Préservation des Intentions et Maintien de la Cohérence des Données Répliquées en Temps Réel

Luc André

▶ **To cite this version:**

Luc André. Préservation des Intentions et Maintien de la Cohérence des Données Répliquées en Temps Réel. Algorithme et structure de données [cs.DS]. Université de Lorraine, 2016. Français. NNT : 2016LORR0089 . tel-01754666v2

HAL Id: tel-01754666

https://theses.hal.science/tel-01754666v2

Submitted on 9 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Préservation des Intentions et Maintien de la Cohérence des Données Répliquées en Temps Réel

THÈSE

présentée et soutenue publiquement le 13 mai 2016

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Luc ANDRE

Composition du jury

Rapporteurs :	Sophie CHABRIDON	Maître de Conférences, Télécom SudParis
	Guillaume PIERRE	Professeur, Université de Rennes
Examinateurs :	Pierre-Etienne MOREAU	Professeur, Université de Lorraine
	Nuno PREGUICA	Assistant professeur, Universidade Nova, Lisbonne
Encadrants :	François CHAROY	Professeur, Université de Lorraine
	Gérald OSTER	Maître de Conférences, Université de Lorraine

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

Je tiens tout d'abord à remercier les membres du jury, pour avoir bien voulu consacrer du temps à la lecture de mes travaux, ainsi que mes encadrants de thèse François CHAROY et Gérald OSTER.

Merci également à Caludia IGNAT, avec qui j'ai souvent eu l'occasion d'échanger et de travailler.

Viennent ensuite les membres de ma famille, pour m'avoir permis de réaliser cette thèse dans les meilleures conditions possibles, tout particulièrement ma grand mère Monique MINELLA, qui n'aura malheureusement pas pu voir cette thèse aboutir.

Mes remerciements vont aussi à tous ceux que j'ai cotoyés au LORIA, de près ou de loin : Stéphane, Weihai, Pascal, Medhi, Ahmed, Raman, Valérie, Matthieu, Meagan, Phillippe, Victorien, Jordi, ,Vinh, Clément, Loïck ; mes plus plates excuses à ceux que j'oublie de citer ici.

Enfin, pour les distractions qu'ils ont pu m'offrir quand il fallait penser à autre chose, merci à mes partenaires d'escrime, d'échecs japonais et de danse.

Sommaire

Chapitre 1 Introduction

1.1 Contexte 1
1.2 Résumé des contributions 2
1.3 Organisation du document 3

Chapitre 2 Problématique **5**

2.1 Réplication, copies et convergence 5
2.2 Respect de la causalité 7
2.3 Préservation des intentions des opérations 8
2.4 Intentions des utilisateurs 9
2.5 Objectif 13

Chapitre 3 État de l'art **15**

3.1 Approche Transformées Opérationnelles 15
3.1.1 Principes de base 15
3.1.2 Intentions des utilisateurs 26
3.1.3 Résumé de l'approche OT 32
3.2 Approche Commutative Replicated Data Types 33
3.2.1 Principes de base 33
3.2.2 Algorithmes 33
3.2.3 Synthèse de l'approche CRDT 40
3.3 Synthèse de l'état de l'art 41

Chapitre 4 Une approche OT pour les documents hiérarchisés **43**

4.1 Modèle et Opérations 44
4.1.1 Modèle de données 44
4.1.2 Opérations 44
4.2 Transformées 46
4.2.1 Transformées par rapport à InsertText 49

4.2.2 Transformées par rapport à DeleteText 51
4.2.3 Transformées par rapport à NewParagraph 53
4.2.4 Transformées par rapport à MoveParagraph 55
4.2.5 Transformées par rapport à MergeParagraph 57
4.2.6 Transformées par rapport à SplitParagraph 58
4.2.7 Transformées par rapport à Style 60
4.3 Implémentation 62
4.4 Discussion 66

Chapitre 5 LogootSplit : CRDT pour blocs à taille variable 69
5.1 Principe et Modèle 69
5.1.1 Modèle de données 69
5.1.2 Identifiants 70
5.1.3 Génération des identifiants et insertion 70
5.1.4 Suppression 75
5.1.5 Scénario explicatif 77
5.1.6 Respect des critères CCI 78
5.2 Implémentation et comparaison 79
5.2.1 Implémentation naïve 80
5.2.2 Implémentation en chaîne 81
5.2.3 Implémentation en arbre 82
5.2.4 Expériences et résultats 85
5.3 Prototype MUTE 89
5.3.1 Architecture générale 89
5.3.2 Processus d'édition et d'intégration 92
5.3.3 Travail en mode déconnecté 93
5.3.4 Maturité du prototype 93
5.4 Bilan de la contribution 94

Chapitre 6 Conclusion 95
6.1 Contributions 95
6.1.1 Transformées pour document structuré 95
6.1.2 CRDT à granularité variable 96
6.2 Perspectives 96

Annexe

Annexe A
Fonctions de transformations complètes de notre approche OT

A.1 Transformées par rapport à InsertText 99
A.2 Transformées par rapport à DeleteText 100
A.3 Transformées par rapport à NewParagraph 102
A.4 Transformées par rapport à MoveParagraph 103
A.5 Transformées par rapport à MergeParagraph 107
A.6 Transformées par rapport à SplitParagraph 110
A.7 Transformées par rapport à Style 113

Bibliographie **121**

Table des figures

2.1 Divergence lors de la résolution d'un conflit 7
2.2 Illustration de la causalité 8
2.3 Illustrations d'intentions différentes pour la même opération 10
2.4 Intentions des utilisateurs non respectées : duplicatas 11
2.5 Intentions des utilisateurs non respectées : entrelacs 13

3.1 Intégration des opérations sans transformation : intentions non respectées 16
3.2 Intégration et transformation des opérations 16
3.3 Divergence avec respect des intentions 17
3.4 Propriété TP2 avec trois opérations concurrentes 18
3.5 Concurrence partielle mal résolue 20
3.6 Exemple de transformées 21
3.7 Concurrence partielle correctement traitée 22
3.8 Concurrence sur trois sites 23
3.9 Exemple de document TreeOpt 27
3.10 Principe de l'approche Transparent Adaptation 30
3.11 Scénario Transparent Adaptation avec recomposition de la commande 32
3.12 Exemple de document WOOT 34
3.13 Insertions concurrentes simples avec WOOT 35
3.14 Insertions concurrentes complexes avec WOOT 36
3.15 Scénario d'intégration RGA 37
3.16 Exemple de document LOGOOT 38
3.17 Exemple de document TREEDOC 40

4.1 Exemple de modèle de document 44
4.2 Illustration de l'opération Style 47
4.3 Insertion et césure en début de feuille 50
4.4 Insertion et style en bordure de feuille 51
4.5 Suppression et césure en début de feuille 52
4.6 Suppression et style en bordure de feuille 53
4.7 Décalages d'indices de chemin après une fusion 57

4.8 Intégration de deux opérations *Style* concurrentes 62
4.9 Exemple de document avec la syntaxe wikitext 64
4.10 Exemple de page générée à partir d'un document wikitext 64
4.11 Exemple de structure en arbre 64
4.12 Architecture de l'éditeur temps réel de wiki 65

5.1 Fonction *scinder* 71
5.2 Fonction *generateBase* 72
5.3 Fonction *LocalInsert* 73
5.4 Procédure *RemoteInsert* 73
5.5 Génération et intégration de deux insertions 74
5.6 Fonction *LocalDelete* 75
5.7 Procédure *RemoteDelete* 76
5.8 Génération et intégration de deux suppressions 77
5.9 Exemple d'édition 78
5.10 Structure d'implémentation en chaîne 81
5.11 Scénario complexe avec la structure en chaîne 82
5.12 Structure d'implémentation en arbre 83
5.13 Scénario complexe avec la structure en arbre 84
5.14 Résultats avec une trace aléatoire 86
5.15 Résultats mémoire avec une trace aléatoire 86
5.16 Résultats avec des opérations en fin de document 88
5.17 Résultats mémoire avec des opérations en fin de document 88
5.18 Architecture globale de l'application MUTE 90
5.19 Exemple de liens websockets clients-serveur 91

1

Introduction

1.1 Contexte

L'édition collaborative [Ellis *et al.*, 1991] en temps réel permet à plusieurs utilisateurs d'éditer ensemble un même document, avec un appui informatique. Chaque utilisateur visualise le document depuis son poste de travail et peut le modifier à loisir, comme avec un éditeur de document classique, la différence étant que le document est partagé. Lorsqu'un changement est réalisé, le document est modifié et chaque utilisateur visualise le nouveau document. La notion de temps réel ici sert à distinguer ce type d'édition d'applications où chaque utilisateur travaille en isolation, c'est-à-dire coupé des autres utilisateurs, comme par exemple les gestionnaires de versions. Dans un gestionnaire de versions, un utilisateur récupère une version du document, la modifie, puis décide d'enregistrer le document modifié comme étant la nouvelle version de référence. Les modifications ne sont donc pas disponibles pour les autres utilisateurs dès leur création, mais plus tard, lorsque l'on décide de les publier. Les éditeurs collaboratifs en temps réel fonctionnent de manière différente et tentent de rendre disponibles les modifications dès qu'elles ont été réalisées. Des exemples d'éditeurs collaboratifs en temps réel sont GoogleDocs [1] et Etherpad [2]. Un exemple de situation où une application d'édition collaborative en temps réel peut être utilisée est en réunion, pour une prise de notes mutualisée.

Quand plusieurs personnes éditent un document, elles peuvent provoquer des conflits d'édition. Un conflit apparaît lorsque deux utilisateurs ou plus décident de modifier en même temps la même partie du document, la délimitation des parties du document est propre à chaque approche et une partie peut être le document complet comme un simple caractère. Résoudre le conflit signifie décider quel sera l'état de la partie modifiée en parallèle. Dans le cas d'un gestionnaire de versions, cette décision peut être laissée aux utilisateurs. Lorsqu'un utilisateur travaille sur une version qui n'est plus à jour car un autre utilisateur a publié une nouvelle version, un conflit va se déclencher quand l'utilisateur va tenter de publier sa version. Sa version va être comparée avec la version de l'autre utilisateur ; les conflits vont être mis en évidence, et l'utilisateur pourra

1. https ://www.google.com/drive/
2. http ://etherpad.org/

réécrire les parties conflictuelles. Dans le cas de l'édition collaborative en temps réel où toutes les modifications sont transmises le plus rapidement possible, il est inenvisageable d'interrompre les utilisateurs dans leur édition à chaque conflit, et la résolution des conflits doit donc obligatoirement être automatique. Les éditeurs collaboratifs en temps réel se doivent d'avoir une résolution efficace des conflits.

La résolution automatique des conflits d'édition est une caractéristique très intéressante pour les utilisateurs, qui ne sont plus tenus de les résoudre eux mêmes. Les champs d'applications de l'édition collaborative en temps réel s'agrandissent donc de plus en plus. Par exemple, l'encyclopédie en ligne Wikipédia, qui permet l'édition uniquement en isolation de ses pages, utilise le moteur MediaWiki, lequel travaille sur la possibilité de les éditer en temps réel [3]. Les éditeurs temps réel peuvent aussi être utilisés en isolation, pour travailler temporairement sans être dérangés par les modifications des autres utilisateurs, tout en profitant de cette résolution automatique.

Ces nouvelles situations sont autant de défis que les nouveaux algorithmes d'édition collaborative en temps réel doivent surmonter. En particulier, l'édition collaborative ne se limite plus aux simples documents textes, mais s'applique aussi à des modèles de données plus évolués, qui ne sont pas modifiés de la même manière par les utilisateurs et qui demandent des méthodes de résolution des conflits améliorées. Dans notre mémoire, nous nous focalisons sur la résolution des conflits, en montrant les limites des algorithmes existants et leurs conséquences sur la qualité de l'édition. Nous proposons deux nouveaux algorithmes qui permettent une résolution des conflits plus proche du résultat attendu par les utilisateurs en cas d'édition complexe d'un document.

1.2 Résumé des contributions

Deux contributions sont présentées dans ce mémoire. Notre première proposition se base sur l'approche des transformées opérationnelles, abréviée OT. Cette approche fonctionne avec des ensembles d'opérations, que l'on essaie généralement de garder petits pour plus de simplicité. Notre approche propose un ensemble d'opérations plus étendu que la moyenne, ce qui a pour conséquence d'augmenter la précision de la résolution des conflits, de réduire le nombre d'opérations nécessaires à l'édition d'un document, et de permettre l'édition de documents structurés.

Notre deuxième contribution est une structure de données qui peut être éditée en parallèle sans générer de conflits. Les structures de ce type sont connues sous le nom de Commutative Replicated Data Type (CRDT), et les CRDT pour l'édition collaborative en temps réel sont généralement modifiées à partir d'opérations d'insertion ou de suppression d'éléments. Notre CRDT ajoute une opération de scission d'un élément, ce qui permet de résoudre plus précisément certains conflits.

3. https ://www.mediawiki.org/wiki/Future/Real-time_collaboration

1.3 Organisation du document

Le chapitre 2 du mémoire présente les bases et le vocabulaire de l'édition collaborative en temps réel, ainsi que la notion de réplication optimiste qui lui est associée. Une attention particulière sera portée sur ce que l'on appelle l'Intention, qui est un point important de la résolution des conflits.

Le chapitre 3 présente les familles d'algorithmes de réplication optimiste OT et CRDT ainsi que leurs représentants les plus pertinents, et examine comment ils gèrent les éditions et résolvent les conflits.

Les chapitres 4 et 5 présentent chacun une des deux contributions du mémoire.

Enfin, le dernier chapitre dresse le bilan du mémoire, et s'ouvre sur les différentes pistes futures à explorer.

2

Problématique

2.1 Réplication, copies et convergence

La notion de temps réel associée à l'édition collaborative signifie que l'on souhaite une édition réactive, où chaque utilisateur peut modifier le document à n'importe quel moment, et où les modifications sont intégrées le plus rapidement possible. Un scénario où les utilisateurs prennent la main tour à tour sur le document avec un système de verrou exclusif n'est pas envisageable. Dans une telle situation, lorsqu'un utilisateur veut éditer le document, il bloque les autres utilisateurs, modifie et enregistre le document, et libère le verrou pour que d'autres puissent modifier le document. Cela induit une latence à chaque édition le temps de sauvegarder le document, et bloque les éditions pendant ce temps. L'édition est ralentie, et notre critère de temps réel n'est pas respecté.

Pour le respecter, une solution est de répliquer le document : chaque utilisateur dispose d'une copie du document, qu'il peut modifier à volonté. On appellera un site un endroit où est stockée une copie. Cette solution peut entraîner une divergence des copies : si deux utilisateurs modifient en même temps leurs copies, chacun va avoir un document différent. Il faut donc réconcilier les copies, en les faisant converger vers la même valeur. Les algorithmes qui répliquent leur données sont appelés des algorithmes de réplication, et on peut les classer en deux familles.

Une première famille d'algorithmes de réplication est la réplication pessimiste. Ces algorithmes répliquent les données pour qu'elles soient accessibles depuis plusieurs sites, et stockées à plusieurs endroits à la fois. Le but n'est pas de permettre à plusieurs utilisateurs de mettre à jour des données en même temps, mais d'avoir un système robuste et disponible. Les copies ne divergent pas, et sont mises à jour uniformément dès que l'une d'entre elles est modifiée [Bernstein and Goodman, 1981]. Par exemple, une base de données peut répliquer ses données sur plusieurs sites et élire un site maître qui est en charge de toutes les transactions sur la base. Quand le site maître est mis à jour, il propage son état aux autres sites. Si le maître tombe en panne, un nouveau maître est élu parmi les sites restants et la base de données peut continuer à fonctionner le temps de réactiver le site défectueux.

L'approche pessimiste n'est pas ce que nous voulons pour l'édition collaborative en temps

réel, à la différence de la réplication optimiste [Saito and Shapiro, 2005]. Les algorithmes optimistes autorisent la divergence des répliques, et assurent que la convergence sera effective. Ils permettent la modification des copies à n'importe quel moment et sans contraintes, et ont été conçus pour permettre de travailler en parallèle sur les mêmes données. Les modifications peuvent être représentées de deux manières différentes. La première est sous forme d'opérations. Chaque fois qu'une copie est modifiée, une ou plusieurs opérations sont créées, et sont immédiatement ou ultérieurement envoyées aux sites distants, et sont utilisées pour faire converger les copies. Par exemple, ajouter une chaîne de caractères au document génère une opération d'insertion de caractères à une position donnée, et en retirer une génère une opération de suppression. La deuxième est sous forme d'état. Modifier le document ne génère pas d'opérations, et c'est donc la copie complète qui est envoyée aux autres sites. Ils pourront la comparer à leurs propres copies pour déterminer comment faire converger les copies. L'approche basée sur les états consiste généralement à déduire depuis les multiples états quelles ont été les opérations exécutées depuis le dernier envoi, puis à utiliser un mécanisme d'intégration basé sur des opérations. De plus, cette approche n'est pas conçue pour envoyer immédiatement les modifications, mais plutôt pour attendre d'en avoir un nombre conséquent avant d'envoyer l'état qui les contient toutes : envoyer le document complet à chaque ajout de caractère nécessiterait trop de ressources réseau. L'approche basée sur les états n'est donc pas compatible avec l'édition collaborative en temps-réel et l'envoi rapide des modifications. GoogleDocs fonctionnait initialement avec des états, et utilise maintenant des opérations. Pour ces raisons, nous ne traiterons dans la suite que des approches de réplication optimiste basées sur les opérations.

Dans une approche de réplication optimiste basée sur les opérations, les opérations sont d'abord générées localement sur une copie, puis elles sont envoyées aux copies distantes, qui les intègrent ensuite. Un défi ici est que les modifications n'arrivent pas dans le même ordre sur toutes les copies. Si on considère deux sites s_1 et s_2 qui insèrent en concurrence chacun une phrase différente au début du document (p_1 et p_2, respectivement), le site s_1 va recevoir la modification correspondant à l'ajout de p_2 alors qu'il contient la phrase p_1, et le site s_2 va recevoir la modification correspondant à l'ajout de p_1 alors qu'il contient p_2. Chaque site va devoir résoudre un conflit lors de l'intégration de la modification distante, et bien que ce conflit semble avoir la même forme sur chaque site (on cherche à insérer une phrase au même endroit qu'une insertion concurrente), une résolution symétrique conduit à une divergence des copies. En effet, si par exemple les deux sites décident de placer l'insertion distante avant l'insertion locale, le site s_1 contient p_2 suivi de p_1 alors que le site s_2 contient p_1 suivi de p_2. La figure 2.1 illustre cette situation. Elle se complexifie bien évidemment avec plus de deux sites.

Comme les éditions ont lieu en même temps que les intégrations, les copies ne peuvent converger que lorsque les copies ne sont plus modifiées, et que suffisamment de temps a été laissé à l'algorithme pour intégrer les modifications sur toutes les copies. Ce type de convergence est appelé la convergence à terme.

La convergence des copies est nécessaire, mais n'est pas suffisante dans le cas de l'édition

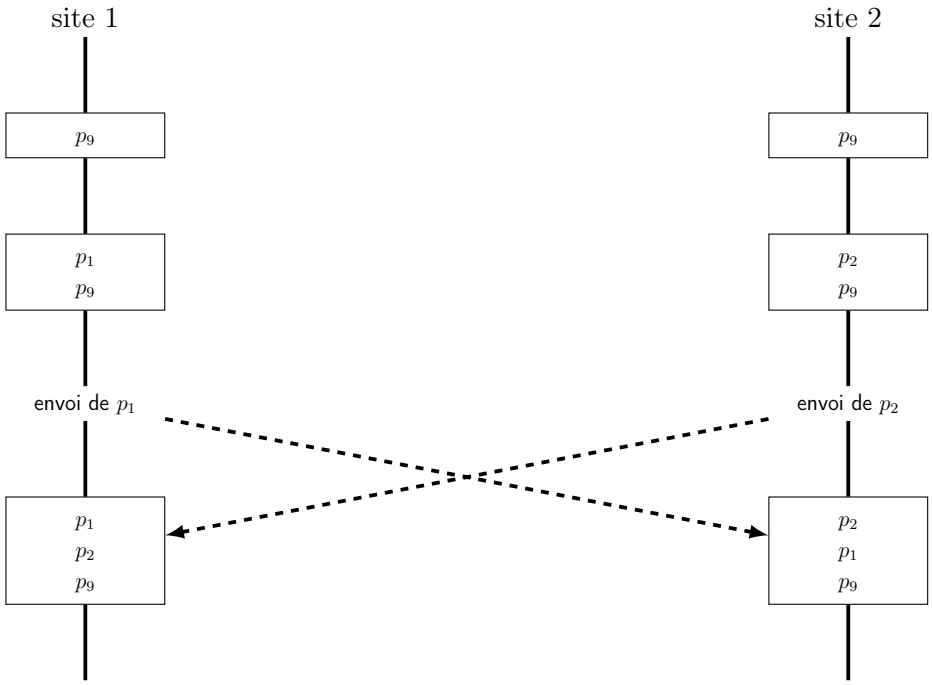


FIGURE 2.1 – Divergence lors de la résolution d'un conflit

collaborative en temps réel. [Sun *et al.*, 1998] présente deux autres prérequis que l'algorithme de réplication doit respecter. A la convergence s'ajoutent le respect de la causalité et la préservation des intentions des opérations.

2.2 Respect de la causalité

Le critère de respect de la *Causalité* des modifications signifie que si deux modifications sont liées par une relation de causalité, c'est à dire si une modification n'a de sens qu'associée à une autre modification, elles doivent être intégrées sur tous les sites en respectant cette relation. Cela permet de garder une certaine cohérence dans le document : si un utilisateur modifie une phrase récemment reçue, il serait étrange de voir apparaître sur la copie d'un site la modification en premier, puis la phrase qui est modifiée ensuite. La relation de précédence causale est difficile à détecter, mais elle est incluse dans la relation happens-before définie par Lamport [Lamport, 1978].

Définition : happens-before Une modification m_1 happens-before une modification m_2 si m_1 a été générée avant m_2 sur le même site, ou si m_2 a été générée sur un site après la réception de m_1 par ce site, ou si il existe une modification m_3 telle que m_1 happens-before m_3 et m_3 happens-before m_2. Cette relation est une relation d'ordre partiel sur les modifications, deux opérations qui ne sont pas en relation sont dites concurrentes.

En effet, si une modification m précède causalement une modification n, alors m happens-before n. La réciproque est fausse, deux modifications qui surviennent l'une après l'autre ne sont pas forcément liées, par exemple dans le cas de la modification de deux paragraphes distincts. Assurer une intégration des modifications en respectant la relation happens-before est un moyen fort d'assurer leur intégration en respectant l'ordre causal. Nous retiendrons ce moyen, et ne ferons plus de distinctions entre ces deux relations dans la suite de ce document. Lorsque les notions d'ordre causal ou de causalité seront évoquées, elle feront référence à la relation happens-before.

Dans la figure 2.2, la modification m_2 précède causalement la modification m_3 puisque m_2 a été générée avant m_3 sur le site 2. De même, m_1 précède causalement m_4. La modification m_1 précède causalement les modifications m_2 et m_3, puisque m_2 et m_3 ont été générées après la réception de m_1 sur le site 2. Enfin, la modification m_4 est concurrente avec les modifications m_2 et m_3.

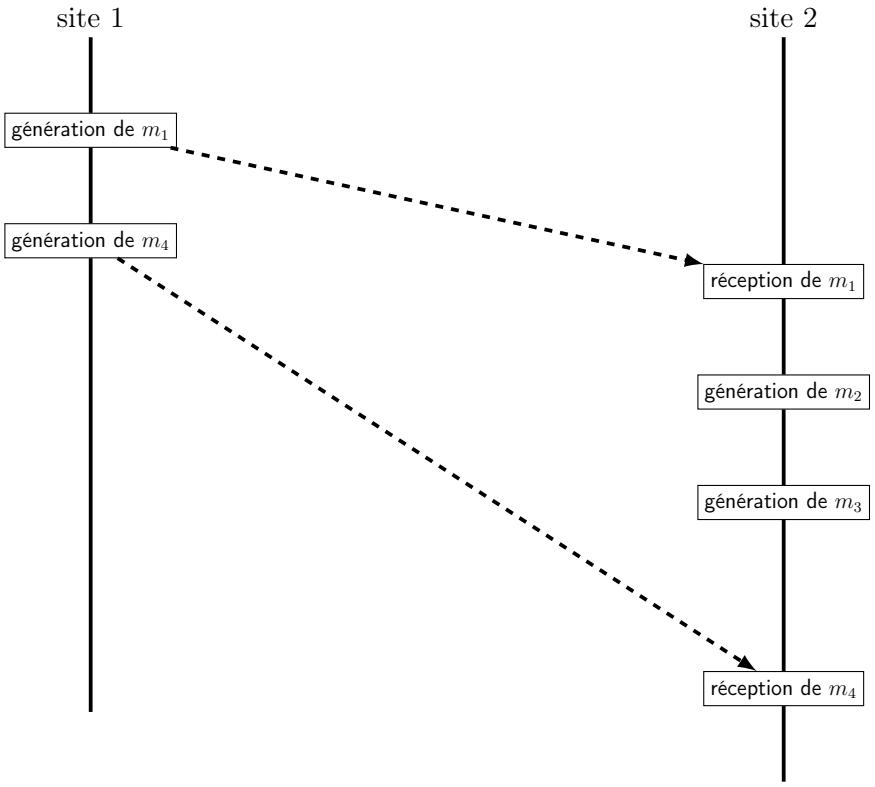


FIGURE 2.2 – Illustration de la causalité

2.3 Préservation des intentions des opérations

Lors de la résolution d'un conflit, la convergence n'est pas suffisante, encore faut-il converger vers quelque chose de pertinent au regard des modifications qui ont été intégrées. Dans l'exemple

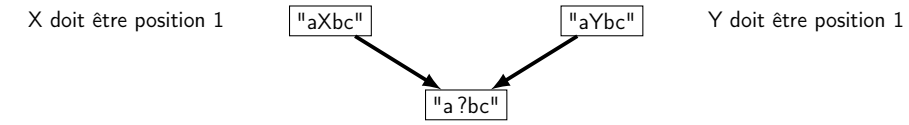
précédent utilisé pour illustrer la définition de la convergence, on peut trouver une manière très simple de résoudre le conflit : on supprime la phrase qui a déjà été insérée et on ignore celle que l'on vient de recevoir. Ainsi, aucun des deux sites ne contient ni p_1, ni p_2, et les copies convergent.

Cette solution est évidemment inacceptable, elle nuit au processus d'édition en supprimant des modifications sans aucune raison valable. La résolution du conflit doit essayer de conserver au mieux l'effet des éditions conflictuelles. Pour cela, on associe à chaque opération une postcondition que l'on nomme *Intention de l'opération*, et on résout les conflits de manière à respecter les intentions des opérations. Ces intentions ne doivent être ni trop strictes, ni trop permissives. Par exemple, considérons l'opération d'ajout d'un caractère c à une position p dans le document. Si on définit l'intention de cette opération comme étant "le caractère c doit être à la position p après l'insertion", on ne peut pas résoudre un conflit entre deux opérations qui insèrent un caractère différent à la même position p, l'un occupera forcément une place différente de p. Ceci est illustré figure 2.3a. Si on définit plus librement l'intention comme étant "le caractère c doit se trouver dans le document", on laisse la porte ouverte à des résolutions de conflits trop approximatives où le caractère c risque de se retrouver bien loin de la position d'insertion initiale. La figure 2.3b illustre ce cas. Une intention possible pour l'opération d'insertion est "Le caractère c doit se trouver entre ses deux caractères voisins initiaux". Si par exemple dans le document "abc", deux utilisateurs insèrent en concurrence les caractères "X" et "Y" à la position 1 entre "a" et "b", les résolutions "aXYbc" et "aYXbc" conviennent toutes les deux. Les caractères "X" et "Y" sont bien entre les caractères "a" et "b", comme le souhaitaient chacun des utilisateurs. Ces deux solutions, dont l'une est représentée figure 2.3c sont les seules possibles avec cette intention, et sont celles qui intègrent au mieux les deux opérations des utilisateurs. On peut noter que la convergence n'est pas garantie par le respect des intentions, notre dernier exemple présentant deux moyens différents de résoudre un conflit entre deux insertions en préservant leurs intentions.

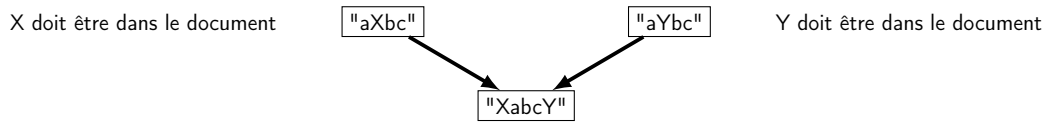
2.4 Intentions des utilisateurs

Les algorithmes d'édition collaborative en temps réel doivent respecter les intentions des opérations pour être corrects. Seulement, il arrive que les intentions des utilisateurs diffèrent des intentions des opérations, et donc que le document ne convienne pas aux utilisateurs, car les intentions des utilisateurs ne sont pas respectées.

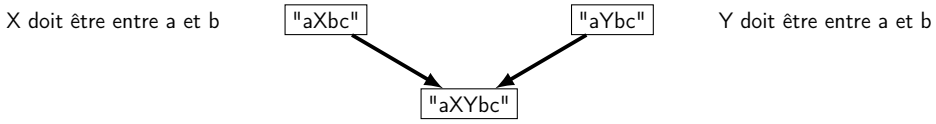
D'une part, certaines intentions des utilisateurs ne sont pas compatibles et on ne peut pas satisfaire deux intentions concurrentes. Par exemple, si un utilisateur souhaite modifier une phrase tandis qu'un autre veut la supprimer, il faut faire un choix. Soit on garde la phrase modifiée, soit on la supprime totalement. Ne faire apparaître que la modification sans la phrase à laquelle elle se rattache semble maladroit. Au final, le document ne contient qu'une seule modification et l'autre est perdue. Dans ce cas, le document convient à un utilisateur et ne convient pas à l'autre, mais il n'y a pas de solution pour qu'il convienne aux deux. Faire apparaître les deux versions d'une manière ou d'une autre et proposer aux utilisateurs un moyen de faire



(a) Intentions trop strictes



(b) Intentions trop permissives



(c) Intentions bien définies

FIGURE 2.3 – Illustrations d'intentions différentes pour la même opération

ultérieurement un choix entre les deux n'est pas envisageable dans notre contexte de temps réel. Ce type de conflit n'est pas lié à l'édition en parallèle du document, mais au fait qu'il est édité à plusieurs. Le fait de ne pas préserver une intention n'est pas une erreur, les deux intentions ne peuvent pas cohabiter. Dans le cadre d'une édition séquentielle, un utilisateur peut modifier la phrase, et un autre la supprimer une fois la modification reçue. Dans ce cas, l'intention du premier utilisateur est aussi perdue. La résolution proposée en cas d'édition en parallèle donne donc l'illusion que le document a été édité de façon séquentielle, et propose en ce sens un contenu correct.

D'autre part, les intentions de l'utilisateur peuvent être trop complexes pour être exprimées correctement avec les opérations et les intentions qui leur sont associées. Dans ce cas, le conflit est mal résolu et le document se retrouve dans un état non satisfaisant, car il ne convient à aucun utilisateur. Par exemple, on peut considérer un algorithme d'édition collaborative qui fonctionne avec des lignes, et dont les opérations sont l'ajout ou la suppression d'une ligne. L'opération de modification d'une ligne n'est pas disponible en tant que telle. Modifier une ligne passe par la suppression de la ligne à modifier, et de l'insertion d'une nouvelle ligne qui contient la modification. En conséquence, du point de vue de l'algorithme, l'intention de modification de l'utilisateur est une intention de suppression d'une ligne suivie d'une intention d'insertion d'une autre ligne quelconque. Cela pose problème lors de la modification en concurrence d'une même

ligne par deux utilisateurs. L'algorithme va chercher à résoudre le conflit tout en respectant les intentions des opérations d'insertion et de suppression, et va donc supprimer la ligne initiale et insérer deux nouvelles lignes, une qui contient la ligne modifiée du premier utilisateur, l'autre qui contient la ligne modifiée du second utilisateur. Une résolution de ce conflit d'édition qui préserve les intentions des utilisateurs devrait conduire à un document contenant une seule ligne modifiée deux fois, ce qui n'est pas le cas ici, et donc le document n'est pas dans un état satisfaisant. Cette situation est illustrée figure 2.4. L'intention du premier utilisateur est de remplacer la ligne "world" par "everyone", ce qui est traduit en la suppression de la ligne 2 et l'insertion d'une nouvelle ligne 2.1. De même, le second utilisateur a l'intention de remplacer "Hello" par "Hi", ce qui est traduit en la suppression de la ligne 2 et l'insertion d'une nouvelle ligne 2.2. Le conflit est résolu en préservant les opérations des opérations. Par rapport à l'état initial, l'état final est donc amputé de la ligne 2 (deux fois) et les lignes 2.1 et 2.2 sont ajoutées. Le document final contient deux lignes au lieu d'une ligne "Hi everyone".

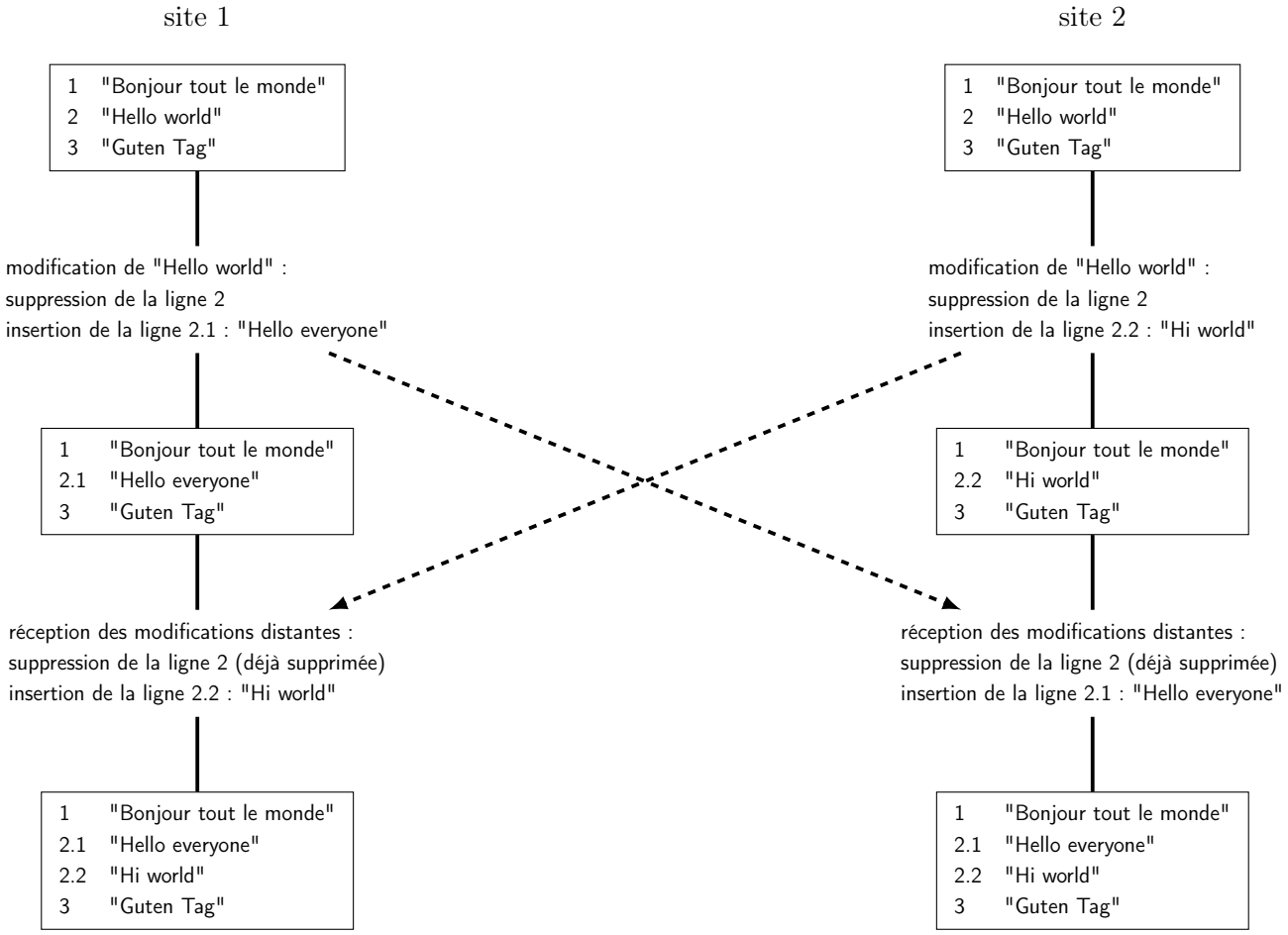


FIGURE 2.4 – Intentions des utilisateurs non respectées : duplicatas

Pour que le document soit dans l'état souhaité, un utilisateur doit éditer à nouveau le docu-

ment et corriger l'erreur produite lors de la résolution du conflit. Cette édition supplémentaire a un coût et donc participe indirectement à la dégradation des performances de l'algorithme, en plus de contraindre l'utilisateur à arrêter son édition pour corriger le problème. L'utilisateur doit être capable de détecter que le conflit a été mal résolu et de déterminer la forme correcte de sa résolution, s'il se rend compte de la forme incorrecte du document. Dans notre exemple, ce sont des tâches a priori aisées, mais d'autres situations peuvent conduire à des conflits plus difficile à détecter. De plus, les algorithmes d'édition collaborative en temps réel peuvent être utilisés dans des contextes non temps réel, notamment en mode déconnecté, où les opérations ne sont pas envoyées immédiatement mais lorsque l'utilisateur se connecte après avoir modifié le document. Dans ce cas, les conflits mal résolus sont plus difficiles à détecter par les utilisateurs, car les opérations qui provoquent le conflit n'apparaissent pas au même moment sur leurs copies. Un autre problème est que la résolution du conflit peut provoquer des autres conflits. Dans notre exemple, si deux ou plusieurs utilisateurs parviennent à détecter l'erreur et souhaitent la corriger au même moment, ils vont supprimer les deux lignes et insérer chacun une nouvelle ligne qui contient les deux modifications. Au final le document contient la ligne dupliquée. S'ils s'en aperçoivent et décident à nouveau de corriger le document, l'un peut choisir de retirer la première ligne, et l'autre la deuxième, et donc la ligne modifiée disparaît entièrement.

Fonctionner avec une granularité plus fine que des lignes, par exemple des caractères, permet de corriger le problème de notre exemple. La ligne est modifiée une fois par chaque utilisateur, ils ajoutent ou suppriment deux caractères différents, et la ligne contient les deux modifications une fois le conflit résolu. Seulement, une granularité plus fine implique plus d'opérations pour modifier le document, ce qui peut dégrader les performances. Un autre problème est l'entrelacs d'éditions : insérer une phrase ou un bloc de texte revient en réalité à insérer une série de caractères qui se suivent. Par exemple, dans le document "Le chat.", insérer "noir et blanc" entre "t" et "." revient à insérer "n" entre "t" et "." puis "o" entre "n" et "." puis "i" entre "o" et "." et ainsi de suite jusqu'à "c" entre "n" et ".". Si un utilisateur insère en concurrence "de mon voisin" entre "t" et ".", un des documents "Le chat noir et blanc de mon voisin." ou "Le chat de mon voisin noir et blanc." est attendu après résolution du conflit. Pourtant, le document "Le chat n doei r moetn vo bilsaincn." par exemple est aussi possible puisqu'il respecte les intentions de toutes les insertions, mais l'entrelacs des caractères des différentes phrases insérées rend le texte illisible. La figure 2.5 présente ce scénario.

