dans une suite d'opérations.

- **insert(n,k,m)**
  L'opération insert nécessite trois paramètres, un noeud m à insérer, un noeud n au dessous duquel m sera inséré et k la position de m dans la liste des enfants de n. Le plus important est le noeud n. Il faut que n se trouve dans l'arbre actuel au moment de l'exécution. Le noeud n peut être un noeud de l'arbre original mais aussi l'objet d'un autre insert prédécesseur (fig. 4.5). Dans le premier cas, insert peut s'exécuter normalement. Dans le deuxième cas, l'exécution de l'insert successeur n'est possible qu'après l'exécution de l'insert prédécesseur :

$$insert(.,.,n) > insert(n,k,m).$$

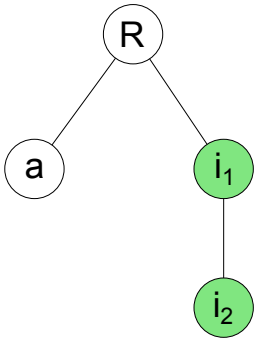


FIGURE 4.5 – Insert i2 dépend de l'insert i1 précédent

Une deuxième condition à satisfaire est que le noeud n a au moins k-1 enfants au moment de l'exécution. Cette condition est particulièrement importante dans le cas où le noeud parent va recevoir plusieurs nouveaux enfants (fig. 4.6). Pour que les nouveaux noeuds soient aux bons endroits, il faut les insérer l'un après l'autre en commençant par la position la plus à gauche. Insert(n,k,m) est donc effectué après d'autres inserts ou moves entrants au niveau du même parent et dont la position est inférieure dans le rang des enfants :

$$insert(n,l,.) > insert(n,k,m) \text{ si } l < k$$
$$move(n,l,.) > insert(n,k,m) \text{ si } l < k$$

En fin, il est important de rappeler que la position k d'un insert delta ne réfère pas à l'arbre actuel mais à l'arbre final. Elle est la position exacte du noeud inséré dans l'arbre final. Rien n'assure que cette position est aussi la position correcte lors de l'exécution de l'insert. Par exemple, le

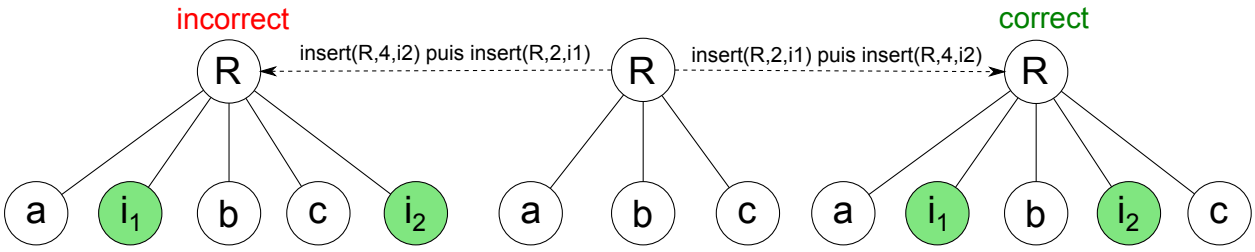


FIGURE 4.6 – Plusieurs inserts au même parent

noeud n, dans l'arbre final, a deux enfants dont le deuxième est m, l'objet d'un insert(n,2,m). En réalité, n avait initialement plusieurs enfants et tous sauf un (qui se voit encore dans l'arbre final) sont l'objet de deletes ou de moves sortants (fig. 4.7). Dans ce cas, si l'insert est exécuté avant les deletes et moves, le noeud m est incorrectement inséré. Il est donc nécessaire, dans une liste d'enfants, d'effectuer les deletes et moves sortants sur les noeuds les plus à gauche avant insérer les noeuds plus à droite :

$$delete(o) > insert(n,k,m) \text{ si } parent(o) = n \; et \; position(o) <= k$$
$$move(p,l,q) > insert(n,k,m) \text{ si } parent(q) = n \; et \; position(q) <= k$$

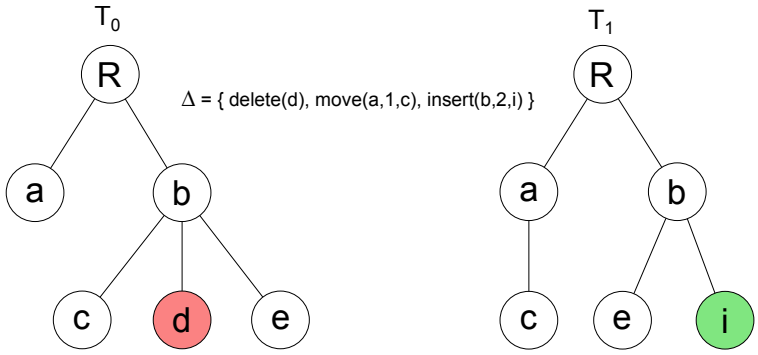


FIGURE 4.7 – La position k d'un insert se référence à l'arbre cible

- **delete(m)**
  L'opération delete a besoin a priori seulement de l'existence du noeud m. Cependant, en supprimant totalement le sous-arbre enraciné à m, on peut ignorer qu'un ou plusieurs éléments au-dessous soient également l'objet d'autres opérations. Ces dernières ne peuvent être ni update ni insert car on ne les trouvera pas dans Δ avec la présence du delete. Par contre, c'est tout à fait le cas pour un move sortant dont l'objet à déplacer se situe dans le sous-arbre supprimé (fig. 4.8). Dans ce cas, exécuter le delete avant le move va causer un problème d'incohérence : l'objet du move a disparu, le move est définitivement non-exécutable. Dans ce cas, le move sortant inférieur devra être exécuté avant le delete supérieur :

$$move(n,k,m) > delete(o) \text{ si } m \in T(o).$$

- **move(n,k,m)**
  L'opération move peut être spécialisée par une combinaison de delete et insert. Par conséquent, elle peut se comporter à la fois comme insert et comme delete en relation avec d'autres opérations.

  Comme l'insert, le move nécessite aussi trois paramètres, un noeud m à déplacer, un noeud n au

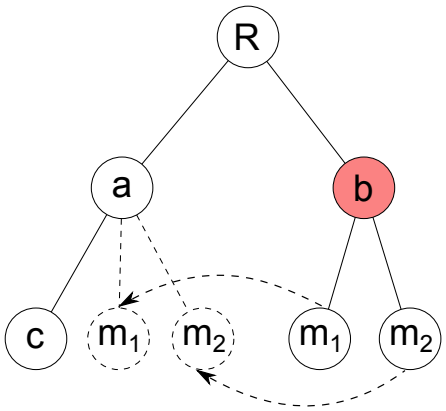


FIGURE 4.8 – Delete dépend de move inférieure

dessous duquel m sera inséré et k la position de m dans la liste des enfants de n. Le move va donc se comporter identiquement à l'insert en relations avec d'autres opérations (fig. 4.9 et fig. 4.10) :

$insert(.,.,n) > move(n,k,m)$
$insert(n,l,.) > move(n,k,m)$ si $l < k$
$move(n,l,.) > move(n,k,m)$ si $l > k$
$delete(o) > move(n,k,m)$ si $parent(o) = n$ *et* $position(o) <= k$
$move(p,l,q) > move(n,k,m)$ si $parent(q) = n$ *et* $position(q) <= k$

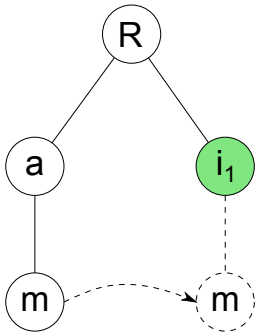


FIGURE 4.9 – Move dépend de l'insert supérieur

L'opération move enlève le sous-arbre de sa place initiale mais ne le supprime pas tel que fait le delete. Tout le sous-arbre étant maintenu, alors les moves inférieurs sont toujours exécutables. Le move n'est pas en relation d'ordre avec des moves inférieurs.

Pour résumer, nous avons les relations d'ordre suivant :

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 $l < k$

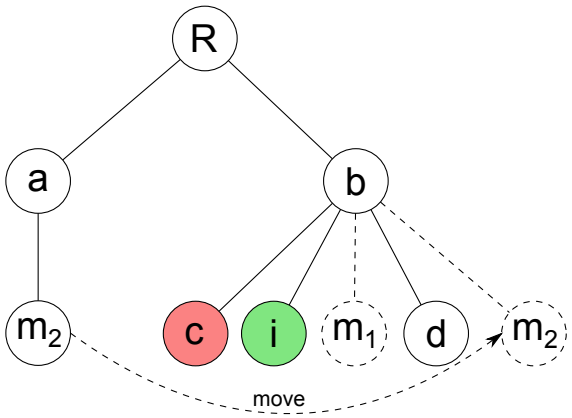


FIGURE 4.10 – Move en relation d'ordre avec d'autres insert et move

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 divisions ces relations d'ordre en deux groupes. Le premier groupe comprend les relations 1, 6 et 11. Ces relations sont des conditions consistantes sans lesquelles l'exécution des opérations concernées n'est pas possible. D'autres relations forment le deuxième groupe. Elles ne conditionnent pas l'exécution des opérations mais assurent leur exactitude en termes de résultat final.

En enchaînant ces relations ensemble, nous avons effectivement identifié des cycles (fig. 4.11) :

– **delete(o)** > **move(n,k,m)** > **delete(o)** où parent(o) = n & position(o) < k et m ∈ T(o)
– **delete(o)** > **insert(n,k,m)** > **move(m,l,p)** > **delete(o)** où parent(o) = n et position(o) < k et m ∈ T(o) & p ∈ T(o)
– **move(n,k,m)** > **insert(o,l,n)** > **move(n,k,m)** où position(m) < l
– ...

Il est impossible d'exécuter les opérations de ces cycles, car leur exécution est mutuellement dépendante. Pour résoudre ce phénomène, une solution consiste à briser l'une (ou plusieurs) des relations composant le cycle. Il n'est pas possible d'annuler l'une des relations appartenant au premier groupe. C'est donc dans le deuxième groupe que nous cherchons à annuler certaines relations. Nous voyons paraître souvent dans les cycles les relations 4, 5, 9 et 10 et constatons que la cause principale est le fait d'utiliser la position exacte, dans l'arbre final, du noeud inséré ou déplacé. Cette position est souvent décalée par rapport à la position correcte dans l'arbre actuel.

Il peut y avoir plusieurs stratégies qui devraient assurer deux choses : l'exécution exacte de toutes les opérations et l'élimination des cycles. Nous proposons, ici, deux solutions possibles :

1. La **première solution** consiste à "relativiser" le paramètre k dans la définition des opérations insert et move. Le valeur de k n'indique pas la position, dans l'arbre final, du noeud inséré (ou déplacé) mais indique la position dans l'arbre actuel où le noeud est inséré (ou déplacé). Il permet d'exécuter correctement les inserts et moves sans dépendre d'autres inserts, moves et deletes.

