Aix-Marseille Université
Laboratoire d'Informatique
Fondamentale de Marseille

Thèse

présentée pour obtenir le grade de
Docteur d'Aix-Marseille Université
Délivré par l'Université de Provence
Spécialité: Informatique

par

Stéphane Martin, LIF

Édition collaborative des documents semi-structurés

Thèse soutenue publiquement le 8 septembre 2011

Composition du jury

Les rapporteurs :

M.me	Véronique Benzaken,	Professeur à l'Université Paris-Sud XI au LRI
M.	Pascal Molli,	Professeur à l'Université de Nantes au LINA

Les examinateurs :

M.	Marc Shapiro,	Professeur à l'UPMC paris, au LIP6
M.	Michaël Rusinowitch,	Professeur à l'INRIA Nancy, au LORIA
M.	Rémi Morin,	Professeur à Aix-Marseille Université, au LIF

Directeur de thèse :

M.	Denis Lugiez,	Professeur à Aix-Marseille Université au LIF

Mots-clés : Edition collaborative, Pair-à-Pair, Documents semi-structurés, Opération d'édition commutatives (CRDT), Transformées opérationnelles, Type pour XML

Remerciements

Le doctorat n'est pas seulement une aventure scientifique, mais aussi une aventure humaine. Je tiens à remercier toutes les personnes qui m'ont aidé durant ma thèse.

Je remercie tout d'abord mon directeur de thèse Denis Lugiez, pour m'avoir proposé ce sujet et guidé pendant toutes ces années. Je le remercie aussi d'avoir passé beaucoup de temps et d'énergie à me relire, me conseiller et faire des critiques constructives. Malgré la fonction de directeur d'UFR qui lui prend pas mal de temps.

A Pascal Urso et Abdessamad Imine avec qui j'ai pu avoir des discussions passionnantes et qui m'ont fait progressé dans ce domaine.

Je remercie mes rapporteurs Pascal Molli et Véronique Benzaken d'avoir pris le temps de lire et d'avoir écrit un rapport aussi rapidement.

Je remercie aussi les membres du jury pour l'attention particulière qu'ils ont portée à mes travaux.

J'aimerais remercier les membres du lif et du cmi, qui m'on suivit pendant mes études et accueilli dans leur laboratoire. Sans oublier les doctorants du LATP avec qui j'ai passé de bons moments.

Je voudrais aussi remercier toutes les personnes de la 301 crew m'ont supporté pendant les moments de stress.

Je remercie bien évidemment ma mère et mes amis proches.

Pour finir, je remercie du fond du coeur ma compagne qui a accepté de déménager loin de sa famille et se retrouver au chômage pour que je puisse continuer cette thèse.

Table des matières

Introduction

1

1.1 Contexte

À l'heure du web 2.0, internet devient de plus en plus interactif. Dans cette évolution, il devient aussi de plus en plus collaboratif avec le développement de systèmes permettant de collaborer afin de réaliser une tâche. Cette tâche peut être l'édition d'un document commun, et on parle alors d'éditeur collaboratif (dans le monde scientifique cvs et svn sont devenus des standards pour l'écriture d'articles ou la réalisation de code partagé). D'un autre coté, le développement et la démocratisation d'internet alliés à la puissance toujours accrue des ordinateurs tout public ont poussé les acteurs du web à se tourner vers des solutions décentralisées. Ce travail est consacré à la réplication "optimiste" dans un milieu pair-à-pair. Dans cette approche tous les sites ont des répliques du document original. Chacun modifie le document à sa guise et envoie ses modifications aux autres sites. Nous supposons qu'aucun message n'est perdu (ce qui ressort d'une autre problématique), mais le système est asynchrone et donc en général les opérations arrivent dans un ordre différent de l'ordre d'édition et peuvent être en conflit. Une première difficulté est donc de faire converger les différentes répliques une fois que le système est au repos, c'est-à-dire que plus aucune opération d'édition n'est effectuée et que tous les messages émis ont été reçus. Une deuxième difficulté est de concevoir un algorithme passant à l'échelle dans un milieu pair-à-pair. En effet, les collaborations deviennent massives avec de très nombreux sites participant à l'édition d'un document, ce qui peut rendre inefficaces des solutions de complexité satisfaisante pour un nombre de participants limités.

Simultanément, le format XML s'est imposé comme référence pour la manipulation de documents textuels ou même des données produites par des logiciels (par exemple, des feuilles Excel seront traduites en document XML). Ce format sert de base à de nombreuses applications[Cov] : RSS, SVG, SOAP, et XHTML.

Dans cette thèse, nous allons nous intéresser aux documents semi-structurés -qui sont une abstraction du format XML sous forme d'arbres étiquetés- comme objets de base manipulés par un éditeur collaboratif.

1.2 Différentes approches

De nombreuses approches d'édition collaborative en pair-à-pair ont été proposées dans la littérature. Peu de ces approches modélisent des objets complexes tels que des arbres ou des graphes. La grande majorité des éditeurs traite le cas des mots, qui sont la forme la plus simple de documents textuels, à savoir une suite de caractères. Beaucoup d'éditeurs proposés dans ce cadre rencontrent des difficultés pour assurer la convergence. Généralement, ils voient un texte comme des suites de caractères, lignes ou mots, la position d'un objet dépend des positions des objets précédents. Lors d'insertions ou de suppressions concurrentes, calculer les positions des nouveaux objets ou des objets à supprimer devient vite un "casse tête".

Certaines approches détectent les conflits et élisent un "leader" pour le résoudre [KRSD01]. D'autres approches imposent le verrouillage de la ressource[Ber90] ou un ordre total sur les opérations[MOSmI03 ; WML10]. Ces modèles ne sont adaptés à un modèle pair-à-pair dans lequelle il n'y a aucune centralisation. Une autre approche basée sur la notion de *transformée opérationnelle* modifie au fur et à mesure les opérations concurrentes reçues en fonction de celles déjà traitées. Cette fonction d'intégration peut être compliquée à concevoir et la propriété de convergence repose sur cette transformation. Ressel [RNRG96] a donné un algorithme qui assure la convergence dans ce cadre à condition que la transformée satisfasse deux propriétés. Ces propriétés sont suffisantes, mais on ne sait pas si elles sont nécessaires. Cette méthode a pour avantage de pouvoir faire converger les documents sans contraintes (ordre total, verrous) et sans pertes de mises à jour, même si les opérations ne sont pas naturellement commutatives. Par contre, trouver des transformées qui satisfont les propriétés requises se révèle extrêmement difficile, et actuellement, il n'en n'existe aucune qui satisfasse les propriétés exigées par l'algorithme de Ressel pour les mots.

L'approche du Commutative Replicated Data Type (CRDT) est une autre façon d'aborder le problème. Au lieu de vouloir transformer des opérations naturellement non commutatives, le but est de concevoir des structures de données et des opérations qui commutent. Ainsi pour le cadre des mots on remplace la numérotation consécutive partant du premier caractère à une numérotation relative [PMSL09 ; OUMaI06] ou absolue (ne changeant pas pendant l'édition) [WUM09]. La structure de données a été légèrement compliquée, mais l'algorithme d'édition devient très efficace.

1.3 Contributions

La première contribution (section 4.2), est de proposer une structure de données de document semi-structurés "simple" (similaire à celle proposée par Pierce et al. [PSG03]) avec des opérations d'édition de base permettant de faire fonctionner le modèle des transformées opérationnelles. Nous avons montré également qu'un jeu d'opérations légèrement différent ne permet pas de faire fonctionner cette approche, quelle que soit la transformée opérationnelle utilisée. Dans un deuxième temps, nous proposons une structure de données un peu plus compliquée, dont l'idée essentielle

est d'utiliser des identifiants uniques pour l'étiquetage des arêtes dans le document. Ces identifiants uniques permettent d'accéder de manière non-ambigüe dans le document. Par ailleurs, ils sont donnés gratuitement par le processus d'édition et ne demandent pas de travail supplémentaire à effectuer sur le document. Nous proposons des opérations et des transformées opérationnelles qui garantissent les propriétés requises pour la convergence. Enfin, nous proposons une généralisation des arbres aux graphes de fonction qui permet également de prendre en compte l'opération de déplacement d'un sous-arbre avec une méthode de transformée opérationnelle.

Une deuxième contribution est la conception d'un algorithme générique de type CRDT (chapitre 5) qui assure la propriété de convergence grâce à une notion de dépendance sémantique. Cet algorithme n'utilise pas l'historique, passe à l'échelle, a été validé par un prototype, et il est décrit dans la section 5.3. Nous proposons ensuite une structure de données pour les documents semi-structurés et un jeu d'opérations satisfaisant la propriété d'indépendance requise par l'algorithme pour assurer la convergence. Dans le chapitre 5, nous avons poussé la modélisation jusqu'au format XML complet en utilisant toujours notre algorithme générique. Nous avons également ajouté une opération d'annulation d'opération extrêmement utile en pratique.

Une autre contribution consiste à prendre en compte la notion de typage dans l'édition, en adaptant l'algorithme générique de manière à pouvoir garantir que la convergence est établie et que de plus le document est bien-typé. Cette solution est valide pour de nombreux schémas de type de documents (dont les plus classiques comme les DTD) pourvu qu'existe un algorithme de réparation pour les types considérés utilisant les opérations d'éditions usuelles (cet algorithme calcule les opérations d'édition à effectuer pour que le document soit bien typé).

Nous avons complété les résultats de l'approche CRDT en prenant en compte l'opération de déplacement d'un sous-arbre en plus des opérations classiques, ce qui est possible en passant par une structure de données de type graphe. Les documents semi-structurés correspondent alors à un cas particulier de ces graphes. Nous donnerons la généralisation des résultats précédents pour en vue de réaliser cette opération de déplacement. Ce qui n'était pas pris en compte dans les travaux antérieurs.

La dernière contribution (chapitre 9) est la réalisation d'un prototype dans lequel nous avons implémenté nos solutions afin de les valider. Ce prototype a permis de tester les algorithmes et de valider l'étude théorique que nous avons faite.

Éditeur Collaboratif 2

2.1 INTRODUCTION

Depuis ces dernières années, le réseau internet est devenu d'usage courant et la puissance de calcul des ordinateurs a considérablement augmenté. Le travail collaboratif assisté par ordinateur (en anglais Computer Supported Cooperative Work - CSCW), prend de plus en plus d'importance tant dans le monde industriel que dans le monde académique et dans l'usage grand public de l'outil informatique (Wiki, Réseaux sociaux,...). Le principe est de faire collaborer des personnes situées en des lieux différents sur un projet commun en interagissant à distance. Voici quelques domaines d'applications :

- Le développement de logiciels "open source" avec CVS qui permet de faire travailler ensemble un bon nombre de développeurs sur un même projet.
- La rédaction d'articles par plusieurs chercheurs travaillant ensemble, mais répartis sur des sites distants.
- La gestion de la production et de la vente dans une multinationale exige la collaboration entre différents acteurs.
- Les systèmes de régulation de trafic visant à détecter les embouteillages ou les perturbations en temps réel par communication entre véhicules.
- Les forums interactifs de discussions.
- De manière plus générale, la réalisation de documents numériques avec plusieurs acteurs travaillant ensemble.

Un système collaboratif est un système partageant un objet (texte, image, son, ...) modifiable par chaque participant. Un tel système, pour être utilisable se doit, d'être de haute disponibilité, réactif et simple pour l'utilisateur afin que ce dernier ne perde pas de temps.

2.2 Bref historique et Motivations

Au début des moyens modernes de communication, il était possible de travailler à distance par exemple en s'envoyant des fax, puis des fichiers informatiques. Conceptuellement, ces deux procédés n'ont pas de différence. Ces procédés sont peu pratiques. Des approches plus évoluées se sont donc développées :

SCCS En 1972 Marc J. Rochkind travaillant à Bell Labs a développé le premier logiciel de contrôle de version appelé Source Code Control System (SCCS)[Roc75]. Son principe est de dater les différents fichiers avec un numéro de version. Ce système n'est pas réellement interactif, puisque le numéro de version est propre à chaque utilisateur et il n'y pas de possibilité de résoudre les conflits entre les utilisateurs. Ce modèle est justifié, car l'architecture standard d'un centre informatique était d'avoir une machine centrale et des terminaux reliés à cette machine, les infrastructures réseaux étant limitées voire inexistantes. Donc les problèmes liés au travail collaboratif se limitaient au partage de fichiers sur une même machine.

Usenet En 1979, l'idée du célèbre forum de discussion Usenet[Sal92] germa dans les têtes de Tom Truscott et Jim Ellis, alors étudiants. Aidés par Steve Bellovin, ils relièrent le serveur de l'université de Duke avec celle de la Caroline du Nord à l'aide d'un script Bourn Shell en utilisant le protocole UUCP [1]. Il fut rapidement relié au réseau ARPANET via l'université de Berkley.

Le principe d'usenet est d'héberger plusieurs forums de discussion répliqués sur plusieurs serveurs. Pour assurer la convergence des répliques, il utilise les règles de Thomas [Tho79] (définies pour assurer la convergence des bases de données distribuées), ce qui peut faire perdre des modifications en cas de conflit, lié au remplacement d'un message par le plus récent. Cela ne pose guère de problème dans un forum de discussion, mais n'est pas adéquat pour l'édition collaborative.

RCS Revision Control System (RCS)[Tic85], développé en 1982, ajoute la gestion et la fusion de plusieurs fichiers d'une même version, procurant une première approche pour la gestion de conflits. En effet un projet ou un fichier n'a pas forcément de développement linéaire. A partir de certains nœuds, il se peut que des personnes partent dans des directions divergentes. Cela conduit à des fichiers différents qui ont chacun le statut de dernière version. C'est au développeur de décider si cela correspond à des branches distinctes du projet ou si ce sont des variantes qui doivent être fusionnées.

CVS Les aspects réseaux vont apparaitre en 1986 avec Concurrent Versions System (CVS) conçu par Dick Grune et complété par Brian Berliner (à l'aide de CVS) en 1989 [Ber90]. Il reprend de façon plus élaborée le concept de branche : Un projet n'évolue pas forcément linéairement, mais selon un arbre. Il peut aussi étiqueter les différentes branches. Il introduit aussi le système de client serveur dans lequel la copie de référence est celle du serveur central. Chaque utilisateur ayant récupéré les fichiers ("CheckOut") peut les modifier et les renvoyer au serveur central ("Commit"). Chacun peut voir les modifications des autres participants en mettant à

[1] Unix-to-Unix Copy Protocol : un ensemble de programmes utilisés pour envoyer des fichiers ou des commandes à distance. Il pouvait être utilisé via un réseau privé ou une ligne téléphonique équipée d'un modem. Il était déjà utilisé pour les courriers électroniques.

jour leur copie locale ("Update") à partir de la copie du serveur central. Les opérations de "commit" sont atomiques. Lorsqu'une personne fait un commit le système s'assure qu'il part de la dernière version du fichier publié par le serveur. Dans le cas contraire, il doit faire un "update" qui modifie sa version locale. C'est à l'utilisateur de s'assurer qu'après cette opération les fichiers sont encore fiables.

Ces deux derniers projets sont "open source" et s'améliorent au fil du temps.

GROVE Elis et Gibbs ont introduit en 1989 [EG89] les transformées opérationnelles avec leur éditeur GROVE. Malheureusement, ils n'ont pas prouvé formellement l'unicité de la copie de référence (propriété de convergence). En fait, on verra au chapitre suivant que leur système ne converge pas. Cela pose une question difficile à résoudre : Peut-on assurer la convergence dans ce modèle de transformée opérationnelle : C'est-à-dire : est-ce que toutes les personnes ont le même document à la fin de l'édition ? De nombreux chercheurs se sont posé cette question et ont proposé des solutions qui se révèlent incorrectes car elles n'assurent pas la convergence. Le chapitre suivant détaille ce modèle et donne un état de l'art de cette approche.

TeamWare Au début des années 90 Sun Microsystem lance TeamWare qui propose de distribuer des copies de référence. De plus, le document de référence n'est pas sur un seul serveur, mais découpé en plusieurs parties qui sont déposées sur plusieurs serveurs. TeamWare permet aussi de regrouper des opérations élémentaires en une opération complexe atomique et aucune opération ne peut s'intercaler dans la suite des opérations correspondant à une opération atomique.

Bitkeeper & Git [git ; Toro5] En 1998, Bitkeeper de McVoy, développé par BitMover, est un outil pour le contrôle de versions de logiciels basé sur une approche pair-à-pair. Git (2005 par Linus Torvalds) et Mercurial (2005 par Matt Mackall) sont basés sur le même principe que Bitkeeper et sont "open source"s. Leur principe est de cloner un dépôt contenant tout l'historique avec les différentes branches et de pouvoir fusionner les clones en générant éventuellement d'autres branches en cas de conflit. On s'approche de l'édition distribuée, mais les fusions et la génération de branche sont très contraignantes.

SVN En 2000, apparait le projet Subversion (SVN), qui fonctionne sur le même principe que CVS, mais avec quelques améliorations comme le verrou de fichier, la gestion des fichiers binaires et qui améliore la sécurité (utilisation de https) entre autres.

IceCube En 2001, IceCube [KRSD01] est un système d'édition collaborative utilisant un algorithme de réconciliation générique. Chaque site possède un historique qui répertorie la suite des opérations effectuées. En cas

de conflit, l'algorithme élit un site responsable en utilisant une méthode classique. Ce site résout le conflit en calculant à partir des historiques des sites en conflit la suite des opérations à effectuer qui minimise le coût de réparation. Puis cette suite est envoyée à tous les sites qui vont se conformer à cette suite. Cela peut amener un site à défaire certaines opérations effectuées et à en effectuer de nouvelles. De nouveaux conflits peuvent apparaitre lors de ce processus amenant à de nouvelles résolutions de conflits. Cet algorithme ne passe pas à l'échelle car d'une part il impose de garder un historique des opérations, mais surtout car il impose d'effectuer un algorithme d'élection à chaque apparition de conflit. Cela revient à introduire de la centralisation dans un système pair-à-pair. Des concurrents propriétaires d'IceCube existent comme Clear Case (IBM), Visual SourceSafe (Microsoft)

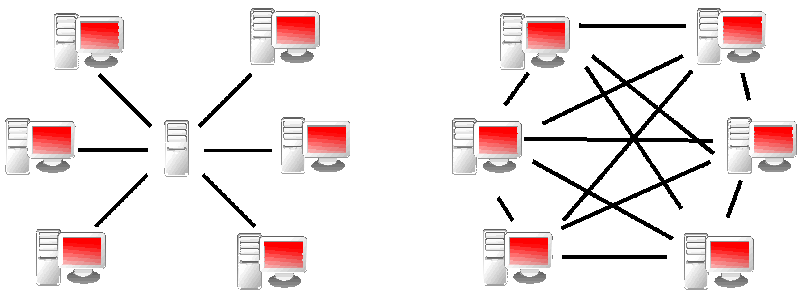
Darcs Darcs est un gestionnaire de version asynchrone dédié à l'édition de programmes. Il groupe les modifications d'un document dans *un patch*. Chaque patch peut dépendre d'un autre patch pour être exécuté, et cette relation de dépendance n'est pas un ordre total[Ka]. La convergence repose sur le fait que deux patchs indépendants devraient pouvoir commuter en donnant le même document. Cette commutativité de patch n'est malheureusement pas prouvée. Une opération a été introduite dans darcs 2.0 appelée le "conflictor" permettant de marquer les patchs étant en conflit (ne pouvant pas être intégrés ou ne pouvant pas commuter)[Jaco9]. C'est à l'utilisateur d'intégrer dans le document les modifications des patchs dits en conflit par le système.

2.3 Modèles d'édition collaborative

2.3.1 Modèles pair-à-pair et centralisés

Nous distinguons deux types de modèles de réseaux :

- Le modèle centralisé.
- Le modèle pair-à-pair.



(a) Serveur central (b) Pair-à-pair

Fig. 2.1 – *Différents modèles*

Pour simplifier, une application distribuée est vue comme un calcul. Dans le système centralisé, ce calcul est effectué par le serveur central qui utilise les clients pour effectuer certains calculs et qui centralise les résultats obtenus pour fournir le résultat final. En particulier dans le système centralisé, les clients communiquent avec le serveur central, mais n'ont pas besoin de communiquer entre eux. Dans le système pair-à-pair, le calcul est complètement délocalisé et chaque site communique avec les autres sites. Aucun site n'est privilégié et tous participent à la réalisation du calcul final. La figure 2.3.1 illustre les deux types de modèles.

Une notion apparue récemment est celle du *Cloud computing* qui consiste à distribuer les calculs faits normalement sur un serveur central sur d'autres serveurs distants. Dans cette thèse ce modèle est considéré comme centralisé, car l'intelligence est toujours en un point et non en périphérie. Google wave[WML10] utilise un nuage pour le coordinateur central de Jupiter[NCDL95]. Ce qui fait que le système passe à l'échelle dans le sens où beaucoup de personnes peuvent se connecter à google wave, mais cela exige d'avoir une grande puissance de calcul.

Il existe plusieurs modèles intermédiaires entre centralisé et pair-à-pair. Le but est de distribuer le rôle ou une partie du rôle du serveur central sur des pairs tout en gardant des propriétés suffisantes au bon fonctionnement du service. En effet certaines propriétés sont non assurées ou difficilement assurées par un système purement décentralisé. Par exemple en P2P, il est difficile de :
– connaître à tout instant le nombre de participants ;
– verrouiller une ressource ;
– etc.
Par exemple : BitTorrent utilise un serveur qui répertorie toutes les adresses IP des ordinateurs ayant téléchargé les parties de ce fichier et leur charge. Ce qui permet d'optimiser les flux et donc d'augmenter la vitesse de téléchargement.

2.3.2 Optimistes vs Pessimistes

Deux approches sont possibles pour l'édition collaborative : l'approche pessimiste[BHG87] et l'approche optimiste [SS05]. L'approche pessimiste consiste à autoriser la lecture à partir de n'importe quelle réplique ce qui entraine que les mises à jour doivent être synchronisées et coordonnées, généralement par un site central. Ces mises à jour nécessitent usuellement de verrouiller le document ou une partie du document pendant la mise à jour et la propagation de celle-ci. En opposition avec ce mécanisme, l'approche optimiste, ne demande pas de mises à jour immédiates, mais accepte les différentes mises à jour au fur et à mesure de l'édition. Cette approche laisse diverger les copies locales, [2] mais elles doivent finir par converger sur un document unique lorsque toutes les mises à jour ont été faites. Nous verrons qu'assurer cette convergence n'est pas trivial et le processus d'édition peut créer des conflits entre les opérations. Par exemple, deux mises à jour simultanées choisissant deux couleurs distinctes sur un

[2] chaque site peut passer par différents états du document.

même objet seront en conflit. Malgré ces problèmes, cette approche reste la plus adaptée à l'édition en pair-à-pair.

2.3.3 Propriétés attendues

Nous passons en revue les avantages et inconvénients des deux types de modèles vis-à-vis d'un certain nombre de propriétés voulues pour l'édition collaborative.

Ressources

L'application ne doit pas nécessiter de ressources trop importantes, une ressource étant un composant physique du réseau, machines incluses (puissance en terme de CPU, mémoire, débit requis pour le réseau,...). Les applications distribuées sont un exemple de partage de ressources permettant de limiter la taille des machines utilisées.

Passage à l'échelle

L'application doit toujours fonctionner de manière satisfaisante lorsque le nombre d'utilisateurs ou la taille des données utilisées augmente.

Centralisé Le serveur central et les communications avec celui-ci sont le goulot d'étranglement du modèle centralisé. Dès que le volume de communication requis par l'application est trop important, le site central risque d'être saturé. De même si le traitement des calculs par le site central est trop coûteux, celui-ci ne pourra pas effectuer le traitement nécessaire.

Pair-à-pair Le problème du passage à l'échelle dans le modèle décentralisé est un problème algorithmique. C'est-à-dire que les ressources se greffent au réseau au fur et à mesure que le nombre d'utilisateurs augmente. Il n'y a pas d'architecte régissant les charges réseau ou de calcul. C'est à l'algorithme de gérer au mieux les ressources disponibles. Certains algorithmes ont des difficultés à passer à l'échelle comme les algorithmes de diffusion basés sur un ordre total[DSU03] contrairement au modèle basé sur une diffusion épidémique (chacun passe l'information à ses voisins les plus proches) [EGmKM04].

Dynamicité

Un réseau dynamique permet aux pairs d'entrer ou de sortir en tout temps sans que les autres subissent une dégradation du service.

Centralisé Dans la limite des ressources disponibles, tous les utilisateurs peuvent se connecter et se déconnecter du service sans l'affecter. On peut connaître facilement le nombre d'utilisateurs. La géométrie du réseau ne change pas. Elle est toujours en étoile.

Pair-à-pair L'entrée et la sortie d'utilisateur dans un réseau pair-à-pair peut être délicate à gérer. On peut observer des dégradations du

service dans les cas simples : un pair se déconnecte pendant qu'il faisait la liaison entre deux réseaux ; qu'il avait un rôle de chef de groupe, etc ... Il se peut aussi que certains algorithmes aient besoins de connaitre la topologie du réseau [EG09] ou le nombre exact de participants en tout temps [HKP+05].

La résistance aux pannes

L'application réseau doit être capable de continuer à fonctionner si l'un de ses composants (site, connection réseau,...) tombe en panne. De nombreux algorithmes distribués sont conçus pour être résistants aux pannes.

Réseau Centralisé Par définition, un réseau de type client-serveur repose sur un serveur. Le service n'est plus assuré s'il est en panne. Par contre, un réseau pair-à-pair résiste à la panne d'un de ses composants, puisque aucun site n'est privilégié et peut être remplacé par un autre site.

Réseau Pair-à-pair Dans un réseau pair-à-pair, si chacun peut jouer tous les rôles, tant qu'il y a un participant opérationnel, le service peut continuer de fonctionner. Sauf dans le cas défaillance byzantine, mais dans cette thèse nous allons rester dans un cadre de confiance.

Résistance à la censure

En 1996 Mike Godwin dit : Je suis tout le temps soucieux au sujet de mon enfant et d'Internet, bien qu'elle soit encore trop jeune pour se connecter. Voilà ce qui m'inquiète. Je redoute que dans 10 ou 15 ans elle vienne me voir et me demande : "Papa, où étais-tu quand ils ont supprimé la liberté de la presse sur Internet ?" » (I worry about my child and the Internet all the time, even though she's too young to have logged on yet. Here's what I worry about. I worry that 10 or 15 years from now, she will come to me and say "Daddy, where were you when they took freedom of the press away from the Internet ?"

De nos jour le réseau internet est de plus en plus en proie à un filtrage de l'information par différents pays. Le 12 mars 2011 Reporters sans frontière ont établi une carte de la censure (2.2) sur internet[sf11]. Il est donc nécessaire de faire des applications capables de "ruser" pour passer outre l'oppression de certains gouvernements.