De plus, d'autres problèmes d'intentions mal exprimées restent non résolus, par exemple dans le cas d'un déplacement. Un déplacement va être interprété comme la suppression des caractères déplacés suivi de leur insertion ailleurs dans le document. Si un bloc de texte est déplacé et qu'un ajout de caractère le modifie en même temps, la modification va rester sur place et le reste sera déplacé, ce qui va satisfaire les intentions des opérations de suppression et d'insertion de caractères déduites à partir du déplacement et l'intention de l'opération d'ajout de caractère de la modification, mais ne va pas déplacer la modification avec le bloc. GoogleDocs a quant à lui une autre manière de résoudre ce conflit : le bloc est déplacé et la modification est perdue. En effet, lors de la situation de suppression et de modification en concurrence expliquée en

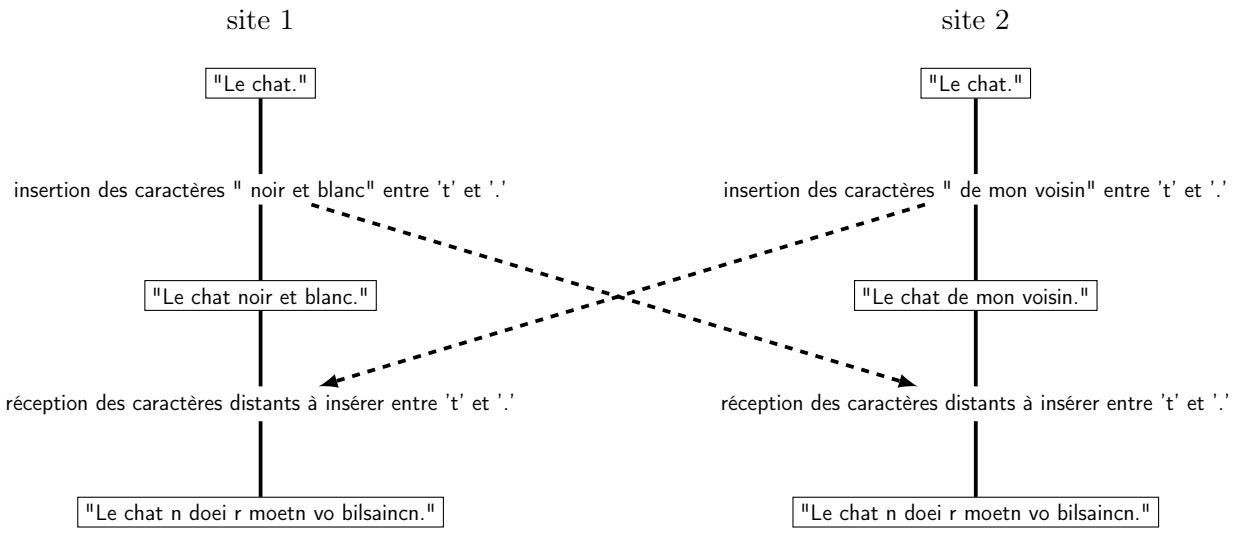


FIGURE 2.5 – Intentions des utilisateurs non respectées : entrelacs

début de section, GoogleDocs choisi de supprimer la partie modifiée. En conséquence, comme le déplacement est interprété comme une suppression puis une insertion, la première étape de la résolution du conflit avec une insertion dans le bloc déplacé est de résoudre le conflit entre la suppression et la modification, ce qui conduit à la suppression totale du bloc ; la seconde étape est l'insertion du bloc déplacé, qui est donc inséré sans la modification, comme si les caractères étaient nouveaux.

Les problèmes liés à l'intention des utilisateurs sont nombreux et laisser aux utilisateurs le soin de les corriger est inconfortable, difficile et coûteux. Progresser dans le domaine du respect des intentions des utilisateurs permettrait d'améliorer la qualité d'édition des algorithmes d'édition collaborative en temps réel.

2.5 Objectif

Notre objectif est de proposer des algorithmes d'édition collaborative en temps réel qui respectent davantage les intentions des utilisateurs. Cela passe par une gestion flexible de la granularité des modifications et par une augmentation des intentions des utilisateurs qu'il est possible de faire comprendre à l'algorithme. Cela améliore la résolution des conflits d'édition et conduit à des documents plus lisibles tout en réduisant le nombre d'éditions nécessaires à la création d'un document. Tout cela participe à l'amélioration des performances globales des algorithmes et permet à plus de personnes d'éditer en même temps.

3

État de l'art

La littérature présente deux familles d'approches de réplication optimiste basées sur les opérations pour l'édition collaborative en temps réel. La première famille est appelée Operation Transformations (désignée par OT), la seconde est appelée Conflict-free Replicated Data Types (désignée par CRDT). La suite de ce chapitre présente ces deux familles.

3.1 Approche Transformées Opérationnelles

3.1.1 Principes de base

Dans l'approche OT, les modifications apportées au modèle considéré sont traduites en opérations, par exemple l'opération d'insertion ou de suppression d'un caractère à une position précise dans un document texte. Ces opérations sont échangées entre les collaborateurs et permettent à chacun de mettre son modèle à jour en fonction des opérations des autres.

Seulement, envoyer simplement les opérations n'est pas suffisant. Les intentions des opérations peuvent ne pas être respectées si deux opérations sont concurrentes. Par exemple, à partir du document "abde", un site peut générer l'opération d'insertion d'un caractère x entre b et d (c'est à dire à la position 2), et un deuxième site peut générer en concurrence la suppression du caractère a (la position 0). Les deux états des copies avant la réception des opérations de l'autre site sont donc "abxde" et "bde". Si ensuite on exécute la suppression sur le premier site, on obtient "bxde". Ici a est bien supprimé. Par contre sur le deuxième site, si on exécute l'insertion on obtient "bdxe", et l'intention de l'opération d'insertion n'est pas respectée : x a été inséré entre d et e.

Ce comportement, illustré figure 3.1, s'explique car l'état du second modèle a été modifié par la suppression. Plus précisément l'indice de la position entre b et d n'est plus 2 mais 1 à cause de la suppression qui a eu lieu. La solution retenue par l'approche OT pour résoudre ce problème est de transformer les opérations concurrentes reçues en fonction des opérations déjà exécutées, pour intégrer les effets qu'elles ont eus sur le modèle.

Pour reprendre l'exemple précédent, on peut déduire lors de la réception de l'insertion que, comme la suppression a supprimé un caractère avant la position initiale de l'insertion, il faut

décrémenter l'indice de l'insertion de 1 avant de l'exécuter. La figure 3.2 présente ce scénario.

Néanmoins, respecter les intentions des opérations n'implique pas forcément la convergence des copies. C'était le cas sur l'exemple précédent, mais on peut considérer une transformation qui, lorsque deux insertions ont lieu en concurrence au même endroit, renvoie une opération transformée qui place systématiquement le caractère de l'insertion locale avant celui de l'insertion distante. Ce scénario est illustré figure 3.3. Les intentions sont bien respectées, le caractère x est bien entre les caractères b et c sur tous les sites comme l'a voulu le premier utilisateur, et le caractère y est bien entre les caractères b et c comme l'a voulu le second utilisateur, pourtant

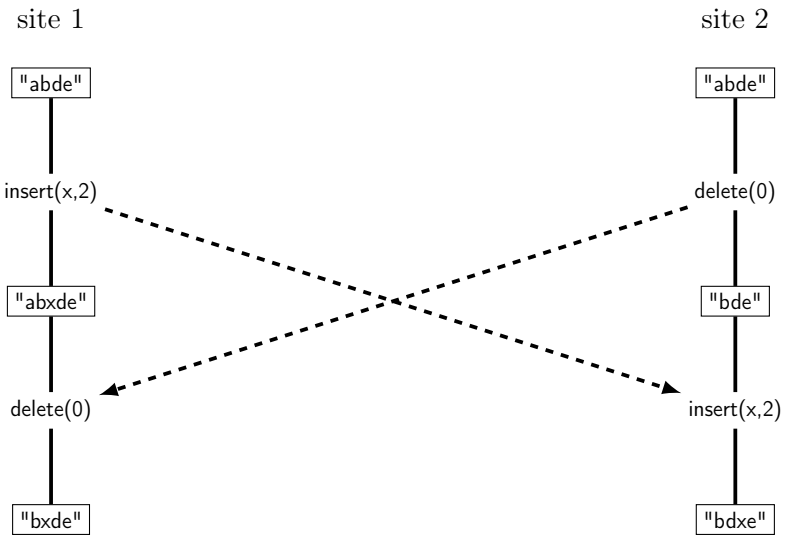


FIGURE 3.1 – Intégration des opérations sans transformation : intentions non respectées

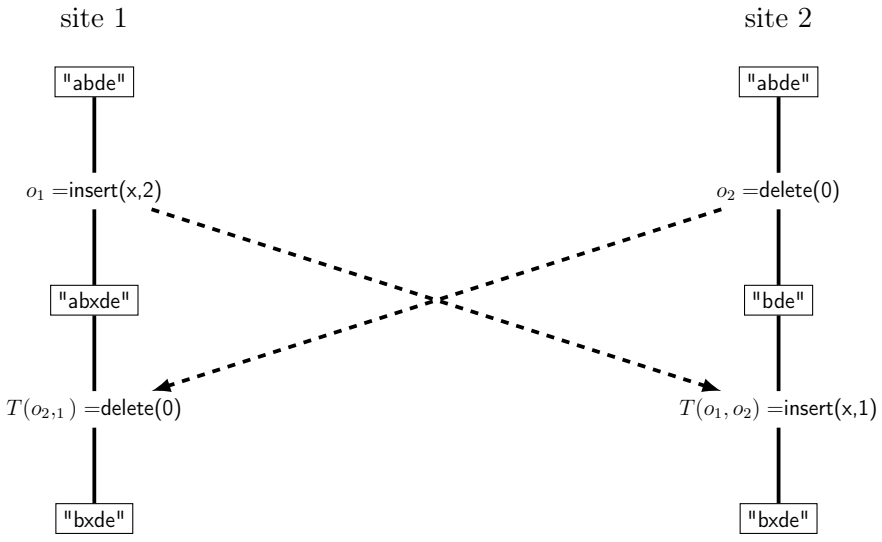


FIGURE 3.2 – Intégration et transformation des opérations

les deux copies divergent.

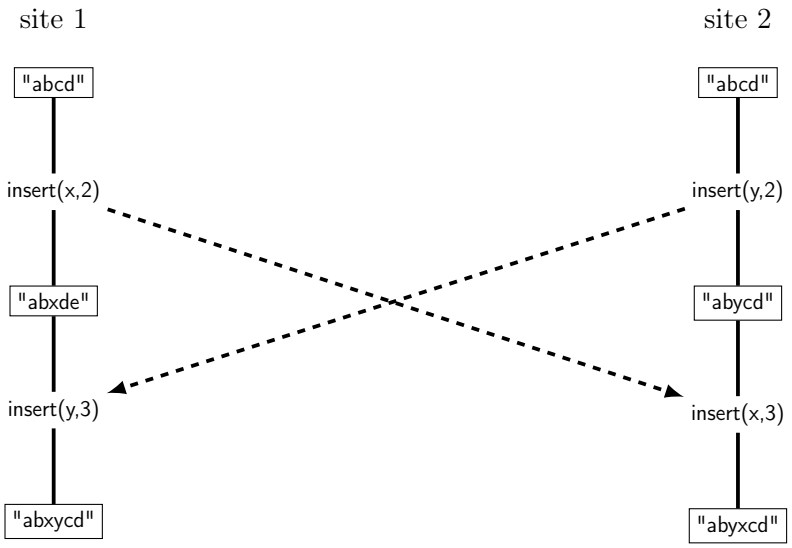


FIGURE 3.3 – Divergence avec respect des intentions

Pour que les copies convergent, les opérations et leurs transformées doivent respecter deux propriétés que nous allons présenter ici. Dans la suite de ce manuscrit, la transformée d'une opération o_1 par rapport à une opération o_2 sera notée $T(o_1, o_2)$, ou plus simplement o'_1 lorsqu'il n'y a pas d'ambiguïté sur l'opération o_2 considérée pour la transformée.

La première propriété, définie dans [Ellis and Gibbs, 1989], est connue sous le nom de TP_1 ou C_1, et s'énonce comme suit :

Définition : Transformation Property 1 Soit un état E du modèle et deux opérations o_1 et o_2 exécutées en concurrence sur cet état, alors $E \circ o_1 \circ o'_2 = E \circ o_2 \circ o'_1$, où $\circ$ désigne l'exécution d'une opération sur un état ou une séquence de deux opérations.

Cette propriété garantit que le scénario figure 3.3 ne peut pas se produire : à partir d'un état E, deux utilisateurs exécutent deux opérations en concurrence, o_1 et o_2. Le premier site a donc pour état $E_1 = E \circ o_1$ et le second $E_2 = E \circ o_2$. A la réception des opérations manquantes, les deux sites détectent que l'opération reçue est concurrente, la transforment, et l'intègrent. Sur le premier site, on aura donc l'état $E'_1 = E \circ o_1 \circ o'_2$ et sur le second l'état $E'_2 = E \circ o_2 \circ o'_1$. Si la propriété TP_1 est respectée, ces deux états sont identiques et donc les deux copies convergent.

Lorsqu'un troisième utilisateur (ou plusieurs) génère des opérations en concurrence, si on reprend l'illustration 3.3 , les deux utilisateurs peuvent recevoir une opération o_3 une fois qu'ils ont intégré les opérations o_1 et o_2 (ils sont à ce moment dans le même état $E'_1 = E'_2$). Alors pour le premier utilisateur, la transformée calculée sera $T(o_3, o_1 \circ o'_2)$, et $T(o_3, o_2 \circ o'_1)$ pour le deuxième. Pour que les copies convergent une fois les opérations intégrées et transformées, il faut que ces

deux transformées soient égales. Ceci est formalisé par la propriété C_2 [Ressel *et al.*, 1996] (ou TP_2), et illustré par la figure 3.4 :

Définition : Transformation Property 2 Soient trois opérations o_1,o_2,o_3 exécutées en concurrence sur le même état, alors $T(o_3, o_1 \circ T(o_2, o_1)) = T(o_3, o_2 \circ T(o_1, o_2))$.

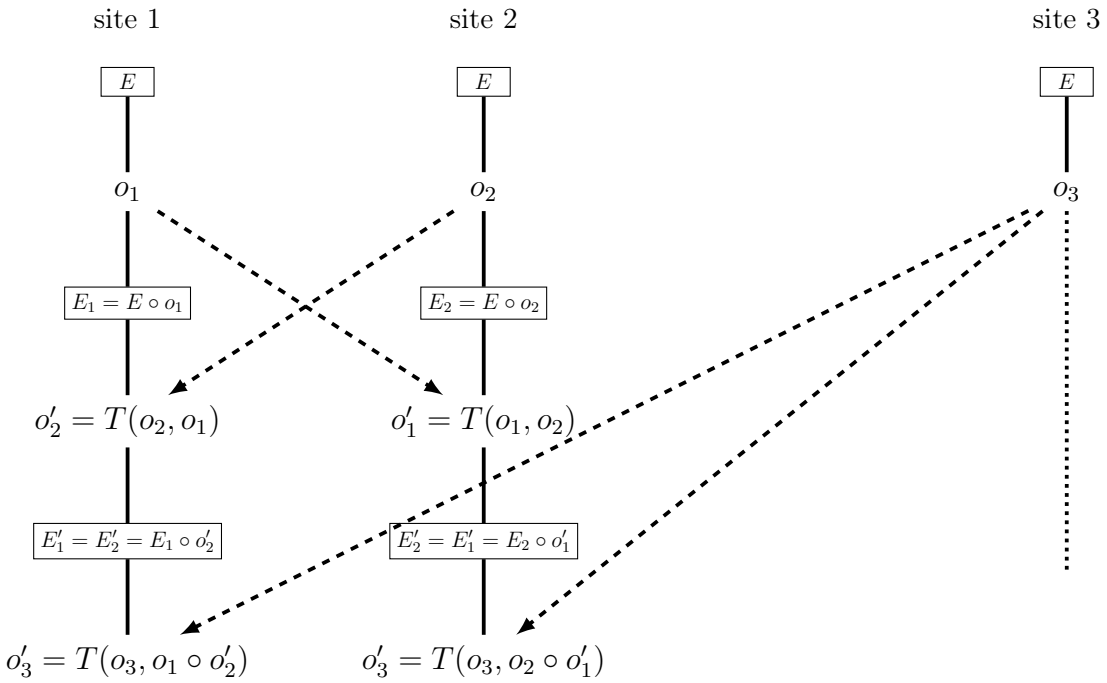


FIGURE 3.4 – Propriété TP2 avec trois opérations concurrentes

Si on s'intéresse au troisième utilisateur de l'exemple précédent, il peut par exemple recevoir les opérations dans l'ordre o_3,o_2,o_1 et générer l'état $E_3 = E \circ o_3 \circ T(o_2, o_3) \circ T(o_1, o_3 \circ T(o_2, o_3))$. Grâce aux propriétés TP_1 et TP_2, on peut montrer que cet état est le même que l'état $E_1 = E \circ o_1 \circ T(o_2, o_1) \circ T(o3, o_1 \circ T(o_2, o_1))$. Il nous faudra utiliser la formule de transformée par rapport à une séquence : pour toutes opérations o_1,o_2,o_3, alors $T(o_1, o_2 \circ o_3) = T(T(o_1, o_2), o_3)$.

$E_3 = E \circ o_3 \circ T(o_2, o_3) \circ T(o_1, o_3 \circ T(o_2, o_3))$
on applique la formule de transformée par rapport à une séquence :
$E_3 = E \circ o_3 \circ T(o_2, o_3) \circ T(T(o_1, o_3), T(o_2, o_3))$
on applique TP_1 à $T(o_2, o_3)$ et $T(o_1, o_3)$:
$E_3 = E \circ o_3 \circ T(o_1, o_3) \circ T(T(o_2, o_3), T(o_1, o_3))$
on applique TP_1 à o_1 et o_3 :
$E_3 = E \circ o_1 \circ T(o_3, o_1) \circ T(T(o_2, o_3), T(o_1, o_3))$
on applique la formule d'une transformée par rapport à une séquence :
$E_3 = E \circ o_1 \circ T(o_3, o_1) \circ T(o_2, o_3 \circ T(o_1, o_3))$

on applique TP_2 sur $T(o_2, o_3 \circ T(o_1, o_3))$:
$E_3 = E \circ o_1 \circ T(o_3, o_1) \circ T(o_2, o_1 \circ T(o_3, o_1))$
on applique la formule de transformée par rapport à une séquence pour supprimer la séquence :
$E_3 = E \circ o_1 \circ T(o_3, o_1) \circ T(T(o_2, o_1), T(o_3, o_1))$
on applique TP_1 à $T(o_3, o_1)$ et $T(o_2, o_1)$:
$E_3 = E \circ o_1 \circ T(o_2, o_1) \circ T(o_3, o_1 \circ T(o_2, o_1))$,
on obtient bien l'état E_1.

Les autres cas d'intégration de trois opérations concurrentes sont symétriques et ceci est généralisable pour quatre utilisateurs et plus, donc le respect des conditions TP_1 et TP_2 est suffisant pour assurer la convergences des copies.

Les opérations et leurs transformées sont une partie de l'approche OT. L'autre partie est l'algorithme de contrôle qui détermine quand une opération distante doit être transformée et par rapport à quelles autres opérations déjà reçues.

Théoriquement, lorsqu'elle est reçue, une opération distante doit être transformée par rapport aux opérations concurrentes qui ont déjà été exécutées, pour répercuter les changements apportés au document sur la forme d'exécution de l'opération. Quand seulement deux opérations sont impliquées, la marche à suivre est simple, mais dans le cas où les opérations concurrentes sont multiples, la situation est plus complexe. Par exemple, la figure 3.5 présente un cas dit de concurrence partielle (ici mal résolu). Le site 1 a généré une opération pendant que le site 2 en a généré deux. Les opérations o_1 et o_3 sont concurrentes, mais transformer o_3 par rapport à o_1 conduit à un résultat erroné. Le principe de l'approche OT est de ne transformer entre elles que des opérations qui ont été exécutées sur le même état, ce qui n'était pas le cas pour o_1 et o_3.

Au final, on définit une transformée comme suit :

Definition : Transformée Soient o_1 et o_2 deux opérations définies sur le même état E, on définit la transformée de l'opération o_1 par rapport à l'opération o_2 comme l'opération o_1', qui est définie sur l'état $E \circ o_2$ et qui réalise les mêmes effets que l'opération initiale o_1. L'opération o_1' peut être considérée comme la forme d'exécution de o_1 sur l'état $E \circ o_2$.

Pour illustrer cette définition, un exemple de transformées issu de [Ressel *et al.*, 1996] est présenté figure 3.6. Les opérations considérées sont $Ins(p, c, u)$ et $Del(p, u)$, qui respectivement insèrent le caractère c à la position p et suppriment le caractère à la position p. Le dernier paramètre u est le numéro unique du site qui a généré l'opération. Ce paramètre est utilisé dans les transformées lorsqu'un choix arbitraire doit être fait : quand deux insertions concurrentes ont lieu à la même position, il faut décider quel caractère est placé avant l'autre. Ce cas est décidé en regardant quels sont les sites qui ont généré les opérations ; le site qui a le numéro le plus petit est prioritaire et ne modifie pas la place de son caractère, tandis que l'autre décale sa position.

Pour expliquer en détail comment les algorithmes détectent quelles opérations doivent être transformées, on peut se tourner vers l'algorithme de contrôle COT [Sun and Sun, 2009] et la

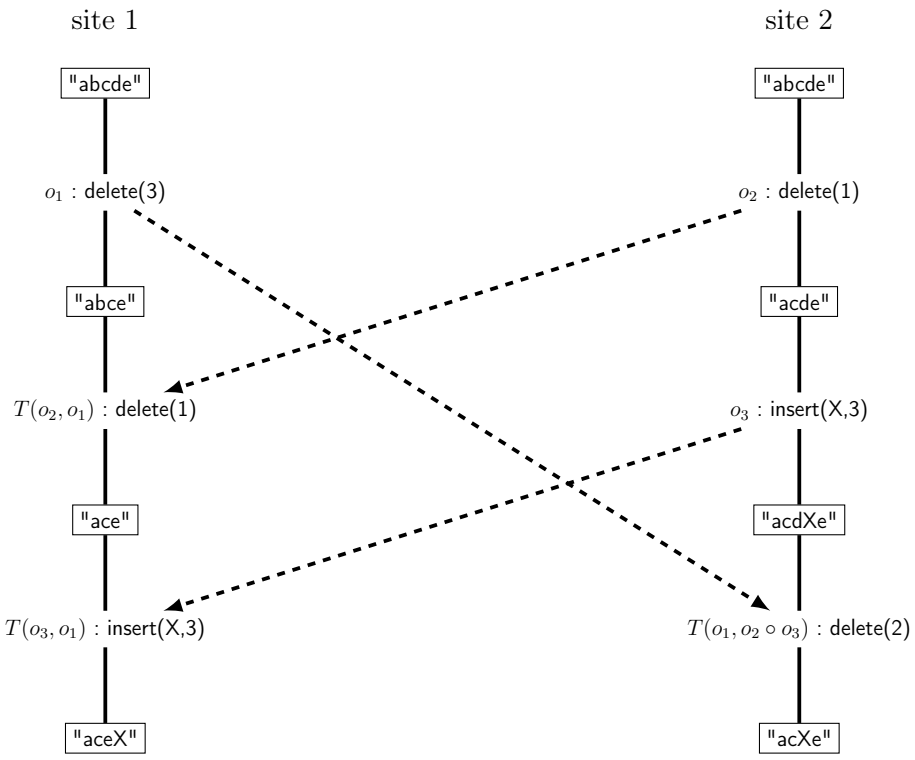


FIGURE 3.5 – Concurrence partielle mal résolue

notion de contexte qu'il introduit. Le contexte d'une opération est la liste de toutes les opérations exécutées avant celle-ci. Il définit donc l'état sur lequel l'opération s'exécute. Deux opérations sont concurrentes si aucune des deux n'apparaît dans le contexte de l'autre. Cette condition de concurrence n'est pas suffisante pour pouvoir transformer une opération par rapport à une autre. Une opération ne peut être transformée que par rapport à une opération qui possède le même contexte qu'elle.

Les contextes peuvent être représentés par des vecteurs d'états[Mattern, 1989, Fidge, 1991]. Un vecteur d'état est un tableau d'entiers. Chaque site possède son propre vecteur d'état, qui est mis à jour en fonction des opérations générées et exécutées. Il y a une entrée dans le tableau par site participant à la collaboration, et la valeur de chaque entrée i correspond au nombre d'opérations émises par le site i qui ont été intégrées localement. On peut trouver dans la littérature des approches similaires à COT, qui fonctionnent directement avec des vecteurs d'état.

Chaque site maintient un historique des opérations qu'il a exécutées. Lorsqu'une opération distante *op* arrive sur un site, cet historique est comparé au contexte de l'opération. Si l'historique et le contexte sont les mêmes, l'opération peut être intégrée sans transformation. Si le contexte de l'opération contient des opérations qui ne sont pas dans l'historique du site, alors l'opération n'est pas prête à être intégrée, il faut attendre de recevoir les opérations manquantes, c'est-à-dire les opérations dont l'opération en attente dépend causalement. Si l'historique contient des

```
1  T(Ins(p1, c1, u1), Ins(p2, c2, u2)) :
2      if ((p1 < p2) or (p1 = p2 and u1 < u2))
3        return Ins(p1, c1, u1)
4      else
5        return Ins(p1 + 1, c1, u1)
6      endif
7  end
8
9  T(Ins(p1, c1, u1), Del(p2, u2)) :
10     if p1 ≤ p2
11       return Ins(p1, c1, u1)
12     else
13       return Ins(p1 − 1, c1, u1)
14     endif
15 end
16
17 T(Del(p1, u1), Ins(p2, c2, u2)) :
18     if p1 < p2
19       return Del(p1, u1)
20     else
21       return Del(p1 + 1, u1)
22     endif
23 end
24
25 T(Del(p1, u1), Del(p2, u2)) :
26     if p1 < p2
27       return Del(p1, u1)
28     else if (p1 > p2)
29       return Del(p1 − 1, u1)
30     else
31       return Id()
32     endif
33 end
```

FIGURE 3.6 – Exemple de transformées

opérations op_i qui ne sont pas incluses dans le contexte de l'opération distante, alors ce sont des opérations concurrentes avec op, et op doit être transformée par rapport à celles-ci.

Une opération op_i dont le contexte est inclus dans celui de op est choisie. En reprenant l'exemple de la figure 3.5, lorsque o_3 arrive sur le site 1 son contexte est o_2, on peut donc choisir o_1, dont le contexte est vide. Lorsque o_1 arrive sur le site 2, on peut choisir o_2 dont le contexte est vide également, mais pas o_3.

Si les contextes de l'opération op et de l'opération op_i choisie sont les mêmes, on peut transfor-

mer directement *op* en *op′*. Le contexte de la nouvelle opération *op′* est le contexte de l'opération initiale *op* augmenté de l'opération op_i. On recommence ensuite la procédure avec *op′* en choisissant une autre op_i, jusqu'à avoir réalisé toutes les transformations. C'est ce qui se passe sur le site 2 de notre exemple : on obtient $o'_1 = T(o_1, o_2)$, puis comme le contexte de o'_1 est le même que celui de o_3 (ce contexte est o_2), on peut transformer o'_1 par rapport à o_3.

Sinon, le contexte de *op* contient plus d'opérations que celui de l'opération op_i. Dans ce cas on ne peut pas réaliser la transformée directement, puisque les contextes ne sont pas les mêmes. Il faut avant augmenter le contexte d'op_i avec les opérations manquantes disponibles dans le contexte d'*op*, c'est-à-dire transformer op_i par rapport à ces opérations. C'est ce que l'on doit faire sur le site 1 de notre exemple lorsque o_3 arrive. La figure 3.7 corrige l'erreur de la figure 3.5. Comme le contexte de o_3 est o_2 est que celui de o_1 est vide, avant de transformer o_3 on calcule o'_1, la transformée de o_1 par rapport à o_2. Puis on peut transformer o_3 par rapport à o'_1.

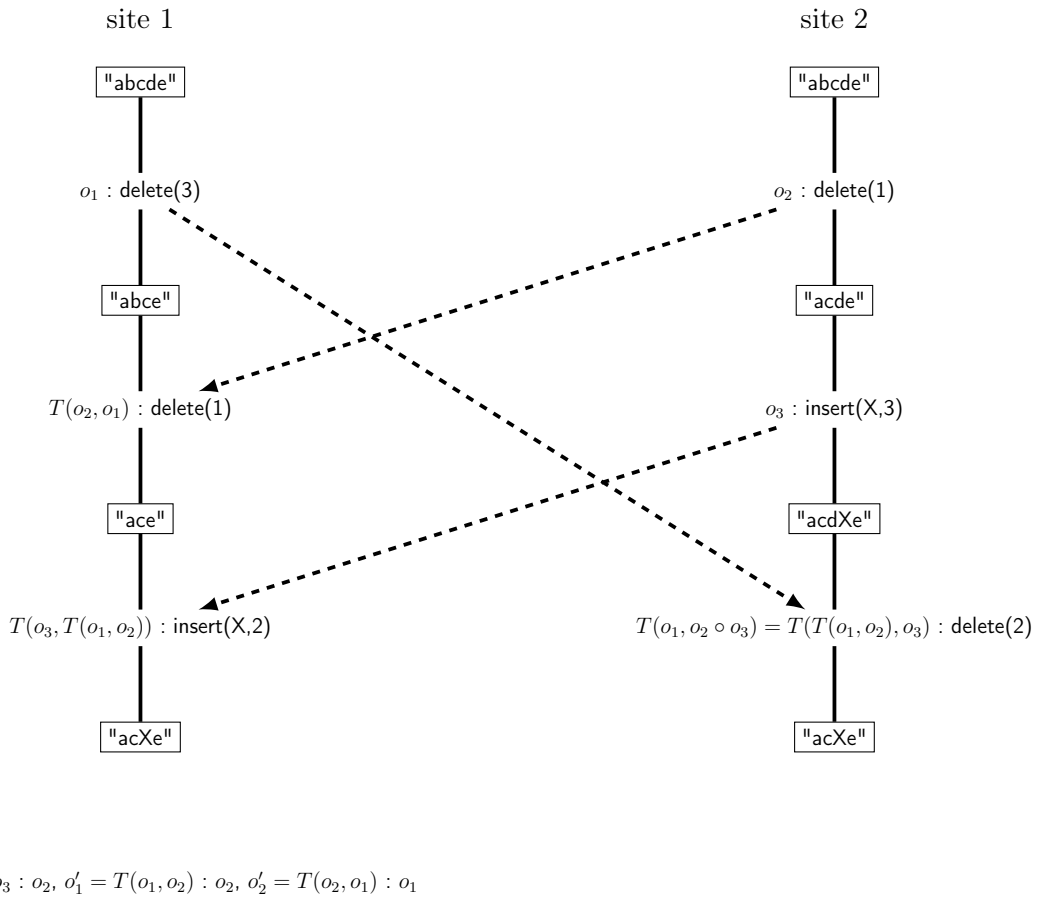


FIGURE 3.7 – Concurrence partielle correctement traitée

L'algorithme COT garde en mémoire les opérations initiales (et non les opérations transformées). En effet, si l'on regarde quelles opérations ont été exécutées site 1 de la figure 3.7, on obtient o_1, puis $o'_2 = T(o_2, o_1)$ et enfin $o'_3 = T(o_3, T(o_1, o_2))$. Si on stocke directement o'_2 dans

l'historique, o_2 n'est plus disponible pour transformer o_1 à la réception de o_3. Par contre, si on veut reconstituer le document à partir de l'historique, il faut recalculer toutes les transformées. L'algorithme SOCT2 [Suleiman *et al.*, 1997] prend le parti de stocker les opérations transformées, et en contre-partie a besoin de transformées inverses, qui permettent de retrouver l'opération initiale à partir de sa transformée et de l'opération qui a servi à la transformer. L'avantage est de ne pas avoir à recalculer des transformées déjà connues dans certaines situations, comme illustré figure 3.8, où les sites 1, 2 et 3 ont besoin deux fois des transformées $T(o_2, o_1)$,$T(o_3, o_2)$ et $T(o_1, o_3)$ respectivement.

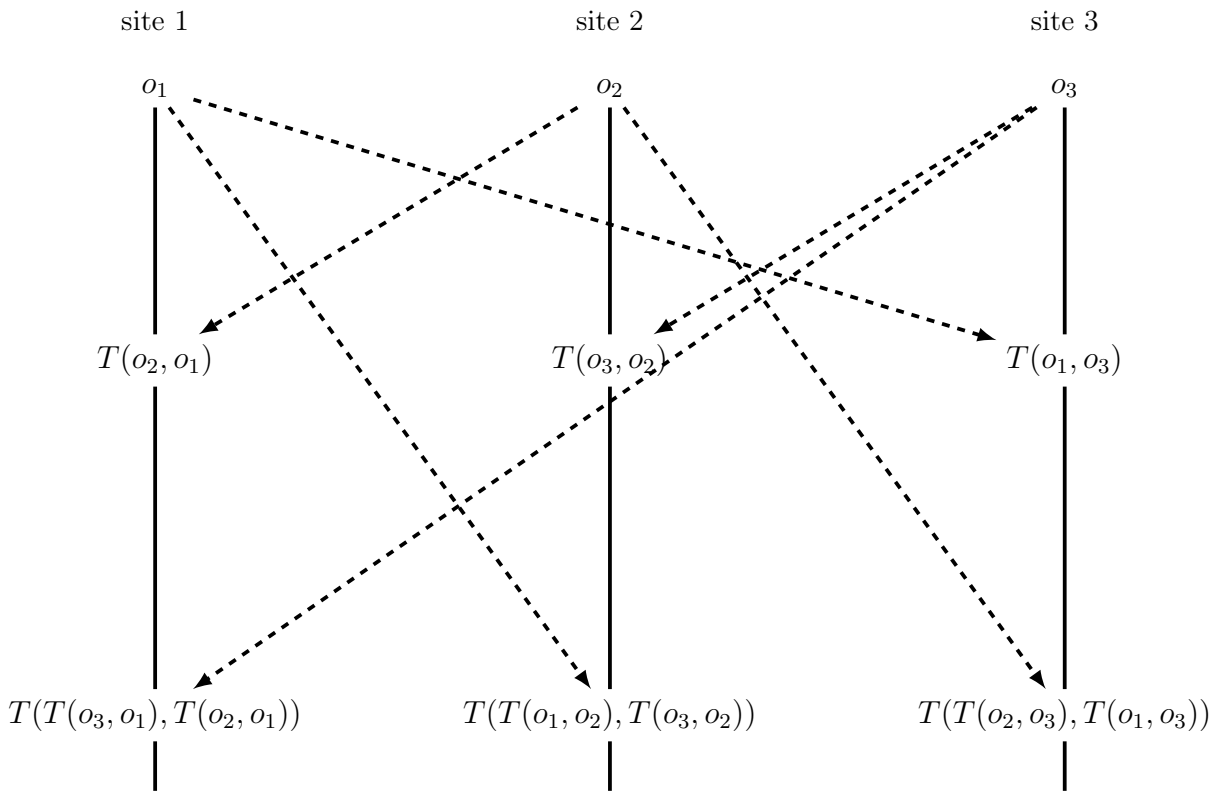


FIGURE 3.8 – Concurrence sur trois sites

Concevoir des transformées qui respectent la condition TP2 est un exercice difficile. Les seules transformées connues pour l'édition de document texte qui respectent TP2 se basent sur une opération de suppression qui ne retire pas du modèle le caractère supprimé mais se contente de le rendre invisible [Oster *et al.*, 2006a]. Un tel caractère est appelé une pierre tombale, et le problème de cette approche est qu'un caractère supprimé continue de prendre de la place en mémoire dans le modèle du document. Des méthodes ont donc été développées pour pouvoir se passer de TP2 dans les algorithmes de contrôle. L'idée générale est la même quelque soit la méthode. Elle consiste à ordonner totalement les opérations et à réaliser les transformations et les intégrations des opérations distantes uniquement suivant cet ordre. C'est-à-dire que l'on

n'intègre pas une opération distante tant que toutes les opérations plus petites selon cet ordre n'ont pas été intégrées, et que lorsqu'une opération doit être transformée par rapport à plusieurs opérations, on la transforme en priorité par rapport aux plus petites. L'ordre en question doit être compatible avec l'ordre causal (si une opération o_1 est causalement avant une opération o_2, alors o_1 est plus petite que o_2).

L'exemple suivant permet de comprendre pourquoi avec cet ordre, seule la propriété TP_1 est requise. Considérons trois opérations concurrentes o_1, o_2 et o_3 générées sur trois sites différents. Chaque site exécute toujours immédiatement ses opérations locales, ce qui conduit aux trois exécutions suivantes en considérant que $o_1 < o_2 < o_3$: $o_1 \circ T(o_2, o_1) \circ T(T(o_3, o_1), T(o_2, o_1))$ sur le premier site, $o_2 \circ T(o_1, o_2) \circ T(T(o_3, o_1), T(o_2, o_1))$ sur le deuxième, et $o_3 \circ T(o_1, o_3) \circ T(T(o_2, o_1), T(o_3, o_1))$ sur le dernier.

Grâce à TP_1 sur $o_2 \circ T(o_1, o_2)$, la séquence d'exécution sur le deuxième site est équivalente à $o_1 \circ T(o_2, o_1) \circ T(T(o_3, o_1), T(o_2, o_1))$, qui est la séquence exécutée sur le premier site.

Grâce à TP_1 sur $o_3 \circ T(o_1, o_3)$, la séquence d'exécution sur le troisième site est équivalente à $o_1 \circ T(o_3, o_1) \circ T(T(o_2, o_1), T(o_3, o_1))$. Cette séquence est elle-même équivalente à $o_1 \circ T(o_2, o_1) \circ T(T(o_3, o_1), T(o_2, o_1))$ (TP_1 sur $T(o_3, o_1) \circ T(T(o_2, o_1), T(o_3, o_1))$), qui est à nouveau la séquence exécutée sur le premier site.

Les trois séquences sont donc équivalentes si uniquement TP_1 est respectée.

Pour ordonner totalement les opérations, on peut se baser sur des vecteurs d'états : une opération o_1 est plus petite qu'une opération o_2 si et seulement si la somme des entrées du vecteur d'états de o_1 est plus petite que la somme des entrées du vecteur d'état de o_2, ou que ces deux sommes sont égales et que le numéro de site qui a généré o_1 est plus petit que celui du site qui a généré o_2. Le défaut d'un tel ordre est que lorsqu'une opération est reçue, il n'est pas possible d'anticiper si une autre opération plus petite va arriver plus tard, ou si cette opération doit être intégrée. Par exemple, si on considère trois sites numérotés 1, 2 et 3, qui réalisent une opération sur le même document vide, les opérations et leurs vecteurs d'états seront respectivement o_1 et [1,0,0], o_2 et [0,1,0], o_3 et [0,0,1]. Le site numéro 1 peut recevoir en premier l'opération du site 3, et comme il n'a aucune connaissance sur l'opération du site 2, il intègre et transforme o_3 par rapport à o_1. Puis o_2 arrive sur le site 1, et comme elle est plus petite que o_3 il faut annuler l'exécution de o'_3 et des opérations locales qui ont eu lieu après son intégration, transformer o_2 par rapport à o_1 et l'exécuter, transformer o_3 par rapport à o_1 et o'_2 et l'exécuter, et rejouer les opérations locales, dont la forme d'exécution a changé suite à l'intégration de o'_2.

Une autre possibilité pour ordonner totalement les opérations est de bénéficier d'un estampilleur centralisé, qui réceptionne toutes les opérations et leur attribue un numéro d'ordre.

Pour contourner le problème, on peut aussi se tourner vers l'algorithme de contrôle Jupiter [Nichols *et al.*, 1995], qui a été conçu pour synchroniser deux sites (un client et un serveur). La propriété TP_2 n'est pas nécessaire, aucun troisième site ne peut interférer. La synchronisation de plusieurs clients se réalise grâce à plusieurs couples clients-serveurs, les différents serveurs se synchronisant de manière atomique [Zafer, 2001] .

Le tableau ci dessous résume les différents algorithmes de contrôle que l'on peut trouver dans la littérature et leurs caractéristiques.

Algorithme	Propriétés requises	Transformées inverses	Ordre d'intégration
Jupiter	TP_1	Non	ordre causal
GOT	Aucune	Oui	ordre total
GOTO	TP_1 et TP_2	Oui	ordre causal
COT	TP_1	Non	ordre causal local, ordre total distant
SOCT2	TP_1 et TP_2	Oui	ordre causal
SOCT4	TP_1	Non	ordre total

L'algorithme GOT[Sun *et al.*, 1998] fonctionne avec des opérations qui n'ont pas besoin de respecter les propriétés TP_1 ou TP_2. A chaque nouvelle opération reçue, il ordonne son historique selon un ordre total basé sur les vecteurs d'états des opérations, et le rejoue dans cet ordre. Plusieurs sites qui ont reçu les mêmes opérations ont donc un état qui est le fruit de l'exécution de séquences d'opérations rigoureusement identiques, et donc les copies convergent. Des transformées sont tout de même requises pour respecter l'intention des opérations.

L'algorithme GOTO[Sun and Ellis, 1998] est l'optimisation de l'algorithme GOT, lorsque les opérations et transformées respectent les propriétés TP_1 et TP_2. Les opérations sont intégrées ici selon l'ordre causal représenté par les vecteurs d'états, l'historique d'un site n'est jamais réorganisé. Cet algorithme est très proche de SOCT2 [Suleiman *et al.*, 1997].

L'algorithme COT[Sun and Sun, 2009] élimine le besoin de la propriété TP_2 comme expliqué plus haut, en transformant les opérations concurrentes selon un ordre total. Les opérations locales sont exécutées immédiatement sur la copie et ne respectent pas cet ordre total, mais uniquement l'ordre causal.

L'algorithme SOCT4[Vidot *et al.*, 2000] utilise un estampilleur centralisé pour affecter des numéros d'ordre aux opérations. Une opération ne peut être transmise aux sites distants que si son numéro est le prochain sur la liste. Par exemple, un site peut vouloir transmettre une opération qui se voit attribuer le numéro 8. Ce site n'a reçu que les opérations de 1 à 5, il doit donc attendre de recevoir les opérations 6 et 7 avant d'envoyer son opération. Le désavantage de cet envoi retardé est compensé par le fait que c'est le site émetteur qui réalise la transformation une fois pour tous. Dans notre exemple, les opérations 6 et 7 sont transformées par rapport à l'opération 8 lors de leur intégration, mais ensuite l'opération 8 est transformée par rapport aux opération 6 et 7 avant d'être envoyée. Ainsi, un site distant qui a reçu les 7 premières opérations et qui reçoit l'opération 8 peut l'exécuter directement sans la transformer s'il n'a pas d'opérations locales en attente.