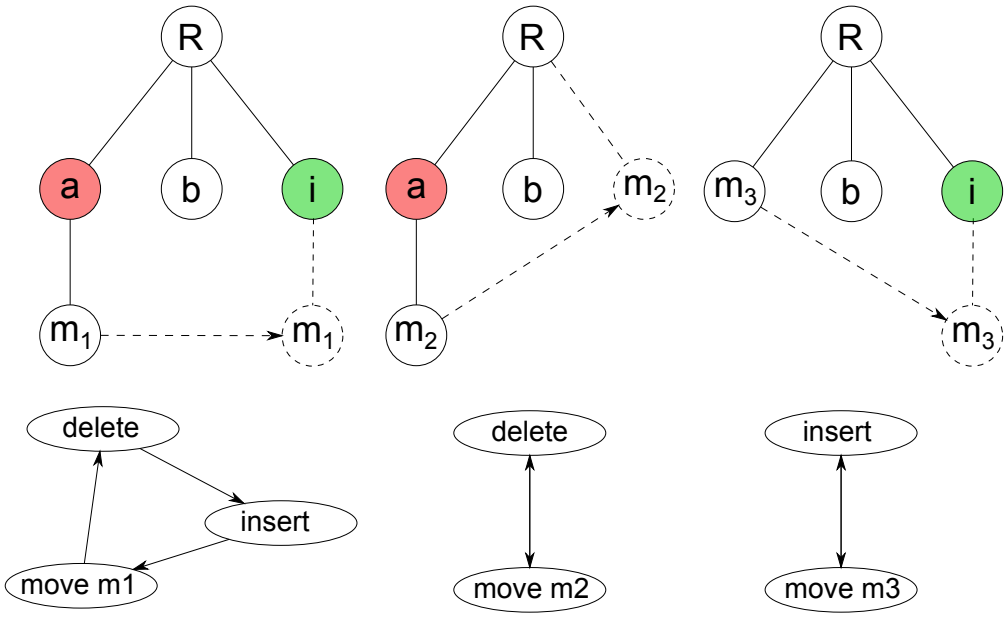


FIGURE 4.11 – Cycles d'opérations

Pour chaque opération insert et move, il faudrait recalculer, tout au début, le paramètre k en fonction du nombre de deletes et moves sortants ainsi que du nombre d'inserts et moves entrants pour les noeuds du même parent et situés à gauche du noeud m (fig. 4.12) :

$$k = k + N(deletes) + N(moves_{sortant}) - N(inserts) - N(moves_{entrant})$$

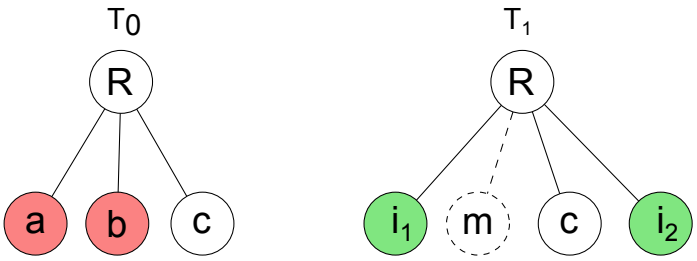
Chaque insert ou delete ou move effectué change la liste des enfants, il nécessiterait également de changer la valeur du paramètre k des opérations insert ou move entrant. Concrètement, après l'insertion ou le déplacement d'un noeud dans la liste des enfants, les positions des noeuds les plus à droite à insérer ou à déplacer, devraient être incrémentées de 1 ; après la suppression et le déplacement d'un noeud en dehors de la liste des enfants, les mêmes positions précédentes devraient être décrémentées de 1.

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 uniquement obtenir une séquence telle que : insert supérieur > insert inférieur > move encore inférieur > delete (contenant l'objet du move) qui ne forme jamais de cycle.

2. La **deuxième solution**, plus radicale, consiste à redéfinir les opérations insert et move. Plus précisément, au lieu d'insérer ou déplacer un noeud à une position précise k, on peut insérer ou déplacer ce noeud après un noeud $left$:

   + $insertAfter(n, left, m)$ insère le noeud m après le noeud left qui est un enfant du noeud n.
   + $moveAfter(n, left, m)$ déplace le noeud m après le noeud left qui est un enfant du noeud n.

   Le noeud n est nécessaire car si left est égal à null, le noeud m est inséré ou déplacé au dessous du noeud n en tant que premier enfant. L'$insertAfter(n, null, m)$ et le $moveAfter(n, null, m)$



Δ	Δ′
delete(a)	delete(a)
delete(b)	delete(b)
insert(R,1,i1)	insert(R,2,i1) où 2 = 1 + delete(a)
move(R,2,m)	move(R,3,m) où 3 = 2 + delete(a) + delet(b) - insert(R,1,i1)
insert(R,4,i2)	insert(R,4,i2) où 4 = 4 + delte(a) + delete(b) - insert(R,1,i1) - move(R,2,m)

FIGURE 4.12 – k est recalculé pour assurer l'exécution correcte des opérations

sont donc possibles si et seulement si le noeud n est présent au moment de l'exécution. Le noeud n peut être lui-même l'objet d'un insertAfter supérieur. Il faudrait 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 du $moveAfter(n, left, m)$ nécessitent que le noeud 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 noeud left.left peut être égal à null. Il faudrait exécuter les insertAfter et moveAfter l'un après l'autre en commençant par les noeuds plus à gauche.

$$insertAfter(n, left.left, left) > insertAfter(n, left, m), left \neq null$$
$$moveAfter(n, left.left, left) > insertAfter(n, left, m), left \neq null$$
$$insertAfter(n, left.left, left) > moveAfter(n, left, m), left \neq null$$
$$moveAfter(n, left.left, left) > moveAfter(n, left, m), left \neq null$$

Il reste à démontrer l'inexistence de cycles. Dans trois opérations delete, insertAfter et moveAfter : les deux derniers peuvent être à la fois dépendante et précédente 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 d'examiner l'une des deux opérations. Examinons donc l'opération insertAfter. L'insertAfter peut dépendre de l'insertAfter (ou moveAfter) à gauche qui peut également dépendre de l'insertAfter (ou moveAfter) encore plus à gauche. Si le noeud 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 dépend d'un insertAfter encore supérieur ou d'un insertAfter (ou moveAfter) à gauche. Ainsi de suite, le processus peut continuer mais dans une seule direction, vers le haut de l'arborescence (fig. 4.13). Il n'y aura jamais d'opération insertAfter (ou moveAfter) qui dépende du tout premier insertAfter, donc il n'y aurai 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 de

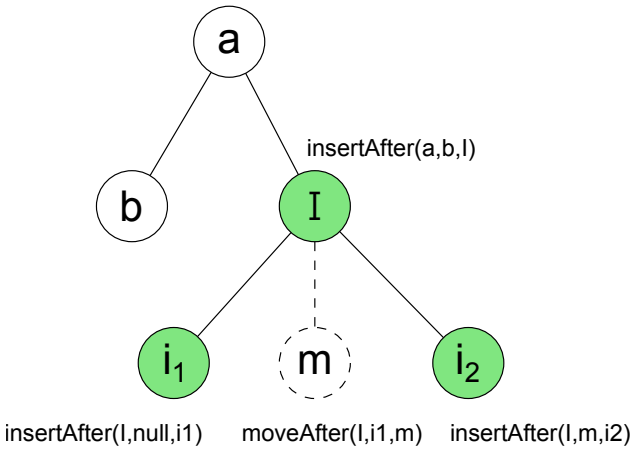


FIGURE 4.13 – InsertAfter dépend d'InsertAfter à gauche et d'InsertAfter supérieur : insertAfter(a,b,I) > insertAfter(I,null,i1) > moveAfter(I,i1,m) > insertAfter(I,m,i2)

calculs qu'elle génère. La deuxième solution n'exige pas de calculs. Par contre, il faudrait enregistrer aussi l'information relative au noeud à gauche pour toutes les opérations insertAfter et moveAfter.

4.1.4 Principe d'acceptation et de refus

Une relation est, pour l'opération dépendante, une condition d'exécution qui est satisfaite si l'opération précédente est exécutée. Une opération est exécutable quand toutes ses conditions d'exécution sont satisfaites et ne l'est pas encore 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 de refuser immédiatement toutes ses opération 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 lie 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 noeud implique dés-incrémenter la position des insert et moves plus à droite (fig. 4.14).
- Refuser un insertAfter ou moveAfter d'un noeud implique changer le paramètre 'noeud à gauche' de l'insertAfter ou moveAfter à droite par le noeud à gauche de l'opération refusée (fig. 4.15).

4.1.5 Opération inversible

Durant l'exécution des opérations, il est parfois nécessaire d'inverser (ou annuler) des opérations acceptées. Mais, avec les opérations telles que définies ci-dessus, cela n'est pas possible. Les opérations ne sont pas inversibles parce que dans leur définition, il manque des données complémentaires. Par

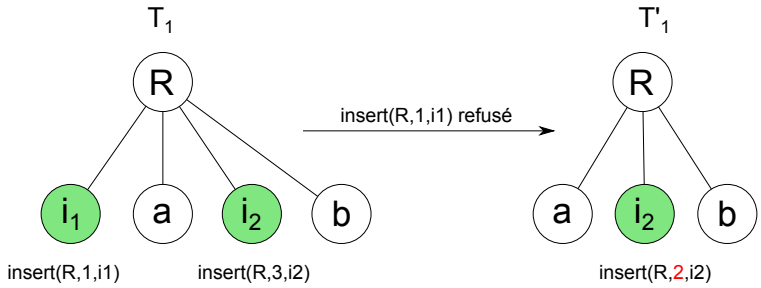


FIGURE 4.14 – Premier insert est refusé, la position du deuxième devrait être dés-incrémenté

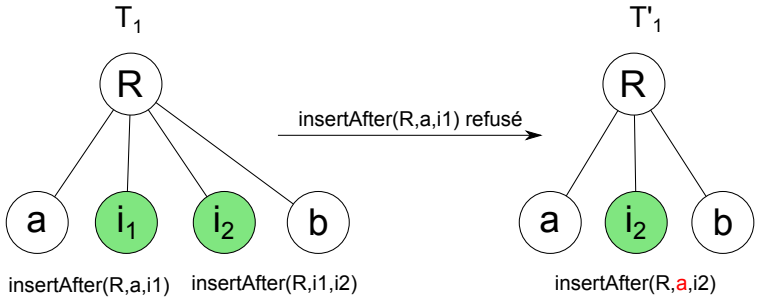


FIGURE 4.15 – Premier insert est refusé, son noeud à gauche devrait devenir le noeud à gauche du deuxième insert

exemple, l'opération delete(m) supprime un sous-arbre enraciné au noeud m. L'inverser nécessite l'opération insert, mais on ne sait pas où exactement réinsérer le sous-arbre supprimé. Marian et al., dans [4], ont défini des *opérations complètes* (*completed deltas*) permettant non seulement d'exécuter mais aussi d'inverser l'opération (voir tableau 4.2). Il est également à noter qu'inverser une opération implique rendre insatisfaites des conditions relatives dans lesquelles l'opération joue le rôle de précédent.