Centralisé Il est facile de s'attaquer à ce genre de modèle par flood [3] ou simplement filtrer le tuyau par lequel, il passe. Ce modèle demandant parfois de grosses structures est plus difficile à déplacer.

pair-à-pair Un réseau pair-à-pair est difficilement attaquable par déni de service. Il faudrait pour cela matraquer (flood) tous les participants en même temps ce qui demande une grande puissance pour l'attaquant.

[3] Action consistant à envoyer une grande quantité de données afin de consommer toute la bande passante ou toute les ressource nécessaire pour traiter l'information

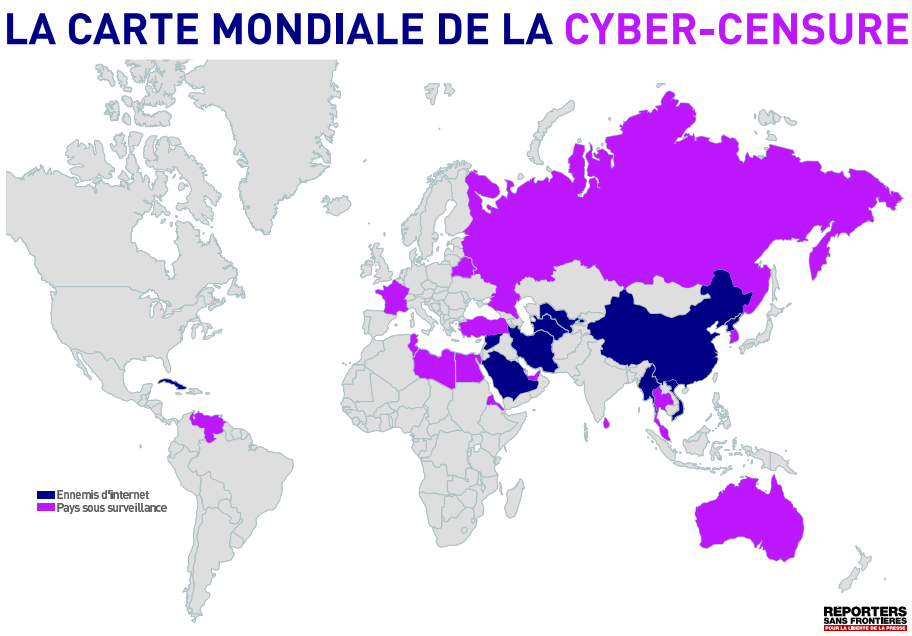


FIG. 2.2 – *Carte de la censure d'internet de RSF.*

> Les réseau pair-à-pair sont difficilement contrôlables par un tiers (censure, interdiction, ...). Des projets ont vu le jour dans cette optique. Le célèbre FreeNet créé en 2000 par Clarke, qui propose de stocker des informations en pair-à-pair de façon anonyme et chiffrée. Pour cela, il fait transiter et stocke des parties de fichiers chiffrés dans différents pairs. Il utilise aussi des routages heuristiques afin de rendre difficile le traçage des paquets.

***Remarque* 2.1** — Utiliser l'approche distribuée permettrait de ne pas sortir d'un réseau de type Freenet pour dialoguer avec un serveur extérieur.

2.4 Divergence, Convergence, Dépendance

2.4.1 Problème de convergence

Dans un système distribué avec réplication des données sur chaque site, un algorithme distribué est *convergent* si toutes les répliques sont identiques sur chaque site après exécution de toutes les opérations considérées. Dans le cas contraire, on parlera de *divergence*.

Cette propriété est difficile à assurer et fait toujours l'objet de recherches actuelles dans différents domaines [ABGM90 ; AD ; TKDP+08]...

Il y a différents types de convergence.

Convergence forte

On parle de convergence forte quand à chaque étape toutes les répliques sont identiques. Cela implique dans un système collaboratif d'avoir un

ordre total sur les opérations et de les exécuter toutes dans le même ordre. Sinon on passe par des états différents. Ces systèmes ne sont pas compatibles avec l'édition pair-à-pair, car il nous faudrait un ordre total sur les opérations et/ou un synchronisme que nous n'avons pas.

Divergence limitée

La divergence limitée ou bornée est le fait d'autoriser le système à diverger de façon bornée selon une métrique. Cette métrique peut être le nombre d'opérations intégrées [YV01] ou une métrique sur les données [ABGM90].Cette solution reste contraignante, car dans notre cas nous n'avons pas de bornes sur le nombre d'opérations concurrentes.

Convergence (à terme)

La convergence à terme implique qu'une fois le système est au repos (toutes les opérations ont été effectuées) toutes les répliques sont identiques. C'est cette propriété qui va nous intéresser dans cette thèse et nous parlerons simplement de *convergence* dans la suite.

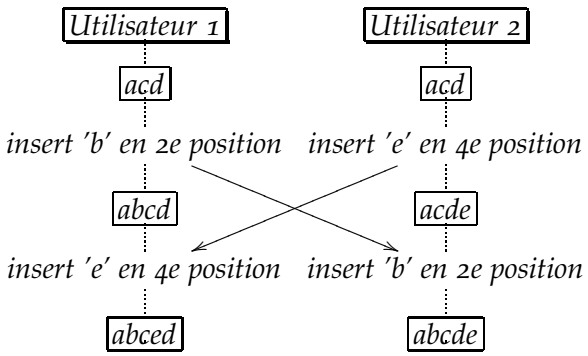
Dans le cadre de l'édition collaborative, de nombreuses approches ont été proposées. Ce sont la majorité des articles cités dans cette thèse.

2.4.2 Comment obtenir la convergence ?

Dans le cadre de l'édition collaborative pessimiste ou centralisée, la convergence est intrinsèque. Le système a un document référent où chaque modification se fait séquentiellement grâce à un verrou.

Dans le cadre de l'édition collaborative optimiste pair-à-pair, il n'y a pas de copie de référence. Même si la causalité des messages est respectée, les messages ne sont pas forcément reçus dans le même ordre sur tous les sites. Deux opérations peuvent avoir la même cause et peuvent être concurrentes [Lam78] et selon l'ordre d'exécution, le résultat peut être différent.

Exemple 2.1



2.4.3 Causalité

La causalité est une notion assez générale. Ce principe est que tout évènement a une cause l'ayant entrainé, cette cause étant un ensemble d'évènements. Dans notre cadre un évènement est une opération d'édition effectuée par un participant. Lamport [Lam78] définit la causalité comme une relation d'ordre partiel entre les opérations, la notation $op \succ op'$ indique que l'opération op a été effectuée avant l'opération op[4]par un des sites participant. Un premier problème est de savoir comment définir *avant* dans un système pair-à-pair qui n'a pas d'horloge globale, mais des horloges locales sur chaque site.

Un autre point soulevé par la définition est que la relation d'ordre ne prend pas en compte la nature des opérations ni des objets.

Dans le modèle CCI, la causalité est définie comme celle de Lamport alors que dans notre approche nous allons utiliser une formulation plus faible de cette relation. Cette formulation prendra en compte la nature des objets et des opérations mis en oeuvre par le processus d'édition tout en gardant l'idée originelle de Lamport.

Dépendance sémantique

La dépendance sémantique est une relation d'ordre partielle sur les opérations, indiquant qu'une opération op dépend sémantiquement d'une opération op' si et seulement si l'opération op a besoin que op' ait été exécutée pour être exécutable. Contrairement à la notion de Lamport, cette relation dépend fortement de la nature des objets utilisés.

Prenons le mode d'emploi pour faire du thé (en sachet) :

1. Faire bouillir de l'eau dans une bouilloire.
2. Mettre le sachet de thé dans un verre.
3. Verser l'eau dans le verre.
4. Mettre le sucre dans le verre.
5. Mélanger le tout.

Supposons qu' Eric, Fabrice, Gaelle et Hugues veulent suivre la même recette en travaillant en parallèle pour optimiser l'exécution de la recette. Chaque participant peut être vu comme un processus qui communique avec les autres processus en informant de l'opération qu'il effectue. Si on considère l'ordre causal de Lamport, seul l'ordre de la recette est possible, car chacun doit attendre l'autre. Mais vous pouvez deviner regardant la propriété des ingrédients qu'il y a d'autres ordres donnant le même résultat, comme :

1. Fabrice met le sachet de thé dans le verre.
2. Eric fait bouillir de l'eau dans la bouilloire.
3. Hugues met le sucre dans le verre.
4. Gaelle verse l'eau de la bouilloire dans le verre.

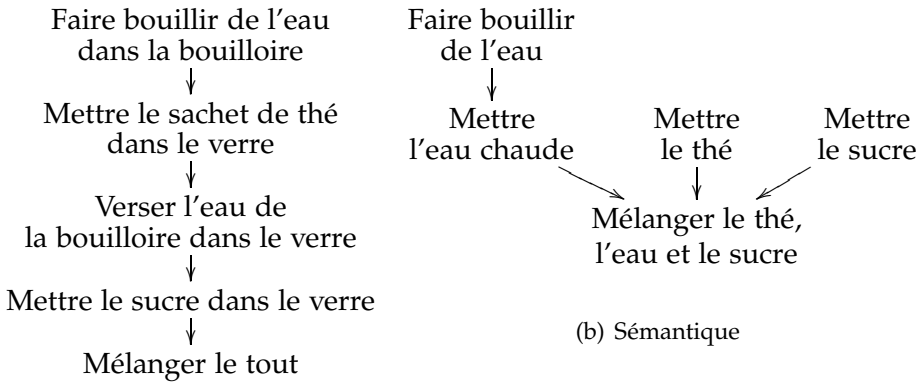
[4]à considérer au sens temporel du terme

5. Hugues mélange le tout.

Mais on ne peut pas avoir l'ordre suivant car sémantiquement incorrect :

1. Hugues met le sucre dans le verre.
2. Hugues mélange (rien !).
3. Fabrice met le sachet de thé dans le verre.
4. Gaelle verse l'eau de la bouilloire dans le verre.
5. Eric fait bouillir de l'eau dans la bouilloire.

Ce thé sera bizarre.



(a) Causale

(b) Sémantique

FIG. 2.3 – *Dépendances*

En fait, on peut définir ce genre de dépendance sur beaucoup de structures de données (et les opérations associées) de façon simple, tout particulièrement celles qui interviennent dans l'édition collaborative (arbres, graphes, documents semi-structurés,...). Par conséquent, la dépendance causale décrite par Lamport n'est pas nécessaire pour faire un éditeur collaboratif consistant (modèle CCI)[SJZ+98 ; Suno2 ; Wei10] et nous allons voir que nous pouvons la remplacer par la dépendance sémantique.

Cette idée nous permet d'avoir un algorithme d'édition générique simple qui simplifie les calculs et qui réduit la quantité d'informations à envoyer (et donc le coût en termes d'envoi de messages). Nous le présenterons dans la section 5.3

2.5 Autres concepts

2.5.1 Préservation de l'intention de l'auteur

Cette notion n'est pas formellement définissable. Pour vraiment respecter les intentions de l'auteur, il faudrait lire dans ses pensées ! Par contre, on peut donner un comportement naturel au système, comme par exemple éviter de faire converger tous les sites en triant les mots par ordre alphabétique.

Généralement, le but est que lorsqu'un auteur est tout seul, le système se comporte comme n'importe quel éditeur. Lorsqu'ils sont plusieurs à éditer

un document, il est préférable pour un éditeur optimiste de ne pas perdre d'informations et d'essayer d'intégrer toutes les données sans "choquer". Mais c'est là encore une notion informelle.

2.5.2 Historique

L'historique est une mémoire où on stocke par ordre d'exécution la suite des opérations d'un site. Il peut être nécessaire pour assurer la convergence. Beaucoup d'algorithmes comme celui de Ressel[RNRG96] d'Imine [Imi06] ont besoin d'un historique complet pour assurer la convergence à terme. Un problème de ce type d'algorithmes est la complexité par rapport à la taille de l'historique, par exemple $O(n^2)$ avec n la taille de l'historique pour l'algorithme de Ressel. Par exemple, un document de 100 pages de 20 lignes de 80 caractères donne 160 000 caractères, chacun correspondant à au moins une opération d'édition. Un coût en $O(n^2)$ devient prohibitif dans ce cas et exclut toute édition de documents avec beaucoup d'opérations.

L'historique est aussi nécessaire pour l'annulation. Il faut en effet garder en mémoire les modifications nécessaires pour pouvoir les défaire. Pour l'annulation, il existe des façons de limiter la taille de l'historique nous en discuterons dans la section 6.4.2 et la taille de l'historique ne rentre pas en compte dans la complexité de l'algorithme.

2.5.3 Annulation

Lors de l'édition de document, il est très classique de faire des erreurs comme tout supprimer ou modifier la mauvaise partie. De plus si le document est partagé ce type d'erreur peut intervenir facilement, parfois de manière volontaire par des participants mal intentionnés (voir certaines modifications de documents sous Wikipédia). Il est donc nécessaire de pouvoir *annuler* certaines opérations pour revenir à l'état précédent (commande undo ou annuler de nombreux éditeurs). Ce processus d'annulation peut souvent être itéré jusqu'à revenir au tout début. Une condition nécessaire pour effectuer ce travail est de garder en mémoire les opérations effectuées depuis le début, donc d'avoir un historique des opérations effectuées. Cet historique peut être délocalisé (chaque site garde la mémoire de ses opérations) ou centralisé. Une variante est de limiter le nombre d'opérations que l'utilisateur peut annuler, ce qui permet de n'avoir qu'un historique limité.

Édition pair à pair

3

Dans ce chapitre nous allons introduire les objets les plus communs (texte, arbres, ...) et les opérations permettant de les construire et les modifier.

Pour finir nous allons passer en revue les principales approches pour réaliser de l'édition collaborative en pair-à-pair sur nos objets : l'approche par transformée opérationnelle et l'approche CRDT.

3.1 Les documents

3.1.1 Textes non formatés

Le document texte non formaté est une simplement une suite de caractères. Ce type de document rudimentaire a été longtemps favorisé par les éditeurs collaboratifs. Tout autre type de document peut se ramener à ce type primitif sachant que la structure du document sera perdue rapidement lors de l'édition de ce type de fichier.

Les opérations d'édition sur ce format texte sont l'insertion et la suppression de caractères. La façon la plus naturelle d'identifier un élément dans un texte non formaté est de prendre sa position dans le texte. La difficulté qui survient est que les opérations d'édition changent cette identification (décalage de +1 en cas d'insertion avant le caractère considéré par exemple), (voir l'exemple 2.1).

3.1.2 Documents structurés

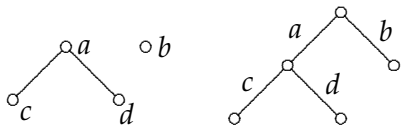
Dans le monde de l'édition, nous remarquons qu'il existe beaucoup de documents structurés. Ils sont beaucoup plus lisibles par l'humain que les documents non-structurés. Le type le plus commun est le document organisé de manière hiérarchique, que ce soit un texte structuré en chapitres, sections et paragraphes, que ce soit un code source avec les fonctions et les objets, voir même un fichier, ... Même une carte peut se voir comme un dessin où des nœuds relient des arêtes. Nous donnons dans la suite des types de documents structurés extrêmement importants en pratique.

3.1.3 Documents arborescents

Les structures arborescentes sont parmi les plus fondamentales en informatique. Un arbre est constitué de sommets et d'arêtes orientées reliant les sommets de manière suivante :
- Tout sommet a un et un seul père (une arête entrante) sauf la racine qui n'en a pas.
- Il n'y a pas de cycle.
- Les nœuds ou les arêtes peuvent avoir des étiquettes
- Les nœuds ou les arêtes peuvent être ordonnés ou pas.

Dans cette thèse nous allons étiqueter les arêtes, mais il est simple de passer de ce type de représentation à la représentation usuelle étiquetant les nœuds. Une forêt d'arbres usuels se représente par un seul arbre avec étiquetage sur les arêtes ce qui est pratique pour modéliser les documents XML.

Exemple 3.1



Dans cet exemple, nous exhibons une forêt d'arbres étiquetés sur les nœuds et son équivalent en arbre étiqueté sur les arêtes. Nous remarquons qu'il suffit de dessiner un nœud supplémentaire reliant les racines et de faire remonter les étiquettes sur les arêtes reliant au père.

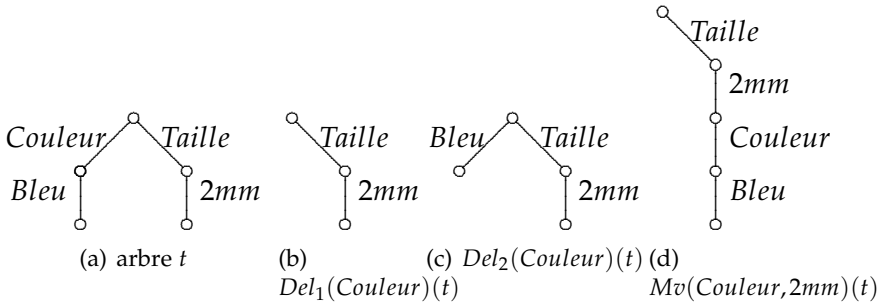
Terminologie Par abus de langage justifié par le souci de ne pas introduire de termes supplémentaires, nous reprenons la terminologie classique des arbres en l'adaptant aux arêtes : Le père (d'une arête)[1] est l'arête au dessus de l'arête et les arêtes filles sont celle en dessous du père. Dans l'exemple 3.1, les arêtes c et d ont comme père l'arête a et elles sont aussi les arêtes fille de a. Deux arêtes sont dites soeurs si elles partagent la même arrête père. Dans l'exemple 3.1 les arêtes c et d sont soeurs.

Opérations

Les opérations d'édition d'arbre sont :
- Ajout d'une arête (Add)
- Suppression d'une arête avec les deux variantes possibles :
  - En supprimant le sous arbre complet sous cette arête (Del_1),
  - En supprimant uniquement l'arête et en faisant remonter d'un niveau les sous-arbres sous l'arête supprimée (Del_2)
- Déplacer une partie de l'arbre d'un endroit à un autre (move Mv)

Nous verrons par la suite que la différence entre les deux opérations de suppression n'est pas anodine. L'opération Mv est très délicate et elle n'est souvent pas prise en compte dans les éditeurs collaboratifs. Dans cette

[1] mère serait plus justifié dans ce cadre...



(a) arbre t (b) $Del_1(Couleur)(t)$ (c) $Del_2(Couleur)(t)$ (d) $Mv(Couleur, 2mm)(t)$

FIG. 3.1 – *Différentes opérations*

thèse nous verrons comment la prendre en compte au prix d'une modification de la structure de données considérée.

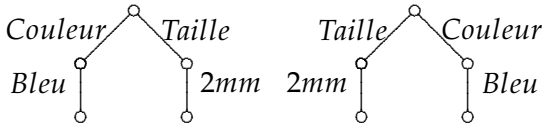
Sélection et identification

Pour réaliser les opérations précédentes, il faut pouvoir identifier clairement les arêtes à modifier.

Nous utiliserons deux façons d'identifier ces arêtes :
- Par un chemin : le chemin est la suite des étiquettes suivies en allant de la racine à l'arête. Une restriction d'unicité des étiquettes sous un même nœud garantira l'unicité de l'arête identifiée.
- Par un identifiant unique : à chaque arête correspond un identifiant qui sera unique dans l'arbre par construction de celui-ci (comme dans un réseau informatique, l'adresse IP identifie une machine unique).

Arbres non-ordonnés et arbres ordonnés

Classiquement, on distingue deux modèles d'arbres, les arbres ordonnés (les fils d'un nœud sont ordonnés) et les arbres non-ordonnés (les fils d'un nœud ne sont pas ordonnés). Dans le modèle ordonné, les arbres *Arbre* et *Arbre*′ sont différents alors qu'ils sont identiques dans le modèle non-ordonné.



(a) *Arbre* (b) *Arbre*′

FIG. 3.2 – *Ordonné vs non ordonné*

Selon l'application visée, certains documents ne requièrent pas d'ordre tandis que celui-ci peut être nécessaire dans d'autres cas.

Dans un carnet d'adresses, les informations utiles correspondent à un nom et l'ordre n'est qu'une commodité d'accessibilité. Le logiciel utilisant ces données classera les contacts par ordre alphabétique ou autre...

Par contre, dans une page web (en xhtml) l'ordre est important : on veut que les items d'une liste restent dans l'ordre donné.

Si on utilise des arbres étiquetés non ordonnés, il est possible de les transformer en arbres ordonnés en rajoutant dans l'étiquette une information qui code l'ordre respectif des éléments entre eux. Ce codage sera effectué dans le modèle d'arbre où l'étiquette de chaque arête contient un identifiant unique et est complétée par une information de priorité permettant de situer cette arête par rapport aux arêtes situées au même niveau. Les opérations d'édition devront être définies de manière cohérente avec cette priorité. Dans notre approche, nous utiliserons une chaine de caractère d'un certain type pour définir les priorités (voir le chapitre 6).

3.1.4 Documents XML

XML est l'abréviation de eXtensible Markup Language , qui définit un standard permettant d'encoder un document dans une forme lisible par la machine et l'être humain. Les spécifications de ce standard sont produites par le *W3C* (World Wide Web Consortium). Ce langage est devenu le standard de référence pour la production et l'échange de document et de très nombreuses applications ou formats s'y réfèrent [Cov] : RSS, SVG, SOAP, et XHTML.

Par définition, un document XML est une chaine de caractère écrite en Unicode [Wikd], structurée avec les constituants suivants :

Des balises : Une balise se construit en commençant par le symbole "<" suivit par le nom du tag et se finit par le symbole ">". Il existe trois types de balises :
- la balise ouvrante, par exemple : `<balise>`.
- la balise fermante, par exemple : `</balise>`.
- la balise orpheline, par exemple : `<orpheline/>`.

Des éléments : Un élément est un composant du document qui commence par une balise ouvrante et fini par une balise fermante correspondante ou simplement une balise orpheline si l'élément ne contient rien. Un élément peut contenir plusieurs autres éléments et des attributs. Ce dernier se met entre les chevrons de la première balise.

Des attributs : Un attribut est une paire nom/valeur et est décrit dans les balises ouvrantes ou orphelines. Par exemple :

```
<img src="toto.jpg" alt="la figure de toto"/>
```

Ici on décrit une image qui aura comme attributs une "src" indiquant un fichier source toto.jpg et un autre attribut qui indique comme description de l'élément : "la figure de toto".

Contrairement au SGML, les noms des attributs dans une même balise doivent être distincts en XML.

Un Début : le début d'un fichier XML appelé aussi un prologue se fait avec des balises orphelines spéciales indiquant sont contenu. Par exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
```

indique que le fichier contient de l'XML en version 1.0 et que le codage utilisé pour les caractère est de l'UTF8. La version 1.1 est sortie

mais le principe reste le même. Les différences sont surtout sur l'internationalisation du format (les caractères et les codages autorisés dans la norme).

La figure 3.1.4 donne un exemple de d'un fichier XML. Dans ce fichier,