Le coût majeur d'une approche OT est le nombre de transformées nécessaires pour intégrer une opération ([Li and Li, 2006]). Un algorithme comme GOT qui doit constamment réorganiser et transformer son historique sera plus coûteux que l'algorithme SOCT4.

3.1.2 Intentions des utilisateurs

Concernant le respect de l'intention des utilisateurs, il est lié au jeu d'opérations et de transformées qui sont utilisées avec l'algorithme de contrôle choisi. Les premiers jeux pour l'édition collaborative de document texte utilisent uniquement deux opérations, celle d'insertion d'un caractère à une position fixe du document, et celle de suppression d'un caractère à une position, les transformations se devant de modifier ces positions pour refléter l'ajout ou la suppression de caractères en concurrence. Les évolutions de ces jeux d'opérations et de transformées conservent ce principe [Ellis and Gibbs, 1989, Ressel *et al.*, 1996, Imine *et al.*, 2003, Suleiman *et al.*, 1997, Sun *et al.*, 1998]. Ces opérations sont suffisantes pour éditer un document texte en collaboration, et respecter des intentions des utilisateurs de bas niveau comme l'insertion de caractères ou la mise à jour d'un mot, mais ne permettent pas de respecter des intentions plus complexes. Par exemple, le déplacement d'une chaîne de caractères sera traduit par la suppression des caractères suivi de leur réinsertion ailleurs dans le document. Si une partie de cette chaîne est supprimée en parallèle par un autre utilisateur, le résultat attendu une fois le conflit d'édition résolu est un document qui contient la chaîne déplacée, avec les caractères supprimés retirés du document. Pourtant, après résolution du conflit le document contient la chaîne déplacée comme si la suppression n'avait jamais eu lieu. En effet, du point de vue de l'algorithme, les caractères insérés par le déplacement sont complètement différents de ceux supprimés, et donc pour que l'intention de cette opération d'insertion soit respectée, tous les caractères sont insérés dans le modèle. Les intentions des deux suppressions sont également respectées : les caractères qu'elles ciblent ne sont plus dans le modèle, comme ils sont différents de ceux de l'insertion.

Ce modèle basé sur les opérations d'insertion et de suppression de texte atteint également rapidement ses limites lorsque l'on souhaite l'intégrer dans les éditeurs ou applications actuelles. Les éditeurs de texte comme *Microsoft Word* ont besoin de métadonnées pour fonctionner correctement et hiérarchiser le document. Par exemple, pour générer automatiquement une table des matières, un tel éditeur est capable grâce aux métadonnées de faire la différence entre des titres et des paragraphes, qui dans un simple document texte sont tous deux des blocs de caractères séparés par un saut de ligne. La structure de données de l'éditeur n'est pas qu'une simple suite de caractères. Bien que ces données et métadonnées peuvent être linéarisées, utiliser l'approche linéaire et les opérations d'insertion et de suppression de texte sur le modèle linéarisé conduit à des mauvaises résolutions des conflits. Par exemple, le langage à balises HTML peut être linéarisé sans aucun problème. Une chaîne de texte “Bonjour à” peut être mise en gras, le document modèle aura pour contenu “Bonjour à”. Une chaîne “à tous.” peut être mise en italique, et le modèle sera “’à tous.”. Sur le texte “Bonjour à tous”, ces deux modifications en parallèle vont ajouter les balises et conduire au modèle linéaire “’Bonjour à tous.”, qui est mal formé et ne correspond pas à un document HTML valide. Le document bien formé serait soit “’Bonjour à tous.”, ou bien “’Bonjour à tous.”, mais rien ne permet à l'algorithme de savoir qu'il faut rajouter ces balises supplémentaires. Il produit donc un contenu erroné, qui est mal formé, bien que

les intentions des opérations d'ajout de caractères soient respectées.

Dans la suite de cette partie, nous allons voir quelles sont les approches existantes qui s'adressent à ces problèmes, et quels sont les modèles et opérations qu'elles utilisent.

L'algorithme TreeOpt

L'algorithme TreeOpt [Ignat and Norrie, 2002] permet de hiérarchiser une document en le séparant en plusieurs parties distinctes bien définies. Les opérations ne modifient plus toutes la même zone de texte (le document complet), mais des zones plus petites et liées entre elles. Ces zones correspondent aux caractères, aux mots, aux phrases et aux paragraphes du document. Les opérations ciblent précisément une zone et transportent plus d'information qu'une simple opération d'insertion. Par exemple, au lieu d'insérer un caractère à la position 10, on peut insérer un caractère dans le troisième mot de la deuxième phrase, à la position 2.

L'algorithme utilise un modèle de document à plusieurs couches. La première couche est le document lui-même. Il est constitué des éléments de la seconde couche, les paragraphes. Ceux-ci sont constitués de phrases, elles-mêmes constituées de mots, qui sont constitués de caractères.

La structure du modèle s'apparente donc à un arbre, et chaque opération cible un sous-arbre. On peut par exemple insérer une phrase en seconde position du deuxième paragraphe, ou insérer un caractère au début du troisième mot de la seconde phrase du premier paragraphe. La figure 3.9 représente un modèle simplifié de document TreeOpt.

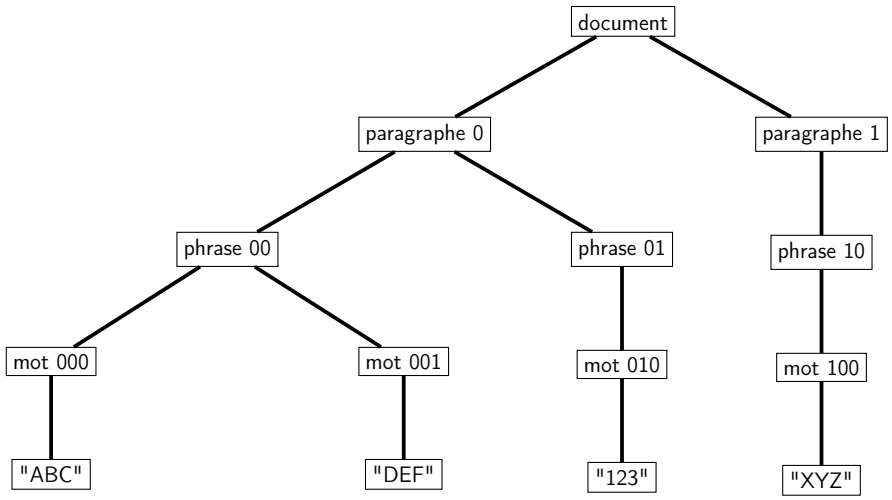


FIGURE 3.9 – Exemple de document TreeOpt

Plus concrètement, les opérations possibles pour modifier le document sont *InsertParagraph*, *InsertSentence*, *InsertWord*, *InsertCharacter*, et quatre fonctions *Delete* nommées selon le même principe. Les fonctions d'insertion prennent en paramètres le contenu qu'elles insèrent et le chemin de la racine vers la position où le contenu est inséré. Les fonctions de suppression prennent en paramètre uniquement un chemin. Deux exemples d'opérations et leurs paramètres sont *In-*

sertSentence(1,1,"hello world") et *InsertCharacter(0,1,2,0,"X")*. Les positions d'insertion ou de suppression ne sont donc plus des positions absolues dans le document, mais sont relatives à un nœud de l'arbre. Seules les opérations concurrentes qui agissent sur le même nœud sont conflictuelles.

Chaque élément d'une couche est traité comme une structure linéaire indépendante qui possède son propre historique et son propre algorithme de contrôle. Les opérations sur les sous-arbres sont découpées en opérations qui agissent sur chacune des couches de l'arbre. L'insertion de la phrase "hello world" de notre exemple insère un nouvel élément de type phrase dans le deuxième élément de la couche paragraphe. Cette insertion consiste à créer une nouvelle structure phrase, qui va contenir deux structures mot, qui vont chacune contenir cinq caractères. L'historique de l'élément paragraphe où la phrase est insérée est modifié par l'ajout de cette opération. Dans notre deuxième exemple, l'élément de type mot voit son historique modifié par l'ajout d'une opération d'insertion d'un élément (le caractère X) en position 0, et le caractère est inséré.

Lorsqu'une opération distante est reçue, les positions de son chemin sont comparées aux positions des opérations concurrentes stockées dans l'historique des éléments qu'elle traverse, et transformées avec l'algorithme OT choisi.

En conséquence, toutes les opérations concurrentes ne sont pas transformées les unes par rapport aux autres. Lorsque par exemple deux opérations insèrent en concurrence une phrase dans deux paragraphes différents, elles n'ont pas connaissance l'une de l'autre et ne sont pas transformées. D'après les auteurs, cela permet de réduire d'un facteur 10 ou plus le nombre de transformées nécessaires à l'intégration des opérations par rapport aux approches classiques, ce qui est un gain important.

Le désavantage de cette solution est qu'elle ne permet pas de modifier de façon satisfaisante la structure du document. Par exemple, si un utilisateur veut couper un paragraphe en deux, les opérations générées sont d'une par la suppression de toutes les phrases qu'il souhaite déplacer, et d'autre part l'insertion de nouvelles phrases, identiques à celles supprimés, dans un nouveau paragraphe. Cela produit plus d'opérations que la simple insertion d'un caractère de saut de paragraphe, mais conduit également à une résolution imprécise de certains conflits. Si par exemple, en concurrence de cette séparation, un utilisateur ajoute une phrase à la fin du paragraphe qui va être coupé en deux, cette phrase ne va pas être déplacée. Elle reste à la fin du paragraphe initial, au lieu de se trouver à la fin du paragraphe qui est créé lors de la coupure.

Une approche pour documents XML

Une approche pour documents XML, qui permettent notamment de représenter des documents texte hiérarchisés, est présentée dans [Davis *et al.*, 2002]. Les auteurs y précisent qu'une approche linéaire classique ne peut pas convenir à l'édition collaborative de documents XML, puisque les éléments XML sont imbriqués et étiquetés. Ils proposent à la place une structure d'arbre et des opérations et transformations génériques pour l'éditer.

Un document peut contenir plusieurs arbres. Chaque nœud d'un arbre est constitué d'un

type (équivalent aux éléments XML), d'une propriété représentant le contenu du nœud (texte pour une feuille, liste de nœuds fils sinon), et d'autres propriétés dépendantes du type du nœud (représentant les attributs XML). La racine d'un arbre est identifiée de manière unique et les nœuds d'un arbre sont identifiés à partir de la racine et du chemin depuis la racine vers le nœud.

Trois opérations permettent de modifier le modèle. L'opération *Insert(N,n,M,T)* insère un arbre de type T dans le modèle, en tant que fils numéro n du nœud identifié par N. Si le modèle contient un arbre dont la racine est M, c'est cet arbre qui est inséré à la position n, sinon un nouveau nœud est créé. Cette opération permet donc soit de créer un nouveau nœud, soit de concaténer deux arbres.

L'opération *Delete(N,n,M)* supprime le sous-arbre numéro n du nœud identifié par N. Il n'est pas retiré du modèle, il est à la place sauvegardé dans un nouvel arbre dont la racine prend l'identifiant M.

L'opération *Change(N,k,f)* change la valeur de la propriété k du nœud N selon la fonction f.

Cette approche définit des nouvelles transformations adaptées à ses nouvelles opérations. Pour synthétiser, les opérations d'insertion et de suppression nécessitent de comparer les chemins (listes de positions) de la même façon qu'on le ferait pour les positions lors d'une approche linéaire, avec toutefois des cas nouveaux lorsqu'un chemin est préfixe de l'autre. Transformer une opération *Change* par rapport à une insertion ou une suppression revient à déterminer sa nouvelle position. Transformer une opération d'insertion ou de suppression par rapport à une opération *Change* revient à retourner l'opération sans transformation. Il en va de même pour deux opérations *Change*, sauf si elles ciblent la même propriété du même nœud. Dans ce cas, les opérations à transformer sont les fonctions paramètres f, et la valeur de la propriété est considérée comme une structure avec son propre mécanisme OT. Comme les valeurs des propriétés sont essentiellement du texte, les auteurs suggèrent d'utiliser une approche OT linéaire classique dans ce cas.

Un des problèmes de cette approche est que, bien qu'elle permette effectivement de représenter des documents XML, elle ne prend en compte aucune considération d'ordre sémantique spécifique à XML. Par exemple, une propriété "italique" d'un nœud peut prendre n'importe quelle valeur, et pas uniquement "vrai" et "faux". Une valeur erronée entraîne des modifications supplémentaires indésirables. Elle souffre aussi du même problème que l'approche TreeOpt, à savoir que des opérations d'édition relativement simples génèrent beaucoup d'opérations. Mettre en gras des caractères au milieu d'une feuille texte équivaut à changer le texte de cette feuille pour qu'il ne contienne que le début, créer une nouvelle feuille qui contient les caractères gras, et créer une troisième feuille qui contient les caractères de la fin. Cela génère trois opérations et est sujet à des résolutions de conflits imprécises en cas de concurrence. De plus, même si les auteurs indiquent que des opérations de plus haut niveau comme l'opération de déplacement peuvent être implémentées sans trop de difficulté grâce aux opérations d'insertion et de suppression, qui permettent la fusion et l'extraction de sous arbres, rien n'est détaillé. On peut être sceptique, d'autant plus que certaines transformées conduisent à des modèles divergents : lorsque le même sous-arbre est supprimé en concurrence, la transformée est équivalente à une opération qui ne fait rien, afin de ne pas le supprimer une seconde fois. En conséquence, comme le sous-arbre

supprimé est toujours présent dans le modèle mais avec un nouvel identifiant de racine, chaque site le stocke avec l'identifiant qu'il a choisi, ce qui conduit à deux modèles différents et peut poser des problèmes lors de la réinsertion éventuelle du sous-arbre.

L'approche Transparent Adaptation

L'approche Transparent Adaptation présentée dans [Xia *et al.*, 2004, Sun *et al.*, 2006] propose une méthode basée sur un mécanisme OT pour permettre de travailler en collaboration sur n'importe quelle application initialement prévue pour un seul utilisateur. Un exemple développé est l'éditeur de texte *Microsoft Word*, qui permet l'édition de documents textes complexes, qui gère du texte hiérarchisé, du style, et de l'insertion de tableaux et de listes.

L'idée de base est de traduire toutes les opérations haut niveau de l'application en opérations bas niveau, et d'utiliser une approche OT sur ces opérations bas niveau. Les opérations de bas niveau sont des opérations d'insertion, de suppression et de mise à jour, et ne dépendent pas de l'application.

Les opérations du niveau de la couche applicative (ajouter un titre, modifier une image...) générées localement par un utilisateur sont interceptées et traduites en opérations bas niveau. Puis, ce sont ces opérations qui sont envoyées aux sites distants, et qui sont utilisées avec l'algorithme OT. Une fois les opérations distantes reçues et transformées, elles sont traduites en opérations de la couche applicative, et exécutées sur la copie locale. La figure 3.10 illustre cette procédure.

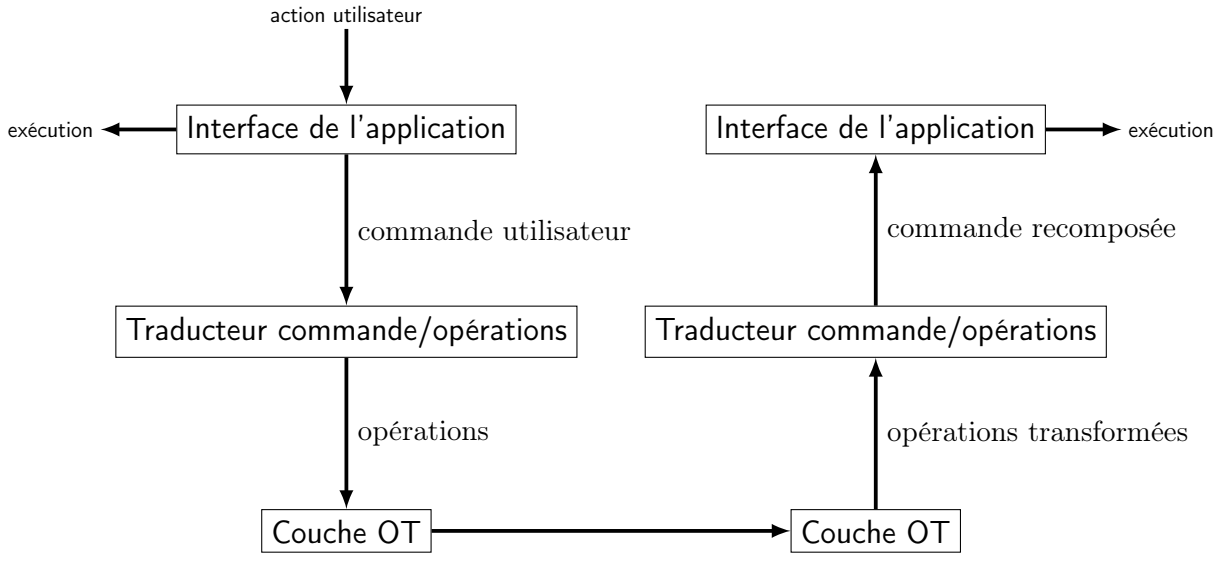


FIGURE 3.10 – Principe de l'approche Transparent Adaptation

Le modèle de l'approche OT est un modèle arborescent qui a des principes similaires à celui décrit précédemment. Pour qu'une application standard puisse être transformée en application collaborative, son modèle de données doit être compatible avec ce modèle arborescent. Par

exemple, dans le cas de l'éditeur *Word*, le modèle OT équivalent est un arbre à deux niveau. Le premier (la racine) contient une liste d'éléments, et le deuxième contient lesdits éléments (zone de texte, titres, tableaux...). Pour l'éditeur de présentation *PowerPoint*, un troisième niveau plus haut que la racine précédente est nécessaire. Il contient la liste des diapositives de la présentation.

Cette approche a l'avantage d'être générique. Quelle que soit l'application, le modèle et les opérations de l'approche OT sont les mêmes. Ce qui change d'une application à l'autre sont les liens entre la couche applicative et la couche OT, c'est-à-dire la façon dont les commandes de la couche applicative sont traduites en opérations de la couche OT, et la façon dont les opérations de la couche OT sont traduites en commandes de la couche applicative. La première traduction est assez immédiate, la deuxième est plus complexe. Par exemple, sur un document collaboratif *Word*, un déplacement d'un bout de texte va être traduit en une opération dite composée, qui contient une opération de suppression de texte et une opération d'insertion de texte. Par contre traduire cette opération composée en commande haut niveau peut se faire de deux façons. On peut reconstituer une commande de déplacement, ou considérer plus simplement les opérations une par une et transmettre une commande de suppression et une commande d'insertion. Ce choix est loin d'être anodin. Dans certaines situations, on doit obligatoirement reconstituer la commande car l'application ne sait pas la décomposer en deux autres commandes. Par exemple, ajouter un commentaire dans un document *Word* conduit dans la couche OT à une opération composée qui d'une part met en valeur le texte auquel se réfère le commentaire, d'autre part insère un élément "commentaire" dans la liste. Bien qu'il existe une commande de l'application pour mettre en valeur le texte uniquement, il n'en existe pas pour simplement insérer un commentaire, sans mise en valeur. Ici, on ne peut que reconstituer. A l'opposé, une commande de type "cherche et remplace" crée une opération composée qui contient une série de suppressions et une série d'insertions. Ici, il ne faut pas reconstituer la commande une fois les opérations transformées. Cela aurait pour effet d'étendre la commande "cherche et remplace" aux commandes concurrentes intégrées avant celle-ci, et conduirait à une divergence des copies. La figure 3.11 illustre cette situation. Un premier site cherche les caractères "a" et les remplace par des caractères "b", tandis qu'un deuxième insère le texte "aaa" dans le document. La commande "cherche et remplace" est traduite en opérations de suppression de "a" et d'insertion de "b", la commande d'insertion en opération d'insertion. Transformer ensemble des insertions et des suppressions revient à mettre à jour la position de l'opération transformée : c'est bien "aaa" qui est inséré sur le premier site. Si donc on reconstitue la commande "cherche et remplace", le premier site exécute d'abord cette commande, puis l'insertion de "aaa", tandis que le deuxième insère "aaa" puis remplace tous les "a" par des "b" : au final il a inséré "bbb", les états ne sont pas les mêmes.

Ces considérations rendent difficile et peu intuitive la reconstitution des opérations de l'application à partir des opérations de la couche OT. Cette approche ne peut pas garantir que les intentions des utilisateurs seront respectées.

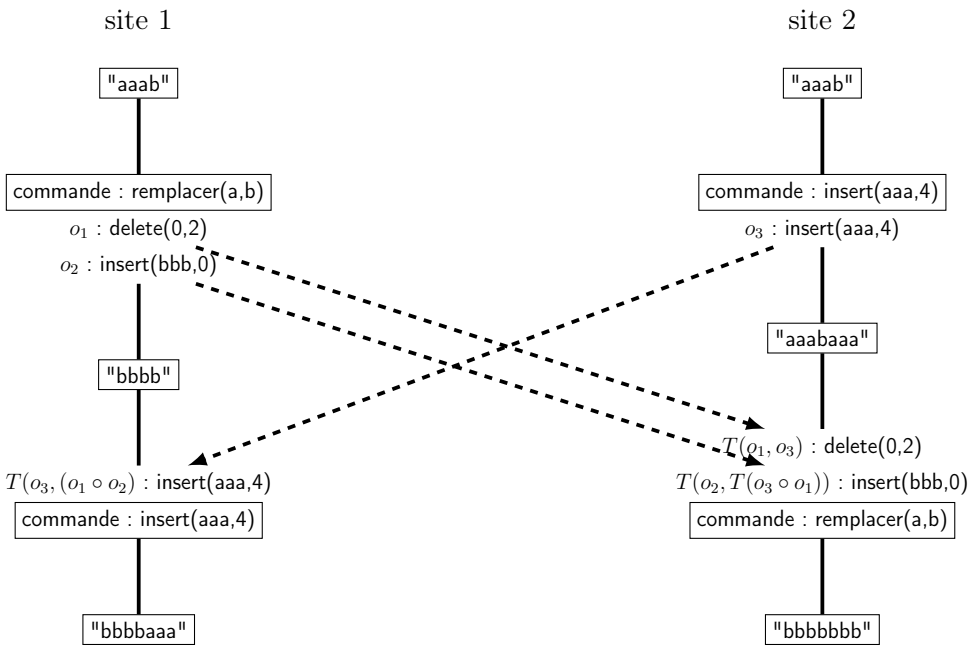


FIGURE 3.11 – Scénario Transparent Adaptation avec recomposition de la commande

3.1.3 Résumé de l'approche OT

L'approche OT permet l'édition en concurrence de structures de données diverses grâce à des algorithmes de contrôle, qui ne dépendent pas de la structure éditée, et d'opérations et de transformées. Les opérations et les transformées sont conçues pour éditer de la manière la plus simple la structure de données, et ne prennent en compte que des considérations d'ordre syntaxique. Pour l'édition collaborative de documents texte, le sens du document est important, et les approches actuelles présentent des limites en terme de respect de la sémantique des opérations de haut niveau. De plus, cette approche est relativement coûteuse, chaque édition en concurrence provoquant des conflits et nécessitant des traitements lourds à effectuer.

Nous allons voir dans le chapitre suivant que l'approche OT peut potentiellement gérer n'importe quels type d'opérations et de structures, pour peu qu'on soit prêt à concevoir un ensemble d'opérations et de transformées complexe.

3.2 Approche Commutative Replicated Data Types

La suite de ce chapitre présente l'approche Commutative Replicated Data Types, ou CRDT, qui base ses éditions sur des positions relatives et non absolues. Le nombre de conflits est réduit et l'intégration d'opérations distantes s'en trouve simplifiée.

3.2.1 Principes de base

L'approche CRDT offre une alternative à l'approche OT. Le principe de l'approche CRDT est de concevoir des structures de données que l'on peut modifier en concurrence sans avoir besoin de résoudre des conflits, grâce à une intégration des opérations concurrentes commutative. L'ordre dans lequel les opérations concurrentes sont intégrées n'a donc plus d'importance. L'exécution de deux opérations concurrentes donne le même résultat quelle que soit la séquence d'exécution choisie, et sans avoir besoin de modifier l'une ou l'autre opération. Les CRDT ne se limitent pas à l'édition de texte, et l'on peut trouver par exemple dans la littérature des CRDT pour des compteurs ou des logs [Shapiro *et al.*, 2011, Wuu and Bernstein, 1984]. Dans notre étude, nous ne considérerons que les CDRTs conçus pour éditer un document texte.

Les CRDT qui permettent l'édition de document texte suivent globalement tous le même principe. Le CRDT est un ensemble d'éléments, muni des opérations d'ajout d'un élément qui n'est pas dans l'ensemble et de suppression d'un élément qui se trouve dans l'ensemble. Les éléments représentent des parties atomiques du document (typiquement, caractères ou lignes), et leur ensemble doit être totalement ordonné et dense, afin de permettre d'une part la création du document linéaire en fonction des éléments de l'ensemble, et d'autre part d'insérer du texte à n'importe quel endroit du document.

Définition : Ordre Dense Soit un ensemble E muni d'une relation d'ordre $<$, on dit que l'ordre $<$ est dense dans E si et seulement si pour tout couple (x, y) d'éléments de E tel que $x < y$, il existe un élément z tel que $x < z < y$.

La suite de cette section présente les CRDTs pour l'édition de texte connus dans la littérature.

3.2.2 Algorithmes

WOOT

Le premier CRDT paru dans la littérature pour l'édition de documents texte est WOOT [Oster *et al.*, 2006b]. Les éléments qui le constituent sont des caractères reliés entre eux par deux relations nommées *Previous* et *Next*. Un document vide contient initialement deux éléments vides qui représentent le début et la fin du document. Insérer un caractère se fait en insérant un élément entre deux éléments du modèle. Le nouvel élément *el* contient le caractère inséré, $Previous(el)$ correspond à l'élément situé avant *el* et $Next(el)$ à l'élément d'après. La figure 3.12 représente un document WOOT édité par un seul site. Les caractères "a", "b" et "c" ont été insérés les uns

après les autres à la fin du document, puis le caractère "X" a été inséré entre "a" et "b". Les flèches représentent les relations *Previous* et *Next*. Ici le document contient le texte "aXbc".

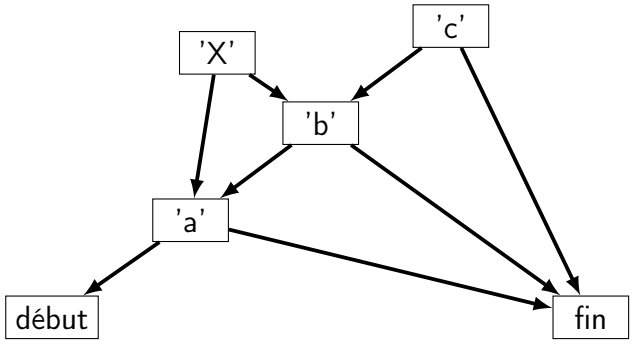


FIGURE 3.12 – Exemple de document WOOT

Lorsqu'un élément créé sur un site distant est reçu, les éléments correspondant à son *Previous* et à son *Next* sont cherchés dans le modèle local, et l'élément est inséré entre les deux. Il n'est pas possible de supprimer définitivement un élément du modèle. Dans l'exemple précédent, si on retire les éléments correspondant aux caractères "a" et "b", "X" ne peut plus être placé dans le document. Au lieu de le retirer du modèle, une suppression d'un élément le rend invisible, mais il reste présent dans le modèle. Seules les données (le caractère) sont supprimées, les métadonnées restent. On parle alors de pierre tombale.

L'ordre qui découle des relations *Previous* et *Next* n'est pas un ordre total, et ne permet que d'ordonner partiellement les éléments : si deux sites insèrent en concurrence un caractère au même endroit, les deux éléments qui vont être créés auront les mêmes *Previous* et *Next*, et seront incomparables. Pour ordonner totalement les éléments, WOOT va linéariser son modèle de manière unique sur tous les sites. Chaque site i possède un numéro de site unique s_i et maintient un compteur local c_i, qui correspond au nombre d'éléments générés par le site. Chaque élément généré par un site contient ce couple (s_i,c_i) et peut être identifié comme étant l'élément numéro c_i du site s_i. Deux couples peuvent être comparés : un couple (s_1,c_1) est plus petit qu'un couple (s_2,c_2) si et seulement si s_1 est plus petit strictement que s_2 ou s_1 et s_2 sont égaux et c_1 est strictement plus petit que c_2. Si lors de l'insertion d'un élément, il y a d'autres éléments dans le modèle entre ses éléments *Previous* et *Next*, on applique l'algorithme suivant. Premièrement, on considère les éléments entre les *Previous* et *Next* de l'élément à insérer qui n'ont pas leurs *Previous* et *Next* entre ceux de l'élément à insérer (ces éléments sont ordonnés suivant leurs couples). Deuxièmement on compare les couples de ces éléments et de l'élément à insérer pour trouver entre lesquels deux de ces éléments il se situe. Troisièmement, on recommence à partir de la première étape avec ces deux nouveaux éléments en guise de *Previous* et *Next*, jusqu'à ce qu'aucun élément ne se trouve déjà entre eux deux. On peut alors insérer l'élément.

L'exemple le plus simple pour comprendre cet algorithme est lorsque deux sites insèrent en concurrence un élément dans un document vide. Ils vont tous deux insérer leur élément entre les

bornes de début et de fin de document, comme illustré figure 3.13. Les caractères “a” et “b” ont les mêmes *Previous* et *Next*. Lorsque par exemple “b” est reçu sur un site où “a” a déjà été intégré, l'algorithme détecte que “b” ne peut pas directement être inséré entre les bornes de début et de fin, car “a” se trouve également entre ces bornes. Il compare donc les couples de “a” et de “b”, et insère “b” après “a” si le couple de “a” est plus petit que celui de “b”, ou “b” avant “a” dans le cas contraire.

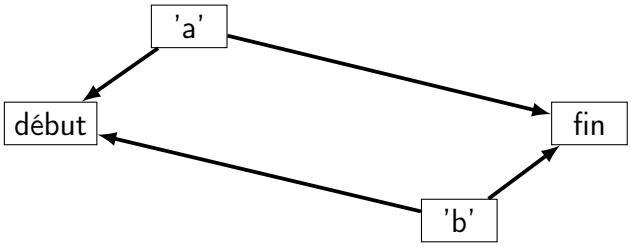


FIGURE 3.13 – Insertions concurrentes simples avec WOOT

La figure 3.14 présente un exemple plus complexe. Dans cette figure, les trois caractères “a”, “b” et “c” on été insérés en concurrence dans un document vide. Le couple associé aux éléments est également représenté sur la figure. On remarque que les éléments correspondant à ces trois caractères, qui ont les mêmes *Previous* et les mêmes *Next*, ont bien été ordonnés en fonction de leurs couples. Puis les caractères “X” et “Y” on été ajoutés par le site 2, entre “a” et “b” et “b” et “c” respectivement. Le scénario illustré est l'intégration d'un nouvel élément “Z” par le site 1 situé entre “a” et “c”, en concurrence de l'insertion de “b”, “X” et “Y”. L'algorithme positionne d'abord “Z” par rapport à “b”, qui est le seul élément entre “a” et “c” qui a ses *Previous* et *Next* en dehors de “a” et “c”. Le couple associé à “Z” est plus petit que celui associé à “b”, “Z” sera donc placé entre “a” et “b”. Ensuite, “Z” est positionné par rapport à “X”. Le couple associé à “X” est plus grand que celui associé à “Z”, “Z” sera donc placé entre “a” et “X”. S'il y avait d'autres éléments entre “a” et “X”, l'algorithme aurait continué de la même manière, dans notre cas il termine, et donc au final “Z” est placé entre “a” et “X” et le texte complet est “aZXbYc”.

WOOT repose sur un principe plutôt simple avec les relations *Previous* et *Next*, et des éléments qui ne nécessitent pas beaucoup de mémoire. Ses faiblesses résident dans les pierres tombales, qui font que le document ne peut que grandir, et dans son algorithme d'intégration en cas d'insertion d'éléments au même endroit, qui nécessite un traitement supplémentaire. Une optimisation de WOOT est présentée dans [Weiss *et al.*, 2007], et améliore le processus d'intégration des opérations.

RGA

L'algorithme RGA [Roh *et al.*, 2011] (pour Replicated Growable Array) garde l'idée d'un ordre généré par chaînage. Le modèle de cette approche est une liste chaînée d'éléments. Lors

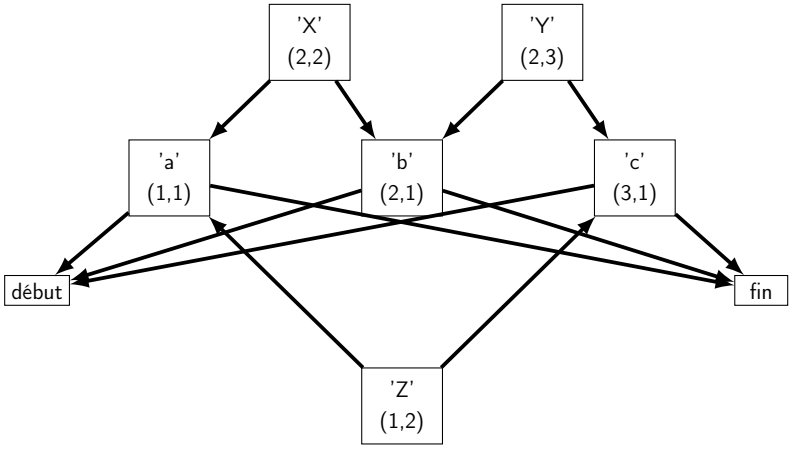


FIGURE 3.14 – Insertions concurrentes complexes avec WOOT

de l'insertion d'un élément, on transmet également l'élément du modèle qui le précède, afin de pouvoir placer l'élément dans la liste.

Pour résoudre le problème posé par l'insertion en concurrence de deux éléments au même endroit, chaque site maintient un vecteur d'état, qui possède une entrée par site et dont la valeur de chaque entrée correspond au nombre d'opérations d'un site qui ont été vues (reçues d'un site distant ou générées localement). Ce vecteur est mis à jour lors de la génération d'une opération locale ou lors de l'intégration d'une opération distante. Chaque élément généré est étiqueté de la somme des entrées du vecteur d'état local, ainsi que du numéro unique qui correspond au site qui a généré l'élément. On peut comparer les étiquettes de deux éléments en comparant d'abord la somme des entrées, puis le numéro de site en cas d'égalité.

Quand plusieurs éléments sont insérés en concurrence après le même élément, ils sont ordonnés selon leurs étiquettes, en insérant les plus grandes d'abord et les plus petites ensuite. En pratique, pour insérer un élément, on remonte la liste chaînée à partir de l'élément qui le précédait localement, jusqu'à trouver un élément qui possède une étiquette plus petite que celle de l'élément à insérer. On l'insère alors avant cet élément. On remarque que l'ordre induit par les étiquettes est compatible avec l'ordre causal : si un élément a été inséré plus tôt qu'un autre, son étiquette sera plus petite. Ainsi, un élément généré a une étiquette plus grande que celle de ses deux voisins locaux, et donc en remontant la chaîne jusqu'à trouver une étiquette plus petite, on est assuré de ne pas insérer l'élément après son voisin de droite lors de la génération.

La figure 3.15 présente un exemple d'édition à laquelle participent deux sites. La première ligne de figure représente une structure de données RGA qui contient quatre éléments, plus un cinquième qui est en train d'être inséré. L'élément e_1 a été le premier inséré, par le site 2, avec pour vecteur d'états <0,1> et donc pour étiquette <1,2>. Le site 2 a ensuite inséré l'élément e_2 après e_1, de vecteur d'états <0,2> et d'étiquette <2,2>. Il insère ensuite e_3, d'étiquette <3,2>, entre e_1 et e_2, c'est-à-dire après e_1. Comme l'élément qui se situe après e_1, c'est-à-dire e_2, a une

étiquette plus petite que e_3, e_3 est inséré avant e_2. De la même manière, l'élément e_4 est inséré entre e_3 et e_2. L'élément e_5 a été ajouté par le site 1 en concurrence de e_3 et e_4, entre e_1 et e_2.

Les lignes suivantes de la figure illustrent la remontée de la chaîne de l'élément e_5 à partir de e_1, jusqu'à ce que son successeur possède une étiquette plus petite que la sienne.

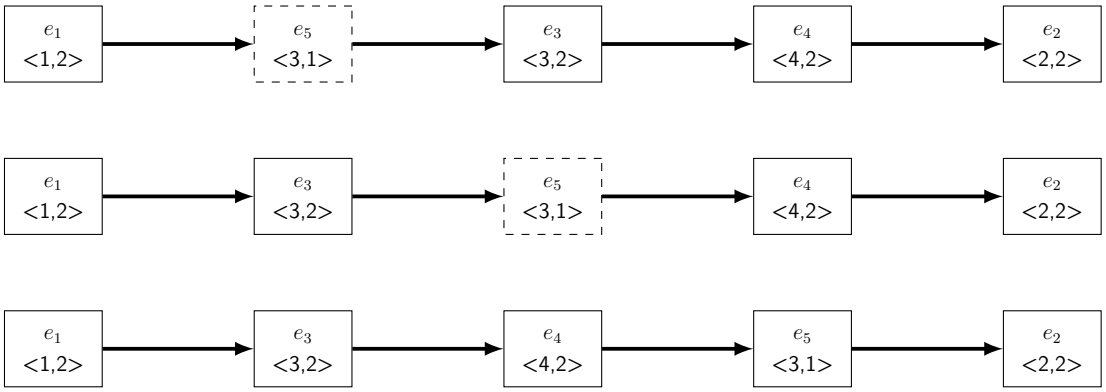


FIGURE 3.15 – Scénario d'intégration RGA

RGA peut être considéré comme une évolution de WOOT. Chaque élément n'a qu'un chaînage au lieu de deux, son algorithme d'intégration des opérations concurrentes est plus simple et moins coûteux que celui de WOOT. Par contre, tout comme WOOT, le fait de baser l'ordre des éléments sur un chaînage implique la nécessité de fonctionner avec des pierres tombales, afin d'être toujours assuré de trouver l'élément qui précède un élément à insérer.

Logoot

L'approche LOGOOT [Weiss *et al.*, 2009, Weiss *et al.*, 2010] est une alternative à ces deux approches basées sur un chaînage et un ordre qui s'obtient par construction. Elle propose d'associer à chaque élément un identifiant unique, et de munir l'ensemble des identifiants d'un ordre total. Les pierres tombales ne sont donc plus utiles, l'intégration d'un élément se réduisant à chercher dans le modèle la place qui lui revient en comparant directement son identifiant aux identifiants des éléments déjà présents dans le modèle.

Le point central de cet algorithme est donc la génération des identifiants, qui doivent provenir d'un ensemble totalement ordonné et dense. De plus, chaque identifiant ne doit être généré qu'une seule fois au plus pendant toute la durée de vie d'un document, pour éviter qu'un identifiant soit présent deux fois dans le modèle à cause d'insertions en concurrence de différents sites, ou à cause d'une réinsertion d'un identifiant supprimé sur un site où la suppression n'a pas encore été reçue.

Les identifiants proposés dans [Weiss *et al.*, 2010] sont des listes de triplets d'entiers. Les entiers des triplets correspondent, dans l'ordre, à une position, un numéro unique de site, et à la valeur d'un compteur local. Pour comparer deux triplets, on compare d'abord leurs positions,

puis leurs numéros de site si leurs positions sont égales, puis leurs compteurs si leurs numéros de site sont égaux.

On compare deux identifiants, c'est-à-dire deux listes de triplets, de manière lexicographique : on compare leurs premiers triplets, si ils sont égaux on compare les deuxièmes triplets, et ainsi de suite jusqu'à ce que deux triplets soient différents où qu'une liste soit totalement parcourue. Dans le premier cas, le résultat de la dernière comparaison des triplets donne celui de la comparaison des listes. Dans le deuxième cas, la liste totalement parcourue est la plus petite (c'est un préfixe de la deuxième liste). La figure 3.16 présente un exemple de document LOGOOT, dans lequel chaque ligne possède un identifiant. Les lignes du document sont ordonnées selon leurs identifiants.

identifiants	lignes
<10,1,1>	Liste de courses :
<20,1,3>	Lait
<20,1,3>,<5,2,2>	Beurre
<20,2,1>	Chocolat
<30,1,2>	Fruits

FIGURE 3.16 – Exemple de document LOGOOT

Lorsqu'un élément est généré localement sur un site entre deux autres éléments, son identifiant doit être compris entre les identifiants de ses deux voisins. Pour générer un identifiant approprié, on le construit itérativement à partir d'une liste vide, en comparant les positions des triplets des identifiants de ses voisins un par un. Si les positions sont égales ou diffèrent de un, elles sont trop proches pour générer une position entre elles. Dans ce cas on ajoute le triplet de l'identifiant du voisin de gauche dans la liste, et on passe au triplet suivant (si la liste arrive à son terme, on considère un triplet nul). Dans le cas contraire, on génère un triplet qui a pour position un entier aléatoire entre les deux positions des triplets des identifiants voisins, pour numéro unique de site le numéro du site qui est en train de le générer, et pour dernière valeur la valeur du compteur d'insertions local, ce qui termine l'algorithme. Ainsi, l'identifiant généré est bien compris entre ceux de ses voisins, et grâce au dernier triplet, qui contient le numéro de site s_i et la valeur du compteur c_i, l'identifiant est bien unique : il s'agit de l'identifiant numéro c_i généré par le site s_i.