Opération complète	Opération forward	Opération backward
$\overline{delete}(n, k, T(m))$ supprime le sous-arbre dont la racine m est le k-ème enfant du noeud parent n.	delete(m)	insert(n,k,T(m))
$\overline{update}(m, v, ov)$ change la valeur actuel ov du noeud m par la nouvelle valeur v.	update(m,v)	update(m,ov)
$\overline{insert}(n, k, T(m))$ insère le sous-arbre enraciné à m en tant que k-ème enfant du noeud parent n.	insert(n,k,T(m))	delete(m)
$\overline{move}(n, k, m, p, q)$ déplace le sous-arbre enraciné à m, q-ème enfant du noeud q, à être le k-ème enfant du noeud n.	move(n,k,m)	move(q,p,m)

TABLE 4.2 – Tableau des pérations complètes

4.1.6 Algorithme de mise en ordre des opérations

La remise en un certain ordre valide des opérations pour un merge interactif n'est pas strictement nécessaire. En effet, durant le merge, l'utilisateur peut sélectionner seulement les opérations exécutables quelque soit leur ordre. Par contre, 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 dans la liste, il faudrait donc pouvoir les mettre ensemble. En particulier, nous voulons être en mesure de rejouer en séquence toutes les opérations, cela nécessite de 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 les opérations appartenant à une même hiérarchie ensemble et à placer une opération après ses opérations précédentes. L'algorithme se divise donc en deux phases. 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 suffira de mettre l'opération dépendante juste après sa dernière précédente dans la liste. La terminaison de l'algorithme est assurée grâce à la caractéristique acyclique des hiérarchies. Il est pourtant à préciser que l'ordre entre des hiérarchies et des opérations indépendantes n'est pas le sujet de l'algorithme.

Exemple : admettons une liste d'opérations { ω_1,ω_2,ω_3,ω_4,ω_5,ω_6, ω_7 } avec les relations imposées suivant :

$$\begin{cases} \omega_1 > \omega_2 \\ \omega_1 > \omega_3 \\ \omega_2 > \omega_4 \\ \omega_3 > \omega_4 \\ \omega_5 > \omega_6 \end{cases}$$

Il y a deux hiérarchies : la première est composée des opérations ω_1,ω_2,ω_3 et ω_4 ; la deuxième est composée des opérations ω_5,ω_6. L'opération ω_7 est la seule opération indépendante. L'ordre initial est ω_4, ω_1, ω_3, ω_2, ω_5, ω_7, ω_6. En appliquant l'algorithme, on obtiendra successivement les séquences suivant :

1. ω_4, ω_1, ω_3, ω_2, ω_5, ω_7, ω_6
2. ω_3 est remontée devant ω_4

   ω_3, ω_4, ω_1, ω_2, ω_5, ω_7, ω_6
3. ω_1 est remontée devant ω_3

   ω_1, ω_3, ω_4, ω_2, ω_5, ω_7, ω_6
4. ω_2 est remontée devant ω_4

   ω_1, ω_3, ω_2, ω_4, ω_5, ω_7, ω_6
5. ω_6 est placée après ω_5

   ω_1, ω_3, ω_2, ω_4, ω_5, ω_6, ω_7

Algorithm 1 Algorithme de mise en ordre

```
1: function SORT(opList)
2:     l1 ← opList
3:     l2 ← Nil
4:     unchanged ← False
5:     while unchanged = False do                ▷ remonter les opérations précédentes
6:         unchanged ← True
7:         for i ← 1, size(l1) do
8:             op1 ← l1[i]
9:             pos ← size(l2)
10:            for j ← 1, size(l2) do
11:                op2 ← l2[j]
12:                if op2.isBelongTo(op1) then
13:                    unchanged ← False
14:                    pos ← j
15:                    break
16:                end if
17:            end for
18:            l2[pos] ← op1
19:        end for
20:        l1 ← l2
21:        l2 ← Nil
22:    end while
23:    for i ← 1, size(l1) do        ▷ remettre les opérations dépendantes après leurs opérations
   précédentes
24:        op1 ← l1[i]
25:        pos ← size(l2)
26:        for j ← 1, size(l2) do
27:            op2 ← l2[j]
28:            if op1.isBelongTo(op2) then
29:                pos ← j + 1
30:            end if
31:            pos ← j + 1
32:        end for
33:    end for
34:    l1 ← l2
        return l1
35: end function
```

4.2 Implémentation

Le merge interactif ne produit pas lui-même des opérations permettant la transformation entre les versions du document. On utilise donc l'algorithme 3DM pour le faire. Le merge interactif récupère la liste des opérations enregistrées par 3DM mais ne l'exploite pas immédiatement car il faut examiner préalablement les relations éventuelles entre les opérations, présentées dans la section précédente. La première implémentation est réalisé au moyen de Java. Elle mobilise également un algorithme de comparaison de texte, google-diff-match-patch, pour compléter 3DM. En plus, elle est essentiellement testée sur les documents issus du modèle documentaire OptimOffice.

4.2.1 Modèle documentaire OptimOffice

l'**OptimOffice**[1] est une chaîne éditoriale XML générique sans orientation métier particulière, appartenant à la suite Scenari. Elle permet de rédiger et de structurer un unique contenu, et de décliner la publication sur différents supports : papier (dossier, rapport, etc.), diaporama, site web. OptimOffice est proche d'une suite bureautique traditionnelle : la création de contenus s'effectue en choisissant des blocs, agrégés dans les sections.

La création d'un document sous forme de support papier se fait en créant d'abord l'élément racine : **papier**. Une fois l'item créé, l'éditeur fait apparaître plusieurs éléments : titre, sous-titre, paternité et version,etc. Ces éléments, qui apportent des renseignements sur le papier, sont suivis de trois éléments de contenus, qui permettent de structurer le papier.

1. La **partie préalable** sert à écrire tous les éléments d'introduction d'un document : préface, introduction, note liminaire, etc.
2. L'**annexe** n'est pas exactement le symétrique de la partie préalable : elle sert spécifiquement à créer une ou plusieurs annexes, et non une partie conclusive.
3. Le **chapitre** est l'élément principal de structuration du papier (comme la page web pour le site et la diapositive pour le diaporama). Un chapitre est composé de blocs et de sections.

   – Le **bloc** est la plus petite unité de contenu dans Optim : c'est un champ dans lequel on peut saisir du texte ou glisser une image. Un bloc peut éventuellement être titré. Un bloc peut avoir une importance plus grande que d'autres, et on parlera de "Bloc mis en relief". Inversement, si un bloc ne sert qu'à apporter une information secondaire, on choisira le "Bloc complément".
   – La **section** est une unité plus grande qui permet d'agréger des blocs. Il n'existe qu'un seul type de section.
   – En plus des blocs et des sections, Optim utilise trois autres items communs. La **liste d'événéments** est utilisée pour structurer une suite d'événement, et en donner une description succincte. La **galerie d'images** permet d'afficher une ou plusieurs images alignées, titrées et/ou décrites. Le **fragment** permet d'externaliser un ou plusieurs blocs, de manière à pouvoir le(s) réutiliser dans d'autres documents. Un fragment ne peut être inséré qu'au niveau des blocs, et permet de rendre disponible une phrase ou un paragraphe, lorsque l'information doit être répétée ou mutualisée entre différents documents, voire différents auteurs. Un fragment peut lui-même contenir plusieurs blocs et un fragment (principe de la récursivité), ce qui indique la grande complexité d'usage.

1. http://scenari-platform.org/projects/optim/fr/pres/co/index.html

Le texte peut être structuré en paragraphes ou listes (à puce ou ordonnée). A l'intérieur même d'un texte, il est possible de préciser des balises de mise en forme (mise en relief, terme spécifique), d'insérer des liens externes (url, adresse email, ...) et des ressources (images) et d'associer une référence à une expression.

Hormis les ressources multimédia (image, son, vidéo), l'ensemble de l'information est structuré dans un format ouvert respectant la norme XML. Ces contenus sont accessibles et peuvent être exploités en dehors du système Scenari (listing 4.1).

Listing 4.1 – Code XML d'un item Optim

```
<sc:item xmlns:sc="http://www.utc.fr/ics/scenari/v3/core">
  <of:paper xmlns:of="scpf.org:office" xmlns:sp="http://www.utc.fr/ics/
      scenari/v3/primitive" xmlns:sc="http://www.utc.fr/ics/scenari/v3/
      core">
    <of:paperM>
      <sp:title>OptimOffice</sp:title>
    </of:paperM>
    <sp:chap>
      <of:section>
        <of:sectionM>
          <sp:title>Présentation</sp:title>
        </of:sectionM>
        <sp:content>
          <of:fragment>
            <sp:info>
              <of:block>
                <of:blockM/>
                <sp:co>
                  <of:flow>
                    <sp:txt>
                      <of:txt>
                        <sc:para xml:space="preserve" sc:id="t7">
                          sc:inlineStyle role="emphasis">OptimOffice</
                          sc:inlineStyle> est une chaîne éditoriale <
                          sc:inlineStyle role="emphasis">Scenari</
                          sc:inlineStyle>. Elle permet de rédiger et
                          de structurer un unique contenu, et de
                          décliner la publication sur différents
                          supports : papier (dossier, rapport, etc.),
                          diaporama, site web. Optim permet d'avoir un
                           outil de type bureautique, avec les
                          avantages d'une chaîne éditoriale.
                          Contrairement aux autres chaînes éditoriales
                           Scenari, OptimOffice est une chaîne
                          éditoriale générique : elle n'est pas
                          orientée métier, et se fonde sur des items "
                          généralistes".</sc:para>
                      </of:txt>
                    </sp:txt>
```

```
                </of:flow>
              </sp:co>
            </of:block>
          </sp:info>
        </of:fragment>
      </sp:content>
    </of:section>
  </sp:chap>
</of:paper>
</sc:item>
```

4.2.2 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 recours à deux algorithmes différents : 3DM de Lindohlm et Google-diff-match-patch. L'algorithme 3DM est destiné à examiner la structure d'arbre de XML tandis que *google-diff-match-patch* est capable de différencier des textes. Ces deux algorithmes sont complémentaires (fig. 4.16). 3DM peut détecter, dans un document, un paragraphe qui a été édité mais il n'a pas de moyen de dire comment le paragraphe est modifié. C'est effectivement là qu'intervient google-diff. Ce dernier va mettre en évidence les mots et même les caractères qui ont été insérés ou supprimés du paragraphe. Google-diff a plusieurs options mais la plus intéressante est peut être la fonctionnalité *Semantic Cleanup*. Cette fonctionnalité permet d'augmenter la clarté et le naturel de la différence. Par exemple, deux phrases contenant par coïncidence quelques petits termes communs sont considérées complètement différentes par Google-diff, les petits mots communs sont simplement ignorés.

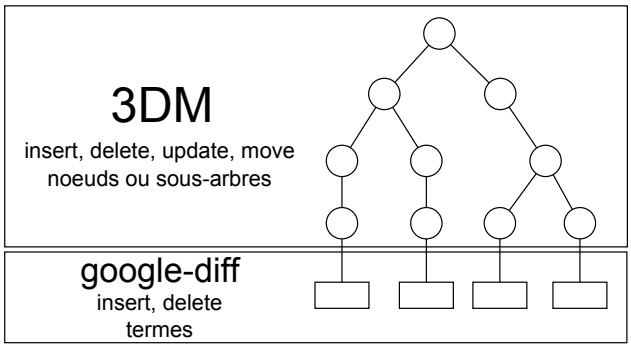


FIGURE 4.16 – 3DM est google-diff sont complémentaires

L'inconvénient principal de 3DM est qu'il s'étend à toutes les entités du document XML même les éléments imbriqués (ou les balises inline). Dans 3DM, un élément devient complètement différent quand on y insère seulement des balises inline sans modifier le texte.