```
<?xml version="1.0" encoding="UTF-8"?>
<names>
  <Pat>
    <Phones>
      <Phone type="Cellular">
        0691543545
      </Phone>
      <Phone type="Home">
        0491543545
      </Phone>
    </Phones>
  </Pat>
  <Henri>
    <Adress type="home">
      45 Emile Caplant Street
    </Adress>
  </Henri>
</names>
```

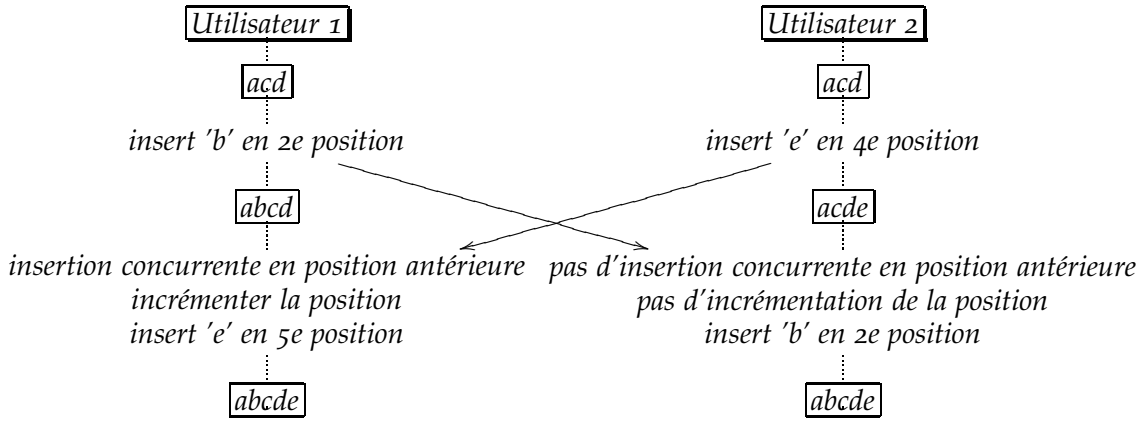
Fig. 3.3 – *Exemple de fichier XML*

nous remarquons qu'il y a Pat et Henri. L'un possède deux numéros de téléphone : un cellulaire et un fixe. Henri quant à lui possède une adresse.

3.2 L'approche transformée opérationnelle pour l'édition collaborative

Principe

Cette approche a été introduite pour la première fois dans GROVE par Elis et Gibbs [EG89]. Le principe est que chaque participant envoie chacune de ses opérations de modification du document à tout le monde. Chaque participant recevant une opération en provenance d'un autre site *l'intègre*, c'est-à-dire qu'elle n'exécute pas directement l'opération reçue, mais une opération qui est la transformation de l'opération reçue qui prend en compte les autres opérations précédemment intégrées.

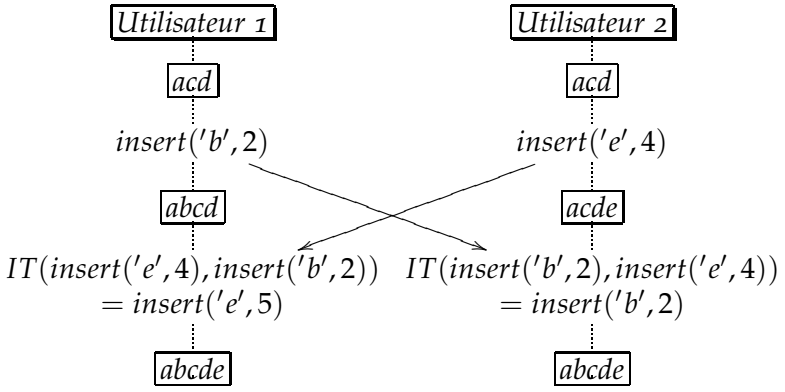


Exemple 3.2

De manière plus formelle et pour cette thèse, on définit une fonction IT : $Op \times Op \longmapsto Op$ pour Integration Transformation prenant en argument deux opérations, celle que l'on veut intégrer et une opération concurrente qui a été déjà intégrée.

Si on effectue une opération d'insertion $insert('b', 2)$ pour insérer un caractère en deuxième position l'exemple 3.2 devient :

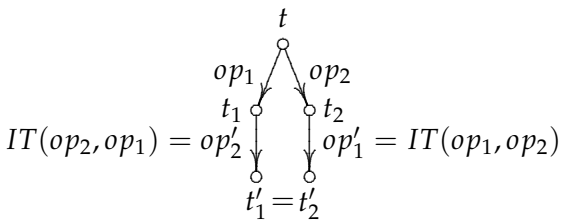
Exemple 3.3



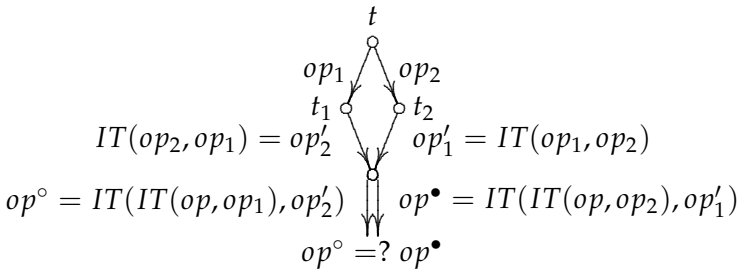
Propriétés sur IT

Ressel [RNRG96] a donné un algorithme pour l'édition collaborative pair-à-pair optimiste qui garantit la convergence, sous réserve que la transformée opérationnelle satisfasse deux propriétés : Soit T l'ensemble d'états du document.

– $TP1$: Soit $op_1, op_2 \in Op$, $\forall t \in T, [op_1, IT(op_2, op_1)](t) = [op_2, IT(op_1, op_2)](t)$ $TP1$ est une sorte de confluence locale assurant qu'après deux opérations concurrentes tous les sites retournent dans le même état.

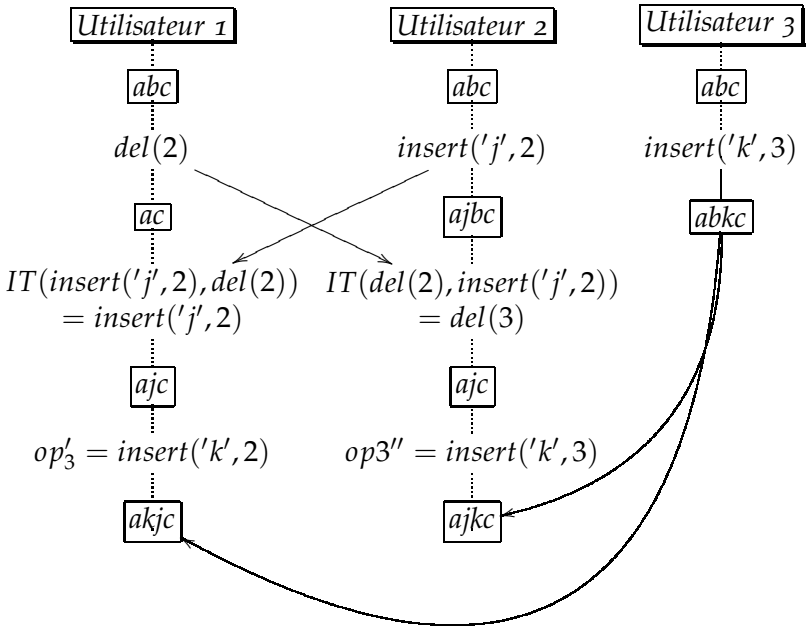


– TP2 : Soit $op, op_1, op_2 \in Op$, $IT(IT(op, op1), IT(op_2, op_1)) = IT(IT(op, op_2), IT(op_1, op_2))$
TP2 : est plus surprenante elle a pour but d'assurer l'unicité d'une transformation après que deux autres soient effectués. Elle est similaire à des commutations partielles sur un cube défini à partir de trois opérations concurrentes.



Le contre-exemple 3.4 est plus connu dans la littérature sous le nom de "TP2-puzzle"[LLS04]. Dans cet exemple, il y a trois utilisateurs voulant exécuter concurremment trois opérations : $op_1 = del(2), op_2 = insert('j', 2), op_3 = insert('k', 3)$. Nous constatons que les utilisateurs 1 et 2 font converger leur état après s'être échangé leur opération. Mais lorsque le dernier utilisateur propage son opération, nous remarquons qu'elle est transformée différemment.

Exemple 3.4 *Contre-exemple du "TP2 puzzle".*



Avec $op'_3 = IT(IT(insert('k', 3), del(2)), IT(insert('j', 2), del(2)))$ *et* $op''_3 = IT(IT(insert('k', 3), insert('j', 2)), IT(del(2), insert('j', 2)))$

Ces propriétés suffisent à assurer la convergence, et la propriété TP1 est nécessaire. Par contre, il n'est pas démontré qu'un algorithme est convergent si et seulement si ces propriétés sont vérifiées. En particulier, la propriété TP2 est très forte, et de nombreux algorithmes ont été proposés en se limitant à des propriétés plus faibles (uniquement TP1). Malheureu-

sement, Imine [Imi06] a montré dans sa thèse que tous les algorithmes et transformées opérationnelles proposés classiquement pour les mots sont incorrects, car ils n'assurent pas la convergence[2].

Algorithmes d'édition collaborative

Plusieurs algorithmes, reposant sur le principe des transformées opérationnelles, ont été proposés pour le cas des mots et servent de base à des éditeurs collaboratifs. La convergence de ces algorithmes nécessite des transformées satisfaisant $TP1$ et $TP2$ (GOTO, Opt,ABT,...) mais malheureusement aucune des transformées proposées pour les jeux d'opérations usuels (ajout, suppression,...) n'assure ces propriétés. D'autres algorithmes proposés ont tout simplement des scénarios divergeant (SDT[SJZ+98], Suleiman et Al[SCF97]). La thèse d'Imine donne une série de contre-exemples qui mettent en défaut les algorithmes ou les transformées opérationnelles proposées pour les mots[Imi06].

adOPTed Ressel en 1996 [RNRG96] a donné un algorithme basé sur la notion de vecteur d'état. Un état correspond à un point dans un hypercube dont la dimension est le nombre de sites. Chaque site correspond à un axe et une opération effectuée par ce site est une translation de longueur 1 selon cet axe. Une suite d'opérations correspond à un chemin dans cet hypercube. La fonction d'intégration correspond à la prise en compte des opérations exécutées sur les autres axes et les propriétés permettent d'assurer que chaque site suit un chemin qui mène au même point. Les positions dans le cube expriment les dépendances entre opérations. La figure suivante correspond à deux sites, l'un horizontal (site 1) qui effectue les opérations $op_{1,i=1,...,5}$ et l'autre vertical, site 2, qui effectue $op_{2,i=1,...,3}$. L'opération $op_{1,2}$ est exécutée après $op_{1,1}$ par définition, mais aussi après $op_{2,1}$ (car elle est "au-dessus"), mais $op_{1,1}$ et $op_{2,1}$ sont concurrentes. Le chemin dessiné en b) décrit les intégrations que doit réaliser le site 1. Les propriétés $TP1$ et $TP2$ vont assurer que le chemin correspondant aux intégrations du site 2 amène bien au même point.

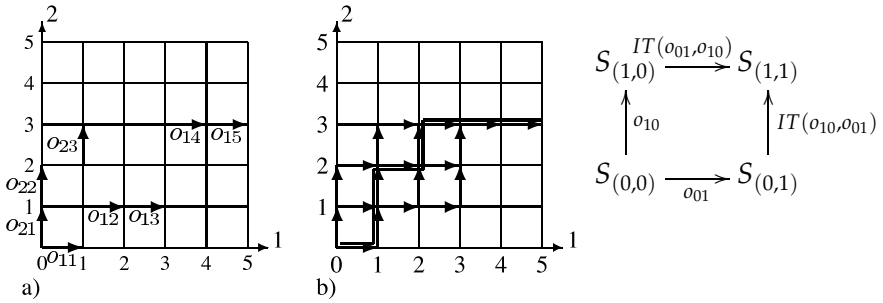


FIG. 3.4 – *illustration de l'algorithme adOPTed*

Cet algorithme a besoin de $TP1$ et de $TP2$ pour converger. Il nécessite d'envoyer les vecteurs de position à chaque message, ce qui peut poser des problèmes de bande passante lorsque le nombre de

[2]celui qu'il propose est correct, mais impose des réordonnancements des opérations couteux

participants augmente. Une preuve différente de correction de cet algorithme a été donnée en 2003 par Lushman, et al., [LC03]. Cet algorithme est de complexité $\mathcal{O}(nb_{op}^2)$ à cause de la recherche d'un chemin. Cet algorithme envoie le vecteur d'état, ce qui rend le passage à l'échelle difficile.

So6 est un projet développé par l'équipe ECOO de l'INRIA Lorraine, basé sur l'algorithme de SOCT4[MOSmI03 ; VCFS00] utilise un serveur qui estampille chaque message de façon unique avec sa date. Ainsi l'ordre d'intégration des messages est total. C'est une approche hybride qui repose sur une notion de temps partagé et correspond plus à une approche centralisée qu'à du vrai pair-à-pair.

Optic [Imi06] est un algorithme de réconciliation nécessitant $TP1$ et une version affaiblie de $TP2$ appelé $TP2'$: $IT(IT(op, op_1), IT(op_2, op_1)) \equiv_s IT(IT(op, op_2), IT(op_1, op_2))$. Dans cette approche, chaque opération d'ajout garde l'historique de ses transformations. Cela permet d'éviter les ambiguïtés dans le calcul de la position, mais impose de garder l'historique des opérations. De plus, une particularité est d'effectuer toutes les insertions avant les suppressions, ce qui peut amener à inverser l'effet des précédentes intégrations. La complexité de l'algorithme est dépendante de son historique $(O)(|h|^2)$ et cette taille d'historique est le nombre d'opération.

ABST [SLG10] est une amélioration de l'algorithme ABT [LL05] en utilisant des groupes d'opérations appelés des transactions. Il est proche de l'algorithme OPTIC. Il va générer un historique de manière canonique (les insertions avant les suppressions). Il utilise une façon spéciale de faire ces intégrations identiques sur les autres sites leur permettant de se passer de TP2. Mais l'envoi des vecteurs d'états peut poser des problèmes de passage à l'échelle. Contrairement à OPTIC qui utilise une relation de dépendance.

COT [SS06 ; SS09] Cet algorithme associe à chaque opération un context qui est l'état du document lors de la création de l'opération. La condition $TP2$ est donné par l'algorithme qui pour un même groupe de contexte va transformer les opérations dans un même ordre. Dans le pire des cas un réordonnancement est requis. L'envois du vecteur d'état, ici appelé contexte passe difficilement à l'échelle.

Google Wave [WML10] est un projet de l'entreprise Google. Le principe est de pouvoir collaborer sur un document hiérarchique à l'image d'un forum où des personnes peuvent poster, répondre et rééditer en temps réel. Il est basé sur l'algorithme de réconciliation nommé Jupiter[NCDL95] conçu pour fonctionner de façon centralisée. Google l'a adapté pour supporter le format XML et l'a virtualisé dans un modèle de *cloud computing* (nuage) en contrôlant la vitesse de propagation. Le nombre de participants est directement lié à la taille du nuage et ce système n'est pas adéquat pour le vrai pair-à-pair.

MOT2 [CF07] est un algorithme de réunification deux à deux. Le but de cet algorithme est de synchroniser plusieurs agendas ou téléphones sans toujours passer par la même copie principale[3]. L'échange d'opé-

[3]ce qui est l'amélioration principale de MOT2 par rapport à MOT1

ration se fait par une phase de synchronisation deux à deux en s'échangeant leur historique. Ce qui malheureusement ne passe pas à l'échelle.

Les arbres La plupart des algorithmes proposés sur les arbres sont des adaptations des algorithmes précédents, qui ne garantissent pas la convergence même pour du texte non-formaté.

En 2002, Davis et al. [DSL02], traite les arbres comme une suite de positions. Il utilise les opérations de Grove pour les vecteurs position, et ne satisfait pas la propriété TP1 [Imi06].

Claudia Ignat et Moira C. Norrie ont adapté plusieurs algorithmes créés pour les documents non structurés aux documents hiérarchiques, notamment les algorithmes Soct2[4] [SCF98] et GOTO[SE98], [IN03 ; Ign06], et ont adapté Soct2 pour les documents XML[IO08].

Oster et al. ont adapté les opérations XML pour So6[OSMMNJ07].

Conclusion Les approches d'édition collaborative sur les documents hiérarchiques en pair-à-pair proposés dans la littérature ne sont pas satisfaisantes. Nous avons vu que certaines ont des exemples de divergence et d'autres ont besoin d'un estampilleur.

Vote : un outil de vérification automatique pour TP1 et TP2

Un obstacle pour construire un algorithme d'édition collaborative convergent basé sur les transformées opérationnelles est la définition d'une fonction d'intégration qui est $TP1$ et $TP2$. La preuve de ces propriétés repose sur une analyse de cas qui est longue et fastidieuse, pouvant contenir facilement des erreurs. En 2003, Imine, Molli, Oster et Urso ont créé un vérificateur de ces propriétes nommées : Vote [IMOU03].

Ce vérificateur utilise SPIKE [BSR95] qui est un prouveur inductif de théorèmes. Il couvre la réécriture conditionnelle et l'analyse cas par cas [ARS02 ; BKR92]. L'outil Vote se charge de traduire les définitions des objets, la fonction d'intégration et les opérations via les fonctions d'observation dans leur modélisation écrite en algèbre universelle comprises par SPIKE. SPIKE renvoie un contre-exemple ou une réponse positive. A partir de ce contre-exemple, Vote donne un contre-exemple plus compréhensible pour l'humain (Voir figure 3.5).

Structure d'un fichier Vote Un fichiers Vote contient 7 parties :

type : Déclaration des types.

Par exemple dans le fichiers A.1.3 à la ligne 1, nous définissons un nouveau type *lbl* (et en même temps un objet *novalue* de type *lbl*). Certains comme *nat* sont prédéfinis, mais doivent être déclarés.

observator : Déclaration des observateurs.

[4] L'algorithme Soct2 nécessite un estampilleur

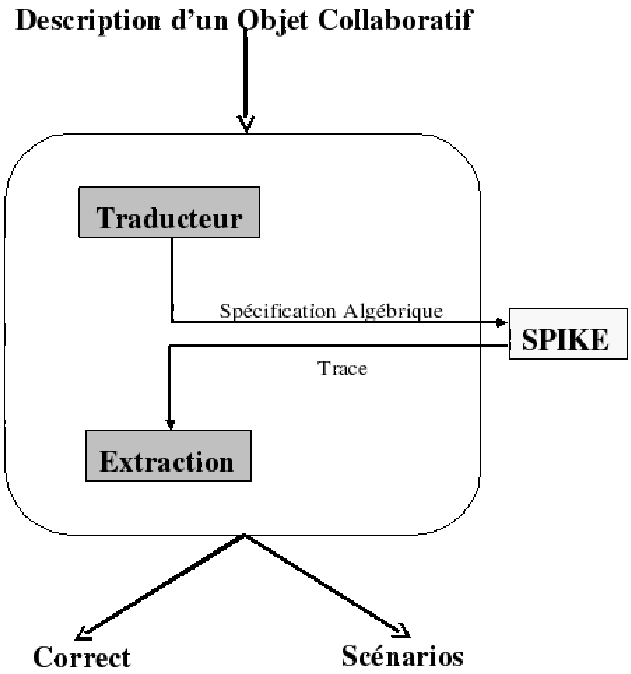


FIG. 3.5 – *Dessin issue de la thèse d'Imine [Imio6].*

Cette partie déclare le type d'un certain nombre de fonctions qui sont utilisées pour définir l'action des opérations d'édition. Dans A.1.3, on définit $exist, childof, getLbl$.

auxiliary : Déclaration des fonctions auxiliaires qui permettent de simplifier l'écriture des autres définitions. Dans l'exemple A.1.3 ligne 11, nous déclarons $childofst$ qui est en fait la version transitive de l'observateur $childof$.

operation : Déclaration des types des opérations d'édition et de leur précondition d'utilisation. Par exemple à la ligne 17 du fichier A.1.3, l'opération d'ajout est déclarée comme prenant deux arguments de type node avec la précondition que le nœud père existe et que le nœud crée n'existe pas déjà.

transform : Définition cas par cas de la fonction d'intégration.

definition : Nous définissons les transformations de l'observateur en fonction des opérations. Par exemple dans le fichier A.1.3 ligne 45, si nous rencontrons l'opération $Del(n_2)$ et que nous souhaitons tester l'existence de n_1 la réponse sera qu'il n'existe pas si $n_1 = n_2$ ou que n_1 est fils de n_2. C'est ainsi que nous supprimons le sous arbre.

lemma : Déclaration de lemmes permettant d'aider SPIKE à effectuer sa preuve. Par exemple, nous avons défini à la ligne 88 du fichier A.1.3 qu'il n'y a pas deux sites avec le même identifiant. Les définitions des fonctions auxiliaires sont données dans cette partie.

L'utilisation de cet outil à permis de montrer que beaucoup d'éditeurs collaboratifs utilisent des opérations et des fonctions d'intégration qui ne sont pas $TP2$ et même parfois qui ne sont pas $TP1$. De plus, l'outil donne des contre-exemples -généralement simples- prouvant la non-convergence des éditeurs [Imio6].

3.3 L'approche CRDT (Commutative Replicated Data Type) pour l'édition collaborative

Nous avons vu que l'approche transformée opérationnelle est difficile, car il est difficile de définir de bonnes transformées pour le format texte. L'approche CRDT évite ce problème. L'idée principale est de trouver des types de données qui commutent "naturellement". N. Ramsey and E. Csirmaz [RC01] ont publié en 2001 un algorithme de synchronisation de fichiers utilisant une façon algébrique de résoudre les conflits. Grâce à des règles de détection d'incohérences, l'algorithme effectue les opérations cohérentes et garde les autres jusqu'à pouvoir résoudre les conflits. Il se peut que des opérations restent en suspens pendant un temps non borné. Cette approche correspond aux paradigmes suivants qui pourraient caractériser un synchronisateur de fichier, voir [BP98] :

1. Propager des opérations non conflictuelles.
2. Si des opérations sont en conflit ne rien faire.

Pour appliquer cette approche pour un document, que faut-il faire et quels sont les objets qui permettent de le faire ?
– Il est nécessaire d'identifier les éléments à modifier. Comme il n'y a plus d'opération d'intégration pour modifier ces valeurs en fonction des opérations précédentes, les identifiants ne doivent pas être ambigus.
– Il est nécessaire d'identifier des objets et des opérations de base avec une notion de commutation. Dans de nombreux travaux, la structure d'arbre non ordonné ou d'ensemble a été choisie. Ainsi, les opérations d'ajout ou de suppression dans un ensemble, commutent.
– Il est nécessaire de pouvoir représenter l'ordre entre éléments indépendamment des opérations de base si on veut pouvoir passer à des applications sur le format XML.
Prenons quelques algorithmes publiés.

3.3.1 Etat de l'art

Woot [OUMaI06] utilise des identifiants uniques d'objet partiellement ordonnés. Chaque identifiant est envoyé avec son prédécesseur et son successeur (pour l'ordre) qui sont uniques, car on travaille avec des documents textuels. On obtient ainsi un diagramme de Hasse. Dans

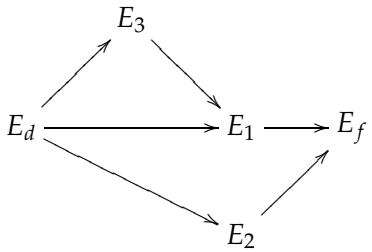


Fig. 3.6 – *Exemple de diagramme de Hasse*

cet exemple figure 3.6, nous pouvons y extraire deux linéarisation : E_d, E_3, E_2, E_1, E_f et E_d, E_2, E_3, E_1, E_f. Pour obtenir la convergence de

tous les sites, le système doit choisir une linéarisation de manière déterministe en utilisant les identifiants uniques afin d'avoir un ordre unique. Tout les sites vont, par exemple, choisir la linéarisation E_d, E_3, E_2, E_1, E_f.

Dans cette modélisation, on ne peut malheureusement pas supprimer d'élément. En effet si on supprime un élément du document toutes les insertions relevant de cet élément ne peuvent plus être intégrées. Pour pallier à ce problème, on remplace les objets supprimés par des pierres tombales ayant le même identifiant. La taille du document augmente au fur et à mesure de l'édition.

TreeDoc [PMSL09] Utilise un arbre binaire pour représenter un document textuel. Chaque nœud possède un identifiant unique. Le contenu

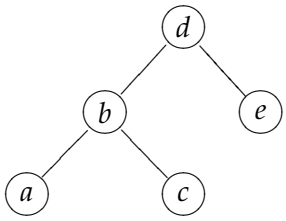
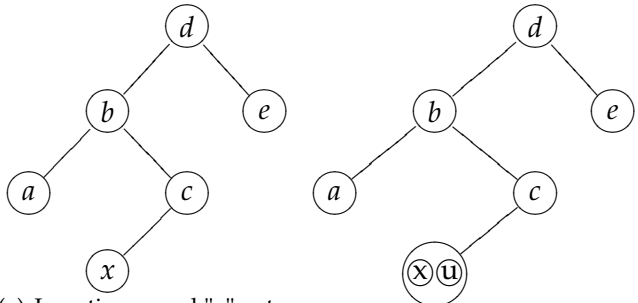


FIG. 3.7 – *TreeDoc : abcdef*

est représenté par le parcours infixe de l'arbre. Le document de l'arbre 3.7 représente le document "abcdef". Dans ce modèle on peut :
- insérer un élément $insert(PosID_n, newatom)$, $PosID_n$ étant en fait le chemin sur le quel on doit insérer le nouvel objet *newatom*.
- supprimer un élément $delete(PosID_n)$. $PosID_n$ est le chemin à supprimer (avec le nœud final).

Deux nœuds insérés à la même position se transforment en un super-Noeud. Ils sont ordonnés avec leur identifiant unique.



(a) Insertion nœud "x" entre "b" et "c"

(b) Super-Nœud car "u" est à la même adresse.

FIG. 3.8 – *TreeDoc : abxucdef*

Comme dans le cas précédent chaque nœud supprimé se transforme en un nœud d'un type particulier qu'on appellera une "pierre tombale". En cas d'un trop grand nombre de pierres tombales dans l'arbre, les auteurs proposent un processus appelé "flatten"qui permet de ne plus voir les pierres tombales dans une partie de l'arbre. Un processus inverse "explode" permet de retrouver ces nœuds.

Logoot [WUM09] utilise un ensemble de lignes toutes identifiées par une

position absolue. Ces identifiants sont utilisé qu'une seule fois, c'est-à-dire qu'ils ne peuvent pas être réutilisés.

Table des identifiants	Document
$\langle 0, NA, NA\rangle$	
$\langle 131, 1, 0\rangle$	"Voici un document Logoot"
$\langle 131, 1, 0\rangle.\langle 2471, 5, 23\rangle$	"Cette ligne se trouve entre 131 et 131"
$\langle 131, 3, 2\rangle$	"Cette ligne était la 2ème générée par la réplique 3"
$\langle MAX, NA, NA\rangle$	

FIG. 3.9 – *Exemple document Logoot tiré de [Wei10]*

Dans cet exemple les identifiants sont des suites de triplets de la forme : $\langle i, s, c\rangle$, i étant un nombre positif borné, s est un identifiant unique et c l'horloge logique. Pour insérer entre deux lignes il suffit de répéter l'identifiant de la ligne qu'on veut avoir au dessus et de générer un identifiant. La première ligne commence par 0 et la dernière ligne par MAX.

L'avantage de cette modélisation est l'absence de pierre tombale. Mais l'utilisation d'une diffusion causale alourdit le processus d'édition. Elle implique d'envoyer tout le vecteur d'état afin de connaitre les opérations qui se sont déroulées sur le site émetteur avec d'exécuter l'opération. Même s'il existe des versions compressées de Prakash et al. [PRS97]. Nous allons voir dans le chapitre 5 comment utiliser une dépendance sémantique. De plus, je ne pense pas que ce système ait besoin d'une délivrance causale. Il suffirait de ne pas supprimer les lignes qui n'ont pas encore été créées. On se rapproche plus d'une dépendance sémantique.

3.4 Conclusion

Nous remarquons que la majorité des travaux traitent le cas des mots, avec des approches qui ne convergent pas ou qui ne passent pas à l'échelle dans le cadre pair-à-pair. Pour les arbres, il n'y avait pas d'approche traitant le cas général permettant de faire de l'édition de document XML.

Nous allons proposer dans le chapitre suivant une structure de données arborescente avec un jeu d'opérations permettant l'édition collaborative qui est compatible avec l'approche par transformée opérationnelle. Cela se fera en définissant des fonctions d'intégration $TP1$ et $TP2$. De plus, nous donnerons des résultats négatifs (i.e. de non-existence de telles transformées) quand on veut prendre en compte certaines opérations.

Dans le chapitre 5, nous passerons au cadre CRDT, et nous donnerons un algorithme générique d'édition collaborative *FCedit* basé sur la notion de dépendance sémantique grâce à notre algorithme. Nous illustrerons la puissance de cette approche en l'instanciant pour une structure arborescente similaire à celle du chapitre précédent.

Le chapitre6 montrera comment prendre en compte l'édition d'un "vrai" document XML avec l'approche CRDT vue dans le chapitre 5.

Modèle des Transformées Opérationnelles

4

4.1 Introduction

Dans ce chapitre nous considérons l'approche des transformées opérationnelles pour l'édition sur des structures de données arborescentes. Le principe de cette approche est de résoudre les conflits provenant d'interactions concurrentes sur un même objet en remplaçant l'exécution d'une opération par l'exécution de l'intégration de cette opération avec l'opération concurrente. Ce principe n'est pas uniquement appliqué dans l'édition collaborative pair-à-pair, voir [Imio6 ; SLG10 ; CF07 ; OSMMNJ07]...La difficulté fondamentale est de définir l'intégration d'une opération par rapport à une autre. Cette définition détermine en quelque sorte l'impact immédiat qu'a une opération -que l'on vient d'exécuter- sur la nouvelle opération que l'on veut appliquer. L'algorithme d'édition collaborative devra lui être capable de prendre en compte tout l'historique des opérations depuis le début de processus d'édition pour appliquer les bonnes transformations. Nous avons vu au chapitre 3 qu'il existe un algorithme d'édition collaborative convergent du à Ressel et al. pourvu que la transformée opérationnelle utilisée satisfasse les propriétés TP1 et TP2.

Les structures de données considérées sont définies par un objet arbre et par un ensemble d'opérations sur cet objet. Plusieurs notions d'arbres sont possibles et plusieurs jeux d'opérations classiques sont envisageables. Ces jeux diffèrent par la notion de suppression : soit tout le sous-arbre disparait (Del_1), soit seul une arête disparait et les arêtes filles "remontent" (Del_2). Les résultats obtenus sont plus intéressants que pour les mots, car nous allons donner des transformées opérationnelles qui sont TP1 et TP2. Par contre nous montrerons aussi que pour certains jeux d'opérations, aucune transformation satisfaisant ces propriétés ne peut exister. Il faut noter que pour le cas des mots aucun résultat négatif similaire n'a été prouvé jusqu'à présent. Nous verrons aussi que certaines opérations sont incompatibles entre elles pour obtenir une bonne transformée opérationnelle.

Les structures qui nous intéressent ne sont pas quelconques mais proviennent d'un processus d'édition. Donc chacun des éléments qui la compose est crée par un des acteurs du processus d'édition. Cela donne l'idée d'attribuer à cet élément une signature *unique* représentant le créateur de cet élément et son numéro d'opération (chaque site numérote les opérations qu'il effectue par ordre croissant $1, 2, 3, \ldots$). Le processus d'édition

va nous donner gratuitement la possibilité d'étiqueter de manière unique chacune des arêtes (ou nœuds) des arbres manipulés. Grace à ces identifiants uniques, nous obtenons des transformées opérationnelles TP1 et TP2 pour les opérations définies mais nous montrons que les deux opérations de suppression sont incompatibles : il n'est pas possible d'avoir de transformée TP1 et TP2 si les deux suppressions sont contenues dans le jeu d'opérations.

Un avantage supplémentaire est de pouvoir prendre en compte l'opération Move qui déplace une partie de l'arbre, opération qui n'est pas pris en compte dans les travaux précédents. Cela permet de définir une transformée opérationnelle TP1 et TP2, sous réserve d'ajouter une notion de mémoire : un conflit entre une suppression et un déplacement impose de pouvoir se rappeler ce qui a été supprimé.

4.2 Un Modèle simplifié : le modèle Harmony

Dans [PSG03], Pierce et al. utilisent un modèle de document qui est un arbre d'arité non-bornée, non-ordonné et dans lesquels les étiquettes des nœuds fils d'un nœud père sont toutes distinctes. Ce modèle est particulièrement adapté pour représenter certains types de documents par exemple un agenda ce qui justifie de l'étudier pour lui-même. C'est un modèle plus simple à priori que le modèle de document XML, et il va permettre de voir comment l'approche des transformées opérationnelles fonctionne -ou pas- sur un type de document arborescent.

4.2.1 Structure de données

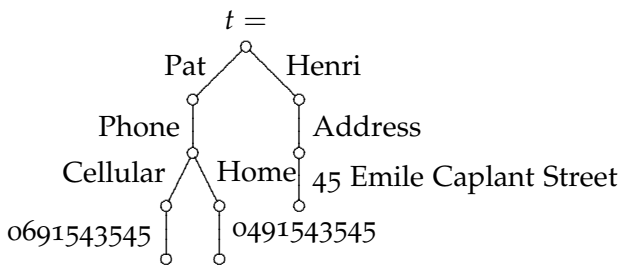
Nous rappelons que dans notre modèle (que nous appellerons modèle *Harmony* par extension) ce sont les arêtes qui sont étiquetées et non les nœuds. Par conséquent, la restriction, du modèle considéré ici, est que les arêtes issues d'un même nœud sont toutes étiquetées par des symboles distincts (noms). Par contre, des arêtes qui ne sont pas directement issues d'un nœud peuvent être étiquetées par le même nom. De plus, les documents sont non-ordonnés donc l'ordre des arêtes sous un même nœud est indifférent.

Par exemple, considérons le document XML suivant qui correspond à un répertoire téléphonique :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Pat>
3    <Phone>
4      <Cellular>
5        0691543545
6      </Cellular>
7      <Home>
8        0491543545
9      </Home>
10   </Phone>
11 </Pat>
```

```
12 <Henri>
13   <Adress>
14     45 Emile Caplant Street
15   </Adress>
16 </Henri>
```

Dans le modèle que nous considérons, il sera représenté ainsi :



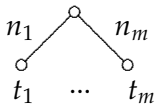
Dans cet arbre, chaque arête possède une étiquette. Il est clair que du point de vue de l'utilisateur de ce répertoire, l'ordre entre les différentes arêtes issues d'un même nœud n'a pas d'importance.