Si on reprend l'exemple de la figure 3.16, le premier identifiant inséré peut être <10,1,1>. La position 10 a été choisie aléatoirement (l'élément n'avait pas de voisins), les deux autres entiers nous indiquent qu'il s'agit du premier élément inséré par le site 1. Le site 1 a ensuite inséré l'identifiant <30,1,2> après <10,1,1>, puis <20,1,3> entre ces deux derniers. En parallèle le site 2 a inséré une ligne au même endroit, et par hasard l'identifiant associé commence également par un triplet dont la position est 20 (toute valeur entre 10 et 30 était possible). Dans ce cas où les positions sont égales, le numéro de site est utilisé pour déterminer quelle ligne est après l'autre, et donc la ligne du site 2 est placée après la ligne du site 1. Enfin, le site 2 a inséré un identifiant

entre <20,1,3> et <20,2,1>. On ne peut pas trouver d'entier entre 20 et 20, donc <20,1,3> sera un préfixe de l'identifiant généré. L'identifiant <20,1,3>,<5,2,2> convient. La position 5 est là aussi choisie aléatoirement, et il s'agit du deuxième élément inséré par le site 2.

Le principal défaut de Logoot est que ses identifiants grossissent de plus en plus si beaucoup d'insertions ont lieu dans la même région du document. Une façon de remédier à ce problème est d'adopter différentes stratégies lors de la génération de la position du dernier triplet de l'identifiant, et de ne pas le choisir aléatoirement. Par exemple, [Weiss *et al.*, 2010] propose de privilégier les petites positions, ce qui est plus efficace lorsque les éditions ont lieu les unes après les autres, de gauche à droite, au même endroit, ce qui est la manière logique de créer un document. Par contre, cette technique est moins efficace lorsque le document est modifié de manière plus erratique, lorsque l'orthographe est vérifiée par exemple. L'approche développée dans [Nédelec *et al.*, 2013] va plus loin, et change dynamiquement de stratégie en fonction de la taille des identifiants. Si une stratégie est mauvaise, elle fait grossir rapidement les identifiants, donc la stratégie change rapidement. Si une stratégie est bonne, la taille des identifiants reste stable, donc la stratégie ne change pas.

Treedoc

Treedoc [Preguica *et al.*, 2009] propose une réponse au problème des identifiants volumineux de Logoot. Son modèle est un arbre binaire, et l'identifiant d'un élément est son chemin depuis la racine de l'arbre. Pour ordonner le document, l'arbre est parcouru de manière infixe : le contenu du sous-arbre gauche (le fils numéro 0) d'un nœud se situe à gauche du contenu du nœud, et le contenu du sous-arbre droit (le fils numéro 1) se situe à droite. Ainsi, lorsqu'un élément est inséré après (respectivement avant) un élément ayant pour identifiant 110, il obtient l'identifiant 1101 (respectivement 1100). Comme dit plus haut, ces identifiants correspondent à un chemin dans l'arbre, et ne sont donc pas stockés dans le modèle, contrairement à Logoot. Quelle que soit la taille de l'identifiant, une insertion conduit à la création d'un nœud, qui prend une place mémoire fixe. Il n'y a pas de phénomène d'augmentation de la taille des identifiants au fur et à mesure des éditions.

Les seuls identifiants ne sont pas suffisants pour travailler en collaboration. Plusieurs utilisateurs peuvent vouloir insérer en concurrence un élément au même endroit. Cela conduit à la création de deux nœuds qui ont le même chemin. L'algorithme crée dans ce cas deux (ou plus) mini-nœuds dans le nœud, chacun contenant un élément. Ces mini-nœuds se distinguent au moyen d'un désambiguateur, qui est un moyen unique de les différencier et qui peut correspondre au maintenant classique couple construit à partir du numéro unique de site et de la valeur d'un compteur local. Les mini-nœuds et leurs sous-arbres sont ordonnés dans le nœud en fonction de leur désambiguateur, les fils gauche et droite du nœud qui les contient restant aux extrémités. Ces désambiguateurs font partie du chemin si nécessaire. La dernière position d'un chemin contient toujours un désambiguateur, tandis que les chemins intermédiaires n'en contiennent que si l'on souhaite insérer un élément entre deux mini-nœuds.

La figure 3.17 présente un exemple de document Treedoc. Un premier nœud contient les mini-nœuds a et b, qui ont pour désambiguateurs respectivement d_a et d_b. Le chemin pour accéder au nœud mini-nœud a est (d_a), celui pour accéder au mini-nœud b est (d_b), celui pour accéder au nœud racine est (). A gauche du nœud racine ont été insérés les mini-nœuds c et d, les chemins pour y accéder sont (0,d_c) et (0,d_d). De même, à droite de la racine, on peut trouver le mini-nœud e, de chemin (1,d_e). Enfin, deux derniers mini-nœuds f et g se situent à droite du mini-nœud a, ils sont atteignables par les chemins (d_a,1,d_f) et (d_a,1,d_g). L'ordre des mini-nœuds de cet exemple est c,d,a,f,g,b,e.

L'approche Treedoc possède quelques pierres tombales. On ne peut en effet pas supprimer du modèle un nœud qui contient des fils visibles. Dans le cas contraire (si le nœud est une feuille ou si tous ses fils sont des pierres tombales), on peut le retirer du modèle. Toutefois, si une opération concurrente insère un élément sur le chemin maintenant supprimé, les pierres tombales nécessaires sont recréées.

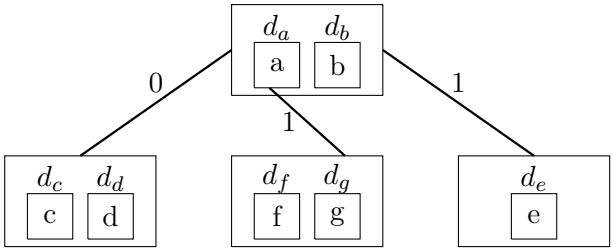


Figure 3.17 – Exemple de document Treedoc

3.2.3 Synthèse de l'approche CRDT

Bien que plus simples en terme d'intégration des opérations distantes que les approches OT, les CRDT pour l'édition collaborative en temps réel ont un besoin de métadonnées de plus en plus volumineuses au fur et à mesure que l'édition du document se poursuit. Les approches Woot et RGA ont besoin de plus en plus de pierres tombales, Logoot a besoin d'identifiants de plus en plus grands.

Ceci fait directement écho au problème de la granularité des éléments présenté en introduction : si on choisi de travailler avec des éléments de grande taille qui contiennent plus de données, pour limiter la quantité de métadonnées, on ne peut pas modifier une partie de ces données sans modifier tout l'élément qui la contient, c'est à dire sans le supprimer puis le réinsérer, l'opération de mise à jour n'étant pas disponible dans les CRDT. La conséquence directe de ce problème est la duplication d'éléments en cas de mise à jour concurrente, alors qu'il est souhaitable de n'avoir qu'un élément modifié deux fois. L'approche OT n'a pas ce problème, des jeux d'opérations comme ceux présentés en [Sun *et al.*, 1998] étant capables de gérer efficacement des chaînes de caractères.

Les CRDTs actuels pour l'édition collaborative de documents texte sont construits à partir

d'ensembles et d'opérations d'ajout ou de suppression d'éléments de cet ensemble. En conséquence, les seules intentions des utilisateurs qui peuvent être préservées sont des insertions et des suppressions.

3.3 Synthèse de l'état de l'art

Deux approches de réplication optimiste sont disponibles dans la littérature pour l'édition collaborative en temps réél, l'approche OT et l'approche CRDT.

Dans ces deux approches, les intentions des utilisateurs qui sont conservées lors de conflits d'éditions dépendent des opérations conflictuelles qui ont modifié le document. Pour préserver plus d'intentions des utilisateurs, il faut augmenter la précision d'édition, en proposant plus d'opérations pour éditer les modèles de données. Pour l'approche OT, la difficulté est de concevoir les transformées associées aux opérations. Pour l'approche CRDT, le défi vient de la condition de commutativité des opérations concurrentes.

Les deux chapitres suivants présentent respectivement nos approches OT et CRDT pour respecter plus d'intentions des utilisateurs, grâce à des ensembles d'opérations étendus.

4

Une approche OT pour les documents hiérarchisés

Comme expliqué dans l'état de l'art, les approches OT actuelles ne respectent pas les intentions des utilisateurs car l'ensemble des opérations qu'elles possèdent représente généralement l'ensemble minimal des opérations dont on a besoin pour éditer la structure de données considérée, alors que les les intentions des utilisateurs sont plus complexes et ne sont pas contenues dans cet ensemble minimal.

Notre proposition est donc de concevoir des opérations non pas à partir de la structure et en cherchant comment elle peut être éditée, mais à partir des utilisateurs et en cherchant quelles peuvent être leurs intentions lorsqu'ils modifient les données. Nos premiers travaux à ce sujet peuvent être trouvés en [André *et al.*, 2012].

Nous considérons un document texte, que les utilisateurs voudront éventuellement organiser en paragraphes, et styliser en ajoutant par exemple du texte en gras ou en italique. Les intentions des utilisateurs que nous avons déterminées pour ce type de document sont décrites ci-après.

Tout d'abord, un utilisateur peut avoir l'intention d'ajouter ou de supprimer un caractère à une position donnée. Il peut vouloir ajouter un nouveau paragraphe. Cette intention est différente d'un ajout de caractère de retour à la ligne. Un utilisateur peut vouloir déplacer un paragraphe, pour réorganiser le document. Dans le même état d'esprit, il est possible de séparer un paragraphe en deux, et de fusionner deux paragraphes. Enfin, il est possible de styliser une partie du document.

La suite de ce chapitre présente le modèle que nous proposons pour représenter le document, ainsi que les opérations qui lui sont associées. Puis, l'ensemble des transformées est expliqué dans une deuxième section. La troisième section présente l'intégration de notre algorithme pour l'édition collaborative en temps réel de wikis, et la dernière section fait le bilan de cette contribution.

4.1 Modèle et Opérations

4.1.1 Modèle de données

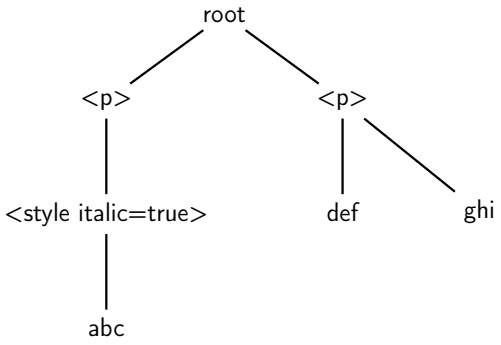


FIGURE 4.1 – Exemple de modèle de document

Représenter nos données sous forme de structure linéaire pose une difficulté. En effet, les styles et paragraphes englobent certaines régions du texte, qui dans une structure linéaire seraient représentés par des caractères de début et fin de région. Mettre à jour correctement ces deux caractères dans les opérations et les transformées, tout en garantissant que le document reste bien formé, n'est pas chose aisée. Nous avons donc adopté un modèle arborescent contenant des nœuds paragraphes et styles, ce qui permet de délimiter plus facilement les différentes régions du document.

Dans notre modèle, les paragraphes sont les fils de la racine de l'arbre. Ils peuvent avoir plusieurs fils : des feuilles qui contiennent du texte, et des nœuds style. Les nœuds styles ont pour fils une unique feuille texte. Par exemple, la figure 4.1 est un modèle qui contient deux paragraphes dont un en italique.

Les différentes zones de notre document sont clairement identifiables. Une zone de texte est en italique si son nœud père est un nœud style de type italique. Les paragraphes sont délimités grâce aux nœuds paragraphes de l'arbre. Certaines opérations sont plus faciles à réaliser que dans un modèle linéaire, notamment l'opération de déplacement d'un paragraphe qui consiste simplement à déplacer un nœud de l'arbre.

4.1.2 Opérations

L'opération *InsertText(pos,path,char,siteId)* sert à insérer un caractère dans une feuille de l'arbre. Elle a pour paramètres la position *pos* à laquelle le caractère est inséré dans le texte de la feuille, le chemin *path*[4] de la feuille depuis la racine, le caractère *char* à ajouter, et le numéro unique de site *siteId* qui sert à définir qui a la priorité en cas d'insertion concurrente au même

4. les chemins seront représentés par des listes d'indices, ainsi [1,0] représente le premier fils du deuxième fils de la racine.

endroit, comme c'est souvent l'usage. Lorsque cette fonction est appelée, le caractère est ajouté à l'endroit spécifié.

L'opération qui fait le contraire de l'insertion est *DeleteText(pos,path)*, qui supprime un caractère du modèle. Elle a pour paramètres la position *pos* à laquelle se trouve le caractère à supprimer dans la feuille considérée, et le chemin *path* qui mène depuis la racine à la feuille en question. Un appel à cette fonction supprime du modèle le caractère désigné.

La fonction *NewParagraph(pos,siteId)* a pour but de créer un paragraphe supplémentaire dans le document. Elle prend en paramètre la position *pos* souhaitée de ce paragraphe parmi les paragraphes déjà existants, et un numéro unique de site *siteId* pour là encore définir une priorité en cas d'insertion concurrente. Cette fonction, en plus de créer lors de son appel un nouveau nœud paragraphe, crée également immédiatement une feuille en dessous de ce nouveau paragraphe. Créer un paragraphe ne rajoute pas de texte au modèle, pour écrire dans un nouveau paragraphe récemment inséré, il faut utiliser un appel à *InsertText(0,[pos,0],...)* après avoir créé le paragraphe, où *pos* est la position donnée lors de sa création.

La fonction *MoveParagraph(origin,dest,siteId)* permet de déplacer un sous-arbre paragraphe à un autre emplacement. Elle utilise pour paramètres l'indice de la position initiale *origin* du paragraphe parmi les fils de la racine, et la nouvelle position *dest*. Le numéro unique de site *SiteId* est une fois de plus nécessaire, si deux paragraphes sont déplacés en concurrence au même endroit ou bien si un même paragraphe est déplacé en concurrence à deux endroits différents. Lors de l'appel de cette fonction, l'indice *dest* est calculé par rapport au modèle avec le paragraphe à sa position d'origine. Par exemple, sur un modèle à cinq paragraphes p_0,p_1,p_2,p_3,p_4 , *MoveParagraph(1,3,...)* déplace le paragraphe p_1 entre p_2 et p_3, qui est bien la position d'indice 3 par rapport à l'état initial du modèle, mais qui après déplacement correspond donc à la position 2, puisque les paragraphes ont maintenant p_0,p_2,p_1,p_3,p_4 pour ordre. Les appels *MoveParagraph(x,x,...)* et *MoveParagraph(x,x+1,...)*, où x est un indice quelconque de paragraphe, n'ont donc aucun effet.

La fonction *MergeParagraph(pos,leftChNb)* permet de fusionner deux paragraphes adjacents en un seul. Ses paramètres sont l'indice *pos* du paragraphe de droite de la fusion, et le nombre de fils *leftChNb* du paragraphe de gauche avant la fusion. A la différence du premier paramètre, le deuxième ne sert pas pour l'exécution de la méthode mais est nécessaire pour transformer les opérations. Nous y reviendrons dans la section dédiée aux transformées. Cette fonction déplace les fils du paragraphe de droite à la suite des fils du paragraphe de gauche.

La fonction *SplitParagraph(pos,path,split))* scinde un paragraphe en deux. Elle a besoin des paramètres *pos* et *path*, respectivement la position dans une feuille et le chemin depuis la racine vers cette feuille, qui permettent d'identifier à quel endroit du texte du paragraphe la césure aura lieu. Si cette dernière se situe entre deux feuilles, *pos* est mis à 0 et *path* référence la feuille de droite. La fonction crée et insère un nouveau paragraphe après le paragraphe à séparer, et déplace les fils du paragraphe initial situés après la césure dans le nouveau paragraphe. Dans

ce cas, un troisième paramètre booléen *split* est mis à faux. Il signifie que les fils ont seulement été déplacés, sans être eux-mêmes séparés en deux, et son utilité est liée aux transformées. Dans l'autre cas, la césure se trouve au milieu d'une feuille texte. La feuille est donc coupée en deux, si elle a un père style ce nœud est copié à l'identique et sert de père à la deuxième partie de la feuille pour que chacune ait son propre père, puis comme dans le premier cas les fils du paragraphe à droite de la césure sont déplacés dans le nouveau paragraphe. Le paramètre *split* est ici mis à vrai : une feuille a été séparée en deux.

Enfin, la fonction *Style(start,end,path,param,value,siteId,splitStart,splitEnd)* ajoute ou supprime du style dans une feuille texte. Les paramètres *start* et *end* délimitent la région de la feuille à considérer, le paramètre *path* indique la feuille elle même. Le style est représenté par un attribut *param* qui possède une valeur *value*, qui sont donc deux paramètres supplémentaires. Par exemple, pour mettre en gras une partie de texte, on peut mettre l'attribut *bold* à *true*, puis pour supprimer le style il faut le mettre à *false*. Les nœuds style peuvent avoir plusieurs attributs, pour par exemple mettre du texte en gras et en italique, et peuvent représenter des styles plus complexes, comme des url (par exemple, *param*=url,*value*="http ://loria.fr"). Le numéro unique de site *siteId* est une fois de plus nécessaire, en cas de modification concurrente d'un même attribut. Deux paramètres booléens *splitStart* et *splitEnd* sont ajoutés. Le premier est mis à vrai si le paramètre *start* est différent de 0 à la génération et à faux sinon. Le second est mis à vrai si le paramètre *end* est différent de la longueur du texte de la feuille à la génération et à faux sinon. Ces deux paramètres permettent de savoir si le style a initialement eu lieu sur le début de la chaîne de caractères, la fin, le milieu, ou la totalité. Cela aura son importance pour les transformées. Lors de l'appel de cette fonction, la feuille est séparée en deux, trois, ou laissée intacte suivant les valeurs de *start* et *end*, afin d'isoler la partie qui reçoit la modification. Si la feuille a déjà un nœud père style, ce père est répliqué pour chaque nouveau morceau de feuille (un nœud style a un unique fils). Sinon, un nœud père est créé entre la racine et le morceau de feuille qui nous intéresse. Puis l'attribut *param* du nœud style est créé si il n'existait pas déjà et prend la valeur *value*. La figure 4.2 présente un exemple de mise à jour d'un modèle après une opération de style issue du site numéro 1, qui met simplement en gras un caractère.

Comme un nœud de type style ne peut contenir qu'une seule feuille, et qu'aucune fonction ne cible ce type de nœud et cible toujours le fils contenant du texte qui lui est associé, on utilisera dans nos fonctions le chemin vers le nœud style à la place du chemin vers le texte quand le texte est stylisé. Cela voudra signifier que l'on cible le fils unique du style, ce qui permet d'avoir des chemins de taille deux au lieu de trois, et simplifie certaines transformées.

Le tableau 4.1 résume les différentes fonctions ainsi que leurs paramètres.

4.2 Transformées

L'ensemble des opérations étant maintenant identifié, nous allons définir les transformées associées à cet ensemble. Nous rajoutons d'abord les opérations *Id* et *Composite*, qui représentent

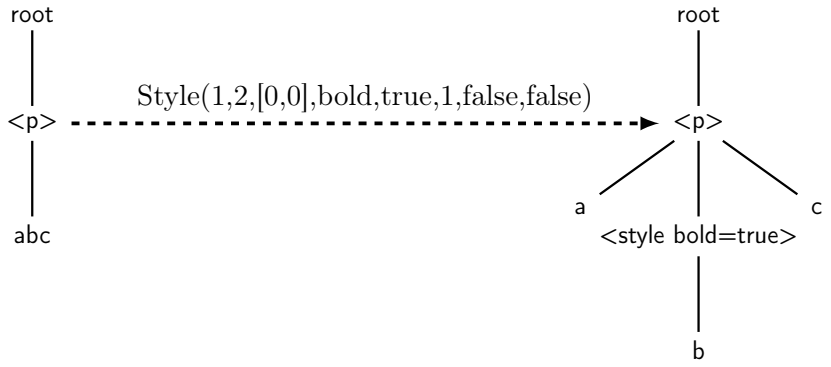


FIGURE 4.2 – Illustration de l'opération Style

InsertText	pos	position de l'insertion dans la feuille
	path	chemin vers la feuille
	char	caractère à insérer
	siteId	identifiant unique du site
DeleteText	pos	position de la suppression dans la feuille
	path	chemin vers la feuille
NewParagraph	pos	indice du nouveau paragraphe
	siteId	identifiant unique du site
MoveParagraph	origin	indice initial du paragraphe
	dest	indice final du paragraphe
	siteId	identifiant unique du site
MergeParagraph	pos	indice du paragraphe de droite
	leftChNb	nombre de fils du paragraphe de gauche
SplitParagraph	pos	position de la césure dans la feuille
	path	chemin vers la feuille
	split	vrai si la césure a scindé la feuille
Style	start	position de début du style dans la feuille
	end	position de fin du style dans la feuille
	path	chemin vers la feuille
	param	type du style
	value	valeur du style
	siteId	identifiant unique du site
	splitStart	vrai si le début de la feuille n'est pas stylisé
	splitEnd	vrai si la fin de la feuille n'est pas stylisée

TABLE 4.1 – tableau récapitulatif des opérations et de leurs paramètres

	L'opération modifie :		
	position des paragraphes	structure arborescente	contenu d'une feuille
InsertText			X
DeleteText			X
NewParagraph	X		
MoveParagraph	X		
MergeParagraph	X	X	
SplitParagraph	X	X	X
Style		X	X

TABLE 4.2 – classification des opérations

	position des paragraphes	structure arborescente	contenu d'une feuille
position des paragraphes	mise à jour de l'indice de paragraphe	rien	rien
structure arborescente	mise à jour de l'indice de paragraphe	mise à jour du chemin	rien
contenu d'une feuille	mise à jour de l'indice de paragraphe	mise à jour du chemin	décalage de position

TABLE 4.3 – résumé des effets des transformations

respectivement l'opération identité qui ne modifie donc en rien le modèle et ne réalise aucune action, et une liste d'opérations à exécuter les unes après les autres. Certaines transformées ne peuvent pas être exprimées sous la forme d'une seule opération, et leur forme d'exécution est composée de plusieurs opérations. Dans ce cas, une opération *Composite* est renvoyée. Dans certaines situations, les deux opérations concurrentes expriment exactement la même intention, qu'il est inutile d'intégrer deux fois au document. La transformée sera l'opération *Id*.

Avant de passer à l'explication détaillée des transformées, le tableau 4.2 classe nos opérations selon trois critères. Le premier critère est si l'opération modifie l'ordre des paragraphes, le second est si l'opération modifie la structure arborescente d'un ou plusieurs paragraphes, le troisième est si l'opération modifie le contenu d'une feuille texte. Le tableau 4.3 résume les transformations à appliquer à l'opération à transformer en fonction de cette classification. En colonnes sont listées les classes de l'opération à transformer, en lignes celles de l'opération par rapport à laquelle la transformation s'effectue.

Nous allons maintenant passer en revue chaque paire d'opérations (o_1, o_2) et chaque cas associé, et expliquer la transformée de o_1 par rapport à o_2. Elle sera notée $T(o_1, o_2)$. Les opérations de type *Id* et *Composite* ne seront pas considérées. Pour les algorithmes de contrôle, une opération *Id* est équivalente à aucune opération, et peut donc être ignorée ; une opération *Composite* est équivalente à la succession des opérations qui la compose, et est donc traitée comme une séquence

d'opérations. Dans les parties qui suivent, le type de l'opération o_2 est fixé, et les différentes possibilités pour o_1 sont examinées. Le pseudo-code complet des transformées est disponible en annexe du document.

4.2.1 Transformées par rapport à InsertText

Les trois opérations *NewParagraph*, *MergeParagraph* et *MoveParagraph* ne sont pas affectées par un ajout de texte (la réciproque est fausse mais nous en parlerons dans les parties appropriées). En effet, elles modifient uniquement la structure des paragraphes, structure qui n'a pas été modifiée par l'insertion. L'opération transformée $T(o_1, o_2)$ est donc o_1 dans ces trois cas.

Si l'opération o_1 est de type *InsertText*, o_2 n'a un impact que si les deux opérations ciblent la même feuille. Dans ce cas, la position *pos* de o_1 doit être mise à jour en fonction de celle de o_2, comme dans un algorithme OT classique qui agit sur un document texte linéaire. La transformée est donc une opération de même type et mêmes paramètres que o_1, à l'exception de la position.

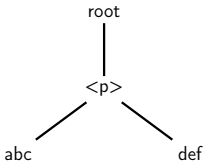
Si la position de o_2 est supérieure strictement à celle de o_1, o_1 n'a pas besoin d'être transformée. Si la position de o_2 est inférieure strictement à celle de o_1, la position de la transformée est celle de o_1 augmentée de 1. Si les deux positions sont les mêmes, on utilise les deux *siteId* pour déterminer quel caractère se place avant l'autre. Généralement, un identifiant plus petit est plus prioritaire, donc si o_1 a le plus petit identifiant elle n'est pas modifiée, sinon la position de la transformée est celle de o_1 augmentée de 1.

Le cas où o_1 est de type *DeleteText* est proche du cas de l'insertion. Si les deux chemins ne ciblent pas la même feuille, il n'y a rien a modifier. Sinon il faut modifier la position de o_1.

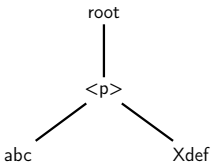
Si la position de o_2 est supérieure strictement à celle de o_1, o_1 n'a pas besoin d'être transformée. Si la position de o_2 est inférieure ou égale à celle de o_1, la position de la transformée est celle de o_1 augmentée de 1.

Lorsque o_1 est de type *SplitParagraph*, si les deux chemins sont différents o_1 n'est pas transformée. Si ils sont semblables la position de la césure doit être mise à jour. Intuitivement, si la césure est avant l'insertion, rien n'est à modifier, et si elle est après on augmente la position de la césure de 1. Reste le cas où les deux positions sont les mêmes. Deux choix sont possibles, soit on conserve le caractère dans la partie à gauche de la scission et on augmente la position de o_1 de 1, soit on le conserve dans la partie à droite et on ne fait rien. Le choix le plus cohérent est de placer le caractère à droite, comme l'illustre la figure 4.3. La figure 4.3a présente un modèle initial avec un seul paragraphe qui possède deux fils texte. Un utilisateur insère X au début du deuxième fils (4.3b) tandis qu'un second utilisateur casse le paragraphe au même endroit (4.3c). Les figures 4.3d et 4.3e donnent les deux états du modèle possibles une fois les deux opérations intégrées, le premier est celui où on a choisi de scinder avant X à partir de 4.3b, et le second après. De ces deux états, seul le premier est accessible facilement depuis l'état 4.3c où la césure a déjà eu lieu. Aucune opération ne nous permet de passer directement de l'état 4.3c à l'état 4.3e. On choisit donc de garder le caractère inséré dans la partie droite de la césure. Ce qui nous

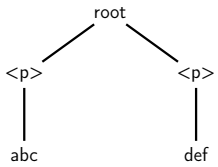
donne, dans le cas où les chemins sont les mêmes : si la position de o_2 est supérieure ou égale à celle de o_1, o_1 n'est pas transformée ; si la position de o_2 est inférieure strictement à celle de o_1, la position de la transformée est celle de o_1 augmentée de 1.



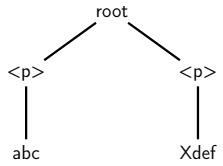
(a) Modèle initial



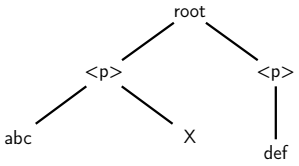
(b) Insertion de X



(c) Césure du paragraphe en début de 2e feuille



(d) Résolution : césure avant X

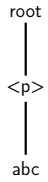


(e) Résolution : césure après X

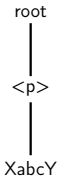
FIGURE 4.3 – Insertion et césure en début de feuille

Pour le cas où o_1 est de type *Style*, là encore l'insertion n'aura de l'influence sur la mise en place du style que si les deux opérations ciblent la même feuille, et si l'insertion se situe entre la borne de début et celle de fin du style. Un choix se pose lorsque l'insertion a lieu en bordure du style, la figure 4.4 illustre pourquoi il est plus simple d'appliquer le style à l'insertion dans ce cas à travers le cas particulier où le style porte sur toute une feuille : ne pas l'appliquer aux caractères insérés en concurrence reviendrait à créer deux fils sur le site qui a intégré le style, opération qui n'est pas disponible dans notre modèle. Nous avons donc, lorsque les chemins sont égaux :

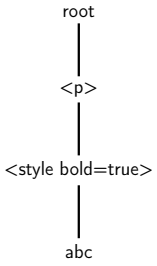
Si la position de o_2 est inférieure strictement à la borne de départ de o_1, cette borne est augmentée de 1 dans la transformée. Si la position de o_2 est inférieure ou égale à la borne de fin de o_1, cette borne est augmentée de 1 dans la transformée.



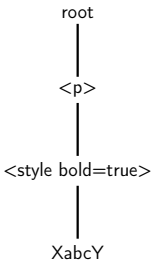
(a) Modèle initial



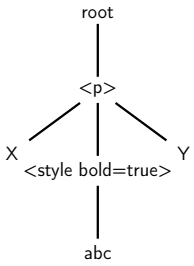
(b) Insertion de X et Y



(c) Ajout de style sur abc



(d) Résolution : X et Y inclus



(e) Résolution : X et Y exclus

FIGURE 4.4 – Insertion et style en bordure de feuille

4.2.2 Transformées par rapport à DeleteText

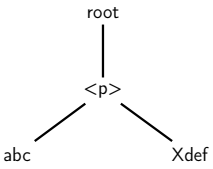
L'opération *DeleteText*, tout comme l'opération *InsertText*, ne modifie pas la structure arborescente et se contente de modifier le contenu des feuilles (une feuille vide n'est pas retirée de l'arbre). Si o_1 est de type *NewParagraph*, *MergeParagraph* ou *MoveParagraph*, $T(o_1, o_2) = o_1$, ces trois opérations n'ayant besoin d'aucune information sur le contenu des feuilles de l'arbre pour fonctionner. Pour tous les autre cas, les chemins des deux opérations doivent être différents pour qu'une transformation soit nécessaire, afin de mettre à jour les indices (des positions ou bornes) utilisés dans o_1.

Si o_1 est de type *InsertText*, la position de o_1 doit être diminuée de 1 quand la position de o_2 est inférieure strictement. Dans le cas contraire aucune transformation n'est nécessaire.

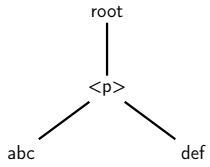
Si o_1 est de type *DeleteText*, la position de o_1 doit également être diminuée de 1 quand la position de o_2 est inférieure strictement. Lorsque les deux positions sont égales, les deux opérations visent à supprimer le même caractère, et la première opération a déjà effectué cette

tâche. Aucune opération n'est nécessaire et donc $T(o_1, o_2)$ est la fonction *Id*. Quand la position de o_1 est supérieure strictement à celle de o_2, rien n'est transformé.

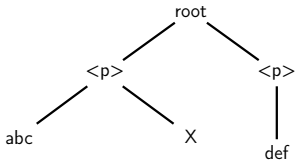
Pour une opération o_1 de type *SplitParagraph*, la transformée se calcule comme pour une insertion : si la position de o_2 est inférieure strictement à celle de o_1, cette dernière est diminuée de 1, sinon o_1 est inchangée. Il est important de remarquer que dans tous les cas, le paramètre *split* n'est pas modifié, même si la position de la transformée est 0. En effet, couper une feuille en deux si la position est 0 n'est pas nécessaire pour réaliser la séparation en deux paragraphes, néanmoins cette coupure a déjà eu lieu sur le site où la position n'était pas égale à 0 (et donc *split* était vrai), il faut donc également réaliser la coupure sur les autres sites. On transmet donc cette information grâce au paramètre *split*. La figure 4.5 illustre cette situation. La figure 4.5a présente un modèle, qui est mis à jour en concurrence par deux sites. Le premier supprime le caractère X (4.5b), le deuxième coupe le paragraphe après X (4.5c). A partir de la figure 4.5c, l'intégration de la suppression de X conduit à l'état figure 4.5d, car un nœud vide n'est pas supprimé. Il peut y avoir des opérations distantes non intégrées qui le ciblent. Depuis la figure 4.5b, il faut donc bien casser la feuille en deux pour intégrer correctement l'opération de scission de paragraphe, même si la position de la scission dans la feuille est 0.



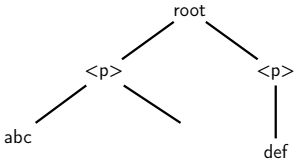
(a) Modèle initial



(b) Suppression de X



(c) Césure après X (*split* vrai)

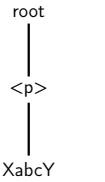


(d) Résolution : *pos*=0, *split* vrai

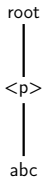
FIGURE 4.5 – Suppression et césure en début de feuille

Pour le dernier cas où o_1 est de type *Style*, les bornes *start* et *end* sont mises à jour en fonction de la position de l'opération o_2 :

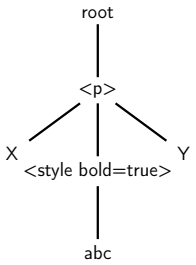
Si la position de o_2 est inférieure strictement à la borne de début de o_1, cette borne de la transformée est celle de o_1 diminuée de 1. Si la position de o_2 est inférieure strictement à la borne de fin de o_1, cette borne de la transformée est celle de o_1 diminuée de 1. Comme pour *SplitParagraph*, les deux paramètres *splitStart* et *splitEnd* ne sont pas mis à jour, même si les bornes de la transformée correspondent au début ou à la fin de la feuille. Cela s'explique de la même manière que précédemment : si la feuille a été séparée sur le site initial lors de la génération de l'opération, elle doit obligatoirement être séparée de la même manière sur les autres sites, et ce pour n'importe quelles nouvelles bornes. La figure 4.6 illustre cette situation.



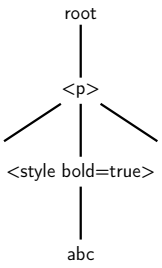
(a) Modèle initial



(b) Suppression de X et Y



(c) Ajout de style après X et avant Y



(d) Résolution : *splitStart* et *splitEnd* à vrai

FIGURE 4.6 – Suppression et style en bordure de feuille

4.2.3 Transformées par rapport à NewParagraph

L'opération *NewParagraph* est une opération qui modifie la structure arborescente du modèle de la manière la plus simple qui soit : elle se contente d'ajouter un nouveau fils (avec une feuille)

à la racine du modèle, à une position donnée. Les transformées par rapport à cette opération vont donc simplement devoir mettre à jour l'indice du ou des paragraphes sur lesquels elles agissent.

Les quatre types d'opérations *InsertText*, *DeleteText*, *SplitParagraph* et *Style* ont comme paramètre le chemin *path* qui désigne la feuille à partir de laquelle le modèle est modifié. Elles se transforment toutes de la même manière par rapport à une opération o_2 de type *NewParagraph* : si la position *pos* du nouveau paragraphe est supérieure strictement au premier indice du chemin *path* de o_1 (c'est à dire à la position du paragraphe ciblé par o_1), rien n'est transformé. Dans le cas contraire, la transformée est une copie de o_1 avec le premier indice de son chemin *path* incrémenté de 1.

Si o_1 est de type *NewParagraph*, il y a trois cas à examiner. Premièrement, si la position de o_1 est inférieure strictement à celle de o_2, rien n'est transformé. Deuxièmement, si la position de o_1 est supérieure strictement à celle de o_2, la transformée est une copie de o_1 avec une position augmentée de 1. Troisièmement, si les deux positions sont égales, on utilise les identifiants uniques de site *siteId* pour déterminer quel paragraphe est inséré en premier. Un identifiant plus faible correspond à une priorité plus haute, et donc si l'identifiant de o_1 est le plus petit des deux on ne transforme pas, sinon la transformée est une copie de o_1 avec une position augmentée de 1. Ce comportement est très similaire à celui de la transformée d'une opération de type *InsertText* par rapport à une autre opération de type *InsertText*. Dans les deux cas, il s'agit en substance de l'insertion de nouveaux éléments dans une structure linéaire.

Pour une opération o_1 de type *MoveParagraph*, les deux positions *origin* et *dest* doivent être modifiées. Si le nouveau paragraphe est créé à la même position que la destination du déplacement, on fait le choix de placer le paragraphe déplacé après le paragraphe créé. L'inverse aurait été possible. Au final, la transformée est une copie de o_1, et si la position *pos* du nouveau paragraphe est inférieure ou égale à la position de départ *origin* du déplacement o_1, le paramètre *origin* de la transformée est augmenté de 1, et si la position *pos* du nouveau paragraphe est inférieure strictement à la position d'arrivée *dest* du déplacement o_1, le paramètre *dest* de la transformée est augmenté de 1.

Dans le cas d'une opération o_1 de type *MergeParagraph*, la position de la fusion change en fonction de la position du nouveau paragraphe. Si la position de la fusion est inférieure strictement à celle du nouveau paragraphe, o_1 n'est pas modifiée. Si la position de la fusion est supérieure strictement à celle du nouveau paragraphe, la transformée est une copie de o_1 avec une position *pos* incrémentée de 1. Reste le cas où les deux positions sont égales, c'est-à-dire lorsque o_1 veut fusionner les deux paragraphes qui sont de part et d'autre du nouveau paragraphe. Dans cette situation, les paragraphes sont réarrangés avec une opération *MoveParagraph* puis la fusion a lieu. La transformée est une opération de type *Composite*. Les paramètres *origin* et *dest* de l'opération *MoveParagraph* de la transformée ont pour valeur respectivement la position de l'opération o_1 et cette position augmentée de 2, pour déplacer le nouveau paragraphe après la partie droite de la fusion. Puis la deuxième opération de la transformée *Composite* est la fusion o_1 initiale. Le nouveau paragraphe est donc placé après les paragraphes fusionnés, l'inverse aurait été possible.

4.2.4 Transformées par rapport à MoveParagraph

L'opération *MoveParagraph* modifie l'arborescence d'une manière un peu plus compliquée que son homologue *NewParagraph*, mais le principe reste le même : les transformées par rapport à cette opération sont des copies des opérations initiales avec des indices de paragraphes différents.

Pour les quatre type d'opérations *InsertText*, *DeleteText*, *SplitParagraph* et *Style*, qui agissent sur une feuille d'un paragraphe précis, la transformation est la suivante. Si la position de départ *origin* de l'opération de déplacement o_2 est inférieure strictement au premier indice du chemin *path* de o_1, cet indice est décrémenté de 1 dans la transformée puisqu'un paragraphe a été retiré avant cet indice, si elle est supérieure strictement l'indice reste le même. Puis si la position d'arrivée *dest* de l'opération o_2 est inférieure ou égale au premier indice du chemin *path* de o_1, cet indice est incrémenté de 1 dans la transformée puisqu'un paragraphe est ajouté avant cet indice, sinon il n'est plus modifié. Il reste le cas où le paragraphe ciblé par o_1 est le paragraphe déplacé. Dans ce cas, le premier indice du chemin *path* de la transformée est égal à la position d'arrivée *dest* de o_2 si la position de départ est supérieure ou égale à celle d'arrivée, et est égal à la position d'arrivée décrémentée de 1 sinon, à cause des choix que nous avons faits dans le comportement de la fonction de déplacement en fonction de ses paramètres.

Pour une opération de type *NewParagraph*, la position *pos* de o_1 est mise à jour dans la transformée en fonction des bornes de o_2 d'une manière semblable à celle explicitée au paragraphe précédent. Si la position de départ *origin* de o_2 est inférieure strictement à la position *pos* de o_1, cette position est décrémentée de 1 dans la transformée, sinon elle ne change pas. Puis si la position d'arrivée *dest* de o_2 est inférieure ou égale à la position *pos* de o_1, cette position est incrémentée de 1 dans la transformée, sinon elle ne change pas. La situation où la position du nouveau paragraphe est la même que la position finale du paragraphe déplacé est capturée par ces deux cas, et aboutit à un modèle où le paragraphe déplacé se situe avant le paragraphe créé, ce qui est bien ce que l'on a également fait pour la transformée d'une opération de type *MoveParagraph* par rapport à une opération de type *NewParagraph*.

Transformer une opération de type *MoveParagraph* par rapport à une autre opération de type *MoveParagraph* nécessite de traiter le cas particulier où deux utilisateurs souhaitent déplacer le même paragraphe. Examinons ce cas en premier.

Si les deux positions de départ *origin* des deux opérations o_1 et o_2 sont les mêmes, et si les deux positions d'arrivée *dest* sont aussi les mêmes, les deux opérations représentent le même déplacement, et il suffit d'en exécuter une. Donc la transformée est *Id* dans ce cas.