La figure 4.17 illustre le matching par 3DM entre deux textes : le premier ne contient aucune balise inline, le deuxième contient une balise $< b >$ autour du terme "inline". 3DM met tout le premier texte "there is no inline tag" en correspondance avec la partie avant le balise inline du deuxième texte "there is no". Le reste, y compris le balise inline et la partie après le balise inline, est considéré comme nouveau et à insérer. Par conséquent, 3DM enregistre jusqu'à 4 opérations :

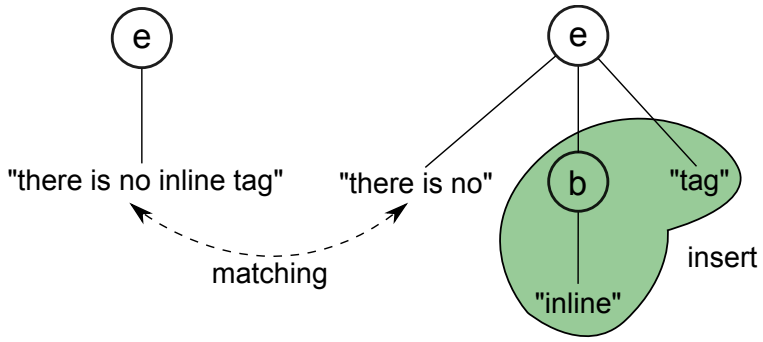


FIGURE 4.17 – Matching en présence des balises inline

– update "Here is not inline tag" en "Here is not"
– insert $< b >$
– insert "inline" dans $< b >$
– insert "tag"

Cet inconvénient est vraiment gênant dans le cas des documents OptimOffice. Puisque leur textes contiennent effectivement beaucoup de balises inline (mise en forme, liens, référence, etc.). Ce qui causera le fonctionnement incorrect de 3DM ainsi que la taille énorme de la liste des opérations.

La meilleure solution est d'empêcher 3DM de s'étendre à toutes les entités du document XML. En plus, il faut déclarer à 3DM les entités à étendre, par exemple bloc ou paragraphe. Actuellement, 3DM ne supporte pas cette fonctionnalité qui dépend beaucoup du modèle documentaire utilisé. Notre solution provisoire est donc faire échapper les balises inline au parser XML de 3DM. Pour ce faire, il suffit d'encadrer les balises inline dans des sections CDATA [2] en utilisant un transformateur XSLT, le texte à l'intérieur CDATA est ignoré par le parser. Le balise inline est considéré comme le texte, 3DM enregistre alors une seule opération update entre deux textes. On peut ensuite montrer comment deux textes sont différents au moyen de google-diff. Par contre, Google-diff considère une balise inline comme une chaine de caractères, il n'est pas capable de distinguer les balises des textes contenus.

There is no inline tag.

En fin, lors de l'enregistrement, des sections CDATA serons enlevées pour rendre les balises inline à nouveau opérationnels.

4.2.3 Opérations de 3DM

A l'opposé des autres outils différentiels, 3DM n'utilise pas un script d'opérations afin de fusionner les fichiers. Il s'appuie totalement sur son matching établi entre deux arbres. Un noeud qui n'est pas matché est soit inséré soit supprimé. 3DM construit l'arbre final par insertion au fur et à mesure des noeuds en commençant par la racine suivie par ses enfants. Les enfants d'un noeud sont insérés si tous les sous-arbres à gauche sont complètement insérés. Pendant le processus, si un noeud a subit du changement ou n'est pas matché, une opération correspondante est enregistrée.

2. http://www.w3schools.com/xml/xml_cdata.asp

3DM supporte les opérations update, delete, insert, move et même copy. Nous avons écarté l'opération copy à cause des raisons mentionnées précédemment à propos de cette opération. Avec 3DM, il est possible via l'option "-c threshold" dont threshold est un seuil de similarité, de fixer le niveau de détection de copy. Dans le cas des documents OptimOffice, avec un seuil fixé à 2000, il n'y a pratiquement plus d'opération copy identifiée.

Edit log

3DM enregistre toutes les opérations dans un fichier de format XML appelé *edit log* (fig 4.2). L'ordre d'enregistrement des opérations est celui de l'insertion des noeuds en vue de construire l'arbre final.

Listing 4.2 – Un edit log de 3DM

```
<edits>
  <delete src="/0/2" originTree="branch1" originList="/0" />
  <update path="/0/0/0" src="/0/0/0" originTree="branch1" originNode="
      /0/0/0" />
  <move path="/0/1/2" src="/0/2/1" plc="/0/1" originTree="branch1"
      originNode="/0/1/2" />
</edits>
```

Chaque enregistrement correspond à une opération précise, les propriétés de l'opération sont décrites par des pairs "attribut-valeur" :

<**operation path**=p **src**=b **plc**=f originTree=T originList=l originNode=n />

Le tagname **operation** indique le type d'opération : insert, delete, update ou move. Nous nous intéressons, parmi d'autres, à trois attributs *path*, *src* et *plc* dans lesquels le plc n'est pas natif dans 3DM mais que nous avons ajouté :

1. **Path** indique la position du noeud dans l'arbre final.
2. **Src** donne la position du noeud dans l'arbre original.
3. **Plc** indique la position du parent adoptif d'un noeud inséré ou déplacé dans l'arbre original.

Une opération ne possède pas nécessairement tous ces trois attributs :

- <**delete** src="." .../>
  src : position dans l'arbre original du noeud auquel le sous-arbre supprimé est enraciné ;
  Src permet de sélectionner le sous-arbre enraciné en m de la définition delete(m).

- <**insert** path="." plc="." .../>
  path : position dans l'arbre final du noeud inséré ;
  plc : position dans l'arbre original du père adoptif du noeud inséré ;
  Path permet retrouver le noeud m ainsi que la position k lorsque plc permet de retrouver le noeud parent n.

- <**update** path="." src="."/>
  src : position dans l'arbre original du noeud mis à jour ;
  path : position dans l'arbre final du noeud mis à jour ;
  Src permet de retrouver le noeud m à modifier lorsque path permet de récupérer la nouvelle valeur v.

- <**move** path="." src="." plc="." .../>
  src : position dans l'arbre original du noeud déplacé ;
  path : position dans l'arbre final du noeud déplacé ;
  plc : position dans l'arbre original du parent adoptif du noeud déplacé ;
  Src permet de sélectionner le noeud à déplacer m quand plc permet de retrouver le noeud parent n. Le paramètre k est calculé à partir de path.

Par rapport aux opérations insert et move, l'attribut plc peut être absent quand le noeud parent d'adoption n est lui-même l'objet d'une autre insertion.

Path de noeud
La valeur des attributs ci-dessus sont des chaînes de caractères telles que "/0/0/1" ou "/0/0/0/1". En effet, il s'agit de la notation de *node path* propre à 3DM (figure 4.18). "/0" est la racine, "/0/0" et "/0/1" sont le premier et le second enfant de la racine et ainsi de suite.

Un noeud identifié par $path_1$ est un descendants d'un autre noeud identifié par $path_2$ quand le $path_2$ est une sous-chaîne commençant par le début du $path_1$ ($pth_2 \subset pth_1$). Par exemple, "/0/0/0/1" et "/0/0/1/0" sont les descendants de "/0/0" car "/0/0" $\subset$ "/0/0/0/1" et "/0/0" $\subset$ "/0/0/1/0". La longueur d'un path est égal au nombre de ses composants, par exemple "0/0/0/1" a quatre composants (0,0,0,1) et a une longueur de 4. La longueur du path est aussi notée par $\|path\|$.

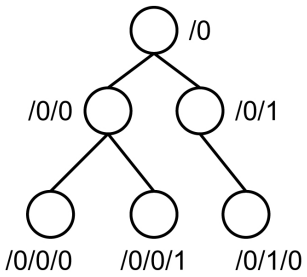


FIGURE 4.18 – Node Path dans 3DM

Pour calculer la position du noeud dans la liste des enfants de son parent, il suffit d'extraire le dernier composant du path. Par exemple, un noeud identifié par "/0/0/0/1" est le deuxième enfant de son parent.

Des noeuds sont frères quand leurs paths ne sont différents que du dernier composant. Par exemple "/0/0/0" et "/0/0/1" et "/0/0/2" sont des frères, leur parent est identifié par "/0/0". Le noeud le plus à gauche est celui dont le dernier composant est moins élevé.

Relations entre les opérations de 3DM
Les opérations 3DM détaillées ci-dessus sont totalement compatibles avec ce que nous avons défini. En plus, elles utilisant des positions référencées dans l'arbre original ou dans l'arbre final, ce sont des opérations deltas. On peut donc appliquer les résultats obtenus dans "Relation d'ordre" pour elles. Pour examiner si une opération est dépendante d'autres opérations, on peut raisonner en fonction des trois attributs (path, src et plc) :

1. $insert_1(o, l, n) > insert_2(n, k, m)$ quand
   path($insert_1$) $\subset$ path($insert_2$) et $\|pth(insert_2)\| = \|pth(insert_2)\| + 1$
   Exemple :
   $insert_1$: <insert path="/0/0/1" etc. />

$insert_2$: <insert path="/0/0/1/1" etc. />
$insert_1 > insert_2$ car path="/0/0/1" $\subset$ path="/0/0/1/1"

2. $insert(o, l, n) > move(n, k, m)$ quand
   path($insert$) $\subset$ path($move$) et $\|pth(move)\| = \|pth(insert)\| + 1$
   Exemple :
   $insert$: <insert path="/0/0/1" etc. />
   $move$: <move path="/0/0/1/1" etc. />
   $insert > move$ car path="/0/0/1" $\subset$ path="/0/0/1/1"

3. $move(n, k, o) > delete(m)$ si $o \in T(m)$ quand
   src($delete$) $\subset$ src($move$)
   Exemple :
   $delete$: <delete path="/0/0" etc. $/ve >$
   $move$: <move src="/0/0/1/2" etc. />
   $move_1 > move_2$ car "/0/0 $\subset$ "/0/0/1/2"

L'implémentation actuelle du merge interactif n'implémente que les relations du premier groupe. Elle ne traite pas encore des cycles. Le positionnement de certains noeuds insérés ou déplacés peuvent être non exact. Dans la prochaine implémentation, pour éliminer les cycles, l'une des deux solutions mentionnées ci-dessus sera utilisée. Néanmoins, la deuxième solution est préférable sachant que l'information sur le noeud à gauche d'un noeud inséré (ou déplacé) peut être enregistrée par 3DM.

4.2.4 Vue globale de l'implémentation Java

Ce premier prototype du merge interactif est implémenté en Java. La totalité du code source se trouve annexe, nous en donnons une vue globale de la structure de l'implémentation.

La classe principale (main class) est `DemoMain`. Il s'agit d'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`. Cette classe 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 de noeuds de 3DM.