Dans cet arbre, les arêtes issues d'un même nœud sont étiquetées de manière unique et une représentation naturelle de cet arbre est la représentation ensembliste suivante :

$$t = \left\{ \begin{array}{l} Pat\left(\left\{ Phone\left(\left\{ \begin{array}{l} Home(\{(0491543545)(\{\})\}) \\ Cellular(\{(0691543545)(\{\})\}) \end{array} \right\}\right)\right\}\right) \\ Henri(\{(Address)(\{45\ Emile\ Caplant\ Street(\{\})\})\}) \end{array} \right\}$$

L'utilisation d'ensemble exprime le fait que les arbres sont non-ordonnés.

Convention de dessin. L'arbre vide sera représenté par $\circ$, l'arbre $\{n_1(t_1), \ldots, n_m(t_m)\}$ par :



Dans notre définition, la représentation d'un document XML, qui est usuellement une forêt, est un arbre étiqueté sur les arêtes et non les nœuds, la racine correspondant à un nœud rajouté pour enraciner tous les arbres de la forêt.

Comme indiqué dans le chapitre précédent, section 3.1, nous utilisons la terminologie père, fils,... pour les arêtes.

Etant donné un ensemble Σ d'étiquettes (qui sont des noms), la grammaire qui définit l'ensemble T des arbres correspondant à cette représentation est :

$$\begin{array}{lll} T \ni t := & \{\} & // \text{ arbre vide} \\ | & \{n_1(t_1), ..., n_m(t_m)\}, & // \text{ arbre avec un ensemble d'arêtes } n_i(t_i) \\ & n_1, ..., n_m \in \Sigma,\ t_1, ..., t_m \in T, & \forall i, j \in [1..m] i \neq j \Rightarrow n_i \neq n_j. \end{array}$$

L'unicité de l'étiquetage des arêtes soeurs est assuré par la dernière condition.

Opérations de base

Nous donnons maintenant les définitions et opérations de service sur ces arbres.

Projection et Chemins

Un chemin est une suite finie de noms, c.a.d. un élément de Σ^*. Le chemin vide est noté ϵ, et $p.p'$ est la concaténation du chemin p avec le chemin p'. L'ensemble des chemins est noté $\mathcal{P}$.

Nous définissons $t_{|_n}$ la projection de t sur $n \in \Sigma$ ainsi :

$$\begin{aligned} \{n_1(t_1), ..., n_i(t_i), ..., n_m(t_m)\}_{|_{n_i}} &= t_i \\ \{\}_{|_{n_i}} &= \bot \end{aligned}$$

Par exemple, $t_{|_{Henri}} = \{Address(\{45 \text{ Emile Caplant Street}(\{\})\})\}$ si t est l'arbre donné précédemment.

Nous étendons cette définition à la projection d'un arbre t sur un chemin p, noté $t_{|_p}$, de la manière suivante :

$$t_{|_p} = \begin{cases} t_{|_\epsilon} & = t \\ \{n_1(t_1), \ldots, n_m(t_m)\}_{|_n} & = t_i \text{ si } n_i = n \\ t_{|_{n.p}} & = (t_{|_n})_{|_p}, n \in \Sigma, p \in \mathcal{P} \end{cases}$$

Nous écrivons $p_1 \triangleleft p_2$, quand le chemin p_1 est préfixe du chemin p_2, i.e. $p_2 = p_1 p'$. Cette relation définit un ordre partiel sur les chemins.

La restriction d'unicité de nom sous chaque nœud dans la définition des arbres permet d'assurer qu'un chemin permet de n'accèder qu'à une seule arête.

Domaine

La fonction $Dom : T \longmapsto \mathbb{P}(\Sigma)$ permet d'obtenir les étiquettes au premier niveau de l'arbre. Elle est définie par :

$$\begin{aligned} Dom(\{\}) &= \emptyset \\ Dom(\{n_1(t_1), ..., n_m(t_m)\}) &= \{n_1, ..., n_m\} \end{aligned}$$

Exemple :

$$\begin{aligned} Dom(t) &= \{Pat, Henri\} \\ Dom(t_{|_{Pat}|_{Phone}}) &= \{Home, Cellular\} \end{aligned}$$

Union d'arbres

L'union d'arbres $\oplus : T, T \longmapsto T$ correspond à l'union ensembliste et se définit ainsi :

$$\{\} \oplus t = t$$

$$t \oplus \{\} = t$$

$$\begin{array}{l}\{n_1(t_1), ..., n_m(t_m), \bar{n}_1(\bar{t}_1), ..., \bar{n}_p(\bar{t}_p)\} \\ \oplus \\ \{n_1(t'_1), ..., n_m(t'_k), \bar{n'}_1(\bar{t'}_1), ..., \bar{n'}_q(\bar{t'}_q)\} \\ = \\ \{n_1(t_1 \oplus t'_1), ..., n_m(t_m \oplus t'_m), \bar{n}_1(\bar{t}_1), ..., \bar{n}_p(\bar{t}_p), \bar{n'}_1(\bar{t'}_1), ..., \bar{n'}_q(\bar{t'}_q)\}\end{array}$$

Opérations d'édition

Les opérations d'édition sont des fonctions $T \longmapsto T$. Le processus d'édition consiste à effectuer une suite d'opérations permettant d'obtenir un arbre à partir d'un arbre initial (usuellement l'arbre vide). Le jeu d'opérations autorisé pour l'édition sera noté Op dans la suite et les résultats dépendront de cet ensemble. Nous définissons une opération d'ajout et deux opérations de suppression.

– $Add(p, n)$: Add ajoute une arête étiquetée n à la fin du chemin p.

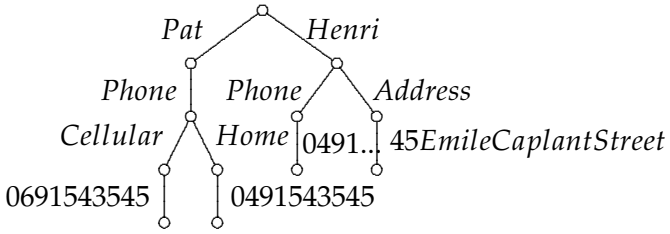
$$Add(n'.p, n)(\{n_1(t_1), ..., n_q(t_q)\}) = \{n_1(t_1), ..., n_q(t_q)\}, \text{ if } n' \notin Dom(t)$$

$$Add(n_i.p, n)(\{n_1(t_1), ..., n_i(t_i), ..., n_q(t_q)\}) = \{n_1(t_1), ..., n_i(Add(p, n)(t_i)), ..., n_q(t_q)\}$$

$$Add(\epsilon, n)(t) = t, \text{ if } n \in Dom(t)$$

$$Add(\epsilon, n)(\{n_1(t_1), ..., n_q(t_q)\}) = \{n_1(t_1), ..., n_q(t_q), n(\{\})\}$$

Exemple 4.1 $t' = Add(Henri.Phone, 0491835469)(t)$



$Add(Henri, Phone)(t')$ ne modifie pas t' car une arête étiquetée par Phone sous le chemin Henri existe déjà.

– $Del_1(p, n)$: Supprime le sous-arbre étiqueté par n atteint par le chemin

p.

$$Del_1(n'.p, n)(t) = t, \text{ si } n \notin Dom(t)$$

$$Del_1(n_i.p, n)(\{n_1(t_1), ..., n_i(t_i), ..., n_q(t_q)\}) = \{n_1(t_1), ..., n_i(Del_1(p, n)(t_i)), ..., n_q(t_q)\}$$

$$Del_1(\epsilon, n)(t) = t, \text{ si } n \notin Dom(t)$$

$$Del_1(\epsilon, n_i)(\{n_1(t_1), ..., n_i(t_i), ..., n_q(t_q)\}) = \{n_1(t_1), ..., n_q(t_q)\}$$

– $Del_2(p, n)$: Supprime n à la fin du chemin p et fait remonter le sous-arbre.

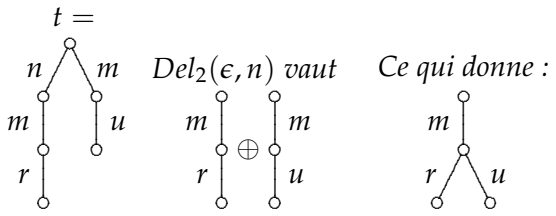
$$Del_2(n'.p, n)(t) = t, \text{ if } n \notin Dom(t)$$

$$Del_2(n_i.p, n)(\{n_1(t_1), ..., n_i(t_i), ..., n_q(t_q)\}) = \{n_1(t_1), ..., n_i(Del_2(p, n)(t_i)), ..., n_q(t_q)\}$$

$$Del_2(\epsilon, n)(t) = t, \text{ if } n \notin Dom(t)$$

$$Del_2(\epsilon, n_i)(\{n_1(t_1), ..., n_i(t_i), ..., n_q(t_q)\}) = \{n_1(t_1), ..., n_q(t_q)\} \oplus t_i$$

Exemple 4.2



– $Nop()$: Ne fait rien.

$$Nop()(t) = t$$

4.3 Résultats

Nous commençons par donner un résultat négatif sur cette structure de données.

4.3.1 Il n'existe pas d'IT avec la propriété $TP1$ par Add et Del_2 dans le modèle Harmony

Le premier jeu d'opérations que nous utilisons est :

$$Op = \{Nop(), Add(p, n), Del_2(p, n) \mid p \in \mathcal{P}, n \in \Sigma\}$$

Cet ensemble d'opérations est non-borné, mais pour simplifier l'écriture nous le désignerons par $Op = \{Add, Del_2, Nop\}$. Cette simplification d'écriture sera utilisée par la suite pour les ensembles d'opérations.

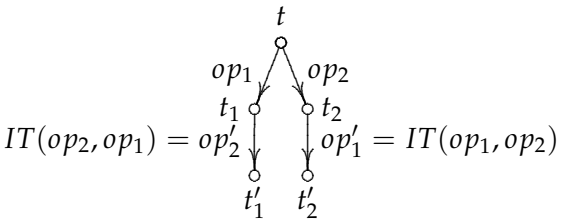
Notre premier résultat est un résultat négatif pour ce jeu d'opérations.

Théorème 4.1 *Il n'y a pas de définition de* $IT : Op \times Op \mapsto Op$ *tel que* IT *satisfasse* $TP1$.

Démonstration. La preuve se fait par un raisonnement par l'absurde : on supposant qu'une fonction IT avec $IT(op_1, op_2) \in Op$ et qui est TP1 existe. La fonction IT satisfait $TP1$ ssi

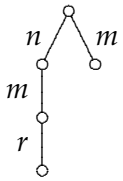
$$[op_1; IT(op_2, op_1)](t) = [op_2; IT(op_1, op_2)](t)$$

i.e. $t'_1 = t'_2$ dans la figure suivante



La preuve utilise un arbre pour lequel aucune définition d'IT ne peut garantir $TP1$.

Soit $n, m, r \in \Sigma$ deux à deux distinct, et soit t l'arbre :



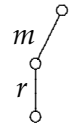
La propriété TP1 implique que pour toutes opérations op_1, op_2, il doit exister des opérations $op'_1 = IT(op_1, op_2)$ et $op'_2 = IT(op_2, op_1)$ de Op telles que $[op_1; op'_2](t) = [op_2; op'_1](t)$.

Nous choisissons $op_1 = Del_2(\epsilon, n)$ et $op_2 = Del_2(n, m)$ exécutées concurremment. L'égalité précédente donne alors

$$[Del_2(\epsilon, n); IT(Del_2(n, m), Del_2(\epsilon, n))](t) = [Del_2(n, m); IT(Del_2(\epsilon, n), Del_2(n, m))](t)$$

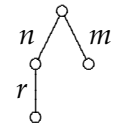
Nous exécutons les premières opérations sur chaque site.
Le premier exécute $Del_2(\epsilon, n)$ et nous avons :

$$t_1 = Del_2(\epsilon, n)(t) =$$



Le second exécute $Del_2(n, m)$ et nous obtenons :

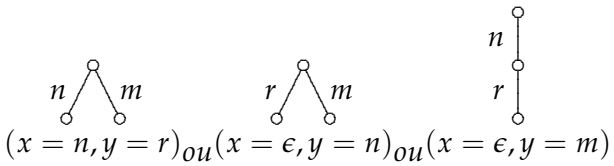
$$t_2 = Del_2(n, m)(t) =$$



Nous montrons que $IT(op_1, op_2)$ ne peut pas être défini dans l'ensemble Op. en considérant toutes les définitions possibles pour $IT(\_, \_)$ et on

montre qu'à chaque fois $[op_1, IT(op_2, op_1)](t) \neq [op_2, IT(op_1, op_2)](t)$. Si une opération ne modifie pas l'arbre nous considérons qu'elle est équivalente à $Nop()$.

1. $op'_2 = Nop()$
   (a) $op'_1 = Nop()$: car $[Nop, Nop](t_1) \neq [Nop, Nop](t_2)$
   (b) $op'_1 = Add(\_, \_)$
   Il y a au moins une arête dans t'_2.
   (c) $op'_1 = Del_2(x, y)$ nous avons :



   Pour chaque cas on a $t'_1 \neq t'_2$
2. $op'_2 = Add(\_, \_)$
   (a) $op'_1 = Nop()$
   Dans t'_1, r est fils de m et dans t'_2, r est le fils de n.
   (b) $op'_1 = Add(\_, \_)$
   Le nombre d'arêtes dans t'_1 et le nombre d'arêtes dans t'_2 sont différents.
   (c) $op'_1 = Del_2(\_, \_)$ idem
3. $op'_2 = Del_2(\_, \_)$

   (a) $op'_1 = Nop()$: Les nombres d'arêtes sont différents dans t'_1 et t'_2.
   Donc $t'_1 \neq t'_2$
   (b) $op'_1 = Add(\_, \_)$ idem
   (c) $op'_1 = Del_2(\_, \_)$ idem

Dans tout les cas il n'existe pas de couple (op'_1, op'_2) tel que $[op_1, op'_2](t) = [op_2, op'_1](t)$. Donc, il n'existe pas de fonction IT satisfaisant $TP1$. □

4.3.2 Une transformée TP1 et TP2 pour $Op = \{Add, Del_1, Nop\}$

Si on remplace la suppression d'une arête seule par la suppression de tout le sous-arbre, il devient possible de définir une transformée opérationnelle qui est satisfaisante. L'algorithme de Ressel peut être utilisé pour l'édition collaborative pair-à-pair sur cette structure arborescente. Nous définissons la fonction IT comme suit.

$IT(op_1, op_2) =$

$$\begin{cases} IT(Add(p,n), Add(p',n')) = Add(p,n), \\ IT(Add(p,n), Del_1(p',n')) = \begin{cases} Nop(), \text{ if } p = p' \wedge n = n' \\ Nop(), \text{ if } p'.n' \lhd p \\ Add(p,n), \text{ else.} \end{cases} \\ IT(Del_1(p,n), Add(p',n')) = Del_1(p,n) \\ IT(Del_1(p,n), Del_1(p',n')) = \begin{cases} Nop(), \text{ if } p = p' \wedge n = n' \\ Nop(), \text{ if } p'.n' \lhd p \\ Del_1(p,n), \text{ else.} \end{cases} \\ IT(op_1, Nop()) = op_1 \\ IT(Nop(), op_2) = Nop(); \end{cases}$$

On peut remarquer que cette transformée opérationnelle a une définition simple et naturelle. Nous avons le théorème suivant :

Théorème 4.2 *IT satisfait TP*1 *and TP*2*.*

***Remarque* 4.1 —** Nous avons eu quelques problèmes pour modéliser les chemins et le déplacement de sous-arbres avec l'outil VOTE décrit dans la section 3.2. C'est pour cela que ces preuves ont été faites à la main en annexe.

Démonstration. La preuve est faite par analyse de cas (voir annexe A.1.1 et A.1.2). Elle ne présente pas de difficultées particulières. Mais le nombre de cas à analyser est très important, notamment pour *TP*2.

□

4.4 Structure de données avec identifiants uniques

Nous allons reprendre l'analyse précédente en modifiant la structure de données pour la rendre plus expressive.

4.4.1 La modélisation d'Harmony n'est pas satisfaisante.

Le jeu d'opérations et la modélisation utilisée ne sont pas totalement satisfaisants. Cette modélisation pose certains problèmes. Par exemple si on essaye d'écrire un document LaTeX.

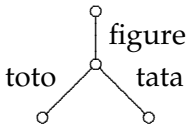
```
1 \begin{figure}
2   tata
3   toto
4 \end{figure}
```

n'aura pas le même effet que :

```
1 \begin{figure}
2   tata
3 \end{figure}
4 \begin{figure}
5    toto
```

```
6 \end{figure}
```

Et pourtant dans cette modélisation, ces deux documents seront représentés par le même arbre.



Ce type d'arbre est utile pour certaines données comme l'exemple du répertoire, mais ne correspond pas à de nombreuses applications, en particulier si on veut pouvoir traiter des documents XML quelconques.

L'idée fondamentale va être d'utiliser un espace de noms plus riche qui va permettre d'identifier de manière unique chaque arête. Ces noms permettront de représenter également les étiquettes usuelles (appelé tag en XML) et nous introduirons une nouvelle fonction permettant de modifier ces étiquettes. Cette opération est nécessaire pour réaliser un système d'édition de documents réaliste.

4.4.2 Structure de données et définitions

Dans la plupart des réseaux chaque machine ou chaque utilisateur possède un identifiant qui est unique par définition (numéro IP, nom d'utilisateur,...). En numérotant les opérations de chaque site de manière incrémentale, nous pouvons associer à chaque opération d'édition un identifiant unique qui est le couple (identifiant du site ; numéro d'ordre de l'opération sur le site). Par conséquent, chaque arête crée par une opération d'édition pourra être identifiée par l'opération qui l'a crée (ou modifié en dernier). Pour un coût pratiquement nul (le numérotage des opérations et l'introduction de noms un peu plus complexes), nous obtenons une structure de données plus spécifique à l'édition collaborative. L'unicité d'identification de chaque arête est une caractéristique très puissante qui va nous permettre de faire de l'édition pair-à-pair de manière simple et d'utiliser des opérations de déplacement et de renommage d'arêtes.

Un exemple de cette structure de données est :

$$t' = \left\{ \begin{array}{l} (Pat,1;1)\left(\left\{ \begin{array}{l} (Phone,2;1)\left(\left\{ \begin{array}{l} (Home,3;1)(\{(0491543545,4;1)(\{\})\}) \\ (Cellular,5;1)(\{(0691543545,6;1)(\{\})\}) \end{array} \right\}\right)\end{array} \right\}\right) \\ (Henri,2;2)(\{(Address,3;2)(\{(45\ Emile\ Caplant\ Street,4;2)(\{\})\})\}) \end{array} \right\}$$

Une étiquette comme $(Phone, 2; 1)$ correspond à l'étiquette *Phone* de l'exemple de la section précédente, mais contient également l'information que cette arête a été crée par le site 2 et que c'était la première opération effectuée par ce site. Si on oublie toute l'information relative aux sites et aux numéros d'opération, on retombe sur l'exemple initial du répertoire (et donc sur le document XML initialement donné en début de chapitre). La différence fondamentale est que l'arbre ci-dessus contient (une partie de) l'historique de sa création[1]

[1] pas tout l'historique : des parties ont pu être supprimées, d'autres modifiées.

Nous supposons que chaque site est répertorié par un identificateur $numsite \in \mathbb{N}$, et qu'il gère un compteur d'opérations $numop \in \mathbb{N}$. En pratique le numéro de site, en plus d'être unique, permet de donner une priorité entre site qui se fait ici par un simple comparaison des entiers. L'ensemble des identifiants ID est définit comme suit :

$$ID = \{numsite; numop \mid numsite \in \mathbb{N}^*, numop \in \mathbb{N}\} \cup \{0;0\}$$

Tous les sites ont un numéro de site strictement positif. L'identifiant $0;0$ est un identifiant spécial réservé comme identifiant initial (on peut considérer qu'il correspond à l'arbre vide de départ). Nous noterons ID^* l'ensemble $ID \setminus \{0;0\}$.

Nous supposons donné un ensemble de labels $\mathcal{L}$ qui contient une valeur particulière $NoValue$ qui ne sera utilisée que pour les initialisations. Une étiquette n de la structure de données précédente est maintenant un couple (l, id) avec $id \in ID, l \in \mathcal{L}$. L'ensemble des étiquettes est toujours dénoté par Σ.

L'ensemble des arbres T est définit par :

$$\begin{array}{rll} T \ni t ::= & \{\} & // \text{ Arbre vide} \\ | & \{n_1(t_1), ..., n_m(t_m)\} & // \text{ Ensemble d'arbre} \end{array}$$

avec $n_i = (l_i, id_i)$, $id_1, ..., id_m \in ID, l_1, \ldots, l_l \in \mathcal{L}$, $t_1, ..., t_m \in T$,
$\forall i, j \in [1..m] i \neq j \Rightarrow id_i \neq id_j$
$\forall i \in [1..m], id_i$ n'apparait dans aucun t_j, $j \in [1..m]$

La terminologie utilisée pour ce type d'arbre est donnée dans le chapitre précédent à la page 18

***Remarque* 4.2 —** Utiliser les identifiants uniques implique d'avoir une fonction recherchant les nœuds à partir des identifiants afin de pouvoir les modifier. Il est possible d'utiliser une fonction de hachage [Wika] afin de limiter le coup de cette recherche. Dans le prototype du chapitre 9, nous utilisons une hashmap [Ora].

Projection

Nous définissons $t_{|_{id}}$ une projection de t sur id comme suit.

$$\begin{array}{l} \{(l_1, id_1)(t_1), ..., (l_i, id_i)(t_i), ..., (l_m, id_m)(t_m)\}_{|_{id_i}} = t_i \\ \{(l_1, id_1)(t_1), ..., (l_m, id_m)(t_m)\}_{|_{id}} = t_{1|_{id}} \cup ... \cup t_{m|_{id}} \\ \{\}_{|_{id_i}} = \{\} \end{array}$$

Par exemple $t_{|_{3;1}} = \{(0491543545, 4; 1)(\{\})\}$.

Nous définissions $t_{\lceil_{id_i}}$ une seconde projection qui conserve l'arête correspondant à l'identifiant :

$$\{(l_1, id_1)(t_1), ..., (l_i, id_i)(t_i), ..., (l_m, id_m)(t_m)\}_{\lceil_{id_i}} = \{(l_i, id_i)(t_i)\}$$
$$\{(l_1, id_1)(t_1), ..., (l_m, id_m)(t_m)\}_{\lceil_{id}} = \{t_{1\lceil_{id}}, ..., t_{m\lceil_{id}}\}$$
$$\{\}_{\lceil_{id_i}} = \{\}$$

Par exemple $t_{\lceil_{3;1}} = \{(Home, 3;1)(\{(0491543545, 4;1)(\{\})\})\}$.