Si les bornes de départ sont les mêmes mais les bornes d'arrivée différentes, l'opération qui a l'identifiant unique de site *SiteId* le plus faible est prioritaire et son paramètre *dest* détermine la position finale du paragraphe. Si o_2 est prioritaire, alors o_1 ne doit pas être exécutée et la transformée est *Id*. Sinon, il faut déplacer le paragraphe qui a déjà été déplacé par o_2, à sa nouvelle position qui nous est donnée par o_1. Ce paragraphe se situe à la position d'arrivée *dest*

de o_2 si la position de départ *origin* de o_2 est supérieure strictement à la position d'arrivée *dest* de o_2. Il se situe à la position d'arrivée *dest* de o_2 décrémentée de 1 sinon. Cette valeur est celle du paramètre de départ de la transformée. La valeur du paramètre d'arrivée de la transformée est la même que celle du paramètre d'arrivée de o_1, modifiée en fonction de l'opération o_2. Si la position de départ de l'opération o_2 est inférieure strictement à la position d'arrivée de o_1, la position d'arrivée de la transformée est décrémentée de 1. Puis si la position d'arrivée de o_2 est inférieure strictement à la position d'arrivée de o_1, la position d'arrivée de la transformée est incrémentée de 1.

Il reste le cas général à expliciter, qui est une double mise à jour des bornes de o_1 dans la transformée en fonction de celles de o_2. Si la position de départ de o_2 est inférieure strictement à la position de départ de o_1, la position de départ de la transformée est décrémentée de 1. Puis si la position d'arrivée de o_2 est inférieure ou égale à la position de départ de o_1, la position de départ de la transformée est incrémentée de 1. Si la position de départ de o_2 est inférieure strictement à la position d'arrivée de o_1, la position d'arrivée de la transformée est décrémentée de 1. Puis si la position d'arrivée de o_2 est inférieure strictement à la position d'arrivée de o_1, la position d'arrivée de la transformée est incrémentée de 1. Enfin, si les deux bornes d'arrivée sont identiques, l'opération qui possède l'identifiant de site le plus petit place son paragraphe avant l'autre. Si l'identifiant de site de o_1 est plus grand que celui de o_2, il faut donc incrémenter de 1 la position d'arrivée de la transformée.

La situation où o_1 est de type *MergeParagraph* présente aussi son lot de cas particuliers. On peut vouloir fusionner deux paragraphes qui ne sont plus adjacents à cause du déplacement d'un autre paragraphe au milieu. On peut également vouloir fusionner deux paragraphes dont l'un a été déplacé. Dans le premier cas on déplace le paragraphe du milieu à droite des deux autres, puis on réalise la fusion. La transformée est donc une opération de type *Composite* qui contient une opération *MoveParagraph* et une opération *MergeParagraph*. La position de départ de l'opération *MoveParagraph* de la transformée est la position d'arrivée de o_2, décrémentée de 1 si la position de départ de o_2 est inférieure strictement à la position de d'arrivée de o_2. La position de son arrivée est celle de son départ augmenté de 2, pour le déplacer d'un indice. L'opération *MergeParagraph* de la transformée a pour position la position de départ de la première opération de la transformée, et le paramètre *leftChNb* de o_1 lui est transmis. Dans le deuxième cas, on déplace le second paragraphe de la fusion près du premier, et on réalise la fusion. La transformée sera une opération de type *Composite* qui contiendra une opération de type *MoveParagraph* et une autre de type *MergeParagraph*. Si le paragraphe de gauche de la fusion a été déplacé, on déplace le paragraphe de droite après lui et on fusionne. Si le paragraphe de droite de la fusion a été déplacé, on déplace le paragraphe de gauche avant lui et on fusionne. Dans le cas général, la transformée est une opération de type *MergeParagraph* qui est une copie de o_1, avec la position incrémentée de 1 si un paragraphe après la fusion a été déplacé avant, décrémentée de 1 si un paragraphe avant la fusion a été déplacé après, inchangé sinon.

4.2.5 Transformées par rapport à MergeParagraph

Une opération *MergeParagraph* modifie le modèle en supprimant un paragraphe, mais ajoute également des fils à un autre paragraphe. Elle reste une opération qui ne modifie pas le contenu des feuilles, et donc les opérations qui ne font qu'en modifier seront peu impactées.

Les transformées pour les opérations de type *InsertText*, *DeleteText*, *SplitParagraph* et *Style*, qui agissent sur une feuille, sont les suivantes : si le paragraphe qui contient la feuille a un indice inférieur ou égal à celui du paragraphe de droite de la fusion (c'est à dire si le premier indice du paramètre *path* de o_1 est inférieur strictement à la position *pos* de o_2), o_1 n'est pas transformée. Si le premier indice du paramètre *path* de o_1 est supérieur strictement à la position *pos* de o_2, les modifications apportées par o_1 ne ciblent aucun paragraphe de la fusion, donc il suffit de répercuter la suppression du paragraphe. La transformée est une copie de o_1 avec le premier indice du *path* décrémenté de 1. Dans dernier cas, les modifications sont apportées à une feuille qui a été déplacée depuis le paragraphe de droite de la fusion dans le paragraphe de gauche de la fusion. Il faut donc d'une part mettre à jour l'indice du paragraphe, mais aussi l'indice du fils à l'intérieur du paragraphe. Le paramètre *leftChNb* de o_2 contient le nombre de fils qu'avait le paragraphe de gauche avant la fusion, c'est de ce nombre que l'on a besoin pour calculer le nouveau chemin. La transformée est ici une copie de o_1, avec le premier indice du chemin *path* décrémenté de 1 et le deuxième indice incrémenté de la valeur du paramètre *leftChNb* de o_2. La figure 4.7 illustre ce cas.

(a) Avant la fusion : X est dans le premier fils du deuxième paragraphe

(b) Après la fusion : X est dans le deuxième fils du premier paragraphe

FIGURE 4.7 – Décalages d'indices de chemin après une fusion

Dans le cas d'une opération o_1 de type *NewParagraph*, la position du nouveau paragraphe est mise à jour en fonction de la fusion. Nous avons fait le choix, lors de la transformée d'une opération de type *MergeParagraph* par rapport à une opération de type *NewParagraph*, de placer le nouveau paragraphe après les paragraphes fusionnés. Il faut conserver ce choix ici. La transformée est donc une copie de o_1, avec sa position *pos* décrémentée de 1 si la position *pos* de o_1 est supérieure strictement à la position *pos* de o_2. La suppression d'un paragraphe est ainsi répercutée.

Pour une opération o_1 de type *MoveParagraph*, les deux positions doivent être modifiées. Le cas général de cette transformée est celui où aucun des deux paragraphes n'est touché par l'opération de déplacement. Il suffit dans cette situation de copier o_1 pour avoir la transformée,

avec les deux paramètres *origin* et *dest* décrémentés de 1 si ceux de o_1 sont supérieurs strictement à la position *pos* de o_2. On peut noter que ce cas englobe celui où le paragraphe est déplacé au milieu de la fusion, puisque le choix a été fait précédemment de placer le paragraphe à droite de la fusion lorsque ce conflit survient. Dans le cas où un paragraphe de la fusion est déplacé, pour être cohérent avec les transformées déjà explicitées il faut déplacer le paragraphe fusionné tout entier. Si le paragraphe de droite est déplacé, la position *pos* de o_2 est égale à la position de départ *origin* de o_1. Si dans ce cas ce paragraphe est déplacé juste avant celui de gauche, la fusion est déjà à la bonne place et la transformée est l'opération *Id*. Sinon la transformée est une copie de o_1, avec le paramètre *origin* décrémenté de 1 (le paragraphe fusionné est à une position de moins que l'ancien paragraphe de droite), et avec le paramètre *dest* décrémenté de 1 si la position de o_2 est inférieure strictement à la destination de o_1 (la fusion supprime un paragraphe). Si le paragraphe de gauche est déplacé, le cas est symétrique et la transformée est la même, sauf qu'il n'y a pas besoin de décrémenter la position de départ puisque le paragraphe fusionné est à la même position que l'ancien paragraphe de gauche.

Lorsque les deux opérations o_1 et o_2 sont de type *MergeParagraph*, la transformée est relativement simple. Si les deux opérations fusionnent les mêmes paragraphes, alors la transformée est *Id*. Sinon, la transformée est une copie de o_1, avec la position de la fusion décrémentée de 1 si la position de o_1 est supérieure strictement à celle de o_2. Cela suffit à mettre à jour la position de la transformée, mais dans le cas où les deux opérations ont lieu côte à côte et fusionnent trois paragraphes en un seul, il faut mettre à jour le paramètre *leftChNb*. Si o_1 est la fusion de droite, alors le paramètre *leftChNb* de la transformée est égal à la somme des paramètres *leftChNb* des deux opérations o_1 et o_2.

4.2.6 Transformées par rapport à SplitParagraph

L'opération *SplitParagraph* a de l'influence sur la structure de paragraphe du modèle et sur une feuille. Elle modifie à la fois la structure du modèle et les données du modèle. Les opérations qui n'agissent que sur la structure auront une transformée proche de celle par rapport à une opération *NewParagraph*.

Si l'opération o_1 est de type *InsertText* ou *DeleteText*, il faut trouver la nouvelle feuille et la nouvelle position dans la feuille où insérer ou supprimer le caractère. Si la feuille ciblée par o_1 se trouve dans un paragraphe situé après le paragraphe ciblé par o_2, la transformée est une copie de o_1 avec le premier indice du chemin *path* incrémenté de 1 pour répercuter l'ajout de paragraphe dû à la césure. Si la feuille ciblée par o_1 se trouve avant celle ciblée par o_2, o_1 n'a pas besoin d'être transformée. Si les deux chemins *path* sont les mêmes, et si la position de o_1 est inférieure strictement à la position de o_2, o_1 n'a pas besoin d'être transformée. Sinon, la modification apportée par o_1 se trouve dans la partie de la feuille qui va être coupée et déplacée dans le nouveau paragraphe. Ce nouveau paragraphe aura pour position (son chemin aura pour premier indice) le premier indice du chemin *path* de o_1 ou o_2 incrémenté de 1, et la feuille sera dans son premier fils, d'indice 0. On construit donc le chemin *path* de la transformée avec ces informations.

Puis, la position *pos* de la transformée, c'est-à-dire la nouvelle position de l'opération dans la feuille, est égale à la position de o_1 décrémentée de la position de o_2 (c'est-à-dire du nombre de caractères restant dans la feuille initiale). Dans le dernier cas, o_1 et o_2 ciblent le même paragraphe mais la feuille ciblée par o_2 est avant celle ciblée par o_1. Dans ce cas la transformée est une copie de o_1, avec son chemin *path* mis à jour : la nouvelle position de la feuille ciblée par o_1 se trouve un paragraphe plus loin qu'initialement, et le nouvel indice du fils dans lequel elle se trouve est égal à l'indice du fils donné par le chemin *path* de o_1 décrémenté de celui donné par o_2.

La transformée par rapport à une opération de type *SplitParagraph* suit le même principe, à un changement près : si les deux opérations sont les mêmes, la transformée est l'opération *Id*.

Pour une opération o_1 de type *Style*, on se rapproche des transformées des opérations de type *InsertText* et *DeleteText*. Les cas où les deux opérations ne ciblent pas la même feuille sont les mêmes que ceux présentés plus haut, seuls diffèrent les cas où les chemins *path* des deux opérations sont les mêmes. Dans cette situation, il y a trois cas à examiner. Premièrement, si le style porte sur la partie gauche de la césure, la transformée est une copie de o_1, avec la particularité de changer le paramètre *splitEnd* à faux si la borne de fin du style tombe à l'endroit de la césure. Deuxièmement, si le style porte sur la partie droite de la césure, la transformée est une copie de o_1, avec son chemin *path* et ses bornes *start* et *end* mises à jour comme expliqué avant pour les autres opérations, et avec le paramètre *splitStart* à faux si la borne de début du style tombe à l'endroit de la césure. Troisièmement, si la césure tombe au milieu du style, la transformée est une opération de type *Composite*, qui contient deux opération de type *Style* : une pour chaque partie de la césure. La première est une copie de o_1 avec sa borne d'arrivée *end* égale à la position *pos* de o_2 et *splitEnd* à faux. La seconde est une copie de o_1 avec sa borne d'arrivée *end* et son chemin *path* mis à jour comme expliqué au début, sa borne de départ *start* égale à 0, et avec *splitStart* mis à faux.

Pour une opération o_1 de type *NewParagraph*, la transformée est immédiate : si la position *pos* de o_1 est inférieure ou égale au premier indice du chemin *path* de o_2, la césure a lieu dans un paragraphe plus éloigné que l'insertion de o_1 et donc rien n'est transformé, et dans le cas contraire la transformée est une copiée de o_1 avec la position *pos* augmentée de 1 pour prendre en compte le nouveau paragraphe issu de la césure.

Dans le cas d'une opération o_1 de type *MoveParagraph*, si le paragraphe à déplacer est le paragraphe qui vient d'être séparé en deux, il faut déplacer ces deux paragraphes. Dans ce cas la position *pos* de o_1 est égale au premier indice du chemin *path* de o_2, et la transformée est une opération de type *Composite* qui contient deux opérations *MoveParagraph*. Si la position de départ *origin* de o_1 est inférieure strictement à sa position d'arrivée *dest*, alors les deux opérations de la transformée sont des copies de o_1 avec leur position d'arrivée incrémentée de 1. Sinon les deux opérations sont des copies de o_1 avec leur position de départ augmentée de 1. Si le déplacement cible un paragraphe quelconque, la transformée est une copie de o_1 avec ses positions de départ et d'arrivée incrémentées de 1 si elles se trouvent après le paragraphe ciblé par o_2.

La transformée d'une opération o_1 de type *MergeParagraph* se comporte presque comme pour une opération de type *NewParagraph*. La transformée est une copie de o_1, avec sa position *pos* incrémentée de 1 si la position *pos* de o_1 est supérieure strictement au premier indice du chemin *path* de o_2. Il reste ensuite à mettre à jour le paramètre *leftChNb* de la transformée. Lorsque la fusion a lieu entre la partie droite de la césure et le paragraphe suivant, l'information contenue dans o_1 n'est plus exacte, et le paramètre *leftChNb* de la transformée est égal à celui de o_1 décrémenté du second indice du chemin *path* de o_2, c'est-à-dire du nombre de fils restant dans le paragraphe de gauche après la césure.

4.2.7 Transformées par rapport à Style

Les dernières transformées à passer en revue sont celles par rapport à une opération o_2 de type *Style*. Comme pour *InsertText* et *DeleteText*, les opérations qui agissent sur la structure et n'ont pas besoin des informations contenues dans les feuilles de l'arbre seront très peu transformées.

C'est le cas des opérations de type *NewParagraph* et *MoveParagraph*, qui ne sont pas transformées du tout. Une opération o_1 de type *MergeParagraph* est transformée dans un cas bien précis : lorsque le style agit dans le paragraphe de gauche de la fusion, alors le paramètre *leftChNb* de o_1 doit être mis à jour et incrémenté, une opération *Style* pouvant créer jusqu'à deux fils supplémentaires dans un paragraphe (un par paramètre *splitStart* et *splitEnd* mis à vrai).

Pour une opération o_1 de type *InsertText* ou *DeleteText*, la transformée met à jour la position et le chemin de l'opération si besoin. Si les paragraphes de o_1 et de o_2 (les premiers indices de leurs chemins respectifs) sont différents, rien n'est transformé. Si les paragraphes sont les mêmes, et que le fils ciblé par o_1 est avant celui ciblé par o_2, là aussi rien n'est transformé. Si le fils ciblé par o_1 est après celui ciblé par o_2, la transformée est une copie de o_1 avec le deuxième indice du chemin incrémenté en fonction des paramètres *splitStart* et *splitEnd* pour répercuter l'ajout de fils avant l'opération. Dans le dernier cas, les deux opérations ciblent initialement le même fils. Si la position *pos* de o_1 est inférieure strictement à la position de départ *start* de o_2, rien n'est transformé car le style agit plus loin que o_1. Si la position de o_1 se situe entre la position de départ (incluse), et la position d'arrivée (incluse pour une insertion, exclue pour une suppression) de o_2, la modification apportée par o_1 est dans la zone modifiée par le style. La transformée est une copie de o_1, avec sa position égale à celle de o_1 décrémentée de la position de départ de o_2. Si le paramètre *splitStart* de o_2 est à vrai, le chemin de la transformée a son deuxième indice incrémenté de 1 pour répercuter l'ajout d'un fils à gauche. Enfin, si la position de o_1 est supérieure (strictement pour une insertion, ou égale pour une suppression) à la position de fin de o_2, la transformée est une copie de o_1, avec sa position égale à celle de o_1 décrémentée de la position de fin de o_2, et si le paramètre *splitStart* de o_2 est à vrai, le chemin de la transformée a son deuxième indice incrémenté de 2, sinon l'incrément est de 1, o_1 agissant maintenant sur le fils à droite du style.

La transformée d'une opération de type *SplitParagraph* par rapport à une opération de type *Style* se comporte comme celle d'une opération de type *DeleteText*, sauf pour la gestion du

paramètre *split* qui n'est pas toujours copié. Il est mis à faux dans le cas où la césure a lieu à la même position que le début ou la fin du style appliqué par o_2.

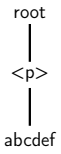
Si o_1 est une opération de type *Style* comme o_2, la transformée est une copie qui suit les mêmes principes que ceux expliqués pour les trois types d'opérations qui précédent lorsque les chemins des deux opérations sont différents. Lorsque les deux chemins sont égaux, la transformée est plus complexe.

Si les deux styles ciblent des zones de texte différentes, une au début du texte, l'autre à la fin, la transformée du style qui agit sur la fin par rapport au style qui agit sur le début doit mettre à jour son chemin et ses positions de début et de fin, car le premier style a créé un ou deux nœuds fils supplémentaires liés au paragraphe père pour isoler le texte à gauche ou à droite de la zone à styliser. Ce n'est donc plus le même indice de fils qui contient la zone de texte à styliser, et ce fils ne contient plus le texte initial mais uniquement la partie de texte qui se situait après la zone que la première opération a stylisée. Si les deux zones sont contiguës, le paramètre *splitStart* de la transformée doit être égal à faux : on n'isole plus de texte à gauche du style, ceci a déjà été réalisé par la première opération. L'opération de style qui agit en début de texte n'a pas besoin d'être transformée, sauf si les zones sont contiguës, dans ce cas la transformée est une copie de l'opération initiale, avec son paramètre *splitEnd* égal à faux, pour des raisons symétriques à celles expliquées plus haut.

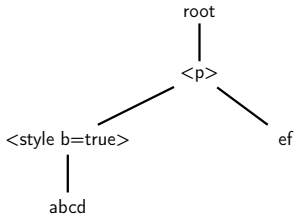
Si les deux styles ciblent exactement la même zone de texte, on va vouloir appliquer les deux styles au texte, mais il se peut qu'ils ne soient pas compatibles. Par exemple, une opération peut ajouter de l'italique, et une autre en retirer. Dans ce cas, on utilise le paramètre *siteId* pour déterminer quel style est appliqué. Le plus petit identifiant a la priorité. Donc, lors de la transformée de o_1 par rapport à o_2, si les styles sont compatibles ou si o_1 est prioritaire, la transformée est une copie de o_1, avec le deuxième indice de son chemin incrémenté de 1 et ses bornes de début et de fin mises à jour si le style n'est pas appliqué dès le début de la feuille texte, c'est-à-dire si o_2 a créé un fils à gauche pour isoler le texte avant la zone stylisée. Les paramètres *splitStart* et *splitEnd* de la transformée sont égaux à faux dans tous les cas, car la zone de texte a déjà été extraite par la première opération. Si les styles ne sont pas compatibles et si o_2 est prioritaire, la transformée est l'opération *Id*.

Si les deux styles ciblent des zones qui se chevauchent, la transformée de o_1 par rapport à o_2 sera une opération de type *Composite*, qui contiendra plusieurs opérations de type *Style*. En effet, une opération de type *Style* ne porte que sur une feuille, la feuille initiale a été coupée en deux ou trois par l'opération o_2, et l'opération o_1 agit maintenant sur le contenu de plusieurs de ces nouvelles feuilles. Il faut une opération de type *Style* par feuille concernée. Les opérations *Style* contenues dans la transformée *Composite* sont créées avec les mêmes principes que ceux expliqués dans les deux cas précédents. Une différence est présente lorsque les styles ne sont pas compatibles : il faut quand même appliquer une opération *Style* sur une feuille stylisée par o_2 lorsque o_1 la stylise également mais n'est pas prioritaire, pour mettre à jour la structure de l'arbre. Le style ajouté sera celui déjà existant de o_2. La figure 4.8 illustre cette situation. La

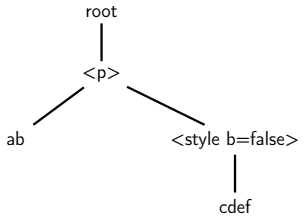
figure 4.8a présente la situation initiale de notre exemple, qui est un simple paragraphe contenant la feuille texte "abcdef". Deux sites s_1 et s_2 vont en concurrence styliser le texte "abcd" et "cdef". Le résultat local de leurs opérations est présenté figures 4.8b et 4.8c. On considère que les styles ne sont pas compatibles et que le style de s_2 a la priorité. Sur le site s_1, il faut donc générer deux opérations de type *Style* pour intégrer l'opération de s_2 : une pour changer le style de "cd", l'autre pour changer le style de "ef". Le résultat de cette intégration est présenté figure 4.8d. Sur le site s_2, il ne faut a priori que styliser "ab", puisque dans la zone conflictuelle "cd", le site s_2 est prioritaire. Mais se contenter de cette seule opération ne mène pas à l'état figure 4.8d. Pour atteindre l'état en question, il faut séparer les textes "cd" et "ef" dans deux feuilles différentes. Ceci est réalisé avec une opération *Style*, qui applique à nouveau sur "cd" le style déjà présent.



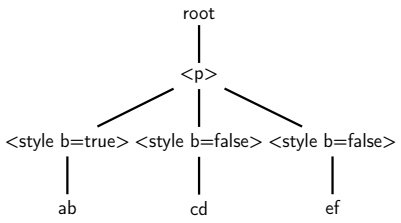
(a) Etat initial



(b) Application du premier style sur le site s_1



(c) Application du second style sur le site s_2



(d) Etat après intégration des deux opérations

FIGURE 4.8 – Intégration de deux opérations *Style* concurrentes

4.3 Implémentation

Pour valider notre algorithme, nous l'avons intégré dans un éditeur collaboratif pour l'entreprise XWiki. Cette entreprise offre à ses clients différents services pour la gestion de wikis, un wiki étant un ensemble de pages web pouvant contenir du texte stylisé et organisé, modifiable

par les lecteurs des pages. C'est donc un environnement de gestion de contenu collaboratif par nature. Par exemple, Wikipédia est une encyclopédie en ligne bien connue qui utilise le service de wikis MediaWiki [5].

Le moyen habituel d'éditer une page wiki est de demander au serveur une copie de la version actuelle de la page, de la modifier, et de soumettre au wiki la nouvelle version. Lors de la soumission en concurrence de plusieurs versions, le conflit est résolu en n'acceptant qu'une version. Seule la première version envoyée au serveur est intégrée, les autres ne sont pas acceptées car l'état à partir duquel elles ont été modifiées n'est plus celui du serveur. Les utilisateurs qui n'ont pas pu soumettre leur version sont notifiés et doivent la soumettre à nouveau, après avoir rechargé la page du serveur et modifié leur copie en conséquence. L'édition collaborative en temps réel de pages wiki permettrait de supprimer ce désagrément, et MediaWiki [6] ainsi que l'application de travail collaboratif Confluence [7] par exemple ont montré un intérêt pour ce type d'édition.

Un frein à l'édition collaborative en temps réel de pages wiki est la syntaxe particulière utilisée lors de l'édition. Le modèle d'une page wiki est différent de la simple suite de caractères qui s'affiche dans le navigateur. Cette syntaxe utilise des caractères spéciaux pour ajouter du style au texte ou spécifier des éléments hiérarchiques comme des titres. Par exemple, la figure 4.9 présente un document utilisant la syntaxe wikitext, qui correspond à un texte contenant un titre principal et un paragraphe contenant du texte non stylisé, puis du texte en gras, du texte en italique, et une url. Cette syntaxe est traduite en code HTML pour être affichée dans les navigateurs. L'équivalent HTML de l'exemple précédent est proposé figure 4.10.

Utiliser les algorithmes de l'état de l'art pour éditer en temps réel un document HTML ou wikitext risque de conduire à un résultat mal formé qui ne peut pas être affiché correctement dans les navigateurs, à cause de conflits d'édition mal résolus qui incluent l'insertion de caractères spécifiques ou de balises. Les intentions des utilisateurs seront mal interprétées. L'intention de mettre en gras une partie du document, par exemple, est différente de l'intention d'ajouter les deux chaînes de caractères "**" et "**".

Notre algorithme propose une solution à ce problème. Ses opérations et transformées permettent le respect d'intentions de haut niveau, et la structure de son modèle de données est adaptée aux wikis. Elle est proche du rendu html désiré. La figure 4.11 propose un moyen de traduire notre page d'exemple dans notre modèle de données.

Contrairement à une session d'édition collaborative où tout le monde participe à la création du document, certains utilisateurs de wiki veulent seulement avoir accès au document en lecture. Ces utilisateurs n'ont pas besoin d'environnement d'édition collaborative, une simple page statique, qui correspond à la version actuelle du serveur, suffit. Cette page n'est pas réactualisée automatiquement, et doit donc correspondre à une version stable, et non à une version en cours d'édition.

Pour cela, une page de wiki qui est en train d'être éditée à plusieurs en temps réel n'est

5. https ://www.mediawiki.org/
6. https ://www.mediawiki.org/wiki/Future/Real-time_collaboration
7. https ://jira.atlassian.com/browse/CONF-8333

```
=Titre=

texte
**texte en gras**
//texte en italique//
[[http://link.com|ceci est un lien.]]
```

FIGURE 4.9 – Exemple de document avec la syntaxe wikitext

```
<h1>Titre</h1>
<p>
texte
<b>texte en gras</b>
<i>texte en italique</i>
<a href=‘‘http://link.com’’>ceci est un lien</a>
</p>
```

FIGURE 4.10 – Exemple de page générée à partir d'un document wikitext

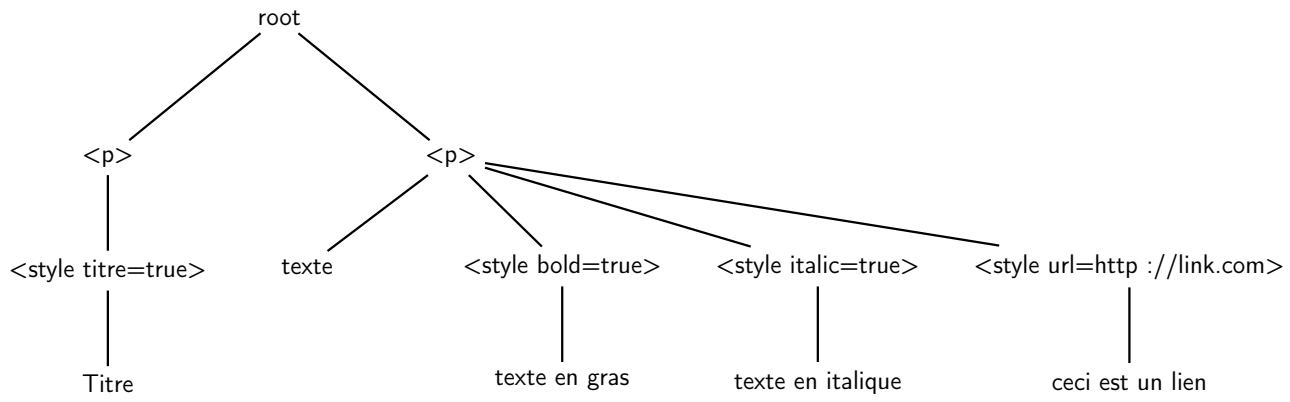


FIGURE 4.11 – Exemple de structure en arbre

pas automatiquement soumise au serveur à chaque modification. Charge aux contributeurs de la soumettre manuellement une fois que son contenu sera satisfaisant. Du point de vue du serveur, l'ajout de la fonctionnalité d'édition collaborative en temps réel ne change rien. Simplement, les versions qui lui sont soumises peuvent provenir d'un effort commun au lieu d'uniquement un contributeur. La figure 4.12 illustre l'architecture de l'application, qui permet de comprendre les différentes actions entre les lecteurs, les contributeurs isolés, les contributeurs qui éditent en collaboration, et le système.

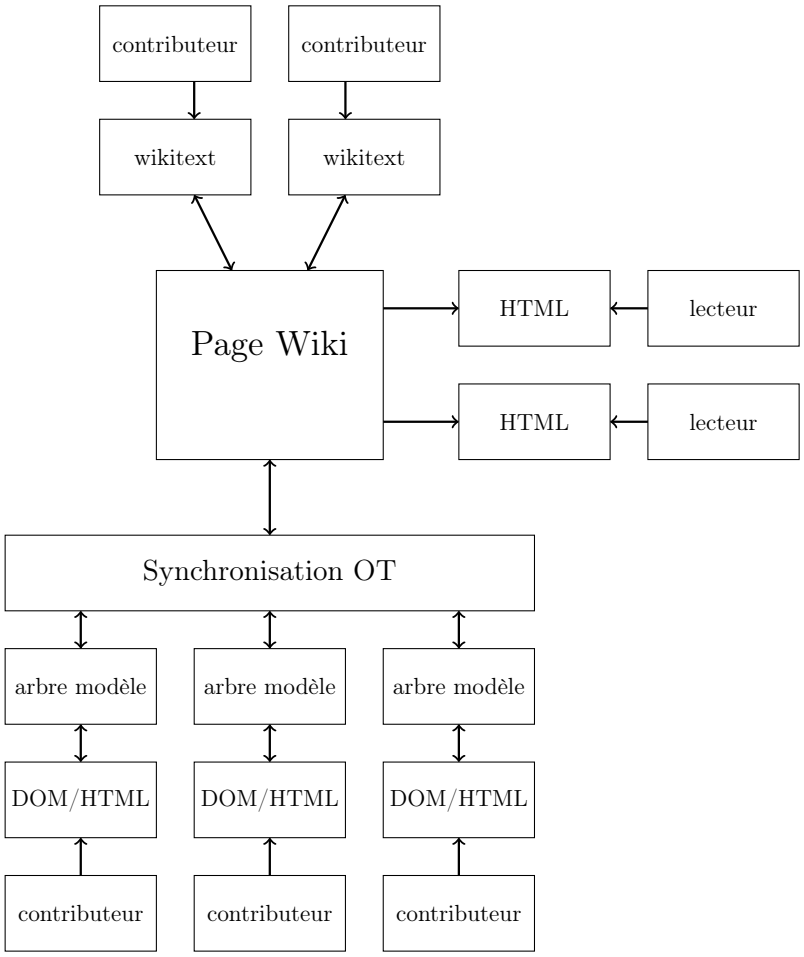


FIGURE 4.12 – Architecture de l'éditeur temps réel de wiki

Un lecteur demande au serveur la page wiki qu'il souhaite consulter. Le serveur lui fourni la page en question, sous forme de document HTML affiché dans son navigateur. Un contributeur qui veut éditer seul une page du wiki l'obtient sous forme de syntaxe wikitext. Ce contributeur peut éditer cette page wikitext, et éventuellement soumettre sa version au serveur, qui l'enregistre au format wikitext s'il n'y a pas de conflit d'édition. Le contributeur qui tente de commencer l'édition d'une page en isolation est notifié de la présence d'une session d'édition en collaboration, et peut revenir sur son choix pour rejoindre la session. Cela réduit le risque de conflits dus à la

soumission de deux versions en concurrence.

Des contributeurs peuvent rejoindre l'environnement d'édition en temps réel. Dans notre implémentation, nous proposons d'éditer la page via un éditeur What You See Is What You Get (WYSIWYG), c'est à dire que l'on édite le modèle à partir de la vue, et non à partir d'un modèle intermédiaire. Comme son nom l'indique, les utilisateurs visualisent et modifient directement ce que tout le monde voit et verra dans son navigateur. L'entreprise XWiki possède un éditeur de Document Object Model (DOM) qui a été réutilisé pour cette application. Le DOM est une façon abstraite de représenter du HTML, XML, ou autre langage à balises, et de le manipuler. L'éditeur de DOM de XWiki est un éditeur WYSIWSG, et présente le document comme une page web. Cet éditeur est le point d'entrée de l'application pour les éditeurs en session collaborative temps réel. Lorsque le DOM est édité, les modifications sont traduites en opérations exécutables sur notre modèle en arbre, et exécutées sur ce dernier. La synchronisation des modifications des différents contributeurs s'effectue à ce niveau. Les modifications des autres contributeurs sont intégrées dans notre modèle en arbre, qui permet de reconstruire l'HTML et le DOM pour afficher les modifications. A tout moment, n'importe lequel des contributeurs peut tenter de sauvegarder la version éditée sur le serveur. Les cas où la soumission a lieu en concurrence avec une autre soumission n'a pas été traité par nos partenaires industriels. Idéalement, il faudrait que si la page wiki du serveur change pendant l'édition en collaboration, les modifications soient transmises à la session collaborative. Les contributeurs de la session seraient donc avertis du changement d'état, en constateraient les conséquences, et travailleraient toujours sur une version à jour.

4.4 Discussion

Les opérations de notre approche sont plus nombreuses et offrent à l'utilisateur plus de moyens d'expression que dans les approches classiques. De plus, ces opérations sont conçues pour représenter les intentions des utilisateurs, de sorte que le respect des intentions des opérations et le respect des intentions de l'utilisateur sont au final liés. Pour arriver à ce résultat, nous avons dû concevoir une structure de données et des moyens de la modifier complexes. Notre approche peut se définir comme une approche orientée utilisateurs, par opposition aux approches existantes que l'on peut qualifier d'orientées données. Au lieu de chercher à rendre possible l'édition en concurrence d'une structure de données spécifique, nous avons créé un modèle qui permet le respect d'intentions des utilisateurs spécifiques.

Par rapport aux approches existantes, la résolution des conflits est plus satisfaisante. Par exemple, notre structure de données peut être éditée dans une certaine mesure avec les opérations de l'approche [Davis *et al.*, 2002] décrite dans l'état de l'art qui s'exécutent sur un arbre quelconque, et l'approche [Ignat and Norrie, 2002] possède également des paragraphes. Dans ces deux approches, scinder un paragraphe en deux se traduit par la suppression de texte dans un nœud et l'insertion d'un nouvel arbre, une modification concurrente de texte dans la partie supprimée puis réinsérée sera perdue, une scission concurrente du même texte dupliquera les données. Par

rapport à l'approche [Sun *et al.*, 2006], nous n'avons pas besoin de décomposer nos opérations de haut niveau en opérations d'insertion et de suppression, puis de les recomposer éventuellement. Dans tous les cas, notre approche produit moins d'opérations que les autres approches. Elle regroupe une intention par opération. Cela a une influence directe sur les performances d'intégration des opérations distantes, en réduisant le nombre de transformées[Li and Li, 2006].

Nos opérations et transformées sont conçues pour respecter la propriété TP_1 uniquement. Une approche pour prouver qu'elles la respectent effectivement à été tentée avec le langage de preuve Coq [8]. Elle a échouée, définir les effets d'une opération sur le modèle en fonction de ses différents paramètres étant trop complexe. Toutefois, des tests en interne, ainsi que l'utilisation de notre algorithme pour l'application de l'entreprise XWiki portent à croire que la propriété est respectée.

Un des désavantages de notre approche et que l'on perd la généricité des approches orientées données. Ces dernières permettent d'éditer n'importe quelle structure de données sur laquelle s'appliquent leurs opérations, tandis que notre approche est conçue pour un ensemble d'applications bien précis. Si l'on souhaite ajouter des intentions à notre modèle, il faut rajouter des opérations, des transformées, ce qui va complexifier cette approche.

Actuellement, on peut simuler certaines intentions qui n'entrent pas dans la liste initiale. Par exemple, le déplacement d'un bloc de texte quelconque et non pas d'un paragraphe complet est simulé en scindant deux fois le paragraphe qui contient le bloc afin d'isoler ce dernier, puis on scinde le paragraphe d'arrivée du bloc à l'endroit où il doit être inséré, on déplace le bloc entre ces deux paragraphes, on les fusionne tous les trois pour recomposer un paragraphe unique, et enfin le paragraphe qui contenait initialement le bloc est recomposé avec une fusion, ce qui génère un nombre conséquent d'opérations. Une opération *moveBlock* serait préférable.

Dans le même état d'esprit, si on veut améliorer l'intégration de notre approche dans l'édition de wikis et proposer les fonctionnalités d'ajout de tableaux et de listes à points, il faudrait ajouter des opérations d'insertion de lignes, de colonnes, de modification de cellules, d'insertion de listes et d'éléments de listes, et les transformées associées. Une solution pour intégrer ces modifications est de fonctionner avec des paragraphes typés. La fonction *NewParagraph* peut prendre un paramètre supplémentaire qui correspond au type du paragraphe. Un paragraphe peut être du type normal, du type élément de liste, de type entrée de tableau sur une nouvelle ligne ou une nouvelle colonne. On peut ainsi directement scinder une liste en deux ou réorganiser ses éléments avec les fonctions déjà existantes qui modifient des paragraphes au sens large. On peut ajouter différents titres de la même manière.

L'approche OT reste une approche lourde qui demande la conception de nombreuses opérations et transformées. Le chapitre suivant présente un algorithme de l'approche CRDT, conçu pour résoudre les problèmes d'intentions des utilisateurs causés par la granularité fixe des éléments d'un CRDT.

8. https ://coq.inria.fr/

5

LogootSplit : CRDT pour blocs à taille variable

Le problème principal des CRDT pour l'édition collaborative de document texte tel qu'expliqué dans l'état de l'art est qu'ils ne supportent que des opérations d'insertion et de suppression d'éléments, ce qui conduit à deux inconvénients majeurs : utiliser des éléments de petite taille peut conduire à des caractères faussement entrelacés ; utiliser des éléments de grande taille ajoute des duplicatas lors de mises à jour concurrentes localisées au même endroit.

Ce chapitre détaille notre solution à ces problèmes, qui se base sur un modèle qui, en plus des opérations standard d'insertion et suppression, autorise la scission d'un élément en éléments plus petits de manière atomique (non simulée par une suppression puis deux insertions). Ainsi, il est possible de créer initialement des éléments de taille importante, qui seront coupés si besoin pour insérer des mises à jour. L'idée a été empruntée à [Yu, 2012], et a conduit à une publication [André *et al.*, 2013].

Ce chapitre présente les principes et algorithmes généraux de notre approche dans une première section. Quelques résultats expérimentaux sont présentés dans une deuxième section, suivis d'une section consacrée à une application implémentant notre approche. Enfin, une dernière section dresse le bilan de cette approche.

Dans notre cas, la structure de données représente un document texte. Les données manipulées par notre algorithme sont donc des blocs ou chaînes de caractères, le caractère étant la plus petite donnée manipulable. En pratique, notre CRDT peut être utilisé avec n'importe quelle structure de données linéaire.

5.1 Principe et Modèle

5.1.1 Modèle de données

Notre modèle est un ensemble d'éléments qui admet trois types d'opérations. L'opération d'insertion, qui ajoute un élément supplémentaire à l'ensemble. L'opération de suppression, qui

retire un élément de l'ensemble. L'opération de scission, qui coupe un élément en deux.

Ce modèle reprend le principe développé dans l'algorithme LOGOOT [Weiss *et al.*, 2009], en affectant aux blocs de caractères des identifiants uniques issus d'un ensemble ordonné et dense, la vue du modèle (le document texte) étant créée en affichant les blocs ordonnés selon leurs identifiants. Un élément est donc un couple (identifiant,bloc).

5.1.2 Identifiants

Nos identifiants sont composés de deux parties. La première est une suite d'entiers, que l'on nommera *Base* de l'identifiant. Elle correspond à l'identifiant global de l'élément. Les entiers composants cette base doivent être compris entre deux entiers donnés (min, Max) (ou égaux à ceux-ci), tels que $min < 0 < Max$. La seconde partie est un couple d'entiers que l'on nommera *Intervalle* de l'identifiant. Elle correspond à l'identifiant des parties de l'élément. Les deux entiers de ce couple doivent se situer entre les entiers min exclu et Max. On définit également le *début* (respectivement la *fin*) de l'identifiant comme étant la liste d'entiers constituée de sa base suivie de la borne inférieure (respectivement supérieure) de son Intervalle. De plus, on considère que chaque site maintient une horloge locale *clock*, qui est incrémentée à chaque génération d'identifiant, et que chaque site possède également un entier unique *id* qui le représente. Les deux derniers entiers de la base de chaque identifiant doivent correspondre au couple *id*,*clock* du site qui l'a généré. La valeur initiale de l'horloge peut être n'importe quel entier strictement supérieur à min et inférieur à Max. Les identifiants uniques de site sont également compris entre min exclu et Max. Pour un algorithme efficace, il convient de prendre un intervalle $[min, Max]$ relativement grand.

Dans la pratique, on ne compare pas des identifiants mais des débuts et des fins d'identifiants, c'est à dire des listes d'entiers. Elles sont simplement comparées de manière lexicographique. Par exemple, le début d'un identifiant ayant pour base la liste 1,1 et pour intervalle [0,1] sera plus petit que sa fin (1,1,0 est plus petit que 1,1,1). Cet identifiant (début ou fin) est également plus petit que n'importe quel élément ayant pour base 1,2. Quant à un identifiant ayant pour base 1,1,0,1,3, il se trouvera entre le début et la fin du premier élément de l'exemple.