Les fichiers XML sont parsés et traités par le 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 [3]. La classe `TreeCellCustomRenderer` permet de changer l'affichage de l'arbre.

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

4.2.5 Interface graphique du merge interactif

L'interface principale du merge interactif (fig 4.19) est constituée de trois panneaux :

1. Le premier panneau affiche la liste des opérations regroupées en hiérarchies (fig 4.20). Une opération est représentée par son type (e.g. update, delete, insert, move), le noeud concerné et d'autres paramètres (e.g. le noeud parent adoptif pour insert et move). Les opérations activées sont susceptibles de s'exécuter immédiatement. Les opérations désactivées sont dépendantes. Elles doivent attendre l'exécution de leurs opérations précédentes afin d'être activée et donc exécutable. 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 noeuds internes et non aux feuilles textuel. En cliquant sur un noeud, le contenu textuel de ce noeud est affiché dans le troisième panneau.
3. Le troisième panneau affiche le contenu textuel du document XML sous un format purement textuel (sans listes, sans tableaux, sans images, ...). Les titres (du papier, du chapitre, du session, ...) sont en gras et les paragraphes sont espacés. Les balises inline ne sont pas affichées, ils seront affichés seulement pour montrer qu'elles sont ajoutées ou supprimées (fig. 4.24).

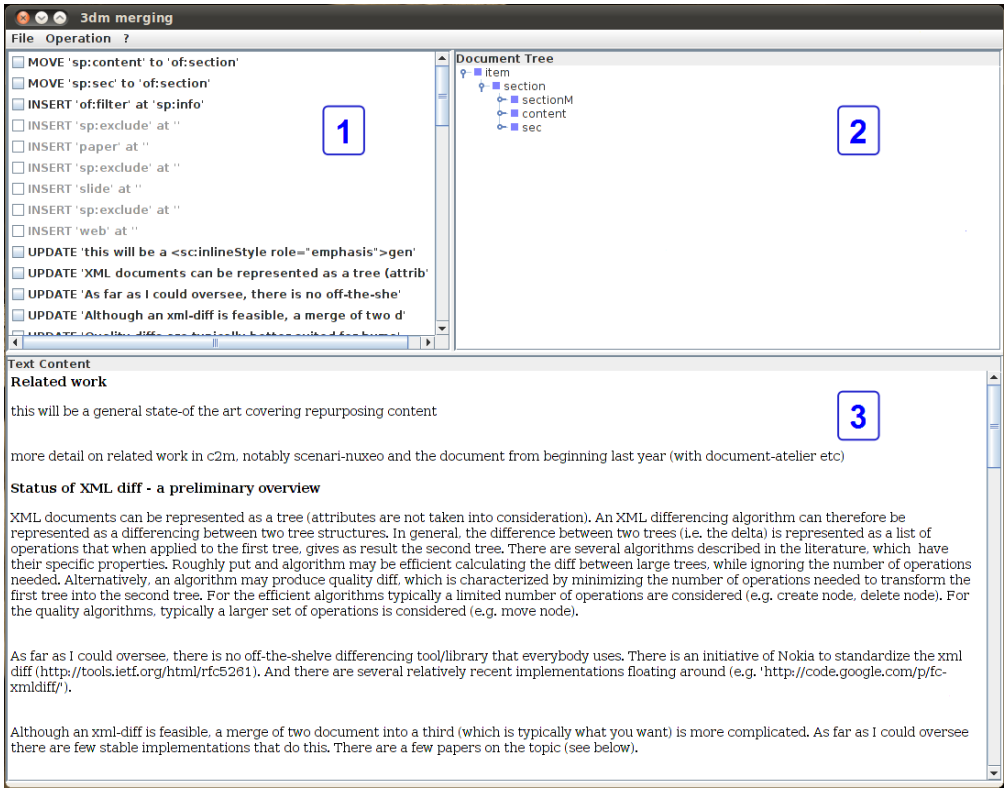


FIGURE 4.19 – Interface principale du merge interactif : 1 - Liste d'opérations ; 2 - Structure d'arbre du document XML ; 3 - Contenu textuel du document XML

En cochant une opération activée dans le panneau des opérations, cette opération sera pré-visualisée à la fois dans le panneau de structure d'arborescence et le panneau de contenu (fig 4.21). Dans le panneau de structure, le code de couleur est vert pour insert, rouge pour delete et orange pour

update. Dans le panneau de contenu, le texte supprimé a la couleur rouge et est rayé tandis que le texte inséré a la couleur verte (fig 4.25). L'opération move est représentée par un noeud supprimé et par un noeud inséré.

L'utilisateur peut éventuellement constater l'activation des opérations dépendantes de l'opération cochée à condition que d'autres conditions aient déjà été satisfaites (fig. 4.22).

Un popup s'affiche pour demander la confirmation de l'utilisateur. L'utilisateur peut accepter (apply), refuser (ignore) ou ne rien faire (cancel). S'il décide d'accepter, le document est réellement changé et il ne pourra pas revenir en arrière. S'il refuse, l'opération est inversée. Il est important de noter que non seulement l'opération en question mais aussi toutes ses opérations dépendantes (directes ou par transitivité) seront effacées du panneau des opérations (fig. 4.23). S'il choisit de ne rien faire, l'opération est inversée et décochée mais toujours présente dans le panneau. En plus, les opérations dépendantes sont à nouveau désactivées.

Il est à préciser que l'utilisateur n'a pas besoin de confirmer chaque opération. Au contraire, il peut cocher plusieurs opérations. En effet, quand il coche une nouvelle opération, l'opération cochée est réellement appliquée sur le document. L'utilisateur peut également décocher une opération cochée. Dans ce cas, l'opération et ses dépendantes cochées sont immédiatement inversées et décochées. L'opération reste activée mais ses dépendantes sont désactivées. La confirmation de plusieurs opération est identique à la confirmation d'une seule opération.

Pour charger les fichiers d'entrée tels que le fichier de base, le fichier final et le fichier des opérations, l'utilisateur doit ouvrir le menu "File" puis choisir "open". Une nouvelle fenêtre s'ouvre en demandant à l'utilisateur d'entrer les noms des fichiers (fig. 4.26). Le menu "File" offre également d'autres fonctionnalités : "save", "reload" et "quit". "save" permet de sauvegarder l'arbre actuel en document XML avec un nom donné par l'utilisateur. "reload" permet de recharger les fichiers d'entrées et donc de recommencer le merge. "quit" permet de quitter l'application.

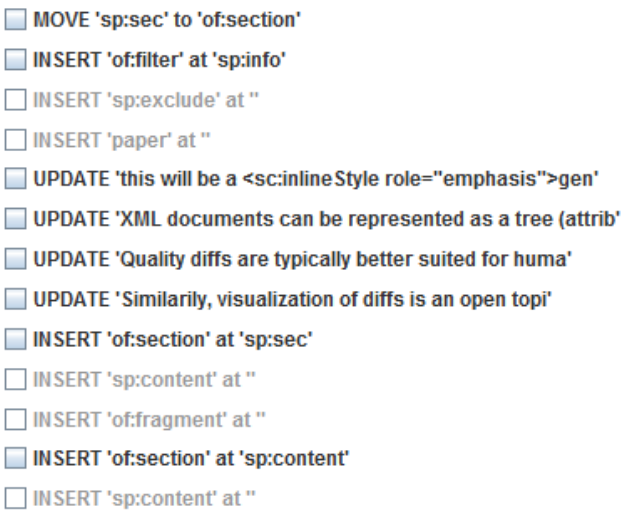


FIGURE 4.20 – Opérations regroupées par hiérarchies : en noir signifie exécutable immédiatement ; en gris signifie non-exécutable

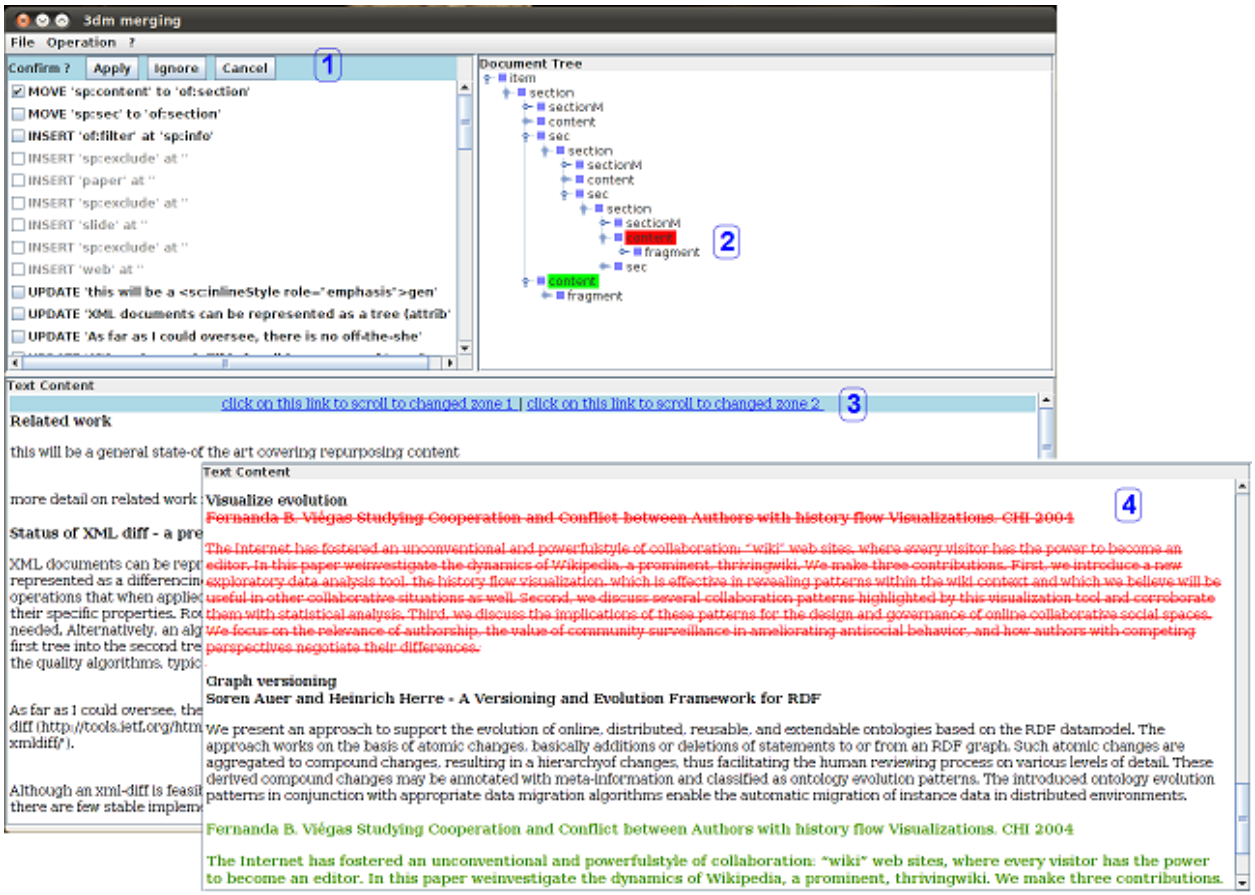


FIGURE 4.21 – Pré-visualiser une opération : 1 - popup de confirmation ; 2 - changement structurel ; 3 - liens permettant de défiler directement aux zones textuels changées ; 4 - changement textuel