4.4.3 Les opérations d'édition

La base de l'édition consiste à supprimer ou ajouter une arête. Pour définir simplement les autres opérations, il est nécessaire de définir deux fonctions auxiliaires :

– $Erase : ID^* \times T \longmapsto T$, $Erase(id, t)$ efface l'arête id et le sous-arbre correspondant.

$$Erase(id, \{\}) = \{\}$$

$$Erase(id, \{(l_1, id_1)(t_1), ..., (l_q, id_q)(t_q)\})$$
$$= \{(l_1, id_1)(Erase(id, t_1)), ..., (l_q, id_q)(Erase(id, t_q))\} \text{ si } id \neq id_i$$

$$Erase(id, \{(l_1, id_1)(t_1), ..., (l_{id}, id)(t_{id}), ..., (l_q, id_q)(t_q)\})$$
$$= \{(l_1, id_1)(t_1), ..., (l_q, id_q)(t_q)\}$$

– $Addtree : ID^* \times T \times T \longmapsto T$, $AddTree(id, t', t)$ greffe l'arbre t' dans l'arbre t sous l'arête identifiée par id.

$$AddTree(id, t', \{\}) = \{\}$$

$$AddTree(id, t', \{(l_1, id_1)(t_1), ..., (l_q, id_q)(t_q)\})$$
$$= \{(l_1, id_1)(AddTree(id_p, t', t_1)), ..., (l_q, id_q)(AddTree(id_p, t', t_q))\}$$
$$id \neq id_i, \forall i \in [1, .., q]$$

$$AddTree(id, t', \{(l_1, id_1)(t_1), ..., (l_{id}, id)(t_{id}), ..., (l_q, id_q)(t_q)\})$$
$$= \{(l_1, id_1)(t_1), ..., (l_{id}, id)(t_{id} \cup t'), ..., (l_q, id_q)(t_q)\}$$

Opération Nop

Cette opération reste nécessaire pour définir l'effet de la transformée opérationnelle dans certains cas.

$$Nop()(t) = t$$

Ajout d'une arête

Pour ajouter une arête avec un identifiant id, il faut savoir à quelle position, et donc connaitre un identifiant père id_p qui correspond à l'arête sous laquelle on fait l'ajout. Par convention l'étiquette de la nouvelle arête sera une valeur par défaut $NoValue$.

$$Add(id, id_p)(t) = AddTree(id_p, \{(NoValue, id)(\{\})\}, t)$$

où id_p est un identifiant présent dans t, et id n'apparait pas dans t.

Supprimer le sous-arbre

Pour Del_1, il faut que l'arête existe et donc que l'identifiant $id \in ID^*$ correspondant, existe dans l'arbre.

$$Del_1(id)(t) = Erase(id, t)$$

Suppression d'une arête avec remontée des sous-arbres

Pour Del_2, nous ajoutons l'identifiant du père lors de l'émission de l'opération dans le message. Cette complication provient de notre souhait de définir une fonction IT ayant les bonnes propriétés. En effet, en cas d'opérations concurrentes de suppression et d'ajout d'une même arête, il faut pouvoir se rappeler sous quelle arête faire l'ajout. Il ne semble pas possible de définir un IT convenable (c'est à dire respectant les souhaits des utilisateurs) sans cette information.

$$Del_2(id, id_p)(t) = AddTree(id_p, t_{|_{id}}, Erase(id, t))$$

si $id_p \in ID$ est le père de $id \in ID^*$

Changer l'étiquette d'une arête

Nous avons vu que l'ajout d'arête donne une étiquette par défaut qui n'est pas significative, et il est courant également de vouloir modifier l'étiquette d'un arbre. L'opération *ChLbl*[2] permet d'effectuer ces modifications d'étiquetage sans changer la structure de l'arbre. Cette opération prend un argument $id \in ID^*$ qui est l'identifiant de l'arête modifiée, $l \in \mathcal{L}$ la nouvelle valeur, et id_s l'identifiant du site émettant la requête. Ce troisième argument est nécessaire pour résoudre les conflits entre deux opérations de réétiquetage concurrentes.

$$ChLbl(id, l, id_s)(\{\}) = \{\}$$

$$ChLbl(id, l, id_s)(\{(l_1, id_1)(t_1), ..., (l_q, id_q)(t_q)\})$$
$$= \{(l_1, id_1)(ChLbl(id, l, id_s)(t_1)), ..., (l_q, id_q)(ChLbl(id, l, id_s)(t_q))\}$$
si $id \neq id_i$ pour $i \in [1..q]$

$$ChLbl(id, l, id_s)(\{(l_1, id_1)(t_1), ..., (l, id)(t_i), ..., (l_q, id_q)(t_q)\})$$
$$= \{(l_1, id_1)(t_1), ..., (l, id)(t_i), ...(l_q, id_q)(t_q)\}$$

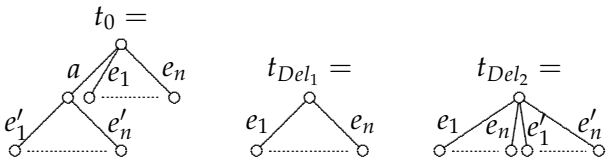
4.5 Résultats

Nous allons définir les transformées opérationnelles possibles sur les différents jeux d'opérations en commençant par voir que Del_1 et Del_2 ne peuvent pas être prises simultanément.

[2] pour Change Label

4.5.1 Incompatibilité de Del_1 et Del_2

L'opération Del_1 supprime le sous-arbre tandis que Del_2 fait remonter les arêtes filles au niveau de celle qui est supprimée. Que se passe-t-il si nous utilisons ces deux opérations de manière concurrente sur le même sous-arbre ? Partant du sous-arbre t_0, nous calculons d'une part t_{Del_1} obtenu en supprimant le sous-arbre correspond à l'arête a, et d'autre part t_{Del_2} obtenu en supprimant l'arête a uniquement. Nous supposons que t_0 est tel que $n > 2$ et que l'identifiant de l'arête a est id_a.



Le premier arbre t_{Del_1} a $n > 2$ arêtes, et le deuxième t_{Del_2} en a $2n$. Supposons que l'on effectue op_{Del_1} sur t_{Del_1} et op_{Del_2} sur t_{Del_2}. Il n'est pas possible de supprimer les deux arbres en une fois par Del_1 car elle devrait s'appliquer à la racine ce qui n'est pas autorisé. Pour assurer une convergence, il faut que les résultats aient le même nombre d'arêtes et donc op_{Del_1} doit être un ajout et op_{Del_2} une suppression. Mais même ainsi on obtient un arbre avec $n+1$ arêtes à partir de t_{Del_1} et un arbre à $2n-1$ arêtes à partir de t_{Del_2}. Comme $n > 2$, nous avons $n+1 < 2n-1$ et donc $op_{Del_1}(t_{Del_1}) \neq op_{Del_1}(t_{Del_1})$. Cela permet d'énoncer la proposition :

Proposition 4.1 *Il n'existe pas d'opérations* $op_{del_1}, op_{del_2} \in \{Add, Del_1, Del_2, Nop\}$ *telles que* $op_{del_1}(t_{del_1}) = op_{del_2}(t_{del_2})$

Par conséquent, aucune transformée opérationnelle TP1 ne peut exister avec l'ensemble d'opérations $\{Add, Del_1, Del_2, ChLbl, Nop\}$.

4.5.2 Une transformée TP1 et TP2 pour $\{Add, Del_1, ChLbl, Nop\}$

Nous définissons une transformée opérationnelle IT_1 pour le jeu d'opération $\{Add, Del_1, ChLbl, Nop\}$ comme suit :

$$
\begin{aligned}
IT_1(Add(id_1, id_{p_1}), Add(id_2, id_{p_2})) &= Add(id_1, id_{p_1}) \\
IT_1(Add(id_1, id_{p_1}), Del_1(id_2)) &= \begin{cases} Nop() \text{ si } id_1 = id_2 \\ Add(id_1, id_{p_1}) \text{ sinon.} \end{cases} \\
IT_1(Del_1(id_1), Add(id_2, id_{p_2})) &= Del_1(id_1) \\
IT_1(Del_1(id_1), Del_1(id_2)) &= \begin{cases} Nop() \text{ si } id_1 = id_2 \\ Del_1(id_1) \text{ sinon} \end{cases} \\
IT_1(ChLbl(id_1, l_1, id_{s_1}), ChLbl(id_2, l_2, id_{s_2})) &= \begin{cases} Nop(), \text{ si } id_1 = id_2, \\ \qquad \text{et } nsite_1 > nsite_2 \\ ChLbl(id_1, l_1, id_{s_1}), \text{ sinon} \end{cases} \\
&\text{avec } id_{s_1} = nsite_1; nbop_1, id_{s_2} = nsite_2; nbop_2.
\end{aligned}
$$

$$
\text{Autres cas : } IT_1(op_1, op_2) = op_1
$$

La proposition suivante nous assure que cette transformée convient pour l'édition collaborative.

Proposition 4.2 *la fonction d'intégration IT_1 est $TP1$ et $TP2$*

Démonstration. La preuve a été effectuée avec l'outil Vote présenté dans la section 3.2. Voir annexe A.1.3. □

Nous avons remarqué qu'il serait possible de définir la fonction IT par $IT(op_1, op_2) = op_1$. Aussi étonnant que cela puisse paraitre cette fonction est $TP1$ dans notre modélisation. La vérification de la propriété $TP2$ pour cette transformée élémentaire ne pose pas de problèmes.

4.5.3 Une transformée TP1 et TP2 pour $\{Add, Del_2, ChLbl, Nop\}$

Pour le jeu d'opération $\{Add, Del_2, ChLbl, Nop\}$, nous définissons $IT_2(op_1, op_2)$ par :

$$
\begin{array}{rcl}
IT_2(Add(id_1, id_{p_1}), Add(id_2, id_{p_2})) & = & Add(id_1, id_{p_1}) \\
IT_2(Add(id_1, id_{p_1}), Del_2(id_2, id_{p_2})) & = & \left\{ \begin{array}{l} Nop() \text{ si } id_1 = id_2 \\ Add(id_1, id_{p_2}) \text{ si } id_2 = id_{p_1} \\ Add(id_1, id_{p_1}) \text{ sinon.} \end{array} \right. \\
IT_2(Del_2(id_1, id_{p_1}), Add(id_2, id_{p_2})) & = & Del_2(Id) \\
IT_2(Del_2(id_1, id_{p_1}), Del_2(id_2, id_{p_2})) & = & \left\{ \begin{array}{l} Nop() \text{ si } id_1 = id_2 \\ Del_2(id_1, id_{p_2}) \text{ si } id_2 = id_{p_1} \\ Del_2(id_1, id_{p_1}) \text{ sinon} \end{array} \right. \\
IT_2(ChLbl(id_1, l_1, id_{s_1}), ChLbl(id_2, l_2, id_{s_2})) & = & \left\{ \begin{array}{l} Nop(), \text{ si } id_1 = id_2, \\ \qquad \text{et } nsite_1 > nsite_2 \\ ChLbl(id_1, l_1, id_{s_1}), \text{ sinon} \end{array} \right. \\
\multicolumn{3}{r}{\text{avec } id_{s_1} = nsite_1; nbop_1, id_{s_2} = nsite_2; nbop_2.}
\end{array}
$$

$$
\text{Autres cas : } IT_2(op_1, op_2) = op_1
$$

Proposition 4.3 *La fonction d'intégration IT_2 est $TP1$ et $TP2$.*

Démonstration. La preuve a également été effectuée avec l'outil Vote. Voir annexe A.1.3. □

Nous remarquons aussi que l'utilisation de Del_2 complexifie le problème.

4.6 L'opération Move

L'opération de déplacement appelée *move* et notée *Mv* consiste à déplacer un sous-arbre entier pour le mettre à un autre emplacement de l'arbre. Cela impose de savoir où se situe le sous-arbre déplacé, et où l'insérer. La notion d'identifiant unique nous permet de définir cette opération. Prendre en compte cette opération complexifie beaucoup les problèmes d'édition et nous n'avons trouvé aucun travail dans la littérature qui considère cette opération dans le cadre de l'édition collaborative. Comme pour

ChLbl nous devons ajouter comme argument l'identifiant id_s correspondant à cette opération (nom du site et numéro d'opération), afin de pouvoir résoudre les conflits de priorité et de définir la transformée opérationnelle.

La première approche que l'on pourrait avoir de l'opération move serait, de prendre comme définition d'opération :

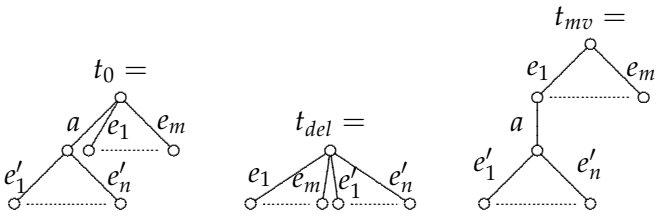
$$Mv(id, id_p, id_s)(t) = AddTree(id_p, t_{\lceil id}, Erase(id, t))$$

avec $id_p \in ID^*, id, id_s \in ID$

4.6.1 Incompatibilité de *Move* et Del_2

Notre premier résultat est qu'il est impossible d'avoir une transformée opérationnelle TP1 et TP2 si on prend les opérations Del_2 et Mv en plus de *Add* et *Nop*.

Nous considérons l'arbre initial t_0 avec id_a l'identifiant de l'arête a, id_{p_a} celui de son père (c.a.d. 0;0) et deux opérations concurrentes $Del_2(id_a, id_{p_a})$ effectuée par le site 1 (identifiée par id_{s_1}) et $Mv(id_a, e_1, id_{s_2})$ effectuée par le site 2 (identifiée par id_{s_2}). Nous supposerons de plus $n, m > 1$ dans ce qui suit. On note $t_{del}(t_0) = Del_2(id_a, id_{p_a})(t)$ et $t_{mv} = Mv(id_a, e_1, id_{s_2})(t_0)$.



Proposition 4.4 *il n'existe pas d'opérations $op_{del}, op_{mv} \in \{Add, Del_2, Mv, Nop\}$ telles que $op_{del}(t_{del}) = op_{mv}(t_{mv})$*

Démonstration. La preuve est similaire à la preuve d'incompatibilité entre Del_1 et Del_2. Notons que l'arbre t_{del} a $n + m$ arêtes et l'arbre t_{mv} en a $n + m + 1$.

L'appplication d'une opération de $Op = \{Add, Del_2, Mv, Nop\}$ soit modifie le nombre d'arêtes de l'arbre (*Add*, Del_2) et ce nombre ne peut être augmenté que de 1 au plus, soit ne modifie pas le nombre d'arêtes de l'arbre (*Mv*, *Nop*).

- $op_{mv} = Add$.
  t_{mv} aurait $n + m + 2$ arêtes et t_{del} a $n + m$ arêtes, donc aucun op_{del} n'est possible.
- $op_{mv} = Del_2(id, id_p)$. Selon l'identifiant choisi, plusieurs possibilités se présentent pour op_{del}
  - $id = ida$. L'arbre $op_{mv}(t_{mv}$ a $n > 1$ arêtes sous l'arête e_1 et t_{del} a $n + m$ arêtes, toutes sous la racine. Seule une opération *move* pourrait déplacer les arêtes e'_i sous e_1, mais ne peut en déplacer qu'une seule à la fois (car elle sont toutes filles de la racine dans t_{del}. Donc aucune opération op_{del} ne convient.

  - $id = id_{e_1}$. L'arbre $op_{mv}(t_{mv}$ a $n+m$ arêtes toutes sous la racine avec un identifiant *ida* non présent dans t_{del}. t_{del} a $n+m$ arêtes, donc seules les opérations Nop ou Mv sont possibles. Aucune ne pourra faire apparaitre *ida* donc aucune ne convient.
  - $id = id_{e_i}$, avec $i \neq 1$. L'arbre $op_{mv}(t_{mv}$ a $n > 1$ arêtes e'_i à profondeur 2, et t_{del} a $n+m$ arêtes sous la racine et ne contient pas *da*. Aucune opération Mv ne peut faire apparaitre *ida* et une opération Add ne peut mettre les arêtes e'_i à leur place.
  - $id = e'_i$. Similaire au cas précédent.
- $op_{mv} = Mv(id_1, id_2, id_s)$. L'arbre résultat a $n+m+1$ arêtes et contient l'arête a qui a comme filles les arêtes e'_i. Seulel'opération $op_{del} = Add$ est possible, mais ne pourra jamais obtenir que les arêtes e'_i sous l'arête a.
- $op_{mv} = Nop$. Seul $op_{del} = Add$ permettrait d'avoir le même nombre d'arête mais il ne permet pas de déplacer les arêtes e'_i sous a.

□

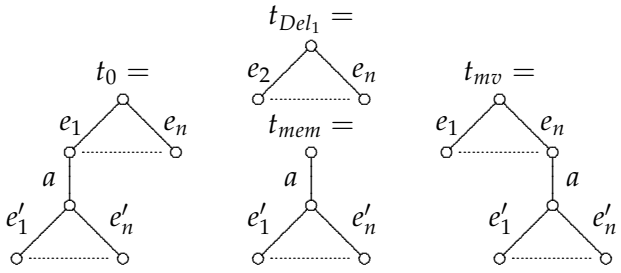
Proposition 4.5 *Il n'existe pas de fonction IT pour l'ensemble d'opération* $Op = \{Add, Del_2, Mv, Nop\}$.

Démonstration. Application directe de la proposition 4.4. □

4.6.2 Mise en mémoire des suppressions

Si on reprend le problème de compatibilité de *move* avec cette fois la suppression Del_1, l'exemple suivant montre qu'il suffirait de mémoriser le sous-arbre disparu.

Exemple 4.3 *Prenons l'arbre t_0 ci-dessous et les opérations concurrentes de suppression de l'arête e_1 (et du sous-arbre correspondant) d'une part et l'opération de déplacement de cette arête (et du sous-arbre correspondant) d'autre part sous l'arête e_n. Au lieu de supprimer brutalement le sous-arbre qui est sous l'arête e_1, sauvegardons-le dans une mémoire t_{mem}.*

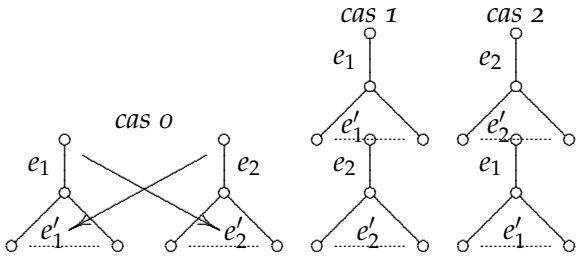


Nous remarquons que sur le site où nous avons fait l'opération Del_1, nous pouvons effectuer l'opération $Mv(a, e_n)$ en utilisant la mémoire t_{mem} (en oubliant momentanément que la structure que nous considérons n'est pas une structure d'arbre). Sur l'autre site nous pouvons exécuter l'opération de suppression de l'arête e_1. A l'arrivée chaque site contient le même arbre et donc l'opération d'intégration IT a simplement consisté à utiliser l'opération à intégrer sans la modifier $IT(op_1, op_2) = op_1$.

L'exemple précédent montre que nous devons modifier notre structure de données afin d'y accueillir une mémoire. Cet ajout ne suffit pas, car la

possibilité de créer des cycles dans la structure de données existe comme illustré dans l'exemple suivant :

Exemple 4.4



Pour des raisons de commodités, nous allons utiliser une structure de graphe orienté. Pour garder le lien avec les arbres et retrouver les arbres à partir d'un graphe, nous nous limitons aux graphes fonctionnels, c.a.d. les graphes tel que chaque nœud possède au plus une arête entrante. Ainsi le cas o de l'exemple sera laissé tel quel jusqu'à un autre déplacement.

4.6.3 Structure de données à base de graphe

Un graphe orienté $G(V, E)$ est un ensemble fini de nœuds V et d'arêtes $E \subseteq V \times V$. Le degré entrant d'un nœud $v \in V$ est l'entier $d^+(v) = \{v' \mid (v', v) \in E\}$. Le degré du graphe est l'entier $\Delta^+(G) = max_{v \in V}(d^+(v))$.

Définition 4.1 *Un graphe fonctionnel est un graphe orienté $G(V, E)$ tel que $\Delta^+(G) = 1$.*

Une racine d'un graphe fonctionnel est un nœud v tel qu'il n'existe pas d'arête (v', v) dans E, i.e. aucune arête n'arrive sur v. Un graphe fonctionnel ne contient pas forcément de racine mais nous ne considérons dans la suite que des graphes fonctionnels contenant au moins une racine, en distinguant l'une d'elle qui sera appelé *la racine* du graphe.

Un nœud v' est accessible depuis un nœud v s'il existe un chemin $v_1 = v, v_2, \ldots, v_n = v'$, i.e. $(v_i, v_{i+1}) \in E$ pour $i = 1, ..., n-1$. Par abus de notation, nous appellerons composante connexe d'un nœud v l'ensemble des nœuds v' accessibles depuis v.

Proposition 4.6 *La composante connexe d'un graphe fonctionnel contenant une racine est un arbre.*

Démonstration. La preuve est par induction sur le nombre n de nœuds du graphe.

- $n = 1$. La composante n'a qu'un nœud qui est la racine et c'est donc un arbre.
- Supposons le résultat vrai pour $p < n$.
  Soit r la racine du graphe et $v_1, \ldots, v_p$ les nœuds tels que $(r, v_i) \in E$. Considérons le graphe G' obtenu en supprimant de G la racine r et les arêtes (r, v). Chaque v_i est une racine de G_i car G est un graphe fonctionnel. Pour chaque v_i considérons G_i la composante connexe de v_i dans G'.
  Fait : $G_i \cap G_j = \emptyset$ si $i \neq j$.
  Sinon nous aurions un nœud v accessible depuis v_i par un chemin $v_i^1 =$

$v_i, \ldots, v_i^l = v$ et accessible depuis v_j par un chemin $v_j^1 = v_j, \ldots, v_j^p = v$. Ces deux chemins sont communs à partir d'un certain rang et donc il existe un nœud v' avec une arête entrante $(v_i^{l_i}, v')$ et une arête entrante $(v_j^{p_j}, v')$ et $v_i^{l_i} \neq v_j^{p_i}$, ce qui contredit le fait que G est fonctionnel.
En appliquant l'hypothèse d'induction sur le graphe G_i en prenant v_i comme racine, nous obtenons que G_i est un arbre pour $i = 1, \ldots, n$. Par conséquent la composante connexe contenant la racine dans G est un arbre.

□

Etant donné un ensemble d'étiquettes $\mathcal{E}$, un graphe étiqueté est un triplet (V, E, λ) avec (V, E) un graphe et $\lambda : E \to \mathcal{E}$ une fonction d'étiquetage des arêtes. Dans la suite nous prendrons $\mathcal{E} = \mathcal{L} \times ID$ avec $\mathcal{L}$ un ensemble de noms et ID un ensemble d'identifiants comme définis précédemment.

La structure de données

La structure de données choisie est un graphe fonctionnel étiqueté (V, E, λ) avec $\lambda : E \to \mathcal{L} \times ID$ telle que $\lambda(e) = (l, id), \lambda(e') = (l, id'), e \neq e' \implies id \neq id'$, c.a.d. un élément de ID apparait au plus une fois dans l'étiquetage. Cela signifie que chaque arête a un identifiant unique. La valeur $\lambda(e)$ est l'étiquette de l'arête. Une conséquence immédiate est qu'un identifiant $id \in ID$ correspond à une arête et une seule au plus, nous l'appellerons l'arête identifiée par id, noté e_{id}. Pour simplifier les notations, nous appellerons toujours graphe fonctionnel une telle structure de données.

Comme dit précédemment nous nous intéressons à des graphes possédant une racine distinguée r. L'arbre associé à un graphe fonctionnel $G = (V, E, \lambda)$, noté $t_r(G)$ est la composante connexe de G contenant la racine r. Si le graphe ne contient pas r (ce qui peut arriver en utilisant l'opération de suppression), alors l'arbre associé est vide.

Opérations d'édition

Nous redéfinissons les opérations d'édition ainsi :
– $Add(id, id_p)$ est définie si id n'est pas un identifiant du graphe. Si id_p est un identifiant du graphe identifiant une arête (w, w') alors l'opération ajoute un nœud v et une arête (w', v) étiquetée par $(NoValue, id)$. Si id_p n'est pas un identifiant présent dans le graphe, alors on ajoute deux nœuds v, v' et une arête (v, v') étiquetée par $(NoValue, id)$. Plus

formellement :

$$\begin{aligned} Add(id, id_p)(V, E, \lambda) &= (V \cup \{v\}, E \cup \{(w', v)\}, \lambda') \\ &\quad avec \begin{cases} \lambda'(e) = \lambda(e) \; si \; e \in E \\ \lambda'((w', v)) = (NoValue, id) \end{cases} \\ &\quad si \; \lambda((w, w')) = (l, id_p) et \; \forall e \in E \; \lambda(e) \neq (l, id) \\ &= (V \cup \{v, v'\}, E \cup \{(v, v')\}, \lambda') \\ &\quad avec \begin{cases} \lambda'(e) = \lambda(e) \; si \; e \in E \\ \lambda'((v, v')) = (NoValue, id) \end{cases} \\ &\quad si \; \forall e \in E \; \lambda(e) \neq (l, id) \; et \; \lambda(e) \neq (l, id_p) \end{aligned}$$

$Del_1(id)$

1. supprime l'arête $e_{id} = (v, v')$,
2. ajoute un sommet $v_{id'}$ pour chaque arête $e_{id'}$ fille de e_{id},
3. remplace $e_{id'} = (v', v'')$ par l'arête $e'_{id'} = (v_{id'}, v'')$ avec $\lambda(e'_{id'}) = \lambda(e_{id'})$.

Les sous-arbres fils de l'arête supprimée restent présents (avec une nouvelle racine), mais le degré entrant des sommets ajoutés est 0, donc le graphe ne sera plus connexe.

– $ChLbl(id, l, ids)$ est crée par le site identifié par id_s et remplace l'étiquette (l', id) d'une arête par (l, id) (si (l', id) est une étiquette du graphe). Formellement :

$$\begin{aligned} ChLbl(id, l, id_s)(V, E, \lambda) &= (V, E, \lambda') \\ avec \;\; \lambda'(e_{id}) &= (l, id) \\ \lambda'(e) &= \lambda(e) \; si \; l'identifiant \; de \; e \; n'est \; pas \; id \end{aligned}$$

– $Mv(id, id', id_s)$ est crée par le site identifié par id_s et n'est définie que si l'arête identifiée par id' n'est pas un descendant de l'arête identifiée par id sur le site émetteur au moment de sa définition.
Cette opération

1. supprime l'arête $e_{id} = (v, v')$,
2. si l'arête $e_{id'} = (w, w')$ existe, crée l'arête $e'_{id} = (w', v)$ avec une étiquette valant $\lambda(e_{id})$,
3. si l'arête $e_{id'}$ n'existe pas, crée un nœud w' et l'arête $e'_{id} = (w', v)$ avec une étiquette valant $\lambda(e_{id})$.

Nous supposerons que si l'opération Mv crée un nœud isolé (sans arête entrante ou sortante), celui-ci est supprimé.

Nous montrons tout d'abord que les fonctions d'édition retournent bien un graphe fonctionnel.