5.1.3 Génération des identifiants et insertion

Un utilisateur peut insérer une chaîne de caractères dans le modèle à partir de la vue (le document texte complet), en plaçant son curseur à la position voulue et en y écrivant la chaîne en question. La chaîne fera partie du nouvel élément à insérer.

Le début de l'insertion nécessite de faire appel à une fonction de recherche, qui prend la position de la vue où insérer en paramètre et qui retourne la position correspondante du modèle, c'est-à-dire entre quels éléments ou à l'intérieur de quel élément il faut insérer.

Si la position de cet élément dans la vue est entre deux éléments, on génère un identifiant plus petit que le début de celui de l'élément de droite et plus grand que la fin de celui de l'élément de gauche, avec pour intervalle [0,x-1], où x est la longueur de l'élément (le nombre de caractères

de la chaîne). Si x-1 est plus grand que Max, on génère un élément pour les Max premiers caractères de la chaîne, puis on relance l'algorithme d'insertion pour les caractères manquants.

Dans le cas où la position est à l'intérieur d'un élément, il faut au préalable le scinder en deux. Les deux parties de l'élément scindé gardent la même base, mais leur intervalle est mis à jour. Si la nouvelle chaîne se situe entre les positions x et x+1 de l'élément, les deux parties de l'élément auront pour intervalle [d,d+x] et [d+x+1,f], où d et f représentent les bornes inférieures et supérieures respectivement de l'intervalle de l'identifiant initial. Cette scission est répercutée dans le modèle, et l'on peut ensuite générer un identifiant pour l'élément qui s'insère entre ces deux parties. Pour reprendre les exemples donnés précédemment, la scission de 1,1,[0,1] entre les positions 0 et 1 mène aux identifiants 1,1,[0,0] et 1,1,[1,1], et 1,1,0,1,3 correspond à une possibilité de base de l'identifiant qui est généré entre les deux, si c'est la troisième insertion du site numéro 1. La fonction *scinder* (figure 5.1) réalise cette opération. Elle prend en paramètre l'élément à scinder (constitué de sa chaîne de caractère, de sa base, de la borne inférieure de son intervalle et de sa borne supérieure) et la position dans l'intervalle où scinder (x+1 dans notre exemple). La fonction *add*(*newel*) ligne 7 ajoute l'élément dans le document trié. Si la structure de données est une liste d'éléments, cette fonction peut se composer d'une recherche dichotomique, qui scinde éventuellement un élément pour insérer ensuite *newel* entre ces deux parties.

```
1 function scinder(el, pos) :
2   newel = el.clone()
3   el.sup = pos − 1 + el.inf
4   el.chaine = el.chaine[0, pos − 1]
5   newel.inf = el.inf + pos
6   newel.chaine = newel.chaine[pos, newel.chaine.size() − 1]
7   document.add(newel)
8   return newel
9 end
```

FIGURE 5.1 – Fonction *scinder*

La fonction *generateBase* (figure 5.2) détaille l'algorithme de génération des bases des identifiants. Elle prend en paramètres deux listes d'entiers, qui correspondent à la fin de l'identifiant de l'élément de gauche et au début de l'élément de droite. La précondition de cette fonction est que la première liste d'entiers doit être plus petite strictement que la deuxième, au sens lexicographique. La fonction $rand(x, y)$ utilisée retourne un entier aléatoire entre x et y exclus. Les variables *site* et *clock* correspondent aux valeurs de l'identifiant unique et de l'horloge locale du site qui fait appel à cette fonction. Les constantes min et Max sont celles définies plus haut.

La ligne 2 déclare une liste d'entiers qui sera remplie au fur et à mesure de l'algorithme et qui contiendra la base que l'on cherche à créer. La boucle ligne 4 à 7 compare chaque entier des deux identifiants un à un, dans le but d'identifier le plus grand préfixe commun des deux identifiants.

```
1  function generateBase(((p_0, p_1...p_{n-1})), ((q_0, q_1...q_{m-1}))) :
2   base = ()
3   i = 0
4   while (i < n and p_i = q_i)  //on extrait le prefixe commun
5      base.add(p_i)
6      i++
7   end
8   if (i = n)  //la première liste est préfixe de la deuxième
9      while (q_i = min)
10        base.add(min)
11        i++
12     end
13     if (q_i = min + 1)
14        base.add(min, site, clock++)
15     else if (q_i > site)
16        base.add(site, clock++)
17     else
18        base.add(rand(min, q_i), site, clock++)
19     endif
20  else  //la première liste n'est pas préfixe de la deuxième
21     if (p_i < site < q_i)
22        base.add(site, clock++)
23     else if (q_i - p_i > 1)
24        base.add(rand(p_i, q_i), site, clock++)
25     else
26        base.add(p_i)
27        i++
28        if (i = n)
29           base.add(site, clock++)
30        else
31           while (p_i = Max)
32              base.add(Max)
33              i++
34           end
35        endif
36        base.add(rand(p_i, Max + 1), site, clock++)
37     endif
38  endif
39  return base
40 end
```


FIGURE 5.2 – Fonction *generateBase*

La nouvelle base commencera également par ce préfixe, qui sera ensuite complété pour aboutir à un identifiant plus petit (au sens de la relation qui ordonne l'ensemble) que l'identifiant de droite et plus grand que l'identifiant de gauche. Le cas ligne 8 à 19 est le cas particulier où l'identifiant de l'élément de gauche est préfixe de l'identifiant de l'élément de droite, le cas ligne 21 à 36 est le cas général.

L'ensemble des listes d'entiers muni de la relation d'ordre lexicographique que nous utilisons n'est pas un ensemble dense. Par exemple, on ne peut pas trouver d'élément situé entre 1,1, et 1,1,min puisque min est le plus petit entier. Toutefois, par construction nos identifiants ne se terminent jamais par cette valeur minimale. Le sous-ensemble dans lequel nous travaillons est bien dense.

L'algorithme complet d'une insertion locale est donc celui disponible ci-dessous (figure 5.3). La fonction $voisins(document, p)$ renvoie les deux éléments qui sont de part et d'autre de la position p du document. Si la position est à l'intérieur d'un élément, cette fonction *voisins* scinde l'élément avec la fonction *scinder* et renvoie les deux parties de l'élément scindé. Si par exemple la structure de données du document est une liste d'éléments, la fonction *voisins* peut parcourir la liste en sommant la taille des chaînes de caractères des éléments jusqu'à atteindre p. La fonction renvoie l'élément inséré, pour qu'il soit envoyé aux sites distants.

```
1 function LocalInsert(chaine, p) :
2   (el_1, el_2) = voisins(document, p)
3   base = generateBase(el_1, el_2)
4   el = (chaine, base, 0, chaine.size() − 1)
5   document.add(el)
6   return el
7 end
```

FIGURE 5.3 – Fonction *LocalInsert*

Une fois le nouvel identifiant associé à l'élément, il peut être envoyé aux sites distants pour intégration. Ces derniers vont l'ajouter à leurs éléments déjà présents en respectant l'ordre des identifiants, grâce à la fonction *add* décrite plus haut (figure 5.4).

```
1 procedure RemoteInsert(el) :
2   document.add(el)
3 end
```

FIGURE 5.4 – Procédure *RemoteInsert*

On peut noter que le résultat de la scission ne dépend que de l'élément scindé et de la position, et non du site qui la réalise. La figure 5.5 illustre une situation où deux sites réalisent une insertion qui nécessite une scission du même élément en concurrence. Chaque site commence

par scinder l'élément à la position de l'insertion, puis lorsque le deuxième élément est reçu la partie appropriée est scindée à nouveau. Au final les deux modèles convergent.

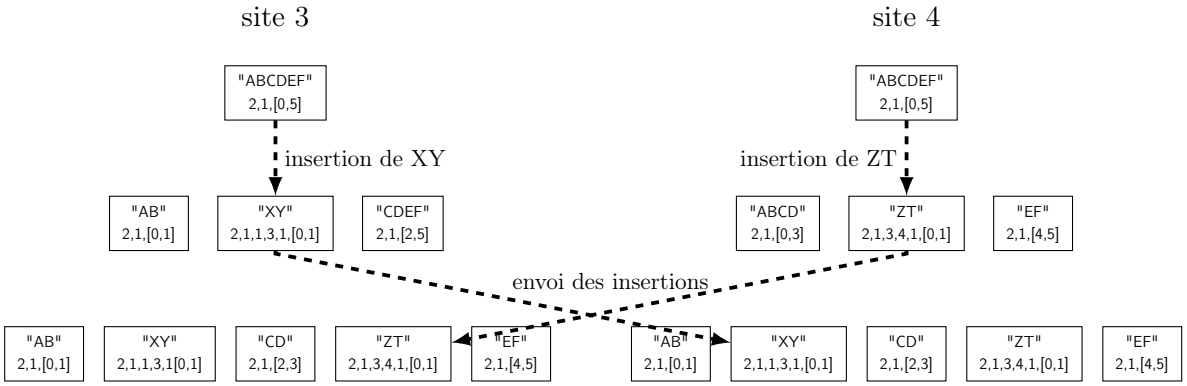


FIGURE 5.5 – Génération et intégration de deux insertions

Pour favoriser la génération d'éléments de grande taille tout en permettant la mise à jour du modèle de manière très fréquente, une deuxième méthode d'insertion, que nous appellerons *append*, permet de compléter un élément déjà inséré plutôt que d'en créer un nouveau. Cela consiste à étendre l'intervalle d'un identifiant et à modifier l'élément associé en y concaténant la chaîne à insérer. Concrètement, du point de vue local, l'algorithme de génération de base n'est pas appelé. A la place, la base d'un voisin est copiée et l'intervalle du nouvel élément suit ou précède immédiatement celui du voisin. Du point de vue distant, lorsque l'élément est reçu, on cherche à le placer normalement dans le modèle, puis on vérifie si il peut étendre ses voisins de gauche ou de droite, auquel cas on étend l'élément au lieu d'en insérer un nouveau.

Par exemple, on peut étendre l'identifiant 1,1,[0,2] en 1,1,[0,4] si on rajoute un élément de deux caractères après lui. On envoie aux sites distants l'identifiant 1,1,[3,4], qui déterminera ensuite que cet identifiant se situe juste après l'élément d'identifiant 1,1,[0,2], et que comme les bases sont les mêmes et les intervalles contigus, on peut étendre l'élément. On aurait aussi pu étendre par la gauche et générer 1,1,[-2,2] si l'insertion avait été avant 1,1,[0,2]. On ne peut pas étendre un intervalle en dehors des bornes (min, Max).

Cette technique n'est pas toujours possible et nécessite de prendre quelques précautions. Tout d'abord, pour empêcher les problèmes de concurrence et le cas où deux sites voudraient étendre le même élément, un site ne peut générer un identifiant étendu que si l'identifiant qu'il étend a été généré par lui-même, ce qui se détecte en regardant l'avant-dernier entier de la base de l'identifiant, correspondant à l'identifiant unique de site. Ensuite, on ne peut pas étendre un élément en utilisant un identifiant qui a déjà été généré et qui se trouve actuellement supprimé. Par exemple, si un site insère l'élément 1,1,[0,4] puis supprime les deux derniers caractères de cet élément, conduisant à l'élément 1,1,[0,2], cet élément ne peut plus être étendu par la droite puisqu'il réintroduirait l'élément 1,1,[3,3], qui a été supprimé, ce qui conduirait à une incohérence. Par exemple, au départ, un document contient le bloc de caractères “AB”, qui avait été inséré

par le site 1. Puis le site 1 supprime “B” et insère le caractère “C” à la place. Si on réalise un *append* à ce moment, le caractère “C” est associé au même identifiant Id_B que le caractère “B” supprimé. En parallèle, le site 3 supprime lui aussi le caractère “B”. Lorsque ces trois opérations vont être reçues par le site 2, si il intègre d'abord les opérations du site 1 puis celle du site 3, le caractère “C” va être supprimé alors qu'aucun site n'a généré cette opération. En effet, lorsque la suppression de l'identifiant Id_B et du caractère “B” arrive sur le site 2, c'est le caractère “C” qui est associé à Id_B et c'est donc ce dernier qui est retiré du modèle.

Ce mécanisme semble utile lors de la saisie en continu d'un grand bloc de texte mais a peu de chances d'être déclenché lors de petites éditions d'un texte déjà construit.

5.1.4 Suppression

Lorsqu'une suppression est réalisée, on doit récupérer les identifiants des éléments ou parties d'éléments qui la composent : une suppression d'une chaîne de caractères peut conduire à la génération de plusieurs suppressions d'éléments dans le modèle, si cette chaîne est contenue dans plusieurs éléments contigus. Pour obtenir ces éléments et leurs identifiants, on utilise une fonction de recherche qui renvoie tous les éléments entre deux positions du document données, puis on génère une suppression par élément trouvé. Dans le cas d'éléments qui ne sont supprimés qu'en partie, une scission a lieu. Par exemple, supprimer les trois premiers caractères d'un élément qui a pour identifiant 1,1,[0,8] génère la suppression de l'identifiant 1,1,[0,2]. La deuxième partie 1,1,[3,8] reste dans le modèle. De la même façon, supprimer le milieu d'un élément nécessite deux scissions, conduit à trois identifiants, celui du milieu étant celui à retirer. La méthode *LocalDelete* (figure 5.6) qui réalise une suppression locale est présentée ci dessous. Les éléments supprimés sont retournés à la fin de la méthode pour envoyer ensuite les ordres de suppression concernant ces éléments aux sites distants.

```
1 function LocalDelete(p1, p2) :
2   elements = between(document, p1, p2)
3   for el in elements
4     document.remove(el)
5   end
6   return elements
7 end
```

FIGURE 5.6 – Fonction *LocalDelete*

Une fois toutes les suppressions générées, elles peuvent être envoyées aux sites distants pour intégration. On réalise une suppression pour chaque élément qui doit être supprimé. Si une suppression a provoqué une scission lors de sa génération, ou si une scission a été réalisée à cause de modifications concurrentes, il est possible que le modèle distant ne contienne pas l'identifiant de l'élément supprimé. Par exemple, on peut vouloir supprimer l'élément ayant pour identi-

fiant 1,1,[0,4] alors qu'un site distant a en concurrence inséré l'identifiant 1,1,2,2,2,[0,1], scindant l'identifiant 1,1,[0,4] en 1,1,[0,2] et 1,1,[3,4]. L'élément ayant pour identifiant 1,1,[0,4] n'est plus disponible sur ce site distant. Toutefois, des identifiants ayant pour base 1,1 sont toujours présents, et on peut retrouver quels sont ceux qui doivent être scindés ou supprimés pour satisfaire la suppression initiale. Dans notre exemple, supprimer 1,1,[0,4] revient sur le site distant à supprimer 1,1,[0,2] et 1,1,[3,4]. On peut donc retrouver où scinder l'élément pour supprimer la partie adéquate.

La méthode *RemoteDelete* (figure 5.7) réalise ces opérations. Elle cherche initialement tous les éléments dans le modèle qui ont un identifiant ayant pour base la même base que l'identifiant à supprimer, les parcourt et les scinde si besoin avant de supprimer la partie contenue dans la suppression.

```
1  procedure RemoteDelete(el) :
2   elements = trouverBase(document, el)
3   for elt in elements
4    if (el.debut ≤ elt.debut and elt.fin ≤ el.fin)
5      document.remove(elt)
6    else if (elt.debut < el.debut and el.debut ≤ elt.fin ≤ el.fin)
7      newel = scinder(elt, el.debut − elt.debut)
8      document.remove(newel)
9    else if (el.debut ≤ elt.debut ≤ el.fin and elt.fin > el.fin)
10     scinder(elt, el.fin − elt.debut + 1)
11     document.remove(elt)
12   else if (elt.debut < el.debut and el.fin < elt.fin)
13     scinder(elt, el.fin − elt.debut + 1)
14     newel = scinder(elt, el.debut − elt.debut)
15     document.remove(newel)
16   endif
17  end
18 end
```

FIGURE 5.7 – Procédure *RemoteDelete*

La figure 5.8 illustre une situation où deux sites réalisent une suppression en concurrence. La suppression du premier site génère deux suppressions d'éléments, et celle du second a lieu au milieu d'un élément. Les deux zones supprimées se chevauchent. A la réception de la suppression du site 2, le site 1 cherche dans son modèle les éléments dont l'identifiant a la même base que celui de la suppression, compare leurs intervalles avec celui de la suppression, et déduit qu'il faut supprimer le début de l'élément de droite. Quant au site 2, il procède de la même manière avec les deux suppressions envoyées par le site 1 pour déterminer qu'il doit supprimer la fin de

l'élément de gauche et son élément central.

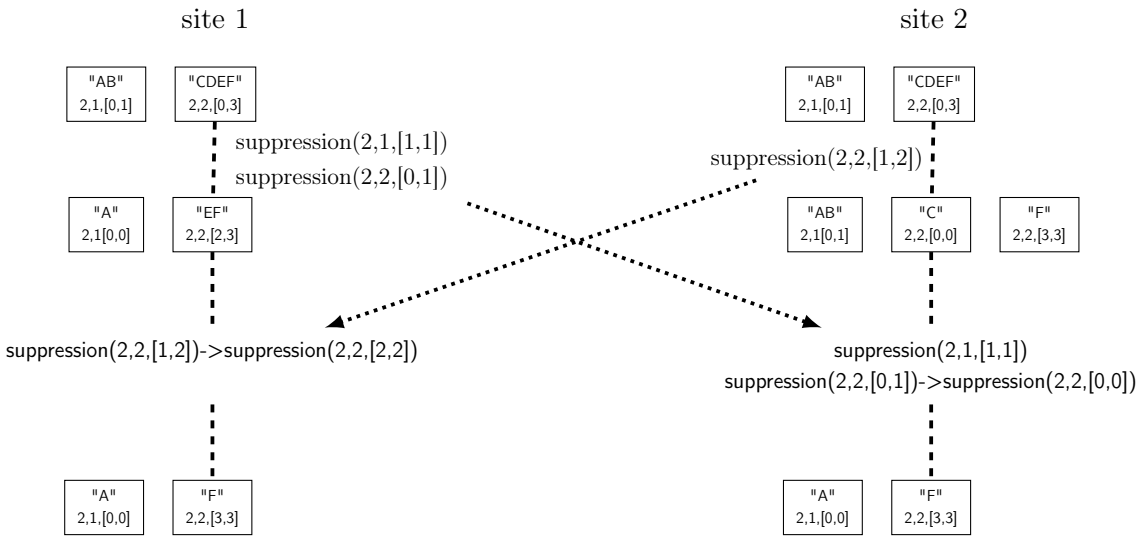


FIGURE 5.8 – Génération et intégration de deux suppressions

5.1.5 Scénario explicatif

La figure 5.9 présente un scénario d'utilisation plus complexe entre deux sites, qui mêle insertions et suppressions. Le site numéro 1 commence par ajouter le texte "CDE" dans un document vide, générant l'identifiant 1,1,[0,2] ; le site 2 reçoit et intègre cet élément. Ensuite, le site 1 ajoute "AB" devant "CDE" tandis que le site 2 ajoute "FGH" après. C'est un *append* pour le premier site, qui complète l'identifiant 1,1,[0,2] avec l'identifiant 1,1,[-2,-1], et une insertion standard pour le site 2, puisque l'élément contenant "CDE" n'a pas été créé par lui-même. L'identifiant 2,1,[0,2] est créé. Puis, ces deux modifications sont transmises à l'autre site et intégrées. Après ceci, le site 1 supprime les caractères "BCD", générant la suppression de l'identifiant 1,1,[-1,1], qui conduit à un modèle contenant les identifiants 1,1,[-2,-2] et 1,1,[2,2], pendant que le site 2 insère "XY" entre "B" et "C", ce qui génère l'identifiant 1,1,-1,2,2,[0,1]. La prochaine étape est l'envoi et l'intégration de ces modifications par les sites. Lorsque le site 1 reçoit l'insertion de "XY", il l'insère entre "A" et "E" puisque l'identifiant associé à "XY" est entre la fin de 1,1,[-2,-2] et le début de 1,1,[2,2]. Lorsque le site 2 reçoit la suppression de "BCD", qui correspond à l'identifiant 1,1,[-1,1], il cherche dans son modèle le début d'identifiant 1,1,-1, le trouve associé au deuxième caractère de l'élément ayant pour identifiant 1,1,[-2,-1], supprime le maximum de caractères possible de l'élément sans dépasser la fin de l'intervalle à supprimer (c'est à dire il supprime "B" et l'identifiant 1,1,[-1,-1]) et relance la suppression avec l'identifiant 1,1,[0,1]. Ce coup-ci la recherche de 1,1,0 est lancée, conduit au premier caractère de l'élément 1,1,[0,2], et à la suppression des deux premiers caractères, donc de l'élément 1,1,[0,1]. La suppression est maintenant totalement intégrée.

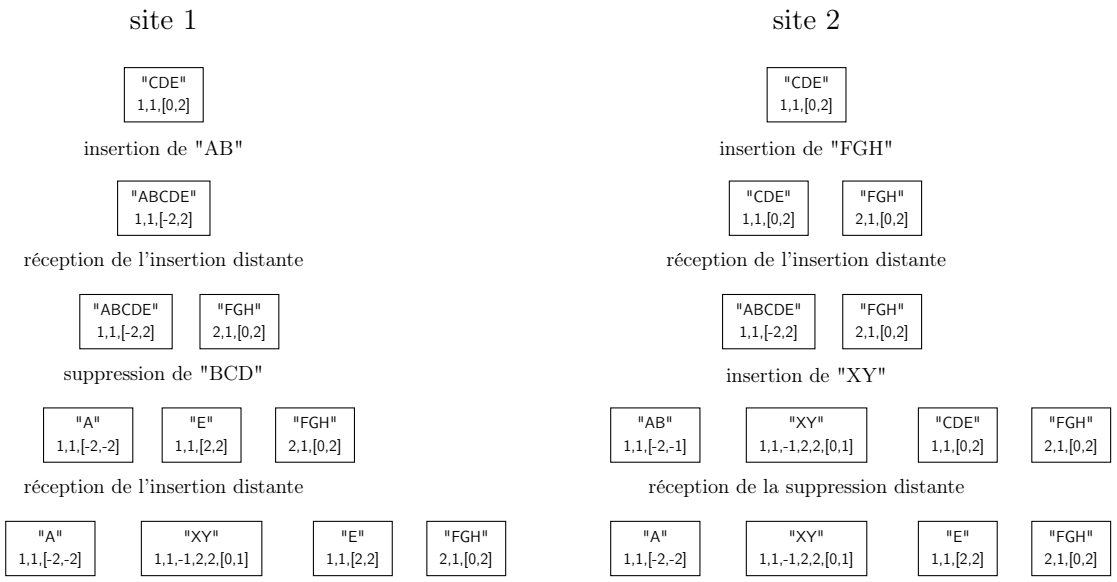


FIGURE 5.9 – Exemple d'édition

5.1.6 Respect des critères CCI

Pour montrer que notre algorithme est correct, nous allons vérifier qu'il respecte les critères du modèle CCI [Sun *et al.*, 1998] : Causalité, Convergence à terme, respect des Intentions des opérations.

Concernant les intentions des opérations, notre CRDT (comme tous les CRDTs) n'a pas besoin de transformer ses opérations. Les deux opérations à considérer sont les insertions et les suppressions. L'opération de scission n'est qu'une étape contenue dans ces opérations. Lorsqu'une insertion est générée, son intégration future sur un site distant sera également une insertion, à la même position (donnée par l'identifiant de l'élément à insérer). Lorsqu'une suppression est générée, son intégration future sera également une suppression, de la même partie du document (des mêmes identifiants).

La convergence à terme est aussi respectée. Comme toutes les opérations concurrentes commutent, si deux sites ont reçu exactement le même ensemble d'opérations, alors leur état est semblable.

Concernant la causalité, le mécanisme de diffusion des opérations n'est ici pas abordé, et laissé à la discrétion de l'implémenteur. Ce critère est important pour les algorithmes OT, qui ont besoin de connaître l'ordre de création des opérations pour les transformer ensuite. Notre algorithme ne requiert pas de diffusion causale des opérations, ce qui facilite son intégration, mais en l'absence de diffusion causale, les ajustements qui suivent sont à prévoir.

Tel qu'il a été présenté, notre algorithme ne permet pas de dissocier le cas de deux suppressions concurrentes du même élément, et le cas d'une opération de suppression d'un élément qui n'a pas été reçu. En effet, dans les deux cas une suppression va être incapable de trouver les parties

d'éléments qu'elle doit supprimer lors de son intégration, et va interpréter cela comme étant le premier cas. Dans le deuxième cas, la suppression n'est donc jamais effectuée. Pour corriger le problème, on peut par exemple maintenir une table des bases complètement supprimées du modèle. Une entrée est ajoutée à cette table lorsqu'une suppression supprime la dernière partie d'un élément, c'est à dire lorsque sa base n'apparaît plus dans le modèle. Cette table est consultée lorsqu'une suppression n'aboutit à aucune modification, pour vérifier si l'élément qu'elle cible a déjà été totalement supprimé (premier cas) ou n'a pas encore été reçu (deuxième cas). Dans ce deuxième cas, il faut également stocker la suppression dans une table des suppression en attente, et tenter régulièrement d'intégrer ces suppressions.

Il est également possible qu'une insertion supposée scinder un élément en deux soit reçue avant l'élément qu'elle scinde. Dans ce cas, lorsque l'on intègre un nouvel élément dans le modèle, il faut vérifier si tout ou seulement une partie de l'élément entre à la position donnée par l'algorithme de recherche, n'insérer que la partie qui s'y insère, et relancer l'algorithme avec l'autre partie si besoin.

Ces ajustements permettent de garantir une véritable intégration causale, par opposition à un mécanisme de diffusion causale qui se baserait sur la relation *happens-before*, qui est plus forte que la causalité.

5.2 Implémentation et comparaison

Le but premier de notre algorithme est de proposer une solution qui évite d'avoir à choisir entre une granularité importante des éléments (lignes, paragraphes), qui conduit à des mises à jour produisant des duplicatas mais nécessitant peu d'identifiants, et une granularité fine (caractères) qui peut conduire à des entrelacs tout en nécessitant beaucoup d'identifiants.

Notre solution permet d'insérer des éléments de taille importante, et permet ensuite de les scinder pour éviter le problème des mises à jour en cas de granularité élevée.

Par rapport aux autres approches de type CRDT, principalement Logoot, on peut remarquer que même si notre approche est conçue pour minimaliser le nombre d'identifiants dans le modèle, ces derniers croissent rapidement en taille et en nombre. En effet, notre approche favorise les identifiants qui représentent un grand nombre de caractères, on peut supposer que les insertions vont majoritairement se trouver à l'intérieur d'autres éléments, et donc conduire à des scissions. Ceci va d'une part ajouter un identifiant supplémentaire lors de chaque scission, puis générer une base plus longue que l'identifiant scindé pour le nouvel élément. De plus, une suppression est susceptible d'augmenter le nombre d'identifiants au lieu de de le réduire : la suppression d'un milieu d'élément va laisser deux éléments dans le modèle, un pour la partie de droite et un pour la partie de gauche.

Nous avons donc réalisé des expérimentations pratiques issues de traces paramétrées générées aléatoirement pour avoir un aperçu du comportement de notre approche. Dans nos expérimentations, trois implémentations de l'algorithme sont considérées, et comparées aux autres CRDTs connus. La suite de cette section présente ces trois implémentations, donne la complexité théo-

rique des fonctions d'insertion et de suppression qui leur sont associées, et présente les résultats pratiques obtenus.

5.2.1 Implémentation naïve

La première implémentation, référencée LogootSplitNaive, stocke les couples (éléments, identifiants) dans une simple liste, ordonnée selon les identifiants. Lors de la génération locale d'une opération, la fonction de recherche d'éléments par position dans la vue s'exécute en un temps $\mathcal{O}(l)$, l étant le nombre d'éléments dans la liste (sa longueur). En effet, cette recherche s'effectue en additionnant la taille de chaque élément un par un depuis le début, jusqu'à atteindre la valeur désirée. La deuxième fonction de recherche, qui renvoie une position dans le modèle en fonction de l'élément passé en paramètre, s'exécute en un temps $\mathcal{O}(i \times \log(l))$, où i est la longueur d'un identifiant. En effet, on utilise une recherche dichotomique sur les identifiants, et une comparaison de deux identifiants nécessite la comparaison de tous les entiers qui les composent. Cette fonction est utilisée à la fois pour la recherche d'éléments présents dans le modèle (typiquement, lors d'une suppression distante), et pour la recherche de l'emplacement où insérer un nouvel élément (lors d'une insertion distante). Elle compare le début de l'identifiant en paramètre avec le début et la fin de l'identifiant du milieu du modèle. En fonction des résultats des comparaisons, on relance la recherche soit dans la partie gauche du modèle, soit dans la partie droite, soit on termine et scinde (lorsque l'identifiant en paramètre est à l'intérieur de l'identifiant considéré). On termine également si l'identifiant en paramètre a le même début que l'identifiant du modèle en train d'être analysé (on a trouvé l'élément à supprimer), ou si on ne peut plus réduire l'intervalle (on a trouvé où insérer l'élément).

De la complexité de ces deux fonctions, on peut déduire la complexité des fonctions d'insertion et de suppression. Une insertion locale nécessite une recherche par position, éventuellement une scission qui se réalise en temps constant, et une génération d'identifiant, qui d'après notre algorithme de génération de base s'exécute en un temps $\mathcal{O}(i)$. Ce qui conduit à une complexité globale de $\mathcal{O}(l + i)$. Une insertion distante utilise la recherche par identifiant, réalise une scission si besoin, et place l'élément à l'endroit trouvé, ce qui nous donne simplement une complexité équivalente à celle de la recherche, c'est-à-dire $\mathcal{O}(i \times \log(l))$. Dans le cas présenté précédemment où la réception causale n'est pas assurée, il est possible de devoir faire appel plusieurs fois à cette fonction d'insertion distante pour un seul élément à insérer. Une suppression locale emploie la recherche par position, récupère les identifiants à supprimer et retire les éléments associés de l'ensemble, ce qui donne une complexité de $\mathcal{O}(l)$. Une suppression distante se fait à partir de la recherche par identifiant. Comme vu au début de ce chapitre, la recherche peut échouer. Si le début de l'élément a déjà été supprimé par une opération concurrente, on ne le trouvera pas. Dans ce cas on relance l'algorithme de suppression, avec un identifiant qui a un début plus élevé. Dans le pire des cas, tout l'élément a déjà été supprimé, et on relance l'algorithme s fois, où s est la longueur de l'élément (la longueur de la chaîne de caractères), ce qui conduit à une complexité de $\mathcal{O}(s \times i \times \log(l))$.

5.2.2 Implémentation en chaîne

Notre seconde implémentation, nommée LogootSplitString, a pour but de réduire le temps de la recherche par position. Nous stockons cette fois les caractères dans un tableau, et non plus les éléments directement. Chaque caractère est associé à la base de son identifiant, et à sa position dans l'intervalle de l'identifiant. La figure 5.10 illustre ce modèle. Elle contient des éléments associés aux trois identifiants 4,1,[0,3], 4,1,1,3,1,[0,3] et 5,1,[0,2]. Le premier élément a été scindé suite à l'insertion du deuxième. Chaque caractère est accessible directement via le tableau du haut, il est lié à sa position initiale dans l'intervalle de l'identifiant, et à la base de l'identifiant. Les bases sont simplement stockées dans une table de hachage.

La figure 5.11 reprend l'exemple développé lors du scénario explicatif général mais avec la structure en chaîne. Seuls les états successifs du site numéro 1 sont représentés.

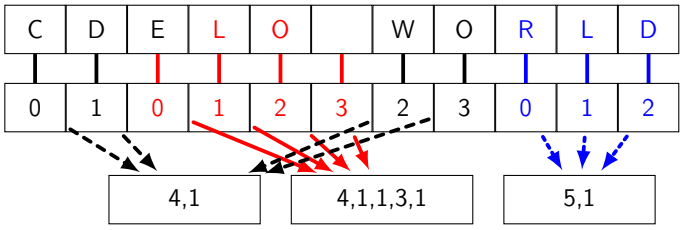


Figure 5.10 – Structure d'implémentation en chaîne

La fonction de recherche par position s'opère en temps constant, en renvoyant directement l'élément du tableau qui est à la position voulue. La fonction de recherche par identifiant se déroule de la même manière que pour l'implémentation naïve, mais sur les caractères et non sur les éléments. Sa complexité est donc $\mathcal{O}(i \times \log(n))$, où n représente la longueur du document.

Dans ce modèle, une insertion locale nécessite une recherche par position, une génération d'identifiant, mais également la création des liens depuis les caractères insérés vers la base. La recherche est en temps constant, la génération en $\mathcal{O}(i)$, et le reste en $\mathcal{O}(s)$, ce qui mène à une complexité globale de $\mathcal{O}(s + i)$. De même, pour une insertion distante, en plus de la recherche par identifiant, il faut réaliser le même travail de création de liens une fois l'élément inséré. La complexité en temps est $\mathcal{O}(i \times \log(n) + s)$. En ce qui concerne les suppressions, une suppression locale s'effectue en temps constant, en supprimant directement les éléments entre les positions recherchées. Une suppression distante suit le même schéma que précédemment : une recherche par identifiant, éventuellement relancée si l'identifiant n'a pas été trouvé, ce qui conduit à une complexité égale à $\mathcal{O}(i \times s \times \log(n))$.

Par rapport à l'implémentation naïve, cette implémentation est théoriquement plus rapide sur le plan des opérations locales, mais plus lente en ce qui concerne les opérations distantes. Elle a également besoin de tableaux plus longs.

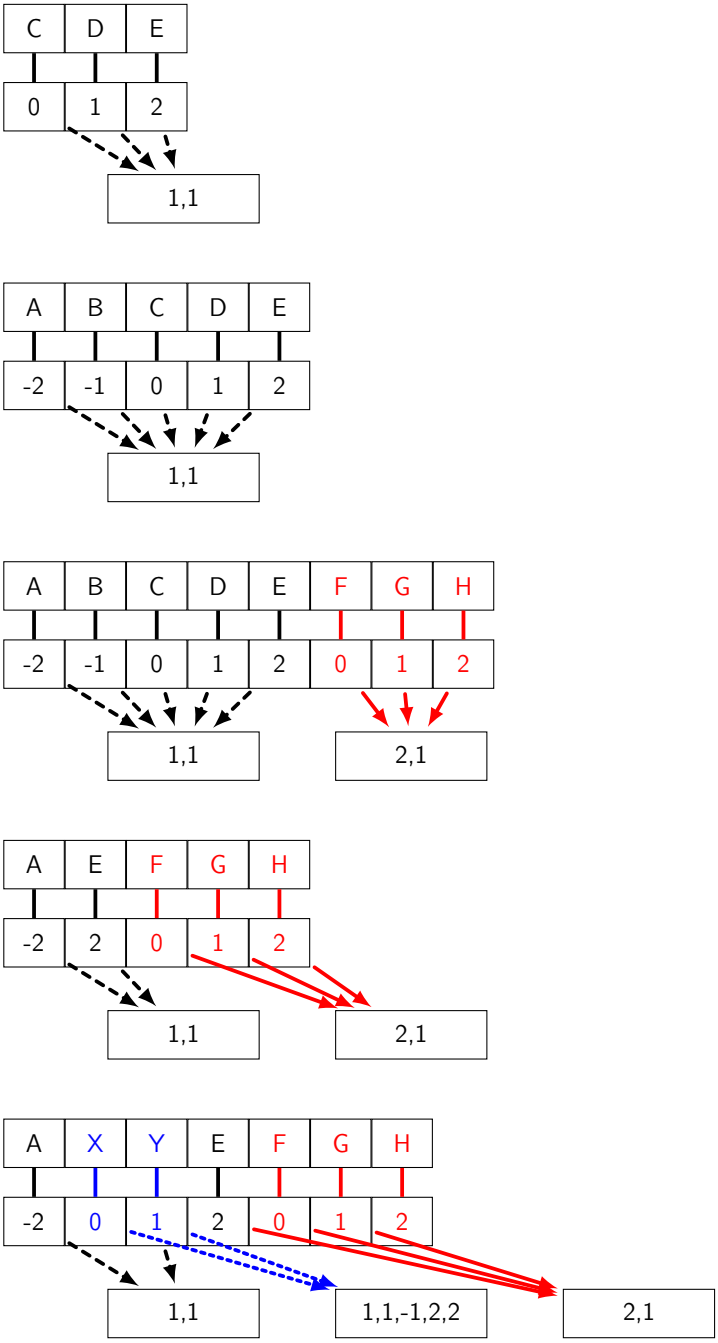


FIGURE 5.11 – Scénario complexe avec la structure en chaîne

5.2.3 Implémentation en arbre

La troisième implémentation utilise un arbre équilibré AVL [Adelson-Velskii and Landis, 1962] pour structure principale, et sera appelée LOGOOTSPLITAVL. Les éléments sont stockés dans les nœuds d'un arbre binaire, le sous-arbre contenu dans le fils gauche d'un nœud contient les éléments situés avant lui (les identifiants sont plus petits), et ceux du fils droit sont situés après

(les identifiants sont plus grands).

Les identifiants des éléments sont représentés comme dans la seconde implémentation : les bases sont stockées dans une table de hachage et un nœud contient une chaîne de caractères, un lien vers la base de l'identifiant et la position de début de l'intervalle.

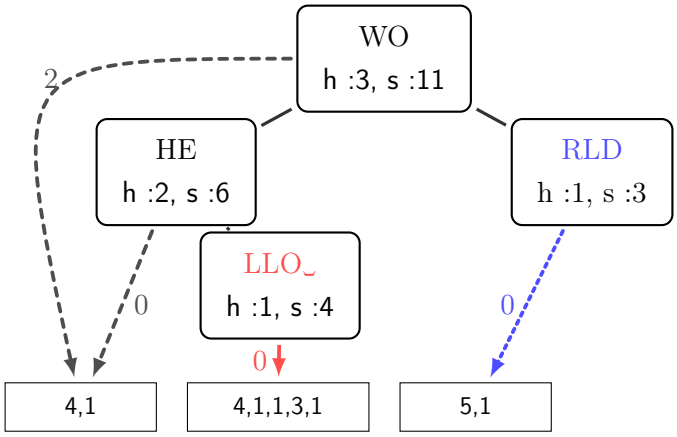


FIGURE 5.12 – Structure d'implémentation en arbre

Les nœuds de l'arbre sont étiquetés avec la taille de la chaîne de caractères contenue dans le sous-arbre qui a pour racine ce nœud et la hauteur de ce sous-arbre. La figure 5.12 présente ce modèle. Le modèle contenait un élément ajouté par le site 4 ayant pour chaîne "HEWO", associé à l'identifiant 4,1,[0,3]. La chaîne "LLO " a été ajoutée par le site 3 à l'intérieur de cet élément pour former "HELLO WO", associée à l'identifiant 4,1,1,3,1,[0,3]. Ceci a conduit à une scission, il y a deux nœuds noirs, qui ont la même base mais un intervalle différent (dont le début est représenté sur la flèche et dont la fin se déduit d'après la longueur de la chaîne du nœud). Enfin, un dernier élément "RLD" a été ajouté par le site 5. Au final, l'arbre a une hauteur de 3 et contient 11 caractères, le sous-arbre gauche est de hauteur 2 et contient 6 caractères, et le sous-arbre droit est de hauteur 1 et contient 3 caractères, comme illustré sur la figure.

La figure 5.13 représente l'évolution du modèle en arbre lors de l'exécution du scénario explicatif.

Lorsque la structure de l'arbre est modifiée, des rotations sont effectuées pour s'assurer que l'arbre binaire reste équilibré.

La fonction de recherche par position s'effectue en un temps $\mathcal{O}(\log(l))$, grâce à l'étiquette qui contient la taille du sous arbre. On parcourt l'arbre depuis la racine, en regardant d'abord la taille t_g du sous-arbre gauche. Si elle est supérieure à la position p recherchée, on relance la recherche depuis le fils gauche. Sinon, on met à jour p en retranchant t_g, et on compare ce nouveau p à la taille t_c de la chaîne contenue dans le nœud. Si p est la valeur la plus petite, la position est à l'intérieur de l'élément de ce nœud. Sinon on met à jour p en retranchant t_c et on relance la recherche depuis le fils droit. L'arbre étant équilibré, cela s'apparente à une recherche dichotomique, la complexité est logarithmique.