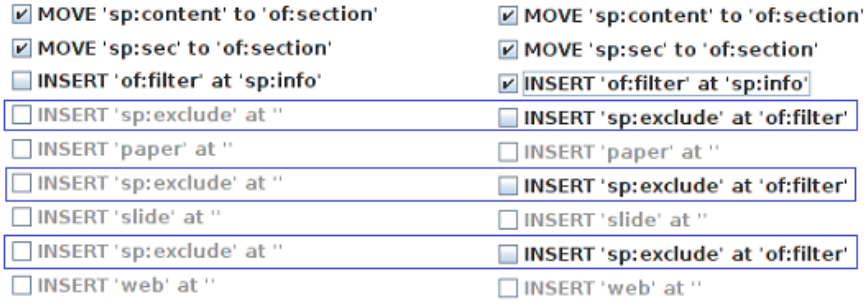


FIGURE 4.22 – Les opérations insert inférieures sont activées et mises à jour sur le noeud parent

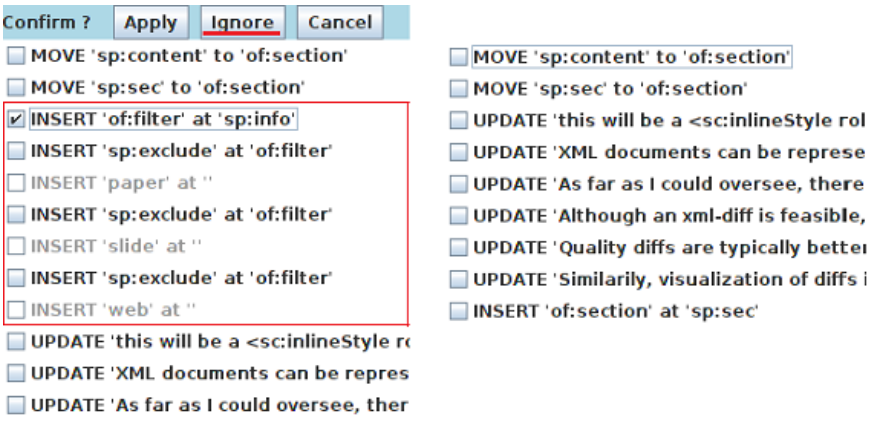


FIGURE 4.23 – Refuser une opération signifie également refuser toutes ses dépendantes

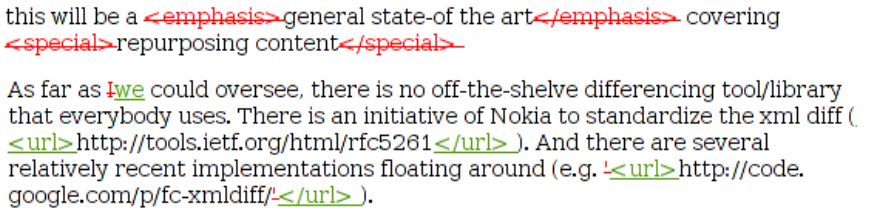


FIGURE 4.24 – Les balises inline sont supprimés et insérés

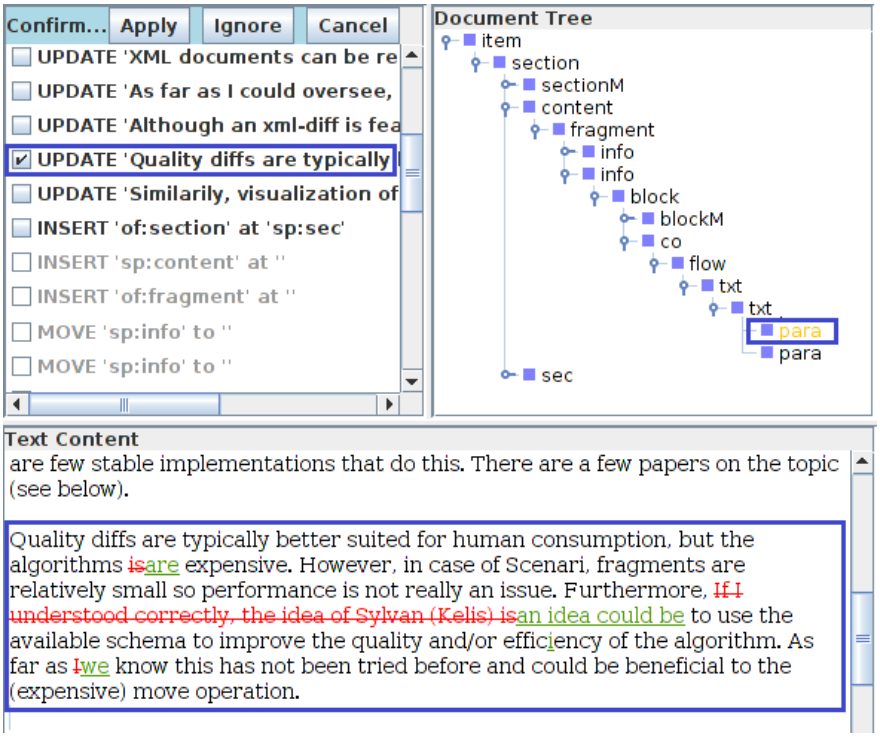


FIGURE 4.25 – Changement du texte d'un noeud updaté

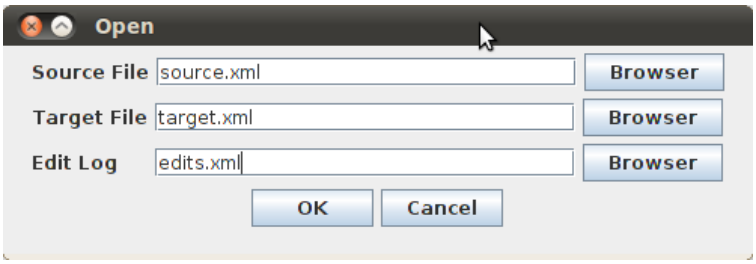


FIGURE 4.26 – Fenêtre de configuration des fichiers d'entrée : source - fichier de base ; target - fichier final ; edit log - fichier des opérations

4.2.6 Fonctionnalités à étudier & implémenter

Étant le premier prototype, l'implémentation actuelle est encore limitée dans son applicabilité. Les fonctionnalités à étudier et implémenter incluent :

1. Gérer les conflits lors du three-way merge
2. Valider le modèle du document
3. Agréger les opérations élémentaires
4. Comparer des réseaux de fragments

Gérer les conflits lors du three-way merge
3DM est en réalité un outil de three-way merge permettant de fusionner trois versions du document (base : T_0 et deux branches : T_1, T_2). Cependant, nous l'utilisons jusqu'à présent, principalement pour fusionner deux versions en substituant T_1 par T_0 ou T_2 par T_0. Ainsi, l'implémentation actuelle du merge interactif est adaptée à cette substitution. Elle est donc pour l'instant un two-way merge interactif. Nous voulons l'étendre à un three-way merge interactif dans lequel l'origine de chaque opération serait précisée (e.g. venant de la branche T_1 ou de la branche T_2).

Étant donné que les versions T_1 et T_2 ont été indépendamment éditées, il est très probable qu'elles contiennent des changements conflictuels. Ces conflits sont détectés pendant la fusion automatique par 3DM. Au lieu d'arrêter le processus, 3DM essaie de résoudre la situation conflictuelle pour que l'algorithme puisse continuer. En effet, il conserve une opération privilégiée. C'est cette dernière qui est enregistrée dans l'edit log par 3DM. Par exemple, un noeud a été différemment mis à jour dans T_1 et T_2, 3DM résout cette situation conflictuelle en choisissant l'update de T_1. Mais, il se peut que cette option ne plaise pas à l'utilisateur. Alors pour palier cet inconvénient, le three-way merge interactif doit être en mesure de présenter à l'utilisateur les deux opérations conflictuelles et lui laisser choisir l'une des deux, même si elle n'était pas enregistrée par 3DM. Il y aura donc deux difficultés à résoudre :

1. Comment récupérer les opérations non-sélectionnées par 3DM ?
2. Lorsque l'utilisateur choisit une opération qui n'est pas celle enregistrée et utilisée par 3DM, est-ce qu'il y aura l'impact sur d'autres opérations enregistrées ?

Pour répondre au premier problème, on pourra utiliser le *conflict log* de 3DM. En effet, c'est dans ce fichier que 3DM enregistre tous les conflits détectés (listing 4.3). Il faudra alors apprendre comment l'utiliser et le fusionner avec l'edit log d'une manière la plus correcte. Une fois que les paires d'opérations conflictuelles étaient identifiées, une question supplémentaire se posera : comment les représenter en sorte que l'utilisateur puisse l'appréhender et prendre une décision adéquate.

A chaque situation conflictuelle, 3DM décide de la débloquer et souvent par choisir l'une des deux opérations. Le processus continue avec cette opération tandis que l'autre opération est abandonnée. L'utilisateur pourrait choisir l'opération non-sélectionnée par 3DM. Il serait donc obligé d'oublier l'opération sélectionnée qui était peut être en relation avec d'autres opérations d'où la question de consistance. Pour vérifier cette question, il faudrait regarder au plus près la nature et les caractéristiques de chacun des conflits détectés par 3DM. 3DM divise les conflits en deux catégories : conflits (tableau 4.4) et avertissements (warning) (tableau 4.3).