Proposition 4.7 *Soit $G = (V, E, \lambda)$ un graphe fonctionnel. L'application d'une opération $op \in \{Add, Del_1, ChLbl, Mv\}$ à G retourne un graphe fonctionnel.*

Démonstration. Nous allons étudier chaque opération :

– $Add(id, id_p)(V, E, \lambda)$ crée un nœud et ajoute une seule arête entrante sur ce nœud avec un nouvel identifiant. Le graphe reste fonctionnel.
– $Del_1(id)(V, E, \lambda)$ supprime des arêtes donc n'augmente pas le degré sortant des nœuds du graphe.

- $ChLbl(l, id, id_s)(V, E, \lambda)$ ne change que l'étiquetage et ne modifie pas les arêtes.
- $Mv(id, id_p, id_s)(V, E, \lambda)$. En reprenant les notations de la définition, l'arête $e_{id} = (v, v')$ est supprimée, et une arête (w', v') est ajoutée. Le degré entrant du nœud v' reste donc 1, celui des autres nœuds n'est pas modifié.

□

La fonction d'intégration et ses propriétés

Nous définissons la fonction d'intégration correspondante.

$$IT(ChLbl(id_1, l_1, id_{s_1}), ChLbl(id_2, l_2, id_{s_2})) = \begin{cases} Nop(), \text{ si } id_1 = id_2 \text{ et } nsite_1 > nsite_2 \\ ChLbl(id_1, l_1, id_{s_1}), \text{ sinon} \end{cases}$$
$$\text{avec } id_{s_1} = nsite_1; nbop_1 \text{ et } id_{s_2} = nsite_2; nbop_2$$

$$IT(Mv(id_1, id_{p_1}, id_{s_1}), Mv(id_2, id_{p_2}, id_{s_2})) = \begin{cases} Nop(), \text{ si } id_1 = id_2 \text{ et } nsite_1 > nsite_2 \\ Mv(id_1, l_1, id_{s_1}), \text{ sinon} \end{cases}$$
$$\text{avec } id_{s_1} = nsite_1; nbop_1 \text{ et } id_{s_2} = nsite_2; nbop_2$$

$$\text{Autres cas : } IT_1(op_1, op_2) = op_1$$

Le principe de cette fonction d'intégration est qu'en cas de conflit entre deux opérations similaires, le site de priorité maximale (i.e. de numéro de site le plus petit) impose son opération. Cette fonction d'intégration a les bonnes propriétés.

Proposition 4.8 *La fonction d'intégration IT est TP*1.

Démonstration. Nous devons prouver, que pour tout $s = (V, E, \lambda)$,

$$[op_1; IT(op_2, op_1)](s) = [op_2; IT(op_1, op_2)](s), \forall op_1, op_2 \in Op$$

avec op_1 et op_2 deux opérations concurrentes effectuées par le site 1 et le site 2 respectivement. Nous considérons tous les cas possibles pour op_1. Excepté pour deux cas (op_1, op_2 sont $ChLbl(\ldots)$ ou op_1, op_2 sont $Mv(\ldots)$), par définition de IT, la propriété revient à prouver la commutativité de op_1 et op_2 i.e. $[op_1; op_2](s) = [op_2; op_1](s)$. Nous supposons que le site 1 effectue op_1 suivi de (l'intégration de) op_2, alors que le site 2 effectue op_2 suivi de (l'intégration de) op_1.

1. $op_1 = Add(id, id_p)$. Cette opération ajoute un nœud v et une arête $e_{id} = (w', v)$ si $e_{idp} = (w, w')$.
   (a) $op_2 = Add(id', id_p{}')$: Les opérations étant concurrentes id et id' sont deux nouveaux identifiants donc $id \neq id_p{}', id'$ et $id' \neq id_p$. L'arête e_{id_p} est (w_{id_p}, w'_{id_p}) et $e_{idp'}$ est $(w_{id_p{}'}, w'_{id_p{}'})$ (on peut avoir $id_p{}' = id_p$).
   $Add(id, id_p)(V, E, \lambda)$ ajoute un nœud v'_{id} à V et l'arête $e_{id} = (w'_{id_p}, v_{id})$ à E. Effectuer $Add(id', id_p{}')$ ajoute un nœud $v'_{id'}$ et l'arête $e_{id'} = (w'_{id_p{}'}, v'_{id'})$. Le résultat est donc un graphe avec

deux nouveaux nœuds $v'_{id}, v'_{id'}$ et deux arêtes (w'_{id_p}, v_{id}) étiquetées par id et $(w'_{id_p{}'}, v_{id'})$ étiquetée par id. Symétriquement, effectuer $Add(id', id_p{}')$ puis $Add(id, id_p)$ crée deux nouveaux nœuds et les arêtes correspondantes, ce qui donne l'égalité.

(b) $op_2 = Del_1(id')$. Nous avons 3 cas dépendants des valeurs de id' et id.
- $id = id'$. Ce cas ne peut exister car un site ne peut détruire une arête qui est ajoutée concurremment par un autre site donc son identifiant n'est pas encore présent.
- $id_p = id'$. Sur le site 1, l'arête $e_{id_p} = (w, w')$ est supprimée et chaque arête fille $e_{id''} = (w', v'')$ (dont e_{id} qui a été ajoutée par op_1) remplaçée par $e'_{id''} = (\bar{w}', v'')$ avec $\bar{w}''$ un nouveau nœud. Sur le site 2, l'arête $e_{id_p} = (w, w')$ est supprimée et chaque arête fille $e_{id''} = (w', v'')$ (aucune de ces arêtes n'est étiquetée par id) remplaçée par $e'_{id''} = (\bar{w}', v'')$ avec $\bar{w}''$ un nouveau nœud. Ensuite $Add(id, id_p)$ est effectuée. Aucune arête id_p existe, il y a donc création de deux nœuds v, v' et de l'arête $e_{id} = (v, v')$ étiquetée par $(NoValue, id)$. Les deux graphes sont donc identiques (à renommage des nœuds près)
- Sinon Del supprime une arête (qui peut être un ancêtre de id_p) qui n'interfère pas avec id_p. Sur le site 2, cette arête est supprimée et l'ajout se fait sous id_p qui existe toujours dans le graphe. Les deux graphes sont donc identiques.

(c) $op_2 = ChLbl(id', l', id_s)$ Nous savons que $id \neq id'$ car on doit avoir d'abord ajouté le nœud pour pouvoir le modifier. Sur le site 1, l'arête e_{id} est ajoutée sous l'arête e_{id_p} puis $ChLbl$ modifie l'étiquette de l'arête $e_{id'}$ (présente avant l'ajout). Sur le site 2, l'étiquette de l'arête $e_{id'}$ est modifiée, puis l'arête id est ajoutée sous l'arête e_{id_p}. Les deux graphes sont donc identiques.

(d) $op_2 = Mv(id', id_p{}', id_s)$. Par définition, $id \neq id'$ et $id \neq id_p{}'$ car il faut ajouter le nœud avant de le déplacer ou de lui greffer un fils.

Sur le site 1, l'arête e_{id} est ajoutée sous l'arête e_{idp}, puis l'arête $e_{id'}$ déplacée sous une arête qui n'est pas e_{id}.

Sur le site 2, l'arête $e_{id'}$ déplacée sous une arête qui n'est pas e_{id}. Ensuite une arête e_{id} est ajoutée sous l'arête e_{idp}

Par conséquent les deux graphes sont identiques.

2. $op_1 = Del_1(id)$. Cette opération supprime l'arête $e_{id} = (v, v')$ et remplace une arête $e_{id''} = (v, w)$ par une arête $e'_{id''} = (\bar{v}, w)$.

(a) $op_2 = Add(id', id_p{}')$: Cas symétrique de 1b.

(b) $op_2 = Del_1(id')$ ou $op_2 = ChLbl(l', id', id_s)$. La commutativité est immédiate : soit les opérations n'interfèrent pas, soit la suppression sera effectuée sur l'arête avant ou après changement de l'étiquette ce qui ne change pas le résultat final.

(c) $op_2 = Mv(id', id_p{}', id_s)$.
- $id = id'$.

Sur le site 1, l'opération $Del_1(id)$ supprime l'arête $e_{id} = (v, v')$ et modifie chaque arête fille $e_{id''} = (v', w')$ en $e'_{id''} = (\bar{v}', w')$ ensuite l'opération $Mv(id, {id_p}', id_s)$ ne fait rien car aucune arête étiquetée par id existe.
Sur le site 2, $Mv(id, id_p, id_s)$ remplace l'arête $e_{id} = (v, v')$ par $e'_{id} = (w', v')$ si $e_{idp} = (w, w')$. L'opération $Del(id)$ supprime cette nouvelle arête et modifie chaque arête fille $e_{id''} = (v', w')$ en $e'_{id''} = (\bar{v}', w')$. Dans chaque cas le graphe résultant est le graphe initial sans l'arête e_{id} et remplacement d'une arête fille $e_{id''} = (v', w')$ en $e'_{id''} = (\bar{v}', w')$.

- $id = {id_p}'$.
  Sur le site 1, l'opération $Del_1(id)$ supprime l'arête $e_{id} = (v, v')$ et modifie chaque arête fille $e_{id''} = (v', w')$ en $e'_{id''} = (\bar{v}', w')$. L'opération $Mv(id', id, id_s)$ supprime l'arête $e_{id'} = (w, w')$ et la remplace par $e'_{id'} = (\bar{w}, w')$ avec $\bar{w}$ un nouveau nœud car id n'apparait plus.
  Sur le site 2, L'opération $Mv(id', id, id_s)$ remplace l'arête $e_{id'} = (w, w')$ par l'arête $e_{id'} = (v', w')$ si $e_{id} = (v, v')$. L'opération $Del_1(id)$ supprime l'arête e_{id} et modifie chaque chaque arête fille $e_{id''} = (v', w')$ dont $e_{id'}$) en $e'_{id''} = (\bar{v}', w')$.
  Les deux graphes sont donc identiques.
- sinon
  Sur le site 1, $Del_1(id)$ supprime l'arête $e_{id} = (v, v')$ et modifie chaque arête fille $e_{id''} = (v', w')$ en $e'_{id''} = (\bar{v}', w')$. $Mv(id', id_p, id_s)$ remplace l'arête $e_{id'} = (w, w')$, par l'arête $e'_{id'} = (u', w')$ si $e_{idp} = (u, u')$ (u, u', v, v', w, w' tous distincts).
  Sur le site 2, $Mv(id', id_p, id_s)$ remplace l'arête $e_{id'} = (w, w')$, par l'arête $e'_{id'} = (u', w')$ (avec $e_{idp} = (u, u')$), puis $Del_1(id)$ supprime l'arête $e_{id} = (v, v')$ et modifie chaque arête fille $e_{id''} = (v', w')$ en $e'_{id''} = (\bar{v}', w')$ (u, u', v, v', w, w' tous distincts).
  Les deux graphes sont donc identiques.

3. $op_1 = ChLbl(id, l, id_{s_1})$. Les seuls cas à traiter sont $op_2 = ChLbl(id', l')$ et $op_2 = Mv(id', {id_p}')$ les autres cas étant symétriques de cas déjà traités en permutant op_1 et op_2. Le cas $op_2 = Mv(id', {id_p}')$ est immédiat op_2 supprimera les mêmes nœuds et arêtes sur les deux sites, et les étiquettes modifiées par $ChLbl$ seront soit supprimées sur chaque site, soit modifiées de manière identique. Si $op_2 = ChLbl(id', l', id_{s_2})$
   - Si $id \neq id'$ les étiquettes de deux arêtes distinctes sont modifiées indépendemment et donc les opérations commutent, donc les résultats sont identiques sur chaque site.
   - si $id = id'$. Si $nsite_1 > nsite_2$ sur le site 1, l'arête e_{id} devient étiquettée par (l, id) puis l'intégration $IT(ChLbl(id', l', id_{s_2}), ChLbl(id, l, id_{s_1}))$ renvoie $ChLbl(id', l', id_{s_2})$. L'arête devient étiquetée par (l', id). Sur le site 2, la première opération étiquette l'arête par $(l', id' = id)$ et l'intégration renvoit $Nop()$, donc les deux sites ont le même étiquetage. Le cas $nsite_1 < nsite_2$ est symétrique.
4. $op_1 = Mv(id, id_p, id_{s_1})$. Le seul cas à traiter est $op_2 = Mv(id', {id_p}', id_{s_2})$, les autres étant symétriques de cas déjà traités.

La propriété ne se ramène pas à une commutation, mais on doit intégrer la deuxième opération avec IT.

- $id = id'$.
  - si $nsite_1 > nsite_2$.
    Sur le site 1, on effectue $= Mv(id, id_p, id_{s_1})$ qui remplace l'arête $e_{id} = (v, v')$ par l'arête $e'_{id} = (w', v')$ si $e_{id_p} = (w, w')$. Ensuite on effectue $IT(Mv(id, {id_p}', id_{s_2}), Mv(id, id_p, id_{s_1})) = Mv(id, {id_p}', id_{s_2})$ car $nsite_2 < nsite_1$. L'arête $e'_{id} = (w', v')$ est remplacée par l'arête ${e''}_{id} = (u', v')$ si $e_{{id_p}'} = (u, u')$.
    Sur le site 2, on effectue $= Mv(id, {id_p}', id_{s_2})$ qui remplace l'arête $e_{id} = (v, v')$ par l'arête ${e''}_{id} = (u', v')$ (car $e_{{id_p}'} = (u, u')$). Esuite on effectue $IT(Mv(id, id_p, id_{s_1}), Mv(id, {id_p}', id_{s_2})) = Nop()$ car $nsite_1 > nsite_2$. Les deux graphes sont donc identiques.
  - $nsite' > nsite$, cas symétrique du précédent.
- sinon : la propriété demandé correspond à une commutation car l'intégration $IT(op_1, op_2)$ renvoie op_1. Les modifications d'arêtes sont sur des parties disjointes du graphe, donc les opérations commutent.

□

La deuxième propriété est également satisfaite.

Proposition 4.9 *La fonction d'intégration IT est TP2.*

Démonstration. Nous allons prouver que : $IT(IT(op, op_1), IT(op_2, op_1)) = IT(IT(op, op_2), IT(op_1, op_2))$ pour tout op, op_1, op_2 dans l'ensemble Op. Nous supposons que l'opération op_1 est effectuée sur le site 1 (identifié par $nsite_1$), l'opération op_2 est effectuée sur le site 2 (identifié par $nsite_2$) et l'opération op est effectuée sur le site 0 (identifié par $nsite_0$). Ces 3 sites sont distincts par définition car les opérations sont concurrentes.

1. $op = Add(id, id_p)$. Par définition de IT, nous avons
   $$\begin{aligned} Add(id, id_p), op_1) &= IT(IT(Add(id, id_p), op_1), IT(op_2, op_1)) \\ &= IT(IT(Add(id, id_p), op_2), IT(op_1, op_2)) \end{aligned}$$
   pour tout op_1 et op_2.
2. $op = Del_1(id)$. Similaire au cas précédent.
3. $op = ChLbl(id, l, id_s)$
   (a) $op_1 = Del_1(id_1)$ ou $op_1 = Add(id_1, id_{p_1}, id_{s_1})$.
   TP2 devient : $IT(op, op_2) = IT(IT(op, op_2), op_1))$. Par définition de IT, $IT(IT(op, op_2), op_1) = IT(op, op_2)$ d'où le résultat.
   (b) $op_1 = ChLbl(id_1, l', id_{s_1})$.
      i. $op_2 = Del_1(id_2)$ ou $op_2 = Add(id_2, id_{p_2})$. Par symétrie, nous avons le même cas que le cas 3a.
      ii. $op_2 = ChLbl(id_2, l'', id_{s_2})$. Tous les résultats d'intégration sont soit $ChLbl(\ldots)$ soit $Nop()$.
         A. $id = id_1 = id_2$: chaque site modifie la même arête.

- $nsite_0 > Max(nsite_1, nsite_2)$.
  Alors $IT(op, op_1) = Nop()$ et $IT(op, op_2) = Nop()$.
  L'égalité à prouver devient
  $IT(Nop(), IT(op_1, op_2)) = IT(Nop(), IT(op_2, op_1))$
  Par définition de IT, chaque terme est égal à $Nop()$.
- $Max(nsite_2, nsite_1) > nsite_0$. Alors op a la priorité maximale et les deux membres de l'égalité sont égaux à op.
- $nsite_1 > nsite_0 > nsite_2$.
  Le premier terme devient $IT(op, IT(op_2, op_1)) = IT(op, op_2) = Nop()$. Le deuxième terme devient $IT(IT(op, op_2), IT(op_1, op_2)) = IT(Nop(), IT(op_1, op_2)) = Nop()$.
- $nsite_2 > nsite_0 > nsite_1$.
  SymÈtrique du cas précédent.

B. $id = id_1 \wedge id \neq id_2$
- si $nsite_0 < nsite_1$. Chaque membre de l'égalité donne op.
- si $nsite_0 > nsite_1$. Le premier terme est $IT(Nop(), IT(op_2, op_1)) = Nop()$ et le deuxième est $IT(Nop(), IT(op_1, op_2)) = Nop()$.

C. $id = id_2 \wedge id \neq id_1$. Cas symétrique du cas précédent.

D. $id_1 = id_2 \wedge id_1 \neq id$ par définition $^{(1)} = op$ et $^{(2)} = op$.

E. $id \neq id_1 \wedge id \neq id_2$ Par définition toutes les fonctions IT renvoient leur premier argument.

iii. $op_2 = Mv(id_2, id_{p_2})$.
Par définition, nous avons $IT(op, op_2) = op$, $IT(op_1, op_2) = op_1$ et $IT(op_2, op_1) = op_2$. Par dÈfinition, $IT(op, op_1) = ChLbl(\ldots)$. Donc le premier membre est $IT(IT(op, op_1), op')$ avec $op' \neq ChLbl(\ldots)$ donc vaut $IT(op, op_1)$. Le deuxième membre devient $IT(op, op_1)$ d'où l'égalité.

(c) $op_1 = Mv(id_1, id_p{}')$

i. $op_2 = Del_1(id_1)$ ou $op_2 = Add(id_1, id_p{}')$ Par symétrie, nous avons le même cas que le cas 3a.

ii. $op_2 = ChLbl(id_2, l'')$ Par symétrie de du cas 3(b)iii

iii. $op_2 = Mv(id_2, id_p 2)$ Par définition, chaque membre donne op.

4. $op = Mv(id, id_p)$ La fonction d'intégration est définie de manière similaire à celle de *ChLbl* et la preuve de ce cas est similaire à celle pour *ChLbl*.

□

Les deux propositions nous permettent d'effectuer l'édition collaborative en incorporant l'opération *move* tout en maintenant la convergence.

4.6.4 Discussion

Le lecteur ne peut manquer de remarquer que dans un grand nombre de cas l'intégration d'une opération par rapport à une autre retourne l'opération elle-même, i.e. $IT(op_1, op_2) = op_1$. Cela revient à établir que les opérations commutent entre elles et cette remarque a conduit à introduire la notion d'édition collaborative avec des opérations commutatives (CRDT) qui est l'objet du chapitre suivant.

Modèle Commutatif (CRDT) 5

5.1 Introduction

Comme nous l'avons vu dans le chapitre précédent, la transformée opérationnelle n'est pas facile à mettre en place, car la définition d'une fonction IT qui permet d'assurer la convergence est difficile. Nous avons même vu que dans certains cas, il n'est pas possible d'en trouver qui ont les propriétés requises pour assurer la convergence de l'algorithme de Ressel. Cela a conduit à explorer d'autres approches parmi lesquelles celle baptisée CRDT. L'acronyme CRDT veut dire "Commutative Replicated Data Type" cela signifie que les opérations indépendantes peuvent commuter entre elles. Ce terme a été introduit par Marc Shapiro [PMSL09]. Cela revient à avoir une transformée opérationnelle de type $IT(op_1, op_2) = op_1$ pour toutes les opérations. Ce qui implique que $TP2$ est toujours vérifié et que $TP1$ devient : $[op_1, op_2](s) = [op_2, op_1](s)$.

L'intérêt majeur de cette approche est de limiter au maximum le contrôle de concurrence et les calculs successifs se faisant depuis le début de l'édition.

5.2 Dépendance sémantique

Nous avons vu dans le chapitre 4 qu'une opération peut avoir un effet sur les autres opérations (que l'opération d'intégration cherche à prendre en compte). Dans certains cas, cet effet est effectif, par exemple changer l'étiquette d'un nœud créé par ailleurs, mais bien souvent deux opérations concurrentes peuvent être effectuées indépendamment par exemple parce qu'elles s'effectuent sur des parties du document sans lien entre elles. Une opération de suppression et une opération d'insertion seront dépendantes ou pas selon les emplacements respectifs des nœuds d'insertion et de suppression. Cette dépendance est *sémantique* et ce concept va nous permettre de définir un algorithme d'édition collaborative générique, qui repose sur cette notion de dépendance sémantique.

Cette relation est définie sur les opérations et permet de déterminer quand elles peuvent commuter. Pour obtenir un algorithme d'édition sur une structure de donnée, il suffira donc de définir cette relation et de montrer qu'elle satisfait bien les propriétés requises. Cette vérification est bien moins difficile en général que celle des propriétés TP1 et TP2 et nous re-

prendrons en fin de chapitre l'exemple des arbres pour montrer comment obtenir un algorithme d'édition collaborative sur les arbres.

Nous considérons une structure de données, avec un ensemble d'opérations Op et une relation binaire $\succ_s$ sur ces opérations qui est un ordre partiel strict (irréflexive, antisymmétrique et transitive). Nous écrivons $op \parallel op'$ si et seulement si $op \not\succ_s op'$ et $op' \not\succ_s op$.

Définition 5.1 *L'ensemble d'opérations Op est indépendant pour $\succ_s$ (on dira que $(Op, \succ_s)$ est indépendant) si et seulement si pour tout s, pour tout $op, op' \in Op$, $op \parallel op' \Rightarrow [op, op'](s) = [op', op](s)$*

Une suite d'opérations $[op_1, \ldots, op_n]$ est *valide* si et seulement si pour tout $1 \leq i, j \leq n$, $op_j \succ_s op_i \Rightarrow j > i$. Une autre formulation est de dire que la suite d'opérations correspond à une linéarisation de l'ordre partiel $\succ_s$. Etant donné une suite d'opérations $[op_1, \ldots, op_n]$ valide, une substitution σ de $[1, 2, \ldots, n]$ respecte $\succ_s$ si et seulement si $[op_{\sigma(1)}, \ldots, op_{\sigma(n)}]$ est une suite d'opérations valide.

La proposition classique suivante permettra d'obtenir un algorithme d'édition collaborative générique.

Proposition 5.1 *Soit $(Op, \succ_s)$ un ensemble d'opérations indépendant. Alors pour toute suite d'opérations $[op_1, \ldots, op_n]$ valide, pour toute permutation σ qui respecte $\succ_s$, pour tout élément s, on a $[op_1, \ldots, op_n](s) = [op_{\sigma(1)}, \ldots, op_{\sigma(n)}](s)$*

Indication. Comme σ respecte $\succ_s$, il suffit de montrer la propriété pour une suite d'opérations indépendantes deux à deux. Il suffit alors de montrer que l'application d'une suite d'opérations commutant deux à deux donne un résultat indépendant de l'ordre d'application, ce qui se montre facilement par récurrence sur la longueur de la suite. □

5.3 Algorithme *FCedit*

L'algorithme *FCedit* (pour Fast Collaborative Edition) que nous présentons est générique et il est dérivé d'un algorithme distribué élaboré par Lamport [Lam78] pour effectuer le calcul d'un ordre partiel.

Cet algorithme, décrit en figure 5.1, a pour but de garantir l'exécution des opérations reçues en respectant l'ordre sémantique défini ci-dessus. Afin que toutes les opérations aient un identifiant[1], nous envoyons aux autres sites des requêtes constituées de l'opération et de son identifiant. L'ensembles des requêtes est noté R. Nous utilisons un vecteur d'état qui ne sera utilisé que localement par le site. Ce vecteur est un tableau d'entiers dont l'entrée est l'identifiant du site et la valeur de l'entrée est le nombre d'opérations émises par ce site. Il va nous permettre de connaitre le numéro de la dernière opération exécutée émise par ce site. Contrairement à l'algorithme de Ressel[RNRG96] ou de ABST [SLG10], le vecteur d'état

[1]Nous reprenons ici la notion d'identifiant du chapitre précédent qui est un couple contenant un numéro de site (adresse mac, ip etc.) et un numéro d'opération. L'ensemble les dénotant est toujours ID. En plus de l'utilisation du chapitre précédent, ils sont utilisés pour identifier de façon unique les opérations.

n'est pas envoyé aux autres sites. Les opérations d'un site sont exécutées dans l'ordre d'émission, ce qui assure la cohérence du vecteur d'état.

Pour connaitre les dépendances d'une opération, nous devons définir une fonction donnant toutes les dépendances d'une opération par rapport à la nature des objets manipulés. La fonction $dependancesOf(R) \longmapsto \mathcal{P}(ID)$ prend une requête et renvoie un ensemble d'identifiants dont la requête dépend. Cette fonction dépend de la structure de données et du jeu d'opérations utilisé. Pour les arbres, une fonction $dependancesOf$ est définie en section 5.6.4.

5.3.1 Description de l'algorithme

Lorsque le document est modifié la fonction $GenerateRequest(op)$ est appelée avec l'opération effectuée sur le document. Un identifiant est alors généré pour créer une requête. Pour la preuve de l'algorithme on vérifie que la requête est effectivement exécutable. C'est-à-dire que les opérations précédentes du site émetteur et que toutes les opérations dépendantes aient été exécutées. Par définition, si on est le site émetteur : toutes les opérations précédentes sont exécutées. Dans la fonction $isExecutable(r)$ ligne 4, on ne retourne $false$ uniquement, si on n'est pas le site émetteur et que l'opération précédente du cite émetteur n'ait pas été enregistrée dans le vecteur d'état. Les lignes 6 à 8 vérifient que les dépendances ont été exécutées.

Nous revenons dans la fonction $GenerateRequest(op)$ à la ligne 5 où nous incrémentons notre compteur d'opération. Nous appliquons l'opération et l'envoyons à tous les autres participants.

Chaque participant reçoit les opérations des autres. À chaque réception la fonction $Receive(r)$ est appelé. Cette fonction ajoute la requête à une liste d'attente ($WaitingList$ - ligne 3) afin de vérifier qu'elle est exécutable. A la ligne suivante, on parcourt toute la liste d'attente pour y extraire toutes les opérations exécutables. On en sort que lorsqu'il n'y a plus d'opération exécutable.

Lors de l'exécution qui est faite avec la fonction $Execute(r)$. On met à jour le vecteur d'état (ligne 4), on retire la fonction de la liste d'attente et on l'exécute.

5.4 Preuve de correction

Le vecteur d'état est en fait un tableau où l'entrée est le numéro du site et la valeur est le nombre d'opération que l'on a exécuté venant de ce site.

Définition 5.2 *Un vecteur d'état est dit* cohérent *si et seulement si pour tout* $i \in \mathbb{N}, S_i = SReceived[i]$ *,le site a exécuté les opérations numéro* 1 *à* S_i *du site* i*.*

Autrement dit chaque index du vecteur d'état est passé de 0 à sa dernière valeur en passant par tous les nombres intermédiaires.