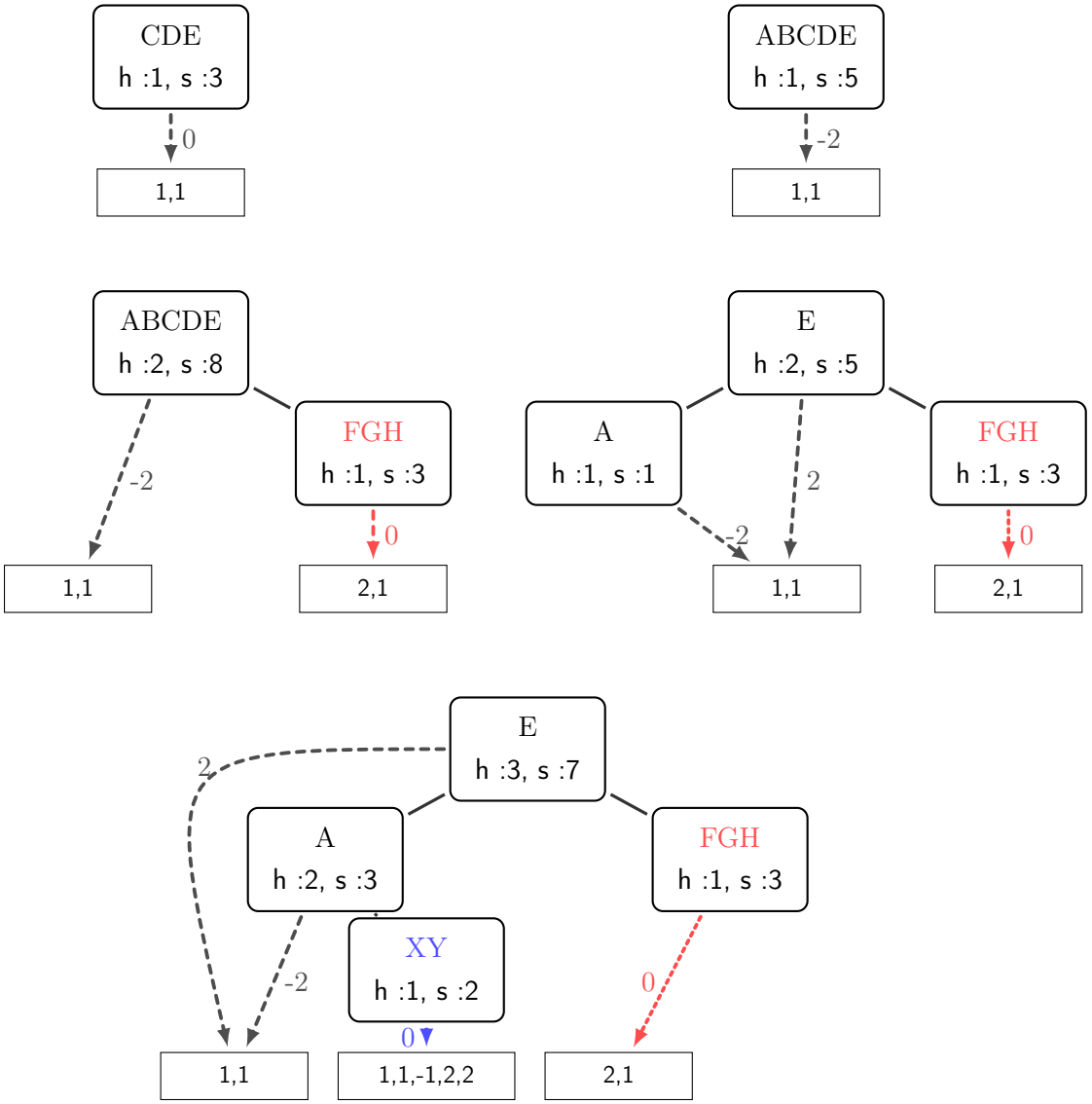


FIGURE 5.13 – Scénario complexe avec la structure en arbre

La fonction de recherche par identifiant se déroule de la même manière. Depuis la racine, on compare l'identifiant en paramètre avec l'identifiant contenu dans le nœud. On relance la recherche avec le fils gauche si le paramètre est plus petit que le début de l'identifiant du nœud, avec le fils droit si le paramètre est plus grand que la fin du nœud, sinon l'identifiant est à l'intérieur du nœud (possible dans le cas où la recherche vise à déterminer où insérer un nouvel élément). Le parcours de l'arbre s'effectue en un temps logarithmique, auquel il faut ajouter le coût des comparaisons, ce qui donne une complexité de $\mathcal{O}(i \times \log(l))$.

En ce qui concerne l'insertion, une insertion locale nécessite, en plus d'une recherche par position, la génération de l'identifiant, ce qui donne une complexité de $\mathcal{O}(i + \log(l))$, et une insertion distante nécessite une recherche par identifiant, d'où une complexité de $\mathcal{O}(i \times \log(l))$.

Il en va de même pour les suppressions, avec toutefois la nécessité de relancer l'algorithme de suppression distante si l'élément n'est pas trouvé, ce qui conduit au pire des cas à une complexité de $\mathcal{O}(i \times s \times \log(l))$.

Cette implémentation semble offrir un bon compromis entre l'implémentation naïve et l'implémentation en chaîne, en proposant une complexité logarithmique pour toute opération, locale ou distante.

5.2.4 Expériences et résultats

Le tableau 5.1 résume les complexités théoriques de ces trois implémentations, ainsi que celles des algorithmes LOGOOT, TREEDOC et WOOT, auxquelles nous allons nous comparer.

TABLE 5.1 – Complexité de différents CRDTS

Algorithme	**Recherche**		**Insertion**		**Suppression**	
	par position	par identifiant	locale	distante	locale	distante
LOGOOTSPLITNAIVE	l	$i \times \log(l)$	$l + i$	$i \times \log(l)$	l	$i \times \log(l) \times s$
LOGOOTSPLITSTRING	1	$i \times \log(n)$	$s + i$	$i \times \log(n) \times s$	1	$i \times \log(n) \times s$
LOGOOTSPLITAVL	$\log(l)$	$i \times \log(l)$	$\log(l) + i$	$i \times \log(l)$	$\log(l)$	$i \times \log(l) \times s$
LOGOOT	1	$i \times \log(n)$	$s \times i$	$i \times \log(n) \times s$	s	$i \times \log(n) \times s$
TREEDOC	i	i	$s \times i$	$s \times i$	$i \times s$	$i \times s$
WOOT	$n + d$	1	$n + d + s$	s	$n + d + s$	s

l : nombre d'éléments, n : taille du document (en caractères visibles), i : taille d'un identifiant, d : nombre de pierres tombales (caractères supprimés), s : taille d'un élément

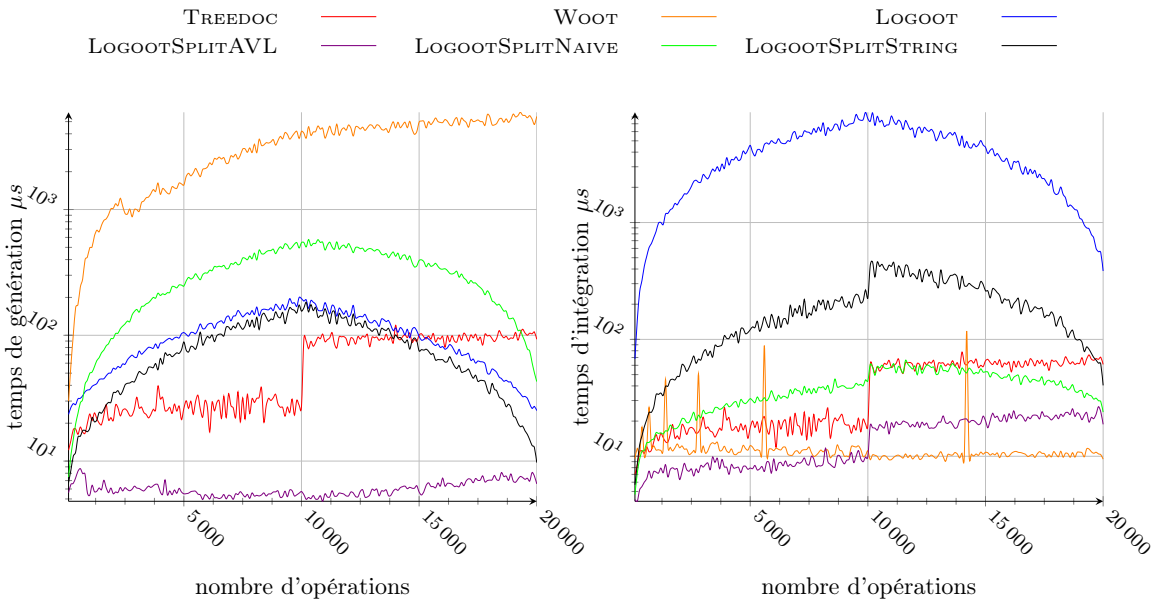
Nous avons lancé plusieurs séries d'expériences en utilisant le projet Replication Benchmarker [9]. Ce projet propose un environnement Java pour implémenter, évaluer et comparer [Ahmed-Nacer, 2015] les algorithmes d'édition collaborative. Les algorithmes WOOT, LOGOOT et TREEDOC étaient déjà implémentés [Ahmed-Nacer *et al.*, 2011] dans le projet, nous avons ajouté les trois implémentations de notre approche.

La première série simule l'exécution des algorithmes du tableau 5.1 sur une même séquence d'opérations. Cette séquence est générée aléatoirement, réalise des opérations qui insèrent ou suppriment en moyenne des chaînes de 50 caractères n'importe où dans le document. La séquence est pour sa première moitié constituée d'un grand nombre d'insertions (80%), et au contraire n'en contient que très peu dans sa seconde moitié (20%). Chaque moitié contient 10 000 opérations. Nous mesurons le temps nécessaire à la génération d'une opération, le temps nécessaire à l'intégration d'une opération, et la place mémoire utilisée.

Les figures 5.14 et 5.15 présentent les résultats de la simulation.

On peut constater que fonctionner avec une taille de blocs élevée est très bénéfique pour nos

9. https ://github.com/score-team/replication-benchmarker



(a) Temps de génération (b) Temps d'intégration

Figure 5.14 – Résultats avec une trace aléatoire

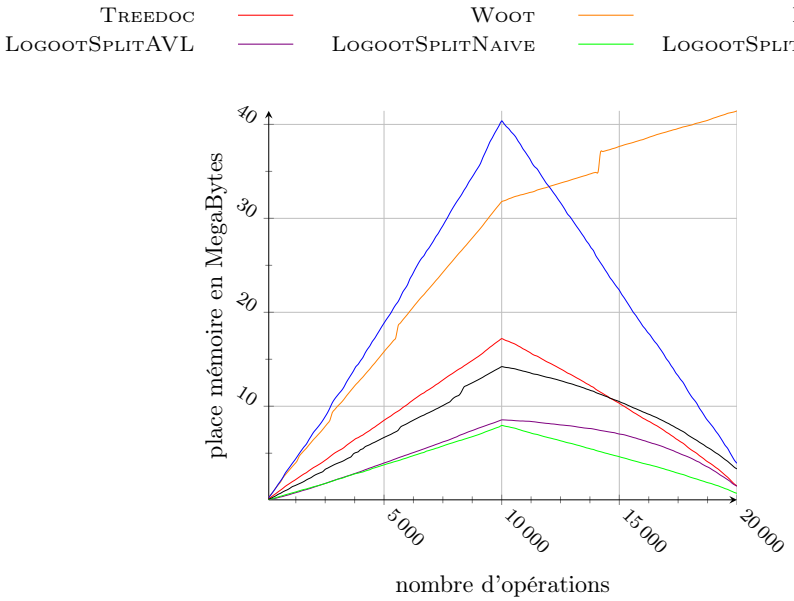


Figure 5.15 – Résultats mémoire avec une trace aléatoire

algorithmes, et qu'ils se placent dans les trois meilleurs en terme de place mémoire utilisée. Lors de la phase des suppressions, la place mémoire prise par LogootSplitNaive décroit linéairement, contrairement à nos deux autres implémentations. Cela est dû au fait que LogootSplitNaive

ne factorise pas les bases des identifiants dans sa structure de données, alors que les autres le font. Tant que la base complète n'est pas supprimée, une suppression dans les algorithmes LogootSplitString et LogootSplitAVL ne retire que peu de choses du modèle.

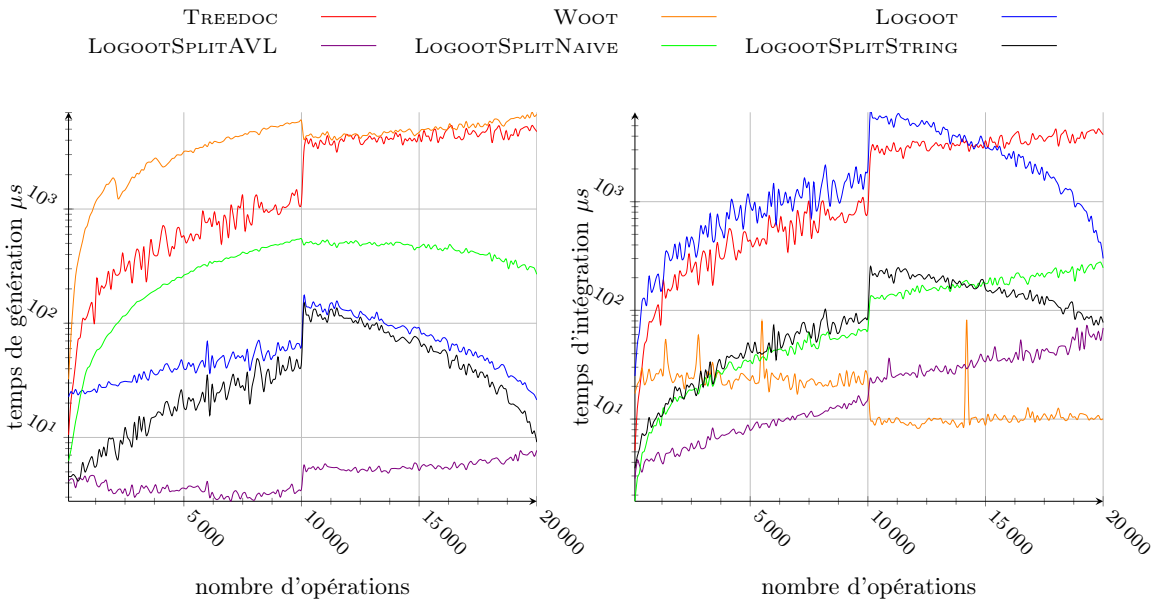
Au niveau des performances locales, LogootSplitAVL est largement plus rapide que nos deux autres implémentations. Cela vient du fait que l'implémentation naïve est obligée de faire une recherche linéaire pour trouver une position dans le document, alors que l'implémentation en arbre est étiquetée et permet une recherche de position en un temps logarithmique. De plus, mettre à jour la structure (rajouter un nœud dans l'arbre) est très rapide. Concernant l'implémentation en chaîne, le fait d'avoir un tableau de caractères et non de blocs permet une recherche d'une position locale instantanée, mais l'insertion d'un bloc dans le modèle est coûteuse car elle réalise un traitement par caractère. En conséquence, cet algorithme se place entre nos deux autres implémentations en ce qui concerne la rapidité de génération des intégrations locales. Treedoc allie structure en arbre et identifiants rapides à générer, et se montre plus rapide que nos implémentations linéaires. Il perd toutefois en efficacité lors de la phase de suppressions à cause des pierres tombales qu'il est obligé de garder en mémoire. On peut remarquer que la courbe de Logoot est très proche de la courbe de notre implémentation en chaîne. Ces deux algorithmes ont en effet un comportement très proche lors de la génération d'opérations et maintiennent tous deux un tableau de caractères.

L'implémentation en chaîne est par contre bien plus rapide que Logoot dans notre expérience lorsqu'il s'agit d'intégrer une opération distante, Logoot en effet va intégrer les caractères un par un alors que nos implémentations intègrent les blocs en une seule fois. Woot est le plus rapide en moyenne pour intégrer les opérations distantes, cas ses éléments sont stockés dans une table de hachage et peuvent être accédés en temps constant, et qu'une intégration d'une opération revient à chercher deux (insertion) ou un seul (suppression) élément. Notre implémentation en arbre se démarque là aussi de l'implémentation naïve et de celle en chaîne. Cette dernière est particulièrement mauvaise par rapport aux autres. En effet, sa recherche dichotomique se fait sur les caractères et non sur les blocs et nécessite donc plus de temps, et à cela s'ajoute également le coût élevé de la mise à jour de la structure de données.

Les figures 5.16 et 5.17 présentent les résultats d'une simulation où les opérations ne sont plus localisées aléatoirement dans le document mais à la fin de celui-ci. On peut observer dans ce cas la dégradation des performances de l'algorithme Treedoc, qui construit un arbre déséquilibré qui se rapproche plus d'une chaîne que d'un arbre.

Localiser les éditions au même endroit entraîne une augmentation rapide de la taille des identifiants de nos approches. En effet, chaque insertion qui nécessite une scission entraîne la création d'un identifiant avec une base plus grande que celle de l'identifiant qu'elle scinde. Dans un document où ces insertions sont réparties au même endroit, la taille des identifiants va donc continuellement grandir à chaque insertion, contrairement à un document où les insertions sont réparties de manière uniforme. En conséquence, la place mémoire prise par LogootSplitNaive, qui ne factorise pas ses bases, augmente énormément.

Au niveau du temps de génération local, LogootSplitNaive et LogootSplitString se



(a) Temps de génération (b) Temps d'intégration

FIGURE 5.16 – Résultats avec des opérations en fin de document

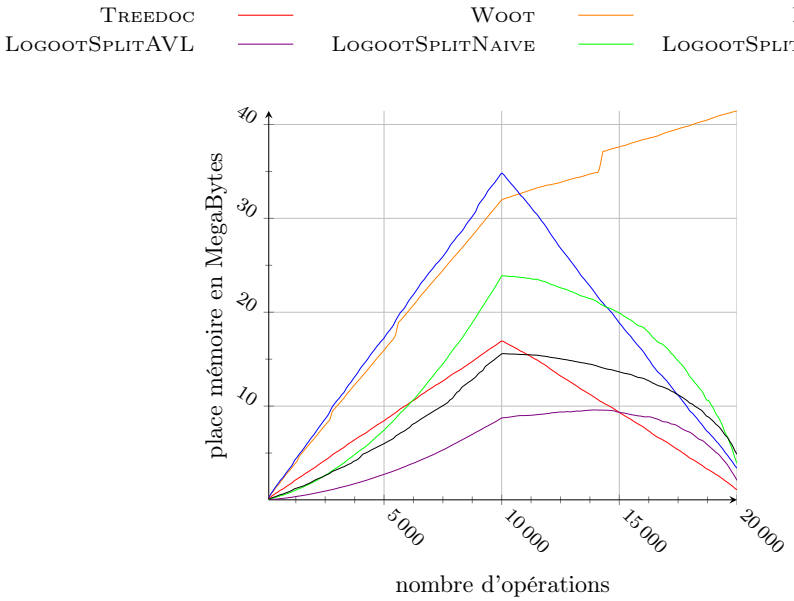


FIGURE 5.17 – Résultats mémoire avec des opérations en fin de document

comportent légèrement plus efficacement lors des insertions qu'auparavant. L'explication est que les structures de données que nous utilisons pour stocker les éléments sous forme de liste sont plus efficaces si on insère un élément à la fin de la structure. On ne constate aucun changement pour

LogootSplitAVL et sa structure d'arbre. Concernant la phase des suppression, LogootSplitAVL se dégrade légèrement, alors que les deux autres implémentations ne présentent aucun changement. La concentration des opérations au même endroit lors de la phase de suppression va mener à un document très fragmenté à cet endroit, et donc la suppression d'une chaîne de caractère va souvent entraîner la suppression de plusieurs éléments. Dans le cas de structures en listes comme LogootSplitNaive et LogootSplitString, pour trouver ces éléments, il suffit de chercher la position du premier puis de prendre les suivants dans la liste. Dans le cas d'une structure d'arbre comme LogootSplitString, il faut chercher tous les éléments un par un, ce qui explique ce temps local additionnel.

Cette localisation des modifications en fin de document entraîne aussi un degré de suppressions concurrentes de la même partie d'un élément plus élevé que lors de nos simulations où elles avaient lieu à des positions aléatoires du document. Dans le cas de suppressions concurrentes de parties d'un même élément, il se peut que l'intégration d'une opération de suppression donne lieu à deux suppressions. Par exemple, supprimer l'élément complet alors que son milieu a déjà été retiré conduit à la suppression de son début et de sa fin. Les implémentations des algorithmes LogootSplitAVL et LogootSplitNaive font un appel à la fonction de suppression par élément à supprimer, et une optimisation a été ajoutée dans LogootSplitString pour ne faire qu'un appel, ce qui explique les courbes de ces algorithmes pour le temps d'intégration des modifications.

Dans le cas général, à la vue de ces résultats, notre implémentation LogootSplitAVL est celle que nous conseillons pour traiter avec des éléments de grande taille.

5.3 Prototype MUTE

Cet algorithme a donné naissance à un éditeur collaboratif web implémenté par l'équipe de recherche nommé MUTE(Multi User Text Editor). Il fonctionne avec un serveur central, qui sert uniquement pour la communication entre les différents utilisateurs et pour récupérer le document lorsqu'un nouvel utilisateur rejoint la session d'édition collaborative. Le cœur de l'application est l'implémentation de LogootSplitAVL en Javascript, reliée à l'éditeur de texte Ace[10]. Les composants qui constituent l'application sont illustrés figure 5.18, la suite de cette partie les présente rapidement.

5.3.1 Architecture générale

Editeur de texte

L'éditeur de texte intégré dans la page web des clients est le point d'entrée de l'application. Les éditions de texte sont traduites en opérations LogootSplitAVL, qui sont ensuite exploitées par le module CRDT. Dans le cas de l'éditeur Ace, chaque édition génère un événement d'inser-

10. https ://ace.c9.io

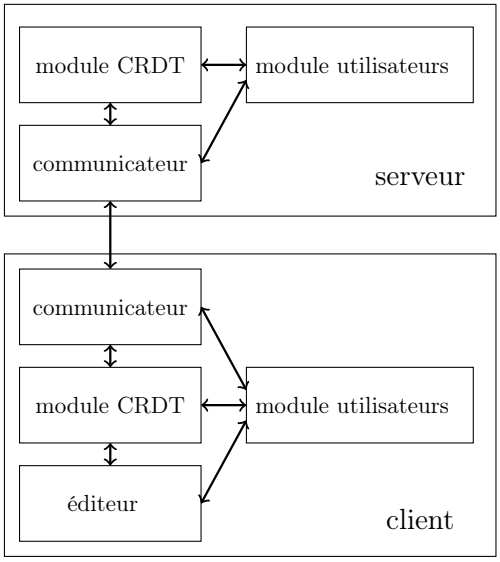


FIGURE 5.18 – Architecture globale de l'application MUTE

tion ou de suppression de texte à une position donnée. Cette position contient un numéro de ligne et l'indice de la position dans la ligne. Les fonctions *LocalInsert* et *LocalDelete* de LOGOOTSPLITAVL prennent en paramètre des positions qui correspondent à l'indice de la modification dans le document complet. La conversion entre ces deux représentations des positions est réalisée avant de les envoyer à module CDRT, qui peut ensuite exécuter les fonctions *LocalInsert* et *LocalDelete* correspondant aux éditions réalisées. Lorsque le module CRDT intègre une opération distante, il envoie à l'éditeur de texte la position où les modifications ont été réalisées, la conversion est réalisée, et le texte est mis à jour en utilisant les fonctions d'insertion ou de suppression de l'API de l'éditeur.

Notre application intègre l'éditeur Ace. Il est tout à fait possible de changer d'éditeur, en remplaçant cette partie de l'architecture.

Module CRDT

Le module CRDT est le composant qui implémente l'algorithme LOGOOTSPLITAVL. Il contient le modèle des documents édités et intègre les opérations générées ou envoyées par les utilisateurs.

Les modules CRDT des clients sont reliés à un éditeur de texte. Les modifications apportées localement au texte via l'éditeur sont transmises au module CRDT, qui les intègre au document modèle et en déduit les éléments du CRDT qui ont été insérés ou supprimés et les opérations CDRT associées. Les opérations CRDT sont ensuite transmises au composant communicateur, qui se charge de les envoyer aux autres utilisateurs via le serveur. De même, à la réception d'une opération CRDT distante, le module CRDT l'intègre et la traduit en opération texte, pour répercuter la modification du modèle sur le texte édité.

Du côté du serveur, il est nécessaire de garder en mémoire l'état des documents pour pouvoir les transmettre lorsqu'un nouvel utilisateur se connecte. Le serveur possède donc lui aussi un module CRDT, qui est mis à jour uniquement par les opérations distantes reçues des clients. Le serveur ne génère jamais d'opérations locales. Il est relié à une base de données qui enregistre les documents pour plus de robustesse, afin de pouvoir les récupérer en cas de panne. Le serveur dialogue avec les clients via le communicateur.

Communicateur

Les clients communiquent entre eux à travers le serveur grâce aux composants communicateurs. Ceux-ci ont pour rôle de gérer les websockets utilisées pour la communication, c'est à dire pour l'envoi du document au client lors de sa connexion, et pour l'envoi des opérations LogootSplitAVL de clients à clients.

Lorsqu'un utilisateur accède à une page web qui correspond à l'édition d'un document, une websocket [11] est créée pour avoir une communication client-serveur bidirectionnelle.

Le client possède une websocket par document ouvert, et le serveur possède une salle par document, lesquelles contiennent une websocket par clients connectés au document. Les messages échangés n'ont donc pas besoin de préciser à quel document ils se rapportent, cela dépend du canal de communication utilisé. La figure 5.19 illustre les liens clients-serveur réalisés à partir des différentes websockets. Sur cette figure, trois clients sont connectés au serveur, et deux documents sont édités. Le premier client a ouvert deux onglets dans son navigateur pour éditer à la fois les documents 1 et 2, les autres clients éditent chacun le document 1 et 2. Quand une opération locale est envoyée à un site distant, elle est envoyée au serveur via la socket qui correspond au document. Le serveur redirige ensuite cet envoi aux autres clients qui ont une socket ouverte pour ce document.

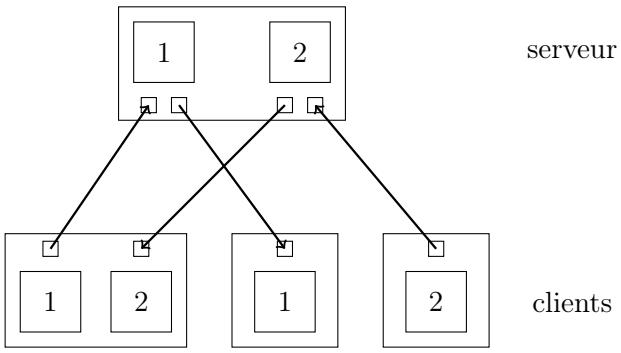


Figure 5.19 – Exemple de liens websockets clients-serveur

Lors de la connexion initiale, un client ne possède pas de numéro de site unique, nécessaire pour la création des identifiants des éléments LogootSplitAVL. Le serveur maintient un comp-

11. https ://www.websocket.org/

teur par document, qu'il incrémente à chaque fois qu'un nouveau client se connecte. Il envoie la valeur du compteur au nouveau client eu guise d'identifiant de site. Ce numéro est ensuite enregistré dans le navigateur du client. Le serveur envoie également l'état du document modèle au client. Lors d'une connexion future, le client transmet cet identifiant au serveur, pour lui indiquer qu'il n'a pas besoin d'en générer un nouveau, et récupère le document modèle.

Module données utilisateurs

Le dernier composant de cette architecture est le module de données utilisateurs. Ce module maintient et transmet des informations sur l'utilisateur, informations qui sont ensuite utilisées pour fournir des outils d'awareness aux utilisateurs. Les données concernées sont les noms des utilisateurs, la liste des utilisateurs connectés, la position de leur curseur ou des bornes de leur sélection actuelle. Elles permettent notamment de voir dans l'éditeur de texte la position des curseurs des autres utilisateurs, ce qui améliore l'édition globale du document [Gutwin and Greenberg, 1999].

5.3.2 Processus d'édition et d'intégration

Lors de la réalisation de notre application, nous nous sommes trouvés confrontés au problème suivant : l'éditeur de texte et le module CRDT peuvent être modifiés en concurrence par deux sources différentes. L'éditeur peut être modifié par l'utilisateur local et par le module CRDT lors de l'intégration d'une modification distante, le module CRDT peut être modifié par l'éditeur texte après une modification locale ou par l'intégration d'une opération distante.

En conséquence, le modèle de données du module CRDT et le texte de l'éditeur peuvent être désynchronisés après l'intégration totale de modifications. Par exemple, un utilisateur peut insérer un caractère 'x' à la position 3 dans le document, pendant que le module CRDT reçoit une opération distante qu'il intègre immédiatement et traduit en opération d'insertion d'un caractère 'y' à la position 6. A ce moment, le module CRDT et l'éditeur n'ont pas encore transmis leurs opérations, mais ont été modifiés chacun différemment par l'une d'entre elles. Ensuite, la transmission va avoir lieu, l'insertion du caractère 'x' à la position 3 dans le module CRDT va décaler le caractère 'y' à la position 7, et le caractère 'y' va être inséré dans le document à la position 6. C'est un cas classique de conflit d'édition lié à la concurrence, qui a déjà été illustré dans l'état de l'art sur l'approche OT par la figure 3.1.

Pour résoudre ce problème, une solution aurait été d'utiliser un mécanisme OT entre ces deux composants pour résoudre le conflit d'édition, mais cela nous a paru trop lourd. La solution retenue est d'interdire les intégrations des opérations distantes pendant l'édition locale.

Les opérations distantes sont stockées dans une file d'attente en attendant d'être intégrées. Nous avons ajouté un timer dans l'éditeur de texte, qui sert à détecter quand un utilisateur est actif et quand il ne l'est pas. Initialement, un utilisateur est considéré comme inactif, et les opérations distantes dans la file d'attente peuvent être intégrées. Dès que l'utilisateur modifie le document, et à chaque fois qu'il le fait, le timer se déclenche et l'utilisateur est considéré comme

actif. La modification de l'utilisateur est retardée le temps que l'opération en cours d'intégration soit complètement intégrée. Lorsque le timer arrive à terme, cela signifie que l'utilisateur n'a pas été actif depuis un certain temps : il est à nouveau considéré comme inactif et les opérations distantes peuvent être intégrées. Notre algorithme donne donc la priorité à l'édition locale sur l'intégration des opérations distantes, pour bloquer les utilisateurs le moins longtemps possible. Pour éviter un retard trop important de l'intégration des opérations distantes en cas d'édition intensive d'un utilisateur, la taille de la file d'attente est affichée dans l'éditeur. Si elle devient trop grande, il suffit à l'utilisateur d'arrêter de modifier le document.

Pour réduire le nombre d'opérations à intégrer, notre application essaye de rassembler les insertion locales contiguës en une seule insertion avant de l'envoyer au module CRDT. Une deuxième file enregistre l'opération texte en attente de transmission. Lorsqu'une deuxième opération est créée, s'il s'agit d'une suppression ou d'une insertion qui ne suit pas immédiatement l'opération de la file, les deux opérations sont envoyées et la file est purgée. Sinon, l'opération dans la file est mise à jour pour insérer en plus les caractères de la deuxième opération. Un deuxième timer contrôle depuis quand la file n'a pas été purgée, et envoie son contenu si cela fait trop longtemps que l'opération est en attente, afin que les utilisateurs distants puissent prendre rapidement conscience des modifications.

5.3.3 Travail en mode déconnecté

Notre application est utilisable en isolation, que ce soit volontairement ou lors d'une coupure réseau. L'utilisateur a la possibilité d'enregistrer les pages de l'application, pour éditer un document à tout moment, ou peut se déconnecter temporairement du serveur. Dans les deux cas, lors d'une édition déconnectée, les opérations et le document modèle sont enregistrés dans le navigateur. A la reconnexion, le client envoie au serveur ses opérations locales. Le serveur les intègre, puis envoie au client le nouveau modèle en entier, comme lors d'une connexion normale. Le client peut alors reprendre l'édition connecté.

Ce travail en mode déconnecté génère beaucoup de concurrence entre les opérations à la reconnexion, et produit des blocs de caractères de taille élevée. Il est d'autant plus intéressant d'utiliser un module CDRT qui fonctionne avec LogootSplitAVL plutôt qu'avec un CRDT décrit dans l'état de l'art, LogootSplitAVL étant très performant quand utilisé avec des blocs et proposant une intégration des opérations plus adaptée en cas de concurrence.

5.3.4 Maturité du prototype

Notre application est disponible à l'url http ://coedit.re. Les utilisateurs qui s'y connectent peuvent créer des nouveaux documents ou rejoindre l'édition de documents déjà créés. Nous avons réalisé ce prototype avec un serveur qui enregistre le document et qui est responsable de la communication de clients à clients, mais le CRDT utilisé peut être utilisé dans un contexte totalement décentralisé, avec l'utilisation d'outils comme par exemple webRTC [12], qui permettent

12. http ://www.webrtc.org/

la communication en temps réel entre deux navigateurs. Ce type d'implémentation est à l'étude dans le projet openPaaS.

5.4 Bilan de la contribution

Notre algorithme résout le problème de la granularité fixe des CRDT posé en début de chapitre. Il élimine la possibilité d'avoir des données concurrentes entrelacées en insérant les données en blocs, et évite le problème des données dupliquées en permettant une scission des données en blocs plus petits.

Au delà de ces améliorations pour l'utilisateur, les performances théoriques de cet algorithme dépassent celles des autres algorithmes connus lorsque les blocs insérés sont de grande taille, et le mécanisme d'*append* permet de créer avec plus de facilité de grands blocs.

Nous avons validé notre approche par l'implémentation d'un prototype, qui est utilisé pour prendre des notes par l'équipe de recherche lors de ses réunions. Des améliorations pour ce prototype sont en cours de développement.

Du point de vue algorithmique, l'apport de cet algorithme est une méthode de scission des éléments d'un CRDT, alors que jusqu'à présent seuls des CRDT avec des méthodes d'insertion et de suppression d'éléments étaient disponibles pour l'édition collaborative en temps réel.

6

Conclusion

L'édition collaborative en temps réel de documents texte se trouve actuellement à un stade bien développé. Plusieurs familles d'approches existent, et on peut trouver à l'intérieur de ces familles de nombreux algorithmes qui permettent d'éditer des structures de données différentes.

Ce n'est toutefois pas un domaine totalement maîtrisé dans lequel on ne peut plus progresser de façon significative. Les algorithmes d'édition collaborative en temps réel ont été conçus pour réconcilier des structures de données divergentes de la manière acceptable la plus simple qu'il soit. L'étape suivante est de rehausser nos attentes et de réconcilier les données de manière plus complexe, mais aussi plus adaptée aux utilisateurs. Nous avons pour cela étudié les deux familles d'approches de la littérature, et proposé des améliorations pour chacune d'entre elles. Ces améliorations vont vers un meilleur respect des intentions des utilisateurs, alors que les approches actuelles ne se contentaient que de respecter l'intention des opérations utilisées par les algorithmes.

6.1 Contributions

6.1.1 Transformées pour document structuré

Dans un premier temps, nous avons analysé l'approche OT et déterminé que le respect des intentions des utilisateurs était lié majoritairement aux opérations utilisées par l'algorithme. Nous avons donc conçu un ensemble de sept opérations basé sur les intentions des utilisateurs, et une structure de données adaptée à ces opérations. Nous avons ensuite écrit les quarante-neuf transformées associées à cet ensemble, afin de le rendre utilisable par les algorithmes de contrôle OT.

Grâce à son ensemble d'opérations étendu, notre algorithme est capable de différencier plus finement les actions de l'utilisateur, et de proposer des résolutions de conflits plus adaptées aux réelles intentions des utilisateurs. Un autre avantage de notre approche est qu'elle génère moins d'opérations que les autres approches, ce qui a théoriquement des effets bénéfiques sur la rapidité d'intégration des éditions.

Nous avons montré le potentiel de notre algorithme en l'appliquant au contexte de l'édition collaborative en temps réel de wikis, à travers un transfert industriel pour l'entreprise XWiki. En plus de mieux respecter les intentions des utilisateurs, notre application permet une édition What You See Is What You Get des pages wiki, améliorant à nouveau le confort d'édition apporté aux utilisateurs.

6.1.2 CRDT à granularité variable

Dans un second temps, nous avons analysé l'approche CRDT pour l'édition de documents texte, plus récente que l'approche OT et plus simple dans ses mécanismes d'intégration. Elle se base sur des ensembles d'éléments ordonnés, que l'on peut ajouter ou retirer du modèle. Une faille commune à tous les CRDT étudiés était l'absence d'opération efficace de mise à jour des éléments, ce qui posait des problèmes de respect de l'intention des utilisateurs en cas de mise à jour en parallèle du même élément.

Nous avons conçu un CRDT qui élimine ce problème, en ajoutant une opération de scission d'un élément qui respecte les intentions des utilisateurs. Une mise à jour peut être réalisée en scindant un élément au niveau de la mise à jour, puis en insérant un nouvel élément entre les deux parties scindées. Cet algorithme permet de fonctionner avec des blocs de caractères de grande taille, ce qui réduit la quantité de métadonnées par caractères nécessaire et réduit la place mémoire requise.

Nous avons proposé trois différentes implémentations de notre algorithme et les avons comparées aux implémentations d'algorithmes existants, avec des résultats satisfaisants. Nous avons ensuite intégré l'implémentation la plus performante dans un éditeur de texte pour réaliser une application web. Cette application est depuis utilisée lors de certaines réunions d'équipe pour prendre des notes en collaboration.

6.2 Perspectives

Concernant notre approche OT, nous n'avons pas conduit d'études sur les performances de notre algorithme. Théoriquement, il est plus rapide que les approches existantes car une édition locale génère moins d'opérations. Toutefois, ces opérations ainsi que les transformées associées sont plus complexes et s'exécutent sur une structure différente. Bien que le point important de notre approche soit le respect des intentions des utilisateurs, il serait intéressant de comparer véritablement notre approche à celles existantes en terme de rapidité d'intégration et de place mémoire.

Concernant notre approche CRDT, le fait de travailler avec des blocs de caractères réduit la taille mémoire utilisée par l'algorithme, mais des éditions répétées au même endroit du document conduisent à une augmentation rapide de la taille mémoire utilisée par caractère. Ceci est dû à une augmentation du nombre de blocs plus rapide que l'augmentation du nombre de caractères.

Un mécanisme de fusion des blocs permettrait de lisser le document et de réduire périodiquement la place mémoire prise par l'algorithme.

De plus, la méthode utilisée pour scinder des blocs en deux peut être adaptée pour d'autres CRDTs, et notre équipe de recherche travaille à l'incorporer dans l'algorithme RGA.