Les cinq premiers type de conflits ne posent pas de problème car :

– Dans 1 et 2, il y a une seule option

Listing 4.3 – Un conflict log de 3DM

```
<conflictlist>
  <conflicts>
    <update>
    Node updated in both branches, using branch 1
    <node tree="merged" path="/0/0" />
    <node tree="base" path="/0/0" />
    <node tree="branch1" path="/0/0" />
    <node tree="branch2" path="/0/0" />
    </update>
  </conflicts>
  <warnings>
    <delete>
    Modifications in deleted subtree.
    <node tree="merged" path="/0" />
    <node tree="base" path="/0/1" />
    <node tree="branch1" path="/0/1" />
    </delete>
  </warnings>
</conflictlist>
```

Type de warning		Description et Résolution
1	Update	**Description** Un noeud a été édité dans deux branches et les nouvelles valeurs sont identiques **Résolution** Utiliser l'update de la branche T_1
2	Insert	**Description** Deux noeuds insérés à la même position dans deux branches et de même valeur **Résolution** Utiliser l'insert de la branche T_1
3	Insert	**Description** Des noeuds de différentes valeurs insérés dans deux branches au même parent **Résolution** Insérer les noeuds venant de la branche T_1 suivi par les noeuds venant de la branche T_2
4	Delete	**Description** Éditer dans un sous-arbre supprimé **Résolution** Supprimer le sous-arbre

Table 4.3 – Tableau des avertissements détectés par 3DM et comment 3DM les résout

Type de conflit		Description et Résolution
5	Update	**Description** Un noeud a été édité différemment dans deux branches **Résolution** Utiliser l'update de la branche T_1
6	Move	**Description** Changer la séquence d'une liste d'enfants différemment dans deux branches **Résolution** Utiliser la séquence de la liste d'enfants de la branche T_1
7	Move	**Description** Un noeud est déplacé au nouveau parent dans une branche et déplacé seulement à l'intérieur d'une liste d'enfants dans l'autre branche **Résolution** Ignorer le déplacement dans la liste d'enfants
8	Move	**Description** Un noeud est déplacé en dehors de la liste d'enfants aux différents nouveaux parents dans deux branches **Résolution** Accepter tous les déplacements ce qui implique des copies supplémentaires
9	Move	**Description** Un noeud est déplacé dans une branche et supprimé dans l'autre branche **Résolution** Ignorer le delete

TABLE 4.4 – Tableau des conflits détectés par 3DM et comment 3DM les résout

– Dans 3, toutes les opérations sont enregistrées
– Dans 4, si l'utilisateur refuse le delete, alors on lui suggère l'update.
– Dans 5, l'utilisateur peut choisir l'un des deux upates sans aucun souci car l'update est indépendant

Le conflit numéro 8 ne pose pas de problème non plus puisque l'utilisateur peut accepter un move et refuser l'autre move. Par contre, d'autres conflits relatifs à l'opération move causeront effectivement des problèmes :

– Dans 6, si dans la même liste d'enfants, il y a également un insert. La position du noeud inséré dépend de l'utilisation de la séquence de la branche T_1 ou de la séquence de la branche T_2.
– Dans 7, de même que 6, le move pourrait impacter sur les inserts.
– Dans 9, si le delete est choisi, il pourrait avoir un effet sur les moves inférieurs.

Il faut recalculer les relations et les positions des noeuds à insérer (ou à déplacer) à chaque changement d'opération non-sélectionnée avec l'opération sélectionnée.

Valider le modèle du document
Actuellement, le merge interactif change le document XML sans tenir compte son modèle. Par ailleurs, 3DM ne le traite pas non plus. Il est probable que certaines opérations identifiées ne valident pas le modèle du document. Un modèle documentaire tel que OptimOffice impose des éléments obligatoires et des éléments optionnels. Par exemple, il demande toujours un titre pour tous les chapitres et les sections insérés. Il est donc nécessaire d'insérer l'élément titre si l'utilisateur acceptait l'insert du contenu de cette section. Un autre exemple est que l'élément fragment doit impérativement contenir au moins un élément bloc. Quand l'utilisateur choisit d'insérer l'élément fragment, il faut aussi insérer au moins un élément bloc. Il n'est pas nécessaire d'avertir l'utilisateur de ces obligations durant le merge mais l'avertir lors de l'enregistrement du document.

Agréger les opérations élémentaires
OptimOffice impose des structures auxiliaires répétitives servant à la génération du document, par exemple : "$< sp : info >< of : block >< sp : co >< sp : flow >< sp : txt >< of : txt >< sc : para >$ ...". Beaucoup de niveaux ne sont pas significatifs. Il vaudrait mieux suggérer une seule opération synthétique au lieu de demander à l'utilisateur d'accepter chacun de ces niveaux d'opération.

Pour réaliser cette fonctionnalité ainsi que la fonctionnalité "Valider le modèle du document", il faudrait étudier de plus près le modèle OptimOffice et sa structure XML interne associée.

Comparer des réseaux de fragments
Les documents issus du modèle OptimOffice sont souvent fragmentés. Ils peuvent être composés de plusieurs 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 avec 3DM ou d'autres outils. Cependant, on peut inclure les contenus de tous les fragments dans le fragment racine en vue de créer un seul fichier. Ensuite, on compare ces fichiers l'un avec l'autre et applique le merge interactif. Lors de l'enregistrement du fichier, il faudra refragmenter le document résultant.

Chapitre 5

Conclusion & Perspectives

Dans les environnements auteurs collaboratifs où les documents numériques sont facilement partagés et édités par plusieurs auteurs, il est très important de pouvoir visualiser les différences entre les sources ainsi que les fusionner efficacement. De multiples outils et méthodes de differencing et merging existent sur le marché. Beaucoup sont spécialisés pour le differencing et moins nombreux sont spécialisés pour le merging. La plupart sont généralistes ou orientés données et très peu sont destinés à traiter les documents XML dans lesquels le texte reste le composant principal. D'une part, ils s'intéressent plutôt à déterminer les changements sur l'arborescence représentant le document que déterminer les changements sur le document même, d'autre part, ils ne favorisent pas la représentation des changements synthétiques et compréhensibles pour l'utilisateur humain.

Après une étude comparative, nous avons choisi *XML three-way Merging and Differencing Tool - 3DM* de Tancred Lindohlm, car il paraît être le meilleur candidat pour notre corpus documentaire. Tout d'abord, 3DM n'est pas limité aux trois opérations basiques insert, delete, update mais supporte aussi l'opération move et même l'opération copy. Il enregistre les changements de manière très compréhensible. Son heuristique tree-matcher est remarquable et fonctionne très bien avec les XML généralistes. En fin, son code source est disponible en open-source et il semblait assez aisé de modifier les modules, le tree-matcher par exemple, pour réaliser des optimisations. Nous avons également eu recours à google-diff-match-patch qui est un comparateur basé texte. 3DM et google-diff-match-patch sont complémentaires, car le premier est destiné à examiner les changements dans la structure du document (l'unité est un noeud) alors que le second est utilisé pour détecter les changements détaillés dans le texte (l'unité est un terme).

En contexte collaboratif, l'utilisateur a besoin de s'assurer de la consistance de chacun des changements qui ont été effectués sur son document. En plus, l'édition concurrente d'un document cause des situations conflictuelles que seule l'intervention humaine peut résoudre. Actuellement, les solutions de fusion automatiques ne satisfont pas à ces besoins. Fort de ce constat, nous avons proposé et développé le merge interactif dont l'objectif est d'aider efficacement l'utilisateur dans ses choix de fusion des changements. L'utilisateur peut accepter certaines opérations de modification et refuser d'autres pendant que le merge interactif calcule et modifie les opérations pour assurer qu'elles soient exécutables et correctes.

Le merge interactif nous a aussi permis d'identifier les relations d'ordre entre les opérations. Ces relations confirment que l'exécution de certaines opérations dépend de l'exécution préalable d'autres opérations. Nous les avons divisées en deux groupes. Le premier groupe est lié à la faisabilité de l'opération (est-elle exécutable ?) alors que le second est lié à l'exactitude de l'opération (est-elle

correcte ?). En explorant les relations possibles, nous avons été conduits à trouver des cas où les opérations sont mutuellement dépendantes et forment un cycle fermé. Nous avons alors dû réviser les définitions initiales des opérations et il était nécessaire de modifier celles de l'opération insert et de l'opération move. Le merge interactif a été partiellement implémenté en Java. L'interface utilisateur est assez intuitive. L'utilisateur peut y visualiser successivement, sans ambiguïté, les changements en même temps sur la structure d'arborescence et sur le contenu textuel du document, avant de prendre une décision finale. Il est possible de vérifier les opérations séparément et aucun ordre d'exécution n'est imposé.

Nous avons conscience que le travail réalisé durant de ce stage doit être directement évalué sur des vrais cas et avec de vraies données. Cependant, le premier prototype du merge interactif, bien qu'ayant été testé sur des jeux de documents réels, nous semble trop prématuré d'être appliqué dès à présent sur le terrain. Il méritait d'être encore approfondi et optimisé. Les perspectives de ce stage peuvent porter sur de nombreux points :

1. **Orienter l'auteur**,
   L'auteur s'intéresse principalement aux changements dans la structure logique (e.g. chapitre, section, sous-section, etc.) et le contenu textuel du document. Il n'a absolument aucun intérêt de connaître le XML ou le code XML du document. Le fait de montrer la code interne XML du document n'est très probablement pas nécessaire pour l'auteur. Il serait plus utile d'afficher la structure logique du document. Dans ce cas, les opérations doivent référencer à cette structure logique, ce qui n'est pas trivial, car il faudrait définir la granularité de la structure logique et expliciter et maintenir la liaison entre la structure logique et la structure interne XML sur laquelle les opérations sont initialement référencées.
   Il est aussi important que les opérations soient intelligibles et en nombre réduit pour l'auteur. Il faudrait donc définir les opérations synthétiques et de haut-niveau qui sont réalisé à de multiples niveaux hiérarchiques dans la structure du document.

2. **Augmenter la performance des algorithmes de différentiels**,
   En profitant la connaissance sémantique du modèle documentaire spécifique, ici le modèle OPTIM, il serait possible d'augmenter la performance des algorithmes de différentiel. D'un côté, cela permettrait d'améliorer le matching entre les arbres représentant des documents à l'aide des correspondances entre les entités significatives. De l'autre côté, cela permettrait d'augmenter la qualité du delta en éliminant les opérations "non-significatives" ou non-valides vis-à-vis du modèle.

3. **Approfondir le principe de comparaison des documents fragmentés**,
   Pour comparer ou fusionner des documents fragmentés, la solution actuelle consiste à défragmenter ces documents par l'inclusion de tous les fragments du document dans un seul fichier, en l'occurrence le fragment racine. La comparaison et la fusion seront effectuées sur ces fichiers d'agrégat. Les données à propos de la fragmentation initiale sont conservées durant la fusion et font également l'objet de la comparaison. Elles seront utiles pour pouvoir refragmenter le document résultant. Dans le cas où l'identité des fragments était connue, il suffirait peut être de comparer seulement les fragments modifiés sans comparer les documents en entier. Les documents fragmentés n'exigent pas une spécification dans l'algorithme de diff ou de merge. Néanmoins, une question posée est comment représenter graphiquement le résultat pour mettre en évidence la structure de fragmentation du document à côté de la structure logique et du contenu du document.

4. **Augmenter la dimension IHM**,
   Comment augmenter la compréhensibilité, la manipulation et la contrôle des utilisateurs finaux, les auteurs ?

5. **Étudier des principes généraux de la genèse documentaire**,
   Comment gérer et exploiter les liens logiques et historiques entre les fragments dans la logique collaborative.