Proposition 5.2 *Soit* $r = (op, SideId : OpCount)$ *une requête telle que* isExecutable(r) *renvoie*

```
1 INITIALIZE():
2 begin
3 |  ∀i, SReceived[i] = 0   // Vecteur d'état des opérations reçues
4 |  (SiteId, t, OpCount, WaitingList) = (n, o, 1, {})
5 end

1 GENERATEREQUEST(op):   // L'utilisateur envoie une opération
2 begin
3 |  Let r = (op, SiteId : OpCount)
4 |  if isExecutable(r) then
5 |  |  OpCount = OpCount + 1
6 |  |  t = op(t)                          // Applique l'opération
7 |  |  broadCast r au autres participants.
8 end

1 RECEIVE(r):   // Cette fonction est exécutée quand une requête
  est reçue.
2 begin
3 |  WaitingList = WaitingList ∪ r
4 |  forall r ∈ {WaitingList|isExecutable(r)} do
5 |  |  execute(r).          // Exécute toutes requêtes exécutables
6 end
1 ISEXECUTABLE(r):        // Vérifie si la requête est exécutable
2 begin
3 |  Let r = (op, #Site : #Op)
  |     // Vérifie que les précédentes requêtes du site émetteur
  |  ont été exécutées
4 |  if #Site ≠ SiteId ∧ SReceived[#Site] ≠ #Op − 1 then
5 |  |  return false
  |  // Vérifie que chaque dépendance a été exécutée.
6 |  for (nSite : cSite) ∈ dependancesOf(r) do
7 |  |  if SReceived[nSite] < cSite then
8 |  |  |  return false
9 |  return true
10 end

1 EXECUTE(r):                              // Exécute la requête r
2 begin
3 |  r = (op, #Site : #Op)
4 |  SReceived[#Site] = #Op       // Mets à jour le vecteur d'état
5 |  WaitingList = WaitingList/r   // Enlève r de la Waiting List
6 |  t = op(t)                              // Applique l'opération
7 end
```

FIG. 5.1 – *L'algorithme d'édition*

la valeur true. Alors toutes les opérations op′ telles que op′ ≻s op ont été exécutées et le vecteur d'état SReceived est cohérent.

Démonstration. Nous allons montrer que cette fonction permet de garder la cohérence du vecteur d'état et qu'elle permet respecter l'ordre $\succ_s$.

1. Premier point, la cohérence du vecteur : Le vecteur d'état étant une

liste de correspondance entre un site et son nombre de requêtes générées. Il faut donc intégrer les requêtes d'un même site dans leur ordre de génération. Ceci est vérifié à la ligne 4 de la fonction *isExecutable(r)*. $\#Site \neq SiteId \wedge SReceived[\#Site] \neq \#Op - 1$ est vrai si le site est différent du notre (le vecteur d'état concerne les autres sites) et si on a pas intégré l'opération précédente du site émetteur. Ce qui a pour effet de nous retourner $false$. Lorsque l'on arrive à la ligne 6, on a bien que : soit on est le site émetteur, soit on a intégré récursivement toutes les opérations précédentes du site émetteur.

2. La numérotation des opérations contiguë : trivial, car à chaque opération nous incrémentons le compteur $OpCount$ et qui deviendra le numéro de l'opération (ligne 5 de *GenerateRequest*).
3. Respect de la relation d'ordre $\succ_s$: les lignes de 6 à 9 vérifient que toutes les opérations directement antécédentes par la relation d'ordre $\succ_s$ grâce à $dependancesOf(r)$ énumérant les dépendances directes et à l'aide du vecteur d'état afin de savoir si elles ont été exécutées.

L'ensemble de ces points prouve la proposition.

□

Proposition 5.3 *L'algorithme FCedit est convergent si l'ensemble des opérations est indépendant.*

Démonstration. Soit $[op_1; \ldots; op_m]$ une suite d'opérations exécutées sur le site s. Nous allons prouver que $[op_1; \ldots; op_m]$ est une linéarisation de l'ordre partiel définit par la relation de dépendance $\succ_s$ sur l'ensemble $\{op_1, \ldots, op_m\}$.

1. Toutes les requêtes générées par $GenerateRequest(op)$ et envoyées aux autres sont exécutables sur le site local. Ce point est garanti par la ligne 4.
2. Toutes les requêtes reçus et exécutées par $Receive(r)$ sont exécutables grâce à la fonction de la ligne 4 de cette fonction.

Ce qui implique, d'après la proposition 5.2 que toutes opérations sont exécutées sur un site respecte l'ordre $\succ_s$. Donc, de manière récursive : $[op_1; \ldots; op_m]$ est une linéarisation de l'ordre partiel induit par $\succ_s$ sur l'ensemble des opérations $\{op_1, \ldots, op_m\}$. Si chaque site exécute une linéarisation de l'ordre $\succ_s$, la proposition 5.1 implique que chaque site possède le même document.

□

5.5 Complexité de l'algorithme

Soit d_{op} le nombre de dépendances de l'opération op. Soit e_{op} le temps d'exécution de op. Soit m nombre de messages n'arrivant pas dans le bon ordre. Soit nb_{op} le nombre d'opération effectués. Soit br le coût du Broadcast.

La complexité de l'algorithme est en $\mathcal{O}(nb_{op} \times (m \times d_{op} + e_{op} + br))$.

***Remarque* 5.1 —** Nous verrons plus tard qu'avec notre jeu d'opérations et la structure de données, nous avons un nombre maximal de dépendances qui est constant $d_{op} = 1$ et nous verrons dans le chapitre 9 que br est en fait le nombre de voisins connus donné par l'utilisateur. Il existe certain type de réseau où le "broadcast" est autorisé, sa complexité devient constante.

Nous remarquons que la complexité de cet algorithme ne dépend plus de la taille de l'historique dans le sens que pour intégrer une opération, nous n'avons pas besoin de la comparer avec toutes les opérations précédentes dans un pire des cas. Nous devons seulement respecter la dépendance sémantique de chaque opération.

Démonstration. Nous allons considérer que la complexité de la recherche dans le vecteur d'état est constante comme dit plus haut. Calculons la complexité de chaque fonction :

isExecutable : Cette fonction ne boucle que sur le nombre de dépendances $\mathcal{O}(d_{op})$.

Execute : Cette fonction modifie le vecteur d'état et exécute l'opération $\mathcal{O}(e_{op})$.

GenerateRequest : Cette fonction vérifie qu'elle est exécutable $\mathcal{O}(d_{op})$, incrémente un compteur, applique l'opération $\mathcal{O}(e_{op})$ et envoie aux autres participant br. La complexité de cette fonction est donc de $\mathcal{O}(e_{op} + d_{op} + br)$.appels $\mathcal{O}(nb_{op} \times (e_{op} + d_{op} + br))$.

ReceiveRequest : Cette fonction est utilisé lorsque l'on reçoit une requête. On ajoute à la waitinglist (liste d'attente) et on parcourt tous les éléments pour vérifier, s'ils sont exécutables. La complexité de cette fonction est de : $\mathcal{O}(m \times d_{op} + e_{op}))$

Dans le pire des cas, les opérations sont reçues dans le sens inverse de l'envoi et toutes sont dépendantes entre elles (par exemple un peigne unaire où l'on reçoit d'abord la feuille).

Pour une opération générée puis intégrée, nous obtenons : $\mathcal{O}(m \times d_{op} + e_{op} + br)$ et pour nb_{op} opérations $\mathcal{O}(nb_{op} \times (m \times d_{op} + e_{op} + br))$.

□

5.6 Application : les arbres étiquetés

Dans cette partie nous allons reprendre les structures de données du chapitre 4 et montrer comment faire de l'édition collaborative avec l'algorithme *FCedit*.

5.6.1 Structure de données

Nous reprenons la structure de données arborescente définie au chapitre 4, avec une notion d'étiquette plus complexe, qui est jusfifiée dans la section suivante.

L'ensemble des identifiants est défini comme précédemment $ID = \{numsite; numop \mid numsite \in \mathbb{N}^*, numop \in \mathbb{N}\} \cup \{0; 0\}$ et Σ est l'ensemble

des étiquettes. Une étiquette sera de la forme (l, id) avec $id \in ID$ et l une expression de la forme (id', v, lbl) où $id' \in ID, v \in \mathbb{N}$ est un numéro de version et $lbl \in \Sigma_L$ est un label XML.

L'ensemble T des arbres est défini

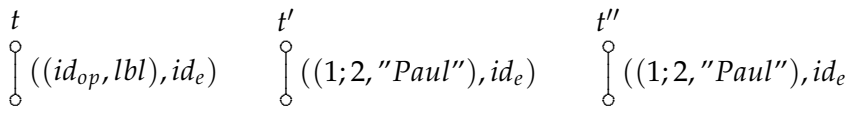
$$T \ni t := \{\} \mid \{n_1(t_1), ..., n_m(t_m)\} \text{ avec } \begin{cases} n_1, ..., n_m \in \Sigma,\ t_1, ..., t_m \in T \\ n_i = (l_i, id_i) \text{ et } l_i = (id'_i, v_i, lbl_i) \\ \text{pour } i = 1, \ldots, n \end{cases}$$

De plus si le multi-ensemble des occurrences d'étiquettes de l'arbre est $\{(l_i, id_i) \mid i \in I\}$ alors nous supposons que $id_i \neq id_j$ pour tout $i \neq j, i, j \in I$. Cette propriété sera appelée unicité des identifiants[2] et étant donné un identifiant $id \in ID$, l'arête identifiée par id sera la seule arête de l'arbre étiquetée par (l, id) (si elle existe).

5.6.2 Absence de causalité et effets sur l'étiquetage des arêtes

Dans la version transformée opérationnelle de *ChLbl* (l'opération ré-étiquetage des arêtes) nous n'avons besoin que de l'identifiant généré par le site émetteur pour donner une priorité sur les demandes de renommage. Nous allons montrer qu'il n'est pas possible de procéder de la même manière. Ce qui justifie la notion d'étiquette plus compliquée introduite dans notre définition d'arbre. L'étiquette (l, id) d'une arête contient l'identifiant id du site qui l'a crée et cet identifiant ne doit pas être modifié. Mais si l est un simple label de Σ_L, il est impossible d'ordonner les réétiquetages et l'étiquette d'une arête serait celle du dernier ré-étiquettage reçu. L'ordre de réception n'étant pas forcément le même sur chaque site, la convergence sera impossible. Donc l doit contenir le label et l'identifiant du site qui a effectué l'étiquetage. L'ordre des priorités, entre sites lu, sur cet identifiant permet de savoir comment modifier le label. L'exemple suivant montre que cela n'est pas suffisant.

Exemple 5.1 *Nous supposons que le site* 4 *reçoit une demande de changement de label du site* 1 *en "Paul". Il veut ensuite renommer ce label en "Jean" de manière non concurrente. Nous modélisons l'étiquette d'une arête identifiée par* id_e (id_{op}, lbl)*, avec lbl le label et* id_{op} *l'identifiant de l'opération qui a donné le dernier label.*



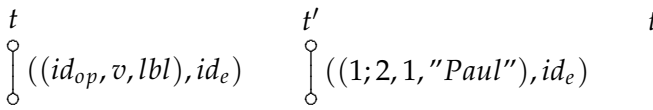
Avec $t' = ChLbl(id_e, "Paul", 1;2)(t)$ *et* $t'' = ChLbl(id_e, "Jean", 4;5)(t')$ *Dans cet exemple le site* 4 *n'a pas pu renommer l'arête car son identifiant n'est pas prioritaire.*

Cette exemple montre que seul le site le plus prioritaire décide du label et que les autres ne peuvent plus le modifier. Çela ne correspond pas à ce

[2] en fait un identifiant peut se retrouver dans une expression l

que l'on attend d'un éditeur collaboratif. Le problème vient du fait que notre algorithme ne garantit que l'exécution des opérations via un ordre partiel qui n'exprime pas toutes les dépendances causales. Pour pallier à ce problème, nous proposons d'ajouter sur chaque arête un numéro de version de son label. Ainsi, si chaque participant joue le jeu, la priorité de chaque site n'est utilisée qu'en cas de version identique.

Exemple 5.2 *Nous reprenons l'exemple précédent avec un numéro de version et en prenant la dernière version du label.*



Avec $t' = ChLbl(id_e, 1;2,1,"Paul")(t)$ *et* $t'' = ChLbl(id_e, 4:5,2,"Jean")(t')$

Nous avons l'effet escompté pour notre exemple.

Comparaisons

Nous définissons les opérations de comparaisons usuelles par : Soit $id_1, id_2 \in ID$ avec $id_i = nsite_i; nbop_i, i \in \{1,2\}$

$id_1 < id_2 \Leftrightarrow nsite_1 < nsite_2$ ou $(nsite_1 = nsite_2$ et $nbop_1 < nbop_2) \Leftrightarrow id_2 > id_1$

5.6.3 Opérations

Nous venons de voir la structure de données et une méthode pour étiqueter les arêtes. Les opérations d'édition sont :

Ajouter une arête. L'opération $Add(id, id_p)$ avec $id_p \neq id$ ajoute une arête étiquetée par (l, id) avec $l = (id, 0, NoValue)$ sous l'arête étiqueté $(\ldots, id_p)$. Quand id_p n'existe pas, l'arbre n'est pas modifié.

$Add(id, id_p)(\{\ \}) = \{\ \}$

$Add(id, id_p)(\{n_1(t_1), \ldots, (l_i, id_i)(t_i), \ldots, n_p(t_p)\})$
$= \{n_1(t_1), \ldots, (l_i, id_i)(t_i \cup ((id, 0, NoValue), id)(\{\ \}) \ldots n_p(t_p)\}$
$si\ id_p = id_i$

$Add(id, id_p)(\{n_1(t_1), \ldots, n_p(t_p)\}) = \{n_1(Add(id, id_p)(t_1)), \ldots, n_p(Add(id, id_p)(t_p))\}$
$si\ n_i = (l_i, id_i)\ avec\ id_i \neq id_p\ pour\ i = 1, \ldots, n$

Suppression d'un sous arbre. L'opération $Del_1(id)$ supprime le sous arbre issu de l'arête $(\ldots, id)$ (incluant l'arête). Quand l'arête (l, id) n'existe pas, l'arbre n'est pas modifié.

$Del_1(id)(\{\ \}) = \{\ \}$

$Del_1(id)(\{n_1(t_1), \ldots, (l_i, id_i)(t_i), \ldots, n_p(t_p)\}) = \{n_1(t_1), \ldots, n_{i-1}(t_i), n_{i+1}(t_{i+1}), \ldots n_p(t_p)\}$
$si\ id = id_i$

$Del_1(id)(\{n_1(t_1), \ldots, n_p(t_p)\}) = \{n_1(Del_1(id)(t_1)), \ldots, n_p(Del_1(id)(t_p))\}$
$si\ n_i = (l_i, id_i)\ avec\ id_i \neq id\ pour\ i = 1, \ldots, n$

Changer une étiquette ou lbl. $ChLbl(id_e, id_{op}, v, lbl)$ avec $id_e, id_{op} \in ID, v \in \mathbb{N}, lbl \in \Sigma_L$ remplace l'expression l_e de l'arête étiquetée par (l_e, id_e), par (L, id_{op}, v), si la version du nouveau label est plus récente.

L'opération *ChLbl* est définie formellement par :

$$
\begin{array}{l}
ChLbl(id_e, id_{op}, v, lbl)(\{n_1(t_1), \ldots (l_e, id_e)(t_e), \ldots n_p(t_p)\}) \\
= \{n_1(t_1), ...(l'_e, id_e)(t_e), \ldots, n_p(t_p)\} \\
\text{où } l_e = (id_e, v_e, lbl_e) \text{ et } l'_e = \begin{cases} (id_{op}, v, lbl), \text{ si } v_e > v \text{ ou } v = v_e \text{ et } id_{op} < id_e \\ l_e, \text{ sinon} \end{cases} \\
ChLbl(id_e, id_{op}, v, lbl)(\{n_1(t_1), ..., n_p(t_p)\})) \\
= (\{n_1(ChLbl((id_{op}, v, lbl), id_e)(t_1)) \ldots n_p(ChLbl(id_e, id_{op}, v, lbl)(t_p))\}) \\
\text{si } n_i = (l_i, id_i) \text{ avec } id_i \neq id_e \text{ pour } i = 1, \ldots, p
\end{array}
$$

5.6.4 Dépendance sémantique

Soit l'ensemble d'opération :
$Op = \{Add(id, id'), Del_1(id), ChLbl(id, id', v, lbl) \mid id, id' \in ID, v \in \mathbb{N}, lbl \in \Sigma_L^*\}$.

La relation de dépendance $\succ_s$ est définit comme suit :

- $Add(id, id_p) \succ_s Del_1(id)$: une arête ne peut-être détruite qu'après avoir été créée.
- $Add(id_p, id_p{}') \succ_s Add(id, id_p)$: ajouter une arête id sous l'arête id_p requiert que id_p soit créée avant.
- $Add(id, id_p) \succ_s ChLbl(id, id_{op}, v, lbl)$: pour changer l'arête id, il faut que l'arête id soit créée avant.

Nous en déduisons la fonction $dependancesOf$ qui est nécessaire à l'algorithme précédent.

$$
dependencesOf(op) = \begin{cases} \{id_p\} \text{ pour } op = Add(id, id_p) \\ \{id\} \text{ pour } op = Del_1(id) \\ \{id\} \text{ pour } op = ChLbl(id, id_{op}, v, lbl) \end{cases}
$$

5.6.5 Preuve d'indépendance

Proposition 5.4 *$(Op, \succ_s)$ est indépendant.*

Démonstration. Nous allons prouver que si $op_1 \parallel_s^* op_2$ alors $[op_1, op_2](t) = [op_2, op_1](t)$ par analyse de cas sur tout les couples possible : op_1, op_2.

1. $op_1 = Add(id_1, id_{p_1})$
   (a) $op_2 = Add(id_2, id_{p_2})$ Par définition aucun Add ne peut dépendre d'une autre opération qu'un Add et $Add(id'_1, id_p 1') \succ_s Add(id'_2, id_{p_2}{}') \Rightarrow id_{p_2}{}' = id'_1$. Nous en déduisons que si $op_1 \succ_s{}^* op_2$ alors id_2 est dans le sous arbre de id_1. Même raisonnement pour $op_2 \succ_s{}^* op_1$
   Ce qui veut dire que si $op_1 \parallel_s^* op_2$, les deux opérations affectent deux sous arbres distincts et on a bien : $[op_1, op_2](t) = [op_2, op_1](t)$.
   (b) $op_2 = Del_1(id_2)$
   - $id_2 = id_{p_1}$ ou id_{p_1} est dans le sous arbre de id_2 :
     Soit t un arbre. $t_1 = Del_1(id_2)(t)$ par définition id_p est supprimé.

$Add(id_1, id_{p_1}) = t_1$. $t_2 = Add(id_2, id_{p_1})(t)$ et $Del_1(id_2)(t_2) = t_1$ parce que le sous arbre a été effacé et les identifiants ne sont utilisés qu'une seule fois.
- $id_2 = id_1$: on a que $Add(id_1, id_{p_1}) \succ_s del(id_1)$.
- Si id_1 est dans le sous arbre de id_2 : Cela implique, par définition, que $Add(id_1, id_{p_1}) \succ_s ... \succ_s Add(id_2, id_{p_2}) \succ_s Del_1(id_2)$ et donc que $op_1 \Vert_s^* op_2$.
- les autres cas : Les deux éléments sont sur deux sous arbres différents.

(c) $op_2 = ChLbl(id_2, id_{op2}, v_2, lbl_2)$
- $id_2 = id_1$: L'arête sera d'abord créée et ensuite renommée car : $Add(id_1, id_{p_1}) \succ_s ChLbl(id_1, id_{op2}, v_2, lbl_2)$.
- Dans les autres cas l'opération n'a pas d'effet sur l'arête créée par $Add(id_1, id_{p_1})$ et vice versa. ◇

2. $op_1 = Del_1(id_1)$

(a) $op_2 = Add(id_2, id_{p_2})$: c'est même cas que le 1b.

(b) $op_2 = Del_1(id_2)$: Si id_1 est un sous arbre de id_2 alors l'ensemble des arêtes supprimées par op_1 est inclut dans l'ensemble des arêtes supprimées par op_2. Quelque soit l'ordre d'exécution le même ensemble d'arêtes sera supprimé. Si id_1 et id_2 sont dans deux sous arbres distincts alors le résultat sera le même.

(c) $op_2 = ChLbl(id_2, id_{op2}, v_2, lbl_2)$

- dans le cas où l'arête identifiée par id_1 est dans le sous arbre de l'arête identifiée par id_2 :
  soit $t' = del(id_1)(t)$. $ChLbl(id_1, id_{op2}, v_2, lbl_2)(t') = t'$ car id_1 n'est plus présent dans t'
  $del(id_1)(ChLbl(id_1, id_{op2}, v_2, lbl_2)(t)) = t'$ car id_1 et son sous arbre ont été supprimés et son nom aussi.
- Dans le cas contraire nous sommes dans deux sous arbres distincts.

◇

3. $op_1 = ChLbl(id_1, id_{op_1}, v_1, lbl_1)$

(a) $op_2 = Add(id_2, id_{p_2})$: c'est le cas 1c.

(b) $op_2 = Del_1(id_2)$: c'est le cas 2c.

(c) $op_2 = ChLbl(id_2, id_{op2}, v_2, lbl_2)$:
- $id_1 \neq id_2$: les deux arêtes sont différentes. Quelque soit l'ordre, cela ne change pas.
- $id_1 = id_2$
  - $v_1 < v_2$ Soit $t_1 = op_1(op_2(t))^{(1)}$
    Soit $t_2 = op_2(op_1(t))^{(2)}$
    Dans $^{(1)}$ le nom de id_1 sera lbl_2 et ne sera pas changé par op_1 (définition). Dans $^{(2)}$ le nom de id_1 sera lbl_1 et sera changé par op_2 en lbl_2 (définition).
    On a bien que $t_1 = t_2$.
  - $v_2 < v_1$: Même démonstration en inversant les indices.

– $v_1 = v_2$ Si $id_{op_1} < id_{op2}$ ce sera le même cas que $v_1 < v_2$
sinon ce sera le même cas que $v_2 < v_1$
Par définition : $id_{op_1} \neq id_{op2}$ (deux opérations différentes)$\diamond$

□

5.6.6 Convergence sur les arbres

D'après les propositions 5.3 et 5.4 nous obtenons le résultat :

Théorème 5.1 *L'algorithme* FCedit *est un algorithme d'édition collaborative convergent sur les arbres étiquetés.*

5.7 Limites de l'approche CRDT

L'approche CRDT ne convient pas à tout type d'opérations, en particulier à la suppression de nœud faisant remonter les sous-arbres que nous avions nommée Del_2. Nous donnons une justification informelle de ce fait. L'exemple suivant correspond à deux opérations concurrentes : un site 1 supprime une arête e dont le père est e_d et un autre site 2 insère une arête e' sous celle-ci. Le résultat sur le site 2 donne une arête e_d qui a comme fils l'arête e' et les fils de l'arête e qui est supprimée. Sur le site 1, exécuter l'opération d'ajout de l'arête e_d ne peut se faire car l'identifiant e a disparu et l'identifiant de son père n'est pas connu.

Exemple 5.3 *Le problème de faire remonter le sous arbre vient du fait que si nous avons indépendamment une opération de suppression d'une arrête b et une opération d'ajout d'une arête d dont le père est b.*

Nous devons le relier au père de d soit b (cf figure 5.2).

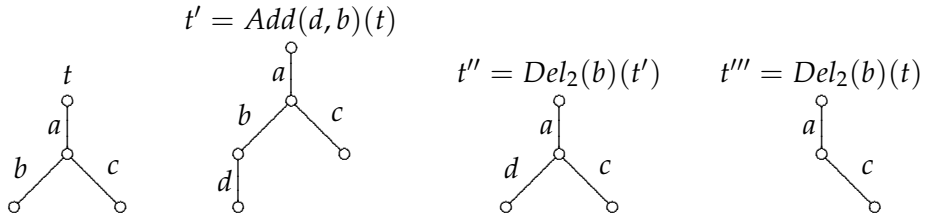


Fig. 5.2 – *Suppression du père e*

Il faudrait donc garder en mémoire, soit les opérations passées (donc un historique), soit maitenir dans la structure de données un lien des fils vers le père. Cette dernière solution n'en n'est pas une. En effet si un arbre a un chemin de n arêtes, on peut imaginer $n-1$ sites qui suppriment chacun une arête distinctes des $n-1$ arêtes du chemin et un site qui insère sous la n^{eme} arête. Un site effectuant les suppressions d'abord, puis l'ajout devrait garder en mémoire tous les pères des nœuds supprimés, ce qui est contradictoire avec l'idée de l'approche CRDT. Dans le cadre de la

suppression d'un sous-arbre (opération Del_1) l'arête et le sous-arbre disparaissent. Donc une opération d'ajout avec un identifiant qui n'apparait plus dans l'arbre ne fait rien, ce qui évite le problème rencontré avec Del_2.

5.8 Discussion sur l'efficacité de notre algorithme dans un réseau pair-à-pair

5.8.1 Complexité des opérations

L'inconvénient de cette modélisation (comme expliqué dans le chapitre précédent) est de trouver les nœuds correspondant à l'identifiant afin de les modifier. Un façon simple est de chercher dans tout l'arbre, mais on se retrouve avec une complexité de la taille de l'arbre pour chaque opération. Ce qui dans le pire des cas rend la complexité de l'algorithme en $\mathcal{O}(nb_{op}^2)$, plus précisément on obtient $\mathcal{O}(nb_{op} \times nb_e)$ avec nb_e le nombre d'arrêtes en prenant la taille du document en compte. En passant par une table de hachage [Ora ; Wika], il est possible d'améliorer la complexité moyenne de l'algorithme pour de grands tableaux. Voir même arriver à une complexité de $\mathcal{O}(1)$. Ce qui finalement permet de passer à l'échelle sur un très grand document.

5.8.2 Autre point de vue

Reprenons les caractéristiques des réseaux pair-à-pair évoqués dans la thèse de Stéphane Weiss [Wei10].

Dynamicité : Le système doit tolérer de grandes variations du nombre d'utilisateurs. Dans notre approche l'algorithme n'a pas besoin de connaitre le nombre de participants. Un nouveau site intégrant le processus n'a qu'à récupérer la structure de données et le vecteur d'état. Une fois la première phase de synchronisation faite à partir de n'importe quel participant, il peut commencer le processus d'édition. La sortie d'un site n'impacte en rien l'édition.

Symétrie : Dans des systèmes pair-à-pair structurés, il peut exister des représentants d'un groupe de nœuds ou des pairs ayant un rôle nécessaire à d'autres pairs. Une défaillance de ce nœud empêche un groupe de participants de fonctionner correctement. Dans notre approche, tous les participants ont le même rôle et envoient à tout le monde ou a des liens répliqués, ce qui rend l'approche robuste aux pannes des autres sites. La contrainte est d'avoir mis en place des mécanismes de délivrance de message sans perte.