A

Fonctions de transformations complètes de notre approche OT

A.1 Transformées par rapport à InsertText

```
1  T(InsertText(pos1, path1, char1, siteId1), InsertText(pos2, path2, char2, siteId2)) :
2     if path1 != path2 then return InsertText(pos1, path1, char1, siteId1)
3     else if pos1 < pos2 then return InsertText(pos1, path1, char1, siteId1)
4     else if (pos1 == pos2 and siteId1 == siteId2) then
5        return InsertText(pos1, path1, char1, siteId1)
6     else return InsertText(pos1 + 1, path1, char1, siteId1)
7     endif
8  end
```

```
1  T(DeleteText(pos1, path1), InsertText(pos2, path2, char2, siteId2)) :
2     if path1 != path2 then return DeleteText(pos1, path1)
3     else if pos1 < pos2 then return DeleteText(pos1, path1)
4     else return DeleteText(pos1 + 1, path1)
5     endif
6  end
```

```
1  T(NewParagraph(pos1, siteId1), InsertText(pos2, path2, char2, siteId2)) :
2     return NewParagraph(pos1, siteId1)
3  end
```

```
1  T(MoveParagraph(origin1, dest1, siteId1), InsertText(pos2, path2, char2, siteId2)) :
2     return MoveParagraph(origin1, dest1, siteId1)
3  end
```

```
T(MergeParagraph(pos1, leftChNb1), InsertText(pos2, path2, char2, siteId2)) :
  return MergeParagraph(pos1, leftChNb1)
end
```

```
T(SplitParagraph(pos1, path1, split1), InsertText(pos2, path2, char2, siteId2)) :
  if pos1 != pos2 then return SplitParagraph(pos1, path1, split1)
  else if pos1 ≤ pos2 return SplitParagraph(pos1, path1, split1)
  else return SplitParagraph(pos1 + 1, path1, split1)
  endif
end
```

```
T(Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1),
 InsertText(pos2, path2, char2, siteId2)) :
  if pos1 != pos2 then
    return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
  else if start1 == pos2 then
    return Style(start1, end1 + 1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
  else if start1 > pos2 then
    return Style(start1 + 1, end1 + 1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
  else if end1 < pos2 then
    return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
  else return Style(start1, end1 + 1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
  endif
end
```

A.2 Transformées par rapport à DeleteText

```
T(InsertText(pos1, path1, char1, siteId1), DeleteText(pos2, path2)) :
  if path1 != path2 then return InsertText(pos1, path1, char1, siteId1)
  else if pos1 ≤ pos2 then return InsertText(pos1, path1, char1, siteId1)
  else return InsertText(pos1 − 1, path2, char1, siteId1)
  endif
end
```

```
1  T(DeleteText(pos1, path1), DeleteText(pos2, path2)) :
2    if path1! = path2 then return DeleteText(pos1, path1)
3    else if pos1 == pos2 then return Id()
4    else if pos1 < pos2 then return DeleteText(pos1, path1)
5    else return DeleteText(pos1 − 1, path1)
6    endif
7  end
```

```
1  T(NewParagraph(pos1, siteId1), DeleteText(pos2, path2)) :
2    return NewParagraph(pos1, siteId1)
3  end
```

```
1  T(MoveParagraph(origin1, dest1, siteId1), DeleteText(pos2, path2)) :
2    return MoveParagraph(origin1, dest1, siteId1)
3  end
```

```
1  T(MergeParagraph(pos1, leftChNb1), DeleteText(pos2, path2)) :
2    return MergeParagraph(pos1, leftChNb1)
3  end
```

```
1  T(SplitParagraph(pos1, path1, split1), DeleteText(pos2, path2)) :
2    if path1! = path2 then return SplitParagraph(pos1, path1, split1)
3    else if pos1\leq pos2 then return SplitParagraph(pos1, path1, split1)
4    else return SplitParagraph(pos1 − 1, path1, split1)
5    endif
6  end
```

```
1  T(Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1),
2   DeleteText(pos2, path2)) :
3    if path1! = path2 then
4      return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
5    else if start1 > pos2 then
6      return Style(start1 − 1, end1 − 1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
7    else if end1 ≤ pos2 then
8      return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
9    else return Style(start1, end1 − 1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
10   endif
11 end
```

A.3 Transformées par rapport à NewParagraph

```
1  T(InsertText(pos1, path1, char1, siteId1), NewParagraph(pos2, siteId2)) :
2    if path1[0] < pos2 then return InsertText(pos1, path1, char1, siteId1)
3    else return InsertText(pos1, [path1[0] + 1, path1[1]], char1, siteId1)
4    endif
5  end
```

```
1  T(DeleteText(pos1, path1), NewParagraph(pos2, siteId2)) :
2    if path1[0] < pos2 then return DeleteText(pos1, path1)
3    else return DeleteText(pos1, [path1[0] + 1, path1[1]])
4    endif
5  end
```

```
1  T(NewParagraph(pos1, siteId1), NewParagraph(pos2, siteId2)) :
2    if pos1 < pos2 then return NewParagraph(pos1, siteId1)
3    else if (pos1 == pos2 and siteId1 < siteId2) then
4      return NewParagraph(pos1, siteId1)
5    else return NewParagraph(pos1 + 1, siteId1)
6    endif
7  end
```

```
1  T(MoveParagraph(origin1, dest1, siteId1), NewParagraph(pos2, siteId2)) :
2    origin = origin1
3    dest = dest1
4    if pos2 ≤ origin1 then origin++ endif
5    if pos2 < dest1 then dest++ endif
6    return MoveParagraph(origin, dest, siteId1)
7  end
```

```
1  T(MergeParagraph(pos1, leftChNb1), NewParagraph(pos2, siteId2)) :
2    if pos1 == pos2 then return Composite([
3      MoveParagraph(pos2, pos2 + 2, siteId1 [13]) ,
4      MergeParagraph(pos1, leftChNb1)])
5    else if pos1 < pos2 then return MergeParagraph(pos1, leftChNb1)
6    else return MergeParagraph(pos1 + 1, leftChNb1)
7    endif
8  end
```

13. il s'agit bien de l'identifiant unique du site qui a généré l'opération 1. Ce n'est pas un paramètre de l'opération de fusion, mais il doit être récupérable pour les besoins de cette transformée

```
1  T(SplitParagraph(pos1, path1, split1), NewParagraph(pos2, siteId2)) :
2    if path1[0] < pos2 then return SplitParagraph(pos1, path1, split1)
3    else return SplitParagraph(pos1, [path1[0] + 1, path1[1], split1)
4    endif
5  end
```

```
1  T(Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1),
2   NewParagraph(pos2, siteId2)) :
3    if path1[0] < pos2 then
4      return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
5    else
6      return Style(start1, end1, [path1[0] + 1, path1[1], param1, value1,
7        siteId1, splitStart1, splitEnd1)
8    endif
9  end
```

A.4 Transformées par rapport à MoveParagraph

```
1  T(InsertText(pos1, path1, char1, siteId1), MoveParagraph(origin2, dest2, siteId2)) :
2    if path1[0] == origin2 then
3      if dest2 ≥ origin2 then return InsertText(pos1, [dest2, path1[1]], char1, siteId1)
4      else return InsertText(pos1, [dest2 − 1, path1[1]], char1, siteId1)
5      endif
6    else if path1[0] < origin2 then
7      if path1[0] < dest2 then return InsertText(pos1, path1, char1, siteId1)
8      else return InsertText(pos1, [path1[0] + 1, path1[1]], char1, siteId1)
9      endif
10   else
11     if path1[0] < dest2 then
12       return InsertText(pos1, [path1[0] − 1, path1[1]], char1, siteId1)
13     else return InsertText(pos1, path1, char1, siteId1)
14     endif
15   endif
16 end
```

```
1  T(DeleteText(pos1, path1), MoveParagraph(origin2, dest2, siteId2)) :
2    if path1[0] == origin2 then
3      if dest2 ≥ origin2 then return DeleteText(pos1, [dest2, path1[1]])
4      else return DeleteText(pos1, [dest2 − 1, path1[1]])
5      endif
6    else if path1[0] < origin2 then
7      if path1[0] < dest2 then return DeleteText(pos1, path1)
8      else return DeleteText(pos1, [path1[0] + 1, path1[1]])
9      endif
10   else
11     if path1[0] < dest2 then return DeleteText(pos1, [path1[0] − 1, path1[1]])
12     else return DeleteText(pos1, path1)
13     endif
14   endif
15 end
```

```
1  T(NewParagraph(pos1, siteId1), MoveParagraph(origin2, dest2, siteId2)) :
2    if origin2 < pos1 then
3      if dest2 ≤ pos1 then return NewParagraph(pos1, siteId1)
4      else return NewParagraph(pos1 − 1, siteId1)
5      endif
6    else
7      if dest2 ≤ pos1 then return NewParagraph(pos1 + 1, siteId1)
8      else return NewParagraph(pos1, siteId1)
9      endif
10   endif
11 end
```

```
1  T(MoveParagraph(origin1, dest1, siteId1), MoveParagraph(origin2, dest2, siteId2)) :
2    if origin1 == origin2 then
3      if dest1 == dest2 then return Id()
4      else
5        if siteId1 < siteId2 then
6          origin = dest2
7          if origin2 < dest2 then origin−− endif
8          dest = dest1
9          if origin2 < dest1 then dest−− endif
10         if dest2 < dest1 then dest++ endif
11         return MoveParagraph(origin, dest, siteId1)
```

```
12          else return Id()
13          endif
14        endif
15      else
16        origin = origin1
17        dest = dest1
18        if origin2 < origin1 then origin−− endif
19        if dest2 ≤ origin1 then origin++ endif
20        if origin2 < dest1 then dest−− endif
21        if dest2 < dest1 then dest++ endif
22        if dest2 == dest1 and siteId2 < SiteId1 then dest++ endif
23        return MoveParagraph(origin, dest, siteId1)
24      endif
25  end
```

```
1   T(MergeParagraph(pos1, leftChNb1), MoveParagraph(origin2, dest2, siteId2)) :
2     if origin2 == pos1 then
3       if dest2 == origin2 − 1 then return Composite([
4         MoveParagraph(origin2, dest2, siteId2),
5         MergeParagraph(pos1, leftChNb1)])
6       else
7         if dest2 > origin2 then return Composite([
8           MoveParagraph(origin2 − 1, dest2 − 1, siteId2),
9           MergeParagraph(dest2 − 1, leftChNb1)])
10        else return Composite([
11          MoveParagraph(origin2, dest2, siteId2),
12          MergeParagraph(dest2 + 1, leftChNb1)])
13        endif
14      endif
15    else if origin2 == pos1 − 1 then
16      if dest2 == origin2 + 2 then return Composite([
17        MoveParagraph(origin2, dest2, siteId2),
18        MergeParagraph(pos1, leftChNb1)
19      else
20        if dest2 > origin2 then return Composite([
21          MoveParagraph(origin2, dest2, siteId2),
22          MergeParagraph(dest2, leftChNb1)
23        else return Composite([
24          MoveParagraph(origin2 + 1, dest2 + 1, siteId2),
25          MergeParagraph(dest2 + 1, leftChNb1)
```

```
26          endif
27        endif
28      else if pos1 < origin2 then
29        if pos1 == dest2 then return Composite([
30          MoveParagraph(dest2, dest2 + 2, siteId2),
31          MergeParagraph(pos1, leftChNb1)
32        else if pos1 < dest2 then return MergeParagraph(pos1, leftChNb1)
33        else return MergeParagraph(pos1 + 1, leftChNb1)
34        endif
35      else
36        if pos1 == dest2 then return Composite([
37          MoveParagraph(dest2 − 1, dest2 + 1, steId2),
38          MergeParagraph(pos1 − 1, leftChNb1)
39        else if pos1 < dest2 then return MergeParagraph(pos1 − 1, leftChNb1)
40        else return MergeParagraph(pos1, leftChNb1)
41        endif
42      endif
43  end
```

```
1   T(SplitParagraph(pos1, path1, split1), MoveParagraph(origin2, dest2, siteId2)) :
2     if path1[0] == origin2 then
3       if dest2 ≥ origin2 then return SplitParagraph(pos1, [dest2, path1[1]], split1)
4       else return SplitParagraph(pos1, [dest2 − 1, path1[1]], split1)
5       endif
6     else if path1[0] < origin2 then
7       if path1[0] < dest2 then return SplitParagraph(pos1, path1, split1)
8       else return SplitParagraph(pos1, [path1[0] + 1, path1[1]], split1)
9       endif
10    else
11      if path1[0] < dest2 then
12        return SplitParagraph(pos1, [path1[0] − 1, path1[1]], split1)
13      else return SplitParagraph(pos1, path1, split1)
14      endif
15    endif
16  end
```

```
1  T(Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1),
2    MoveParagraph(origin2, dest2, siteId2)) :
3      if path1[0] == origin2 then
4        if dest2 ≥ origin2 then
5          return Style(start1, end1, [dest2, path1[1]], param1, value1,
6            siteId1, splitStart1, splitEnd1)
7        else return Style(start1, e, d1, [dest2 − 1, path1[1]], param1, value1,
8          siteId1, splitStart1, splitEnd1)
9        endif
10     else if path1[0] < origin2
11       if path1[0] < dest2 then
12         return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
13       else return Style(start1, end1, [path1[0] + 1, path1[1]], param1, value1,
14         siteId1, spliStart1, splitEnd1)
15       endif
16     else
17       if path1[0] < dest2 then
18         return Style(start1, end1, [path1[0] − 1, path1[1]], param1, value1,
19           siteId1, splitStart1, splitEnd1)
20       else return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
21       endif
22     endif
23 end
```

A.5 Transformées par rapport à MergeParagraph

```
1  T(InsertText(pos1, path1, char1, siteId1), MergeParagraph(pos2, leftChNb2)) :
2      if path1[0] < pos2 then return InsertText(pos1, path1, char1, siteId1)
3      else if path1[0] > pos2 then
4        return InsertText(pos1, [path1[0] − 1, path1[1]], char1, siteId1)
5      else return InsertText(pos1, [path1[0] − 1, path1[1] + leftChNb2], char1, siteId1)
6      endif
7  end
```

```
1  T(DeleteText(pos1, path1), MergeParagraph(pos2, leftChNb2)) :
2    if path1[0] < pos2 then return DeleteText(pos1, path1)
3    else if path1[0] > pos2 then return DeleteText(pos1, [path1[0] − 1, path1[1]])
4    else return DeleteText(pos1, [path1[0] − 1, path1[1] + leftChNb2])
5    endif
6  end
```

```
1  T(NewParagraph(pos1, siteId1), MergeParagraph(pos2, leftChNb2)) :
2    if pos1 ≤ pos2 then return NewParagraph(pos1, siteId1)
3    else return NewParagraph(pos1 − 1, siteId1)
4    endif
5  end
```

```
1  T(MoveParagraph(origin1, dest1, siteId1), MergeParagraph(pos2, leftChNb2)) :
2    if origin1 == pos2 then
3      if dest1 == origin1 − 1 then return Id()
4      else if dest1 > origin2 then
5        return MoveParagraph(origin1 − 1, dest1 − 1, siteId1)
6      else return MoveParagraph(origin1 − 1, dest1, siteId1)
7      endif
8    else if origin1 == pos2 − 1 then
9      if dest1 == origin1 + 2 then return Id()
10     else if dest1 > origin1 then
11       return MoveParagraph(origin1, dest1 − 1, siteId1)
12     else return MoveParagraph(origin1, dest1, siteId1)
13     endif
14   else
15     origin = origin1
16     dest = dest1
17     if pos2 < origin1 then origin − − endif
18     if pos2 < dest1 then dest − − endif
19     return MoveParagraph(origin, dest, siteId1)
20   endif
21 end
```

```
1  T(MergeParagraph(pos1, leftChNb1), MergeParagraph(pos2, leftChNb2)) :
2    if pos1 == pos2 then return Id()
3    else if pos1 == pos2 + 1 then
4      return MergeParagraph(pos1 − 1, leftChNb1 + leftChNb2)
5    else if pos1 < pos2 then return MergeParagraph(pos1, leftChNb1)
6    else return MergeParagraph(pos1 − 1, leftChNb1)
7    endif
8  end
```

```
1  T(SplitParagraph(pos1, path1, split1), MergeParagraph(pos2, leftChNb2)) :
2    if path1[0] < pos2 then return SplitParagraph(pos1, path1, split1)
3    else if path1[0] > pos2 then
4      return SplitParagraph(pos1, [path1[0] − 1, path1[1]], split1)
5    else return SplitParagraph(pos1, [path1[0] − 1, path1[1] + leftChNb2], split1)
6    endif
7  end
```

```
1  T(Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1),
2   MergeParagraph(pos2, leftChNb2)) :
3    if path1[0] < pos2 then
4      return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
5    else if path1[0] > pos2 then
6      return Style(start1, end1, [path1[0] − 1, path1[1]], param1, value,
7        siteId1, splitStart1, splitEnd1)
8    else return Style(start1, end1, [path1[0] − 1, path1[1] + leftChNb2], param1, value1,
9      siteId1, splitStart1, splitEnd1)
10   endif
11 end
```

A.6 Transformées par rapport à SplitParagraph

```
1  T(InsertText(pos1, path1, char1, siteId1), SplitParagraph(pos2, path2, split2)) :
2    if path1 == path2 then
3      if pos1 < pos2 then return InsertText(pos1, path1, char1, siteId1)
4      else return InsertText(pos1 − pos2, [path1[0] + 1, 0], char1, siteId1)
5      endif
6    else if path1[0] < path2[0] or (path1[0] == path2[0] and path1[1] < path2[1]) then
7      return InsertText(pos1, path1, char1, siteId1)
8    else if path1[0] > path2[0] then
9      return InsertText(pos1, [path1[0] + 1, path1[1]], char1, siteId1)
10   else return InsertText(pos1, [path1[0] + 1, path1[1] − path2[1]], char1, siteId1)
11   endif
12 end
```

```
1  T(DeleteText(pos1, path1), SplitParagraph(pos2, path2, split2)) :
2    if path1 == path2 then
3      if pos1 < pos2 then return DeleteText(pos1, path1)
4      else return DeleteText(pos1 − pos2, [path1[0] + 1, 0])
5      endif
6    else if path1[0] < path2[0] or (path1[0] == path2[0] and path1[1] < path2[1]) then
7      return DeleteText(pos1, path1)
8    else if path1[0] > path2[0] then return DeleteText(pos1, [path1[0] + 1, path1[1]])
9    else return DeleteText(pos1, [path1[0] + 1, path1[1] − path2[1]])
10   endif
11 end
```

```
1  T(NewParagraph(pos1, siteId1), SplitParagraph(pos2, path2, split2)) :
2    if pos1 ≤ path2[0] then return NewParagraph(pos1, siteId1)
3    else return NewParagraph(pos1 + 1, siteId1)
4    endif
5  end
```

```
1  T(MoveParagraph(origin1, dest1, siteId1), SplitParagraph(pos2, path2, split2)) :
2    if origin1 == path2[0] then
3      if origin1 < dest1 then return Composite([
4        MoveParagraph(origin1, dest1 + 1, siteId1),
5        MoveParagraph(origin1, dest1 + 1, siteId1)])
6      else return Composite([
7        MoveParagraph(origin1, dest1, siteId1),
8        MoveParagraph(origin1, dest1 + 1, siteId1 + 1)])
9      endif
10   else
11     origin = origin1
12     dest = dest1
13     if path2[0] < origin1 then origin++ endif
14     if path2[0] < dest1 then dest++ endif
15     return MoveParagraph(origin, dest, siteId1)
16   endif
17 end
```

```
1  T(MergeParagraph(pos1, leftChNb1), SplitParagraph(pos2, path2, split2)) :
2    if pos1 == path2[0] then return MergeParagraph(pos1, leftChNb1)
3    else if pos1 == path2[0] + 1 then
4      return MergeParagraph(pos1 + 1, leftChNb1 − path2[1])
5    else if pos1 < path2[0] then return MergeParagraph(pos1, leftChNb1)
6    else return MergeParagraph(pos1 + 1, leftChNb1)
7    endif
8  end
```

```
1  T(SplitParagraph(pos1, path1, split1), SplitParagraph(pos2, path2, split2)) :
2    if path1 == path2 then
3      if pos1 < pos2 then return SplitParagraph(pos1, path1, split1)
4      else if pos1 == pos2 then return Id()
5      else return SplitParagraph(pos1 − pos2, [path1[0] + 1, 0], split1)
6      endif
7    else if path1[0] < path2[0] or (path1[0] == path2[0] and path1[1] < path2[1]) then
8      return SplitParagraph(pos1, path1, split1)
9    else if path1[0] > path2[0] then
10     return SplitParagraph(pos1, [path1[0] + 1, path1[1]], split1)
11   else return SplitParagraph(pos1, [path1[0] + 1, path1[1] − path2[1]], split1)
12   endif
```

```
13 end
```

```
1  T(Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1),
2   SplitParagraph(pos2, path2, split2)) :
3     if path1 < path2 then
4       return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
5     else if path1 == path2 then
6       if end1 < pos2 then
7         return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
8       else if end1 == pos2 then
9         return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, false)
10      else if start1 > pos2 then
11        return Style(start1 − pos2, end1 − pos2, [path1[0] + 1, 0], param1, value1,
12          siteId1, splitStart1, splitEnd1)
13      else if start1 == pos2 then
14        return Style(start1 − pos2, end1 − pos2, [path1[0] + 1, 0], param1, value1,
15          siteId1, false, splitEnd1)
16      else return Composite([
17        Style(start1, pos2, path1, param1, value1, siteId1, splitStart1, false),
18        Style(0, end1 − pos2, [path1[0] + 1, 0], param1, value1, siteId1, false, splitEnd1)])
19      endif
20    else if path1[0] > path2[0] then
21      return Style(start1, end1, [path1[0] + 1, path1[1]], param1, value1,
22        siteId1, splitStart1, splitEnd1)
23    else return Style(start1, end1, [path1[0] + 1, path1[1] − path2[1]], param1, value1,
24      siteId1, splitStart1, splitEnd1)
25    endif
26 end
```

A.7 Transformées par rapport à Style

```
1  T(InsertText(pos1, path1, char1, siteId1),
2   Style(start2, end2, path2, param2, value2, siteId2, splitStart2, splitEnd2)) :
3     if path1[0]! = path2[0] then return InsertText(pos1, path1, char1, siteId1)
4     else if path1[1] < path2[1] then return InsertText(pos1, path1, char1, siteId1)
5     else if path1[1] == path2[1] then
6       if pos1 < start2 then return InsertText(pos1, path1, char1, siteId1)
7       else if pos1 ≤ end2 then
8         if splitStart2 == true then
9           return InsertText(pos1 − start2, [path1[0], path1[1] + 1], char1, siteId1)
10        else return InsertText(pos1 − start2, path1, char1, siteId1)
11        endif
12      else
13        if splitStart2 == true then
14          return InsertText(pos1 − end2, [path1[0], path1[1] + 2], char1, siteId1)
15        else return InsertText(pos1 − end2, [path1[0], path1[1] + 1], char1, siteId1)
16        endif
17      endif
18    else
19    i = 0
20    if splitStart == true then i++ endif
21    if splitEnd == true then i++ endif
22    return InsertText(pos1, [path1[0], path1[1] + i], char1, siteId1)
23    endif
24 end
```

```
1  T(DeleteText(pos1, path1),
2   Style(start2, end2, path2, param2, value2, siteId2, splitStart2, splitEnd2)) :
3     if path1[0]! = path2[0] then return DeleteText(pos1, path1)
4     else if path1[1] < path2[1] then return DeleteText(pos1, path1)
5     else if path1[1] == path2[1] then
6       if pos1 < start2 then return DeleteText(pos1, path1)
7       else if (pos1 == start2 or pos1 < end2) then
8         if splitStart2 == true then
9           return DeleteText(pos1 − start2, [path1[0], path1[1] + 1])
10        else return DeleteText(pos1 − start2, path1)
11        endif
12      else
13        if splitStart2 == true then
```

```
14            return DeleteText(pos1 − end2, [path1[0], path1[1] + 2])
15          else return DeleteText(pos1 − end2, [path1[0], path1[1] + 1])
16          endif
17        endif
18      else
19      i = 0
20      if splitStart == true then i++ endif
21      if splitEnd == true then i++ endif
22      return DeleteText(pos1, [path1[0], path1[1] + i])
23      endif
24  end
```

```
1  T(NewParagraph(pos1, siteId1),
2   Style(start2, end2, path2, param2, value2, siteId2, splitStart2, splitEnd2)) :
3     return NewParagraph(pos1, siteId1)
4  end
```

```
1  T(MoveParagraph(origin1, dest1, siteId1),
2   Style(start2, end2, path2, param2, value2, siteId2, splitStart2, splitEnd2)) :
3     return MoveParagraph(origin1, dest1, siteId1)
4  end
```

```
1  T(MergeParagraph(pos1, leftChNb1),
2   Style(start2, end2, path2, param2, value2, siteId2, splitStart2, splitEnd2)) :
3     if pos1 == path2[0] + 1 then
4       i = 0
5       if splitStart2 == true then i++ endif
6       if splitEnd2 == true then i++ endif
7       return MergeParagraph(pos1, leftChNb1 + i)
8     else
9       return MergeParagraph(pos1, leftChNb1)
10    endif
11 end
```

```
1  T(SplitParagraph(pos1, path1, split1),
2   Style(start2, end2, path2, param2, value2, siteId2, splitStart2, splitEnd2)) :
3     if path1[0]! = path2[0] then return SplitParagraph(pos1, path1, split1)
4     else if path1[1] < path2[1] then return SplitParagraph(pos1, path1, split1)
5     else if path1[1] == path2[1] then
6       if pos1 < start2 then return SplitParagraph(pos1, path1, split1)
```

```
7       else if pos1 == start2 then
8         if splitStart2 == true then
9           return SplitParagraph(0, [path1[0], path1[1] + 1], false)
10        else return SplitParagraph(0, path1, false)
11        endif
12      else if pos1 < end2 then
13        if splitStart2 == true then
14          return SplitParagraph(pos1 − start2, [path1[0], path1[1] + 1], split1)
15        else return SplitParagraph(pos1 − start2, path1, split1)
16        endif
17      else
18        if splitStart2 == true then
19          return SplitParagraph(pos1 − end2, [path1[0], path1[1] + 2], split1)
20        else return SplitParagraph(pos1 − end2, [path1[0], path1[1] + 1], split1)
21        endif
22      endif
23    else
24    i = 0
25    if splitStart == true then i++ endif
26    if splitEnd == true then i++ endif
27    return SplitParagraph(pos1, [path1[0], path1[1] + i], split1)
28    endif
29  end
```

```
1   T(Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1),
2    Style(start2, end2, path2, param2, value2, siteId2, splitStart2, splitEnd2)) :
3     if path1[0]! = path2[0] then
4       return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
5     else if path1[1] < path2[1] then
6       return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, splitEnd1)
7     else if path1[1] == path2[1] then
8       if start1 < start2
9         if end1 < start2 then
10          return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, true)
11        else if end1 == start2 then
12          return Style(start1, end1, path1, param1, value1, siteId1, splitStart1, false)
13        else if end1 ≥ end2 then
14          o1 = Style(start1, end1, path1, param1, value1, siteId1, splitStart1, false)
15          o2
16          if (param1! = param2 or siteId1 < siteId2) then
```

```
17            if splitStart1 == true then
18              o2 = Style(0, end2 − start2, [path1[0], path1[1] + 2], param1, value1,
19                siteId1, false, false)
20            else
21              o2 = Style(0, end2 − start2, [path1[0], path1[1] + 1], param1, value1,
22                siteId1, false, false)
23            endif
24          else
25            if splitStart1 == true then
26              o2 = Style(0, end2 − start2, [path1[0], path1[1] + 2], param1, value2,
27                siteId1, false, false)
28            else
29              o2 = Style(0, end2 − start2, [path1[0], path1[1] + 1], param1, value2,
30                siteId1, false, false)
31            endif
32          endif
33          if end1 == end2 then
34            if splitStart1 == true then return Composite([o1, o2,
35              Style(0, end1 − end2, [path1[0], path1[1] + 3], param1, value1,
36                siteId1, false, splitEnd1)])
37            else return Composite([o1, o2,
38              Style(0, end1 − end2, [path1[0], path1[1] + 2], param1, value1,
39                siteId1, false, splitEnd1)])
40            endif
41          else return Composite([o1, o2])
42          endif
43        else
44          if (param1! = param2 or siteId1 < siteId2) then
45            if splitStart1 == true then return Composite([
46              Style(start1, end1, path1, param1, value1, siteId1, splitStart1, false),
47              Style(0, end1 − start2, [path1[0], path1[1] + 2], param1, value1,
48                siteId1, false, true)])
49            else return Composite([
50              Style(start1, end1, path1, param1, value1, siteId1, splitStart1, false),
51              Style(0, end1 − start2, [path1[0], path1[1] + 1], param1, value1,
52                siteId1, false, true)])
53            endif
54          else
55            if splitStart1 == true then return Composite([
56              Style(start1, end1, path1, param1, value1, siteId1, splitStart1, false),
```

```
57              Style(0, end1 − start2, [path1[0], path1[1] + 2], param1, value2,
58                siteId1, false, true)])
59            else return Composite([
60              Style(start1, end1, path1, param1, value1, siteId1, splitStart1, false),
61              Style(0, end1 − start2, [path1[0], path1[1] + 1], param1, value2,
62                siteId1, false, true)])
63            endif
64          endif
65        endif
66      else if start1 == start2 then
67        if end1 > end2 then
68          if (param1! = param2 or siteId1 < siteId2) then
69            if splitStart1 == true then return Composite([
70              Style(0, end2 − start2, [path1[0], path1[1] + 1], param1, value1,
71                siteId1, false, false),
72              Style(0, end1 − end2, [path1[0], path1[1] + 2], param1, value1,
73                  siteId1, false, splitEnd1)])
74            else return Composite([
75              Style(0, end2 − start2, path1, param1, value1, siteId1, false, false,
76              Style(0, end1 − end2, [path1[0], path1[1] + 1], param1, value1,
77                siteId1, false, splitEnd1)])
78            endif
79          else
80            if splitStart1 == true then return Composite([
81              Style(0, end2 − start2, [path1[0], path1[1] + 1], param1, value2,
82                siteId1, false, false),
83              Style(0, end1 − end2, [path1[0], path1[1] + 2], param1, value1,
84                siteId1, false, splitEnd1)])
85            else return Composite([
86              Style(0, end2 − start2, path1, param1, value2, siteId1, false, false),
87                Style(0, end1 − end2, [path1[0], path1[1] + 1], param1, value1,
88                  siteId1, false, splitEnd1)])
89            endif
90          endif
91        else if end1 == end2 then
92          if (param1! = param2 or siteId1 < siteId2) then
93            if splitStart1 == true then return
94              Style(0, end2 − start2, [path1[0], path1[1] + 1], param1, value1,
95                siteId1, false, false)
96            else return
```

```
97              Style(0, end2 − start2, path1, param1, value1, siteId1, false, false)
98            endif
99          else
100           if splitStart1 == true then return
101             Style(0, end2 − start2, [path1[0], path1[1] + 1], param1, value2,
102               siteId1, false, false)
103           else return
104             Style(0, end2 − start2, path1, param1, value2, siteId1, false, false)
105           endif
106         endif
107       else
108         if (param1! = param2 or siteId1 < siteId2) then
109           if splitStart1 == true then return
110             Style(0, end1 − start1, [path1[0], path1[1] + 1], param1, value1,
111               siteId1, false, true)
112           else return
113             Style(0, end1 − start1, path1, param1, value1, siteId1, false, true)
114           endif
115         else
116           if splitStart1 == true then return
117             Style(0, end2 − start2, [path1[0], path1[1] + 1], param1, value2,
118               siteId1, false, true)
119           else return
120             Style(0, end1 − start1, path1, param1, value2, siteId1, false, true)
121           endif
122         endif
123       endif
124     else
125       if start1 > end2 then
126         if splitStart2 == true then
127           return Style(start1 − end2, end1 − end2, [path1[0], path1[1] + 2], param1, value1,
128             siteId1, true, splitEnd1)
129         else
130           return Style(start1 − end2, end1 − end2, [path1[0], path1[1] + 1], param1, value1,
131             siteId1, true, splitEnd1)
132         endif
133       else if start1 == end2 then
134         if splitStart2 == true then
135           return Style(start1 − end2, end1 − end2, [path1[0], path1[1] + 2], param1, value1,
136             siteId1, false, splitEnd1)
```

```
137         else
138           return Style(start1 − end2, end1 − end2, [path1[0], path1[1] + 1], param1, value1,
139             siteId1, false, splitEnd1)
140         endif
141       else if end1 < end2 then
142         if (param1! = param2 or siteId1 < siteId2) then
143           if splitStart2 == true then
144             return Style(start1 − start2, end1 − start2, [path1[0], path1[1] + 1], param1, value1,
145               siteId1, true, true)
146           else
147             return Style(start1 − start2, end1 − start2, [path1[0], path1[1] + 1], param1, value1,
148               siteId1, true, true)
149           endif
150         else
151           if splitStart2 == true then
152             return Style(start1 − start2, end1 − start2, [path1[0], path1[1] + 1], param1, value2,
153               siteId1, true, true)
154           else
155             return Style(start1 − start2, end1 − start2, [path1[0], path1[1] + 1], param1, value2,
156               siteId1, true, true)
157           endif
158         endif
159       else if end1 == end2 then
160         if (param1! = param2 or siteId1 < siteId2) then
161           if splitStart2 == true then
162             return Style(start1 − start2, end1 − start2, [path1[0], path1[1] + 1], param1, value1,
163               siteId1, true, false)
164           else
165             return Style(start1 − start2, end1 − start2, [path1[0], path1[1] + 1], param1, value1,
166               siteId1, true, false)
167           endif
168         else
169           if splitStart2 == true then
170             return Style(start1 − start2, end1 − start2, [path1[0], path1[1] + 1], param1, value2,
171               siteId1, true, false)
172           else
173             return Style(start1 − start2, end1 − start2, [path1[0], path1[1] + 1], param1, value2,
174               siteId1, true, false)
175           endif
176         endif
```

```
177         else
178           if (param1! = param2 or siteId1 < siteId2) then
179             if splitStart2 == true then return Composite([
180               Style(start1 − start2, end2 − start2, [path1[0], path1[1] + 1], param1, value1,
181                 siteId1, true, false),
182               Style(0, end1 − end2, [path1[0], path1[1] + 2], param1, value1
183                 , siteId1, false, splitEnd1)])
184             else return Composite([
185               Style(start1 − start2, end2 − start2, path1, param1, value1,
186                 siteId1, true, false),
187               Style(0, end1 − end2, , [path1[0], path1[1] + 1], param1, value1,
188                 siteId1, false, splitEnd1)])
189             endif
190           else
191             if splitStart2 == true then return Composite([
192               Style(start1 − start2, end2 − start2, [path1[0], path1[1] + 1], param1, value2,
193                 siteId1, true, false),
194               Style(0, end1 − end2, [path1[0], path1[1] + 2], param1, value1,
195                 siteId1, false, splitEnd1)])
196             else return Composite([
197               Style(start1 − start2, end2 − start2, path1, param1, value2, siteId1, true, false),
198               Style(0, end1 − end2, [path1[0], path1[1] + 1], param1, value1,
199                 siteId1, false, splitEnd1)])
200             endif
201           endif
202         endif
203       endif
204     else
205       i = 0
206       if splitStart2 == true then i++ endif
207       if splitEnd2 == true then i++ endif
208       return Style(start1, end1, [path1[0], path1[1] + i], param1, value1,
209         siteId1, splitStart1, splitEnd1)
210     endif
211 end
```

Bibliographie

[Adelson-Velskii and Landis, 1962] Georgii Maximovich Adelson-Velskii and Evgenii Mikhailovich Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146 :263–266, 1962. (English translation in Soviet Mathematics Doklady, vol 3, pp. 1259–1263).

[Ahmed-Nacer *et al.*, 2011] Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating CRDTs for Real-time Document Editing. In ACM, editor, *11th ACM Symposium on Document Engineering*, DocEng, pages 103–112, Mountain View, California, United States, September 2011.

[Ahmed-Nacer, 2015] Mehdi Ahmed-Nacer. *Evaluation methodology for replicated data types.* Thèse de doctorat, Université de Lorraine, May 2015.

[André *et al.*, 2012] Luc André, Claudia-Lavinia Ignat, and Gérald Oster. Collaboration over Wiki Content. In *IWCES'12 - The Twelfth International Workshop on Collaborative Editing Systems*, Seattle, United States, February 2012.

[André *et al.*, 2013] L. André, S. Martin, G. Oster, and C. L. Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *Collaborative Computing : Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*, pages 50–59, Oct 2013.

[Bernstein and Goodman, 1981] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2) :185–221, June 1981.

[Davis *et al.*, 2002] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. Generalizing operational transformation to the standard general markup language. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, CSCW '02, pages 58–67, New York, NY, USA, 2002. ACM.

[Ellis and Gibbs, 1989] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2) :399–407, June 1989.

[Ellis *et al.*, 1991] Clarence A Ellis, Simon J Gibbs, and Gail Rein. Groupware : some issues and experiences. *Communications of the ACM*, 34(1) :39–58, 1991.

[Fidge, 1991] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8) :28–33, 1991.

[Gutwin and Greenberg, 1999] Carl Gutwin and Saul Greenberg. The effects of workspace awareness support on the usability of real-time distributed groupware. *ACM Trans. Comput.-Hum. Interact.*, 6(3) :243–281, September 1999.

[Ignat and Norrie, 2002] Claudia-Lavinia Ignat and Moira C. Norrie. Tree-based Model Algorithm for Maintaining Consistency in Real-time Collaborative Editing Systems. *Fourth International Workshop on Collaborative Editing, CSCW 2002, IEEE Distributed Systems online*, November 2002.

[Imine *et al.*, 2003] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving Correctness of Transformation Functions in Real-Time Groupware. In Kari Kuutti, EijaHelena Karsten, Geraldine Fitzpatrick, Paul Dourish, and Kjeld Schmidt, editors, *ECSCW 2003*, pages 277–293. Springer Netherlands, 2003.

[Lamport, 1978] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21 :558–565, July 1978.

[Li and Li, 2006] Du Li and Rui Li. A performance study of group editing algorithms. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, pages 8 pp.–, 2006.

[Mattern, 1989] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23) :215–226, 1989.

[Nédelec *et al.*, 2013] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. Lseq : An adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng '13, pages 37–46, New York, NY, USA, 2013. ACM.

[Nichols *et al.*, 1995] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, UIST '95, pages 111–120, New York, NY, USA, 1995. ACM.

[Oster *et al.*, 2006a] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *Collaborative Computing : Networking, Applications and Worksharing, 2006. CollaborateCom 2006. International Conference on*, pages 1–10, Nov 2006.

[Oster *et al.*, 2006b] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p collaborative editing. *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 259–267, November 2006.

[Preguica *et al.*, 2009] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 395–403, June 2009.

[Ressel *et al.*, 1996] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors.

In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297, New York, NY, USA, 1996. ACM.

[Roh *et al.*, 2011] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types : Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3) :354–368, 2011.

[Saito and Shapiro, 2005] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1) :42–81, March 2005.

[Shapiro *et al.*, 2011] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011.

[Suleiman *et al.*, 1997] Maher Suleiman, Michèlle Cart, and Jean Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work : the integration challenge*, GROUP '97, pages 435–445, New York, NY, USA, 1997. ACM.

[Sun and Ellis, 1998] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors : issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

[Sun and Sun, 2009] David Sun and Chengzheng Sun. Context-based operational transformation in distributed collaborative editing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 20(10) :1454–1470, Oct 2009.

[Sun *et al.*, 1998] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5 :63–108, March 1998.

[Sun *et al.*, 2006] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction*, 13(4) :531–582, December 2006.

[Vidot *et al.*, 2000] Nicolas Vidot, Michèlle Cart, Ferrié Jean, and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, CSCW '00, pages 171–180, New York, NY, USA, 2000. ACM.

[Weiss *et al.*, 2007] Stéphane Weiss, Pascal Urso, and Pascal Molli. Wooki : a p2p wiki-based collaborative writing tool. In *Web Information Systems Engineering–WISE 2007*, pages 503–512. Springer, 2007.

[Weiss *et al.*, 2009] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot : a scalable optimistic replication algorithm for collaborative editing on p2p networks. *International Conference on Distributed Computer Systems (ICDCS)*, jun 2009.

[Weiss *et al.*, 2010] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo : Distributed collaborative editing system on p2p networks. *Parallel and Distributed Systems, IEEE Transactions on*, 21(8) :1162–1174, Aug 2010.

[Wuu and Bernstein, 1984] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

[Xia *et al.*, 2004] Steven Xia, David Sun, Chengzheng Sun, David Chen, and Haifeng Shen. Leveraging single-user applications for multi-user collaboration : the coword approach. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 162–171. ACM, 2004.

[Yu, 2012] Weihai Yu. A string-wise crdt for group editing. In *Proceedings of the 17th ACM International Conference on Supporting Group Work*, GROUP '12, pages 141–144, New York, NY, USA, 2012. ACM.

[Zafer, 2001] Ali Asghar Zafer. *Netedit : A collaborative editor.* PhD thesis, Citeseer, 2001.

Résumé

L'édition collaborative en temps réel permet à plusieurs utilisateurs d'éditer un même document simultanément grâce à des outils informatiques. Les applications d'édition collaborative en temps réel, telles GoogleDocs ou Etherpad, répliquent les données éditées chez chaque utilisateur, pour garantir une édition des données réactive et possible à chaque instant. Les conflits d'édition sont fréquents, et doivent être gérés automatiquement par l'application. L'application doit faire converger toutes les répliques vers un document commun, qui contient toutes les modifications exprimées par tous les utilisateurs.

Les algorithmes actuels fonctionnent de manière satisfaisante pour des types de données simples (des documents linéaires) et des possibilités d'édition minimes (insérer ou supprimer du texte). Lorsque le document est plus complexe (document XML, texte structuré), ou qu'il peut être édité avec un ensemble élargi d'opérations (déplacement de texte, styliser du texte), lors de la résolution de conflits d'édition, les algorithmes échouent à proposer un contenu qui convienne aux utilisateurs. Les intentions des utilisateurs ne sont pas respectées. L'objectif de cette thèse est de proposer des algorithmes d'édition collaborative en temps réel qui respectent mieux les intentions des utilisateurs que les algorithmes actuels.

La première contribution de cette thèse est un algorithme basé sur l'approche des transformées opérationnelles (OT), qui permet d'éditer et de styliser du texte hiérarchisé tout en respectant les intentions de style, de déplacement ou d'insertion de texte ou de paragraphes. Cet algorithme a donné lieu à un transfert technologique pour un de nos partenaires industriels.

La deuxième contribution est basée sur l'approche des modèles de données répliquées commutatives (CRDT), et propose un algorithme capable de respecter l'intention de mise à jour d'un élément, tout en améliorant les performances globales de l'approche par rapport aux autres CRDTs. Notre algorithme a débouché sur la création d'une application web d'édition collaborative en temps réel, en continuelle amélioration par notre équipe de recherche.

Abstract

Real-time collaborative editors, like GoogleDocs or Etherpad, allow the simultaneous edition of a document by several users. These applications need to replicate the edited document, for the so called real-time purpose of permitting a fast and reactive editing by any user at any time. Editing conflicts frequently occur, and must be automatically handled by the application, in order to provide every users with the same copy of the document, containing every modifications issued.

Most of current real-time collaborative editing algorithms were designed for simple data structures, like linear text, and simple editing ways, like inserting or removing a character only. These algorithms fail to offer an appropriate editing conflict resolution when used with a complex data structure, like XML, or with complex operations, like moving some text or adding some style. Copies are the sames but users' intentions are not preserved. The goal of this thesis is to design new real-time collaborative editing algorithms that ensure a better preservation of users' intentions.

The first contribution of this thesis is an algorithm based on the Operational Transformation approach (OT). Our contribution is designed to handle rich text document (with stylized text and paragraphs) and to preserve the intentions of a set of high editing level operations (add style, merge paragraphs...). This contribution has lead to an industrial transfer for our industrial partner.

The second contribution uses the Commutative Replicated Data Types approach (CRDT), and offers an algorithm which preserves the update intention, while improving global performance of the approach when dealing with large blocs of data. This contribution has lead to a web real-time collaborative editing application developed by our research team, currently available and in continuous improvement.