Annexe A : Bilan des algorithmes (outils) différentiels

	Reference	Time complexity		Memory	Supported operations	Ordered / unordered	Section	Notes
LaDiff	[CRGW96]	linear	O(ne+e^2)	linear	basic, move	ordered	4.1	for LATEX elements
MH-Diff	[CGM97]	quadratic	O(n^2 log n)	?	basic, move, copy	unordered	4.2	
XMLTreeDiff	[IBM98]	quadratic	O(n^2)	quadratic	basic	ordered	4.3	
MMDiff	[Cha99]	quadratic	O(n^2)	quadratic	basic	ordered	4.4	
XMDiff	[Cha99]	quadratic	O(n^2)	linear	basic	ordered	4.4	quadratic I/O cost
IBM's XML Diff and Merge Tool	[IBM01]	?	?	?	basic	?	4.5	commercial tool
3DM's matching algorithm	[TL01, TL03, TL04]	linear	O(n)	?	basic, move	ordered	4.6	
XyDiff	[CAM02]	linear	O(n log n)	linear	basic, move	ordered	4.7	
VM Tools	[VM02]	?	?	?	?	unordered	4.8	
DiffXML	[AM02]	linear	O(ne+e^2)	linear	basic, move	ordered	4.9	
KF-Diff+	[XWW02]	linear	O(n)	?	basic	both	4.11	
XML Diff and Patch	[XDP02]	?	?	?	?	both	4.12	commercial tool
X-Diff	[WDC03]	quadratic	O(n^2)	quadratic	basic	unordered	4.13	
DeltaXML	[MEL03, RLF03]	linear	?	linear	basic	both	4.14	
TreePatch	[KK03]	linear	O(ne+e^2)	linear	basic, move		4.15	
BioDIFF	[SB04]	quadratic	O(n^2)	quadratic	basic	unordered	4.17	for genomic and proteomic data

FIGURE 5.1 – Tableau extrait de "Change Detection in XML Trees : a Survey" [9]

Program Name	Author	Time	Memory	Moves	Minimal Edit Cost	Notes
fully tested						
DeltaXML	DeltaXML.com	linear	linear	no	no	
MMDiff	Chawathe and al.	quadratic	quadratic	no	yes	(tests with our implementation)
XMDiff	Chawathe and al.	quadratic	linear	no	yes	quadratic I/O cost (tests with our implementation)
GNU Diff	GNU Tools	linear	linear	no	-	no XML support (flat files)
XyDiff	INRIA	linear	linear	yes	no	
not included in experiments						
LaDiff	Chawathe and al.	linear	linear	yes	no	criteria based mapping
XMLTreeDiff	IBM	quadratic	quadratic	no	no	
DiffMK	Sun	quadratic	quadratic	no	no	no tree structure
XML Diff	Dommitt.com					we were not allowed to discuss it
Constrained Diff	K. Zhang	quadratic	quadratic	no	yes	-for unordered trees -constrained mapping
X-Diff	Y. Wang, D. DeWitt, Jin-Yi Cai (U. Wisconsin)	quadratic	quadratic	no	yes	-for unordered trees -constrained mapping

FIGURE 5.2 – Tableau extrait de "A comparative study for XML change detection" [8]

Annexe B : Illustration du principe Merging de 3DM

3DM merging utilise le résultat du tree-matching pour pouvoir merger les pairs de noeuds correspondants. Il n'utilise pas les opérations pour construire le document final. Ce dernier est initialisé vide et est inséré récursivement les éléments fusionnés des pairs de noeuds. Les opérations sont identifiées durant le processus, ayant pour vocation d'expliquer à l'utilisateur le processus.

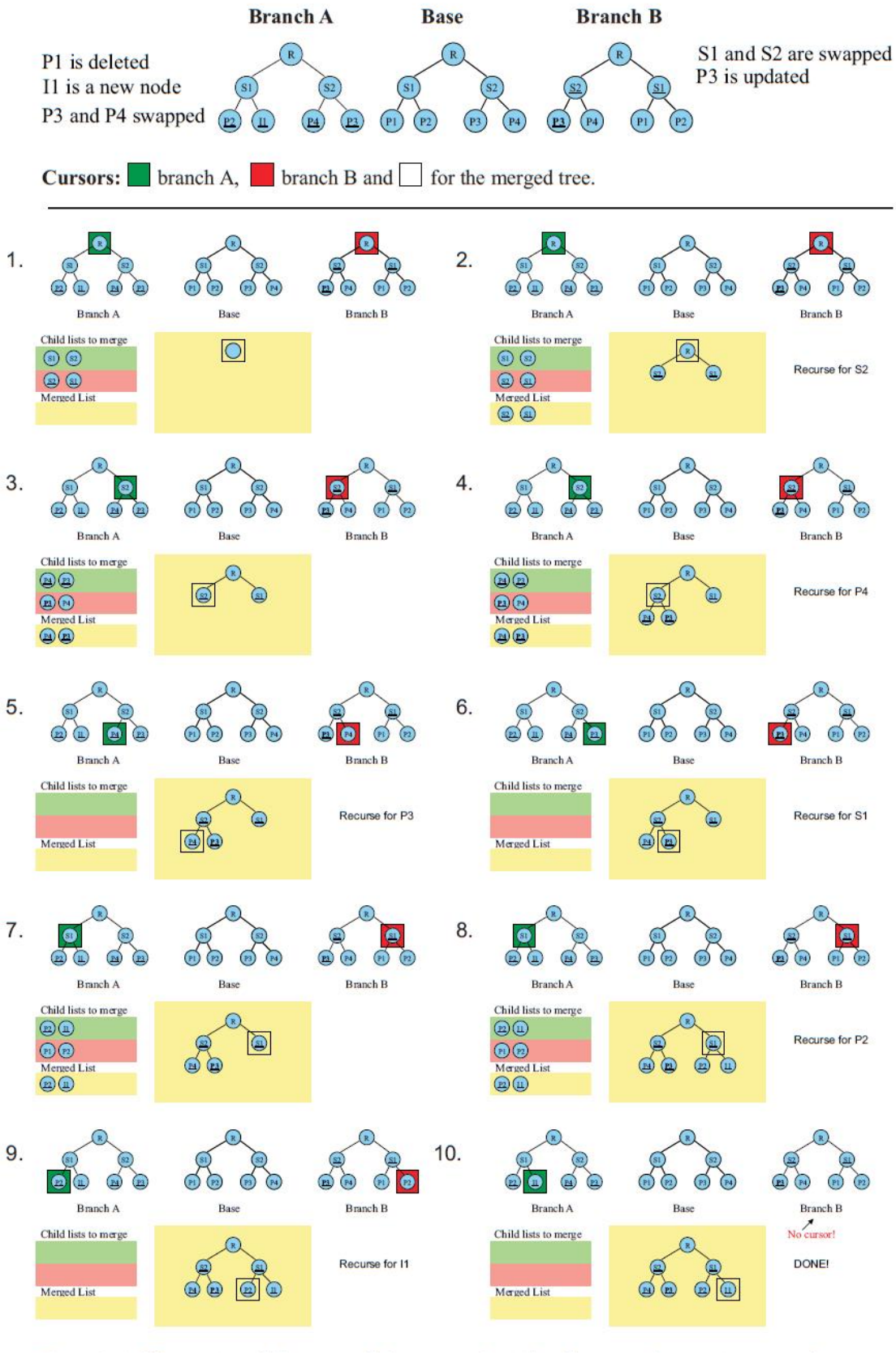


FIGURE 5.3 – extrait de la thèse de Tancred Lindholm

Bibliographie

[1] W3C, Extensible Markup Language (XML) 1.0 (Fifth Edition), http://www.w3.org/TR/2008/REC-xml-20081126/, 2008.

[2] SELKOW S. M., *LaTeX : The tree-to-tree editing problem*, Information Processing Letters,6, , 1977, p. 184-186.

[3] Tai K-C. *The tree-to-tree correction problem*, Journal of the ACM, 1979, p. 422-433

[4] 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

[5] Wu S., Mander U., Myers G., *An O(NP) Sequence Comparison Algorithm*, Univ. of Arizona, Tuscon, 15th September 1990, Information Processing Letters 35 (1990) 317-323

[6] Pekka K. *Tree Matching Problems with Applications to Structured Text Databases* Report A-1992-6, Helsinki, Finland, November 1992

[7] Zhang K., Shasha D. *Approximate tree pattern matching in Pattern matching in strings, trees and arrays*, A. Apostolico and Z. Galil (eds.). Oxford University Press, 1997, pp. 341-371

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

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

[10] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, Jennifer Widom *Change Detection in Hierarchically Structured Information* Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, p. 493-504. June, 1996

[11] Sudarshan S. Chawathe and Hector Garcia-Molina, *Meaningful change detection in structured data* Proceedings of the 1997 ACM SIGMOD international conference on Management of data, p. 26-37. May. 1997.

[12] XML TreeDiff by IBM alphaWorks, retired Nov.1998. Idea of David Epstein, designed and implemented by Francisco Curbera. http ://alphaworks.ibm.com/tech/xmltreediff/.

[13] Sudarshan S. Chawathe, *Comparing Hierarchical Data in External Memory* Proceedings of the 25th International Conference on Very Large Data Bases, Sept. 1999

[14] XML Diff and Merge Tool by IBM alphaWorks, last update Mar. 2001, http ://www.alphaworks.ibm.com/tech/xmldiffmerge.

[15] XML Diff and Patch, Microsoft Corporation, http ://apps.gotdotnet.com/xmltools/xmldiff/, 2002

[16] Adrian Mouat, Diffxml, June 2002, http ://diffxml.sourceforge.net

[17] Gregory Cobéna, Serge Abiteboul, Amélie Marian *Detecting Changes in XML Documents*, Proceedings of the 18th International Conference on Data Engineering, 41-52. Feb. 2002.

[18] Yuan Wang, David J. DeWitt, Jin-yi Cai. X-Diff : *An Effective Change Detection Algorithm for XML Documents*, Proceedings of the 19th International Conference on Data Engineering, 519-530. Mar. 2003.

[19] Robin La Fontaine, *A Delta Format for XML : Identifying Changes in XML Files and Representing the Changes in XML*, XML Europe 2001, Berlin, May 2001

[20] Robin La Fontaine, *Merging XML files : a new approach providing intelligent merge of XML data sets*, XML Europe 2002, Barcelona, May 2002

[21] Robin La Fontaine, *DeltaXML, Change Control for XML : Do It Right* XML Europe, May 2003.

[22] Tancred Lindholm *A 3-way Merging Algorithm for Synchronizing Ordered Trees-the 3DM merging and diferencing tool for XML*, Master ?s thesis, Helsinki University of Technology, Dept. of Computer Science, Sept. 2001.

[23] Tancred Lindholm *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.

[24] Tancred Lindholm *A three-way merge for XML documents*, Proceedings of the 2004 ACM symposium on Document engineering, 1-10, Oct. 2004.

[25] Sebastian Rönnau , Christian Pauli , Uwe M. Borghoff, *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

[26] Neil F. *Differential synchronization*, Proceedings of the 9th ACM symposium on Document engineering, September 16-18, 2009, Munich, Germany

[27] Fuhr N., Grôbjohann K. *XIRQL : A Query Language for Information Retrieval in XML Documents*, Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval, 2001, 12, 32. p. 172-180, New York City, USA

[28] Danny H. , Jarke J. W. *Visual Comparaison of Hierarchically Organized Data*, Eurographics/IEEE-VGTC Symposium on Visualisation 2008.

[29] Fernanda B. V. , Martin W. , Kushal D. *Studying Cooperation and Conflict between Authors with history flow Visualizations*, Proceedings of the SIGCHI conference on Human factors in computing systems, p.575-582, April 24-29, 2004, Vienna, Austria

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

[31] Angelo Di Iorio, Michele Schirinzi, Fabio Vitali, Carlo Marchetti *A Natural and Multi-layered Approach to Detect Changes in Tree-Based Textual Documents*, In Proceedings of ICEIS'2009. pp.90-101

[32] Crozat S. *Scenari - La chaîne éditoriale libre*, Accès libre. Eyrolles, 1st edition.

[33] Site web officiel du projet ANR C2M, http://scenari.utc.fr/c2m/.