Passage à l'échelle : Un système passe à l'échelle si un nombre important d'utilisateurs peuvent y êtres connectés, sans dégradation du service. Nous avons vus que pour chaque opération l'intégration est de faible complexité. Contrairement à beaucoup d'autres approches, nous n'envoyons pas de vecteur d'état, il ne sert que pour le site et nous n'utilisons pas d'historique pour faire converger les copies.

5.9 Conclusion

L'algorithme *FCedit* que nous venons d'introduire est un algorithme convergent extrêmement efficace comme nous le verrons à l'occasion des résultats obtenus par l'implémentation. Sa deuxième caractéristique est d'être générique. Ce qui permet de le décliner pour de multiples structures de données dès lors qu'il est possible de définir une relation de dépendance sémantique intéressante. Il faut noter que dans le pire des cas, toutes les opérations sont dépendantes, ce qui conduit à ce que tous les sites exécutent toutes les opérations dans le même ordre. L'algorithme reste convergent, mais le processus correspond à un processus distribué dans lequel chaque site attend tous les autres ce qui est contradictoire avec le principe de l'édition collaborative.

Édition de Documents XML

6

Dans ce chapitre nous montrons comment réaliser un éditeur *XML* sans et avec annulation d'opérations à l'aide de la modélisation du chapitre précédent. Nous avons introduit dans la section 3.1.4, qu'un fichier XML est un arbre ordonné, alors que nous n'avons traité jusqu'à présent que des arbres non-ordonnés. Dans un premier temps, nous montrons comment l'ordre peut se coder dans les étiquettes des arêtes. Nous verrons ensuite une façon de modéliser les tags et attributs d'un arbre XML. Ensuite nous étendons les résultats du chapitre précédent à cette structure de données et puis nous montrons comment ajouter l'annulation d'opérations.

6.1 Ordonner les arêtes

Dans cette section, nous montrons comment passer d'arbres non-ordonnés à des arbres ordonnés dans notre éditeur collaboratif..

Nous devons résoudre différents problèmes :
- avoir un ordre total (pouvoir comparer chaque arête issue d'un même père),
- avoir un ordre identique sur tous les sites,
- pouvoir insérer avant ou après n'importe quel nœud, et pouvoir insérer entre deux nœuds.

A chaque arête e identifiée par $id_e \in ID$, nous associons un mot p_e d'un alphabet ordonné Σ et nous posons qu'une arête e précède une arête e' si et seulement si $p_e \prec_{lex} p_{e'}$ avec $\prec_{lex}$ l'ordre lexicographique sur les mots de Σ^*. Mais s'il est toujours possible de trouver un mot plus grand que tous les mots d'un ensemble fini et de trouver un mot compris entre deux mots donnés, il n'est pas toujours possible de trouver un mot plus petit qu'un autre (si celui-ci est le plus petit mot pour $\prec_{lex}$). Cela explique que la solution proposée est plus compliquée que ce qui précède.

Nous considérons un alphabet ordonné $\Sigma_0 = \{a_1, \ldots, a_n\}$. Nous notons par $\phi : ID \to \Sigma_0^*$ une fonction injective. Il est toujours possible de définir une telle fonction. Par exemple, si un identifiant est une adresse IP, celle-ci est un mot sur un alphabet $\Sigma_0 = \{0, 1, \ldots, 9, \cdot\}$ et la fonction ϕ est simplement l'identité. Nous introduisons un nouvel alphabet $\Sigma = \{\bot\} \cup \{\#\} \cup \Sigma_0$ ordonné par $\bot \prec \# \prec a_1 \ldots \prec a_n$ et $\prec_{lex}$ est l'ordre lexicographique sur les mots de Σ^*.

A chaque arête e nous associons un mot p_e de Σ^*, et nous disons qu'une arête e précède une arête e' si et seulement si $w_e = p_e \# \phi(e) \prec_{lex} w_{e'} =$

$p_{e'}\#\phi(e')$. La position d'une arête (c.a.d. sa place parmi les arêtes soeurs) fera partie des informations contenues dans l'étiquette de cette arête.

La proposition suivante donne la première propriété requise.

Proposition 6.1 *L'ordre sur les arêtes défini par $e \prec e'$ si et seulement si $w_e \prec_{lex} w_{e'}$ est un ordre total sur l'ensemble des arêtes.*

Démonstration. La fonction ϕ est injective et $\#\phi(id_e)$ est le plus petit suffixe de w_e contenant une occurrence de #, donc les mots associés à deux arêtes distinctes sont distincts. Comme $\prec_{lex}$ est un ordre total sur les mots, la proposition est prouvée. □

La proposition suivante montre que les trois propriétés requises pour gérer les précédences entre arêtes sont vérifies. Nous posons $\mathcal{W}$ l'ensemble des mots de la forme $\Sigma^*\#\phi(ID)$.

Proposition 6.2 *Soit $w, w' \in \mathcal{W}$ avec $w \prec_{lex} w'$.*

1. *Il existe $w'' \in \mathcal{W}$ calculable tel que : $w \prec_{lex} w''$ et $w'' \prec_{lex} w'$.*
2. *Il existe $w_m, w_M \in \mathcal{W}$ tel que $w_m \prec_{lex} w$ et $w' \prec_{lex} w_M$.*

Démonstration.

1. Soit $w = w_p\#w_i \prec_{lex} w' = w_{p'}\#w'_i$. Nous construisons $w'' \in \mathcal{W}$ tel que $w \prec_{lex} w'' \prec_{lex} w'$. Par définition de l'ordre lexicographique tout mot $\overline{w} = w\alpha$ avec $\alpha \in \Sigma^+$ vérifie $w \prec_{lex} \overline{w} \prec_{lex} w'$. Soit $w''_i = \phi(id)$ pour $id \in ID$ différent des identifiants associées à w et w'. Par construction, le mot $w'' = w\#w''_i \in \mathcal{W}$ est tel que $w \prec_{lex} w'' \prec_{lex} w'$ (il est possible d'optimiser la taille des mots "positions".).
2. Soit $w = w_p\#w_i \prec_{lex} w' = w_{p'}\#w'_i$. Nous construisons w_m tel que $w_m \prec_{lex} w$. Dans ce qui suit $s[k]$ dénote la k^{ieme} lettre du mot s et $|s|$ la longueur du mot s.
   Soit $w^m_p[k] = \perp$ pour $i = 1, \ldots, |w_p| + 1$. Soit $w^m_i = \phi(id)$ pour id différent des identifiants associés à w et w'. Par construction le mot $w_m = w^m_p\#w^m_i$ est tel que $w_m \prec_{lex} w$. Une construction similaire permet de construire w_M tel que $w' \prec_{lex} w_M$ (utiliser a_n à la place de $\perp$).

□

En ajoutant l'information d'ordre à l'étiquette d'une arête (en plus de l'identifiant et des labels), il devient possible d'ordonner les arêtes issues d'un même père. Lors de la création d'une arête par *Add*, nous assignons à l'arête créée, une position par défaut, qui pourra être modifiée par une fonction similaire à *ChLbl* (dans la suite ce sera une des fonctionalités de l'opération *SetAttr*).

Nous venons de voir qu'à l'aide des identifiants uniques et d'une étiquette, nous pouvons ordonner les arêtes. Ceci n'est pas seulement valable pour le modèle CRDT, mais peut être appliqué également pour les modèles du chapitre 4 sur les transformées opérationnelles. Par conséquent, tous les résultats donnés dans ce cadre pour les arbres non-ordonnés sont aussi valables pour les arbres ordonnés.

6.2 Format XML

Nous avons vu dans le chapitre précédent que nous pouvons assurer la convergence de documents ayant une structure arborescente non-ordonnée. La section précédente montre qu'on peut aussi faire de l'édition sur une structure ordonnée. Le format XML[1] est un format arborescent, qui contient de plus des attributs de balise. Ces attributs ont un nom - qui doit être unique- et une valeur. Donc chaque arête devra contenir des informations supplémentaires.

Nous allons décrire l'approche que nous avons mise au point avec Pascal Urso et Stéphane Weiss [MUW10].

6.2.1 Structure de données choisie

Nous allons reprendre la structure précédente sauf que chaque arête possède une liste d'attributs. Pour des raisons de commodités, nous plaçons le nom et l'information d'ordre dans deux attributs spéciaux : $@tag$ et $@pos$. Ainsi nous n'avons qu'à modéliser les fonctions structurelles (ici Add et Del) et une fonction d'attribut gérant les étiquettes ($SetAttr$). La suppression d'un attribut (autre que l'ordre ou le nom) se fait par l'envoi de la valeur 'nil'. Il serait aussi possible d'effectuer la suppression en envoyant l'identifiant de la dernière opération ayant modifié l'attribut. Mais dans notre modélisation il n'est pas possible de supprimer définitivement un attribut, car nous oublierions la version du label et pouvons retomber dans le cas décrit dans la section 5.6.2. De plus, si nous attribuons la version 0 à une opération de suppression d'un label, l'opération générée avant cette suppression aurait une version plus récente que celle générée après la suppression. Nous aurions une divergence selon l'ordre d'arrivée.

Pour simplifier, la version d'un label et l'identifiant de la dernière l'opération l'ayant modifié est stocké dans un couple de type $VID = \mathbb{N} \times ID$.

Nous rappelons la comparaison des éléments de l'ensemble ID : Soit $id_1, id_2 \in ID$ avec $id_i = nsite_i; nbop_i, i \in \{1,2\}$

$$id_1 = (nsite_1; nbop_1) <_{id} id_2 = (nsite_2; nbop_2) \Leftrightarrow \begin{cases} nsite_1 < nsite_2 \text{ ou} \\ (nsite_1 = nsite_2 \text{ et } nbop_1 < nbop_2) \end{cases}$$

Maintenant, définissons l'ordre pour les éléments VID :
Soit $Vid_1, Vid_2 \in VID$ avec $Vid_i = (v_i, id_i), i \in \{1,2\}$

$$Vid_1 = (v_1; id_1) <_{vid} Vid_2 = (v_2; id_2) \Leftrightarrow \begin{cases} v_1 < v_2 \text{ ou} \\ (v_1 = v_2 \text{ et } id_1 <_{id} id_2) \end{cases}$$

Définition La définition formelle de la structure de données.

[1] Introduit dans le section 3.1.4.

$$
\begin{array}{rll}
T \ni t ::= & \{\} & \text{// Arbre vide} \\
| & \{(map_1, id_1)(t_1), ..., (map_m, id_m)(t_m)\} & \text{// Ensemble d'arbre} \\
& \text{avec } map_i = \{(attr_{i,1}, value_{i,1}, Vid_{i,1}), ..., (attr_{i,k}, value_{i,k}, Vid_{i,k})\} & \\
& , attr_{i,j}, value_{i,j} \in \Sigma^*, Vid \in VID \text{// un ensemble d'attributs} & \\
& id_1, ..., id_m, id'_1, ...id'_m \in ID,\ t_1, ..., t_m \in T, & \\
& \forall i, j \in [1..m] i \neq j \Rightarrow id_i \neq id_j &
\end{array}
$$

6.2.2 Définition des fonctions modifiant la structure

Pour rendre plus lisible cette structure de données, nous allons définir quelques fonctions.

Modification d'attributs La fonction $SetAttrMap$ va modifier ou ajouter l'attribut à une liste d'attributs. Voici sa définition formelle :

$$
SetAttrMap((attr, value, vid) :: q, nAttr, nValue, nVid) =
\begin{cases}
(attr, value, Vid) :: SetAttrMap(q, nAttr, nValue, nVid) \text{ si } nAttr <_{lexi} attr \\
(attr, nValue, nVid) :: q \text{ si } nAttr = attr \wedge nVid > Vid' \\
(attr, value, Vid) :: q \text{ sinon}
\end{cases}
$$

$$
SetAttrMap([], Attr, nValue, nVid) = (nAttr, nValue, nVid) :: []
$$

Récupération d'une valeur d'attribut $ReadAttr(Map, Attr)$ récupère la valeur d'un attribut $Attr$ à partir de la map.

$$
ReadAttr((attr, value, vid) :: q, Attr') = \begin{cases} value \text{ si } attr = Attr' \\ ReadAttr(q, Attr') \text{ sinon} \end{cases}
$$

$$
ReadAttr([], Attr') = nil
$$

Lister les attributs : $ListAttr(Map)$ extrait une liste de couple valeur-attribut à partir de la map. Nous la définissons simplement par :

$$
ListAttr((Attr, value, vid) :: q) = \begin{cases} (Attr, value) :: ListAttr(q) \text{si } value \neq nil \\ ListAttr(q) \text{ Sinon} \end{cases}
$$

$$
ListAttr([]) = []
$$

Pour la lisibilité de la modélisation, nous allons écrire la fonction Do qui appliquera l'opération à toutes les arêtes de l'arbre. Grâce à cette fonction, nous pouvons définir des opérations seulement pour une arête et elle sera appliquée récursivement sur toutes les arêtes de l'arbre.

Fonction de propagation : $Do : T \times Op \longmapsto T$

$$
\begin{array}{rcl}
Do(op, n(\{\})) & = & op(n(\{\})) \\
Do(op, n(\{n_1(t_1), \ldots, n_p(t_p)\})) & = & op(n(\{Do(op, n_1(t_1)), \ldots, Do(op, n_p(t_p))\}))
\end{array}
$$

Pour simplifier la construction nous introduisons les fonctions intermédiaires suivantes :

$$
\begin{array}{rcl}
getId((id, map)) & = & id \\
getMap((id, map)) & = & map
\end{array}
$$

***Remarque* 6.1 —** Dans le chapitre précédent nous avons vu qu'il est difficile de faire remonter le sous-arbre dans cette modélisation. Nous allons donc nous concentrer sur la suppression du sous-arbre, ici l'opération $Del = Del_1$.

Ajouter une arête : L'opération $Add(id, id_p)$ avec $id_p \neq id$ ajoute une arête possédant l'étiquette $n = (id, map_{def})$ en dessous du nœud étiqueté id_p. Quand id_p n'existe pas, il ne se passe rien. Cette opération est formellement définit par :

$$Add(id, id_p)(n(t)) = \begin{cases} n(t \cup \{(id, map_{def})(\{\})\}) \text{ si } getId(n) = id_p \\ \qquad \text{avec (pour l'instant) } map_{def} = [] \\ n(t) \text{ sinon} \end{cases}$$

Supprimer une arête : L'opération $Del(id)$ supprime l'arête et son sous-arbre identifié par id.

$$Del(id)(n(t)) = \begin{cases} \epsilon \text{ si } getId(n) = id \\ n(t) \text{ sinon} \end{cases}$$

Mettre à jour ou ajouter un attribut : L'opération $SetAttr(id, id_{op}, v, Attr, Value)$ appelle la fonction $SetAttrMap(id_{op}, Attr, Value)$ sur une arrête identifiée par id. Cette fonction ajoute un attribut, s'il n'existe pas ou le modifie s'il existe.

$$SetAttr(id, id_{op}, v, Attr, Value)(n(t)) = \begin{cases} n'(t) \text{ si } getId(n) = id \\ n(t) \text{ sinon} \end{cases}$$
$$\text{où } n' = (SetAttrMap(getMap(n), Attr, Value, (v, id_{op})), getId(n))$$

Maintenant nous pouvons définir les fonctions de changement d'attribut et d'ordre.

$$ChLbl(id, id_{op}, v, lbl) = SetAttr(id, id_{op}, v, @tag, lbl) \text{ où } lbl \neq ""$$

$$ChPos(id, id_{op}, v, p) = SetAttr(id, id_{op}, v, @pos, p)$$

où p est une position définie au début du chapitre.

6.2.3 Dépendance sémantique

- $Add(id, id_p) \succ_s Del(id)$: pour supprimer une arête id, celle-ci doit avoir été créée avant.
- $Add(id', id_p) \succ_s Add(id, id_p)$: pour ajouter l'arête id en dessous de l'arête id_p, l'arête id_p doit avoir été créée avant.
- $Add(id, id_p) \succ_s SetAttr(id, id_{op}, v, Attr, Value)$: pour ajouter ou modifier un attribut, l'arête possédant cet attribut doit avoir été créée avant.

6.2.4 Correction

Dans cette section, nous allons montrer que nos opérations commutent si la dépendance est respectée.

Théorème 6.1 *Soit* $Op_1 = \{Add, Del, SetAttr\}$*, alors l'ensemble* $(Op_1, \succ_s)$ *est* $\succ_s$ *indépendant.*

Nous définissons ${\succ_s}^*$ par : $op_1 \succ_s op_2 \wedge op_2 \succ_s op_3 \Rightarrow op_1 {\succ_s}^* op_3$ et $\|_s^*$ par $op_1 \not\succ_s {}^* op_2 \wedge op_2 \not\succ_s {}^* op_1 \Leftrightarrow op_1 \|_s^* op_2$

Nous définissons la liste d'opérations : $Do(op_n, Do(op_{n-1}, ...Do(op_1, t)...)) = [op_1, ..., op_n](t)$.

Démonstration. Nous montrons que si $op_1 \|_s^* op_2$ alors $[op_1, op_2](t) = [op_2, op_1](t)$ par analyse de tous les cas possibles pour la paire op_1, op_2.

1. $op_1 = Add(id_1, id_{p_1})$
   (a) $op_2 = Add(id_2, id_{p_2})$
   - si $id_{p_1} = id_{p_2}$ alors, nous allons ajouter dans le même ensemble deux arêtes e_1 et e_2 ou vice versa.
     $id = id_{p_1} = id_{p_2}$ et $t'_{id} = t_{id} \cup \{e_1\} \cup \{e_2\}$
   - Sinon par définition les deux opérations sont soit dépendantes soit dans deux sous-arbres différents.

   (b) $op_2 = Del(id_2)$
   - $id_2 = id_{p_1}$ or id_{p_1} sont dans le sous-arbre id_2 :
     Soit t un arbre. $t_1 = Do(Del(id_2), t)$ par définition id_p est supprimé.
     $Do(Add(id_1, id_{p_1}), t) = t_1$. $t_2 = Do(Add(id_2, id_{p_1}), t)$ et $Do(Del(id_2), t_2) = t_1$ car un sous-arbre a été supprimé.
   - $id_2 = id_1$: Par définition nous avons que $Add(id_1, id_{p_1}) \succ_s Del(id_1)$.
   - Autres cas : les opérations sont dans deux sous-arbres différents donc quel que soit l'ordre le résultat sera le même.

   (c) $op_2 = SetAttr(id_2, attr_2, val_2, ts_2)$
   - $id_2 = id_1$: L'arête est créée avant l'attribut car $Add(id_1, id_{p_1}) \succ_s SetAttr(id_1, attr_2, val_2, ts_2)$.
   - Les deux opérations concernent deux nœuds différents, donc l'ordre d'exécution n'a pas d'influence. ◇

2. $op_1 = Del(id_1)$
   (a) $op_2 = Add(id_2, id_{p_2})$: C'est le cas 1b.

   (b) $op_2 = Del(id_2)$ si id_1 est dans un sous-arbre de id_2 alors $[Del(id_1), Del(id_2)](t)$ et il n'y a pas d'arêtes à supprimer avec $Del(id_1)$ car elles ont déjà été supprimées avec $Del(id_2)$. $[Del(id_2), Del(id_1)](t)$ Les arêtes sous id_1 sont supprimées en premier et celle de id_2 avec id_1 sont supprimées aussi. Sinon les deux arbres sont distincts.

   (c) $op_2 = setAttr(id_2, attr_2, value_2, ts_2)$

- $id_1 = id_2$
  Soit $t' = Do(Del(id_1), t)$. $Do(SetAttr(id_1, attr_2, value_2, ts_2)(t') = t'$ parce que id_1 n'est pas présent dans t'.
  $Do(Del(id_1), Do(SetAttr(id_1, attr_2, value_2, ts_2), t)) = t'$ parce que id_1 et son sous-arbre seront supprimés avec leur attributs.
- Dans les autres cas, l'arête supprimée n'a aucune incidence sur le changement d'attribut.

◇

3. $op_1 = SetAttr(id_1, attr_1, value_1, ts_1)$
   (a) $op_2 = Add(id_2, id_{p_2})$: on se retrouve dans le cas 1c.
   (b) $op_2 = Del(id_2)$: c'est le cas 2c.
   (c) $op_2 = SetAttr(id_2, attr_2, value_2, ts_2)$:
   - $id_1 \neq id_2$: les deux arêtes sont différentes.
   - $attr_1 \neq attr_2$: par définition les deux listes sont les mêmes, car elles sont rangées par ordre lexicographique.
   - $id_1 = id_2 \wedge attr_1 = attr_2$
     - $vid_1 <_{vid} vid_2$ Soit $t_1 = op_1(op_2(t))^{(1)}$
       Soit $t_2 = op_2(op_1(t))^{(2)}$
       Dans $^{(1)}$ la valeur de l'attribut $attr_1$ est $value_2$ et ne sera pas changé par op_1 (définition). Dans $^{(2)}$ l'attribut de id_1 est $value_1$ et changé par op_2 en $value_2$ (définition).
       alors $t_1 = t_2$.
     - $vid_2 <_{vid} vid_1$: même cas en inversant les valeurs.
     - $vid_1 = vid_2$ par définition $vid_1 \neq vid_2$ ◇

□

6.3 Les débuts de l'annulation pour le XML

Obtenir un système d'annulation n'est pas chose aisée [RG99 ; CD95 ; WUM08]. Nous allons définir de manière informelle une solution permettant d'avoir des opérations d'annulation, puis nous formaliserons cette solution.

6.3.1 Annulation des opérations d'ajout et de suppression

L'opération qui annule *Add* n'est pas l'opération *Del*. Prenons le scénario suivant : (Voir la figure 6.1)

1. un utilisateur ajoute un élément,
2. un autre utilisateur supprime cet élément,
3. l'ajout de l'élément est annulé,
4. la suppression est annulée de manière concurrente.

Utiliser l'opération *Del* comme annulation de l'opération *Add* (vice et versa) mène à différents résultats selon l'ordre de réception. Par exemple, un élément est visible si une annulation de suppression a été faite après une suppression et pas après l'annulation d'un ajout.

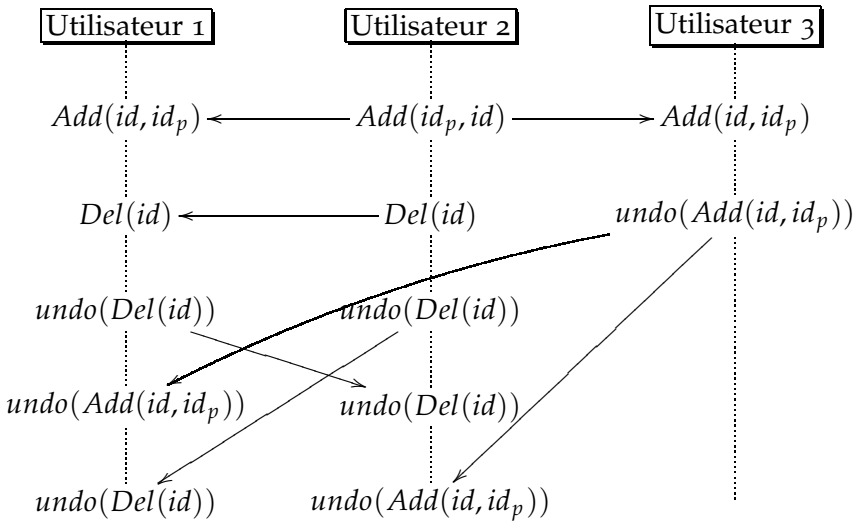


FIG. 6.1 – *Annulations concurrentes.*

Nous devons donc garder les informations des arêtes supprimées. Pour cela nous utiliserons des "pierres tombales". Contrairement aux autres approches de l'annulation, nous pouvons les ranger de telle sorte qu'il n'y ait pas d'impact significatif sur le processus d'édition, car elles ne servent que pour les opérations d'annulations. La génération de l'arbre final ou les placements d'autre nœuds n'en dépendent pas. Un mécanisme de ramasse-miettes pour gérer ces informations supplémentaires permettra de garder l'efficacité de la méthode.

***Remarque* 6.2** — Nous avons montré que l'opération Del_2 n'était pas compatible avec l'approche CRDT, car la structure de donnée ne permettait pas de garder suffisamment d'informations (le père des arêtes supprimées). Comme nous gardons toute l'information, l'opération Del_2 pourrait être utilisée, mais la description de la structure associée à un document XML serait beaucoup plus compliquée. Nous n'aborderons pas cette opération dans ce chapitre.

Pour savoir si un élément est visible, nous allons simplement compter le nombre d'opérations d'annulation d'ajout, les suppressions et les annulations de suppression. Nous appelons cela un compteur d'effets. Le premier ajout va créer réellement l'arête ensuite l'annulation de cet ajout va décrémenter le compteur d'effet de l'ajout et la suppression va incrémenter le compteur d'effet de la suppression. L'annulation de suppression va décrémenter son compteur. Un seul compteur ne suffit pas car comme dit plus haut l'annulation d'un ajout ne peut pas compter comme une suppression.

De plus nous voulons que chaque opération d'annulation d'une suppression d'arête soit identifiée afin d'éviter de la confondre avec deux annulations concurrentes. Nous pensons que cette modélisation combine le respect des intentions de l'auteur et un surcoût (ajout d'informations) limité.

***Remarque* 6.3** — Ceci est une modélisation possible illustrant la capacité du modèle. Nous pouvons imaginer un modèle où la suppression n'existe pas et seule l'annulation d'ajout serait disponible pour effacer une arête.

Différentes possibilités pourrait être explorées pour de futur éditeurs collaboratifs utilisant cette approche.

6.3.2 Annulation des mises à jour des attributs

De façon similaire aux opérations d'ajout et de suppression, nous devons garder aussi les différents états de chaque attribut avec un compteur de visibilité. Nous devons aussi garder un attribut appelé 'nil' qui modélisera la suppression de celui-ci. Nous choisirons l'attribut le plus récent, dont le compteur d'effet est supérieur à 0. L'ordre choisi est le même que le précédent, c'est-à-dire un compteur incrémenté par rapport à la dernière opération sur cet attribut vu par le site émetteur.

6.3.3 Nouvelle Structure de données

Nous reprenons la structure de données précédente où nous modifions les listes d'attributs en ajoutant tous les changements. Avec l'annulation, les attributs d'une arête deviennent une liste ordonnée de *valeurs*, chaque valeur contient trois éléments :

- $v.value$: la valeur de l'attribut (une chaîne de caractères).
- $v.Vid$: le couple de la version du label avec l'identifiant de la dernière modification.
- $v.effect$: le compteur d'effet de cet attribut. (le nombre d'annulations et de répétitions de l'opération modifiant l'attribut)

Cette liste est ordonnée par les couples version et identifiant de la dernière modification (Vid). Les attributs ressemblent à la map illustrée dans la figure 6.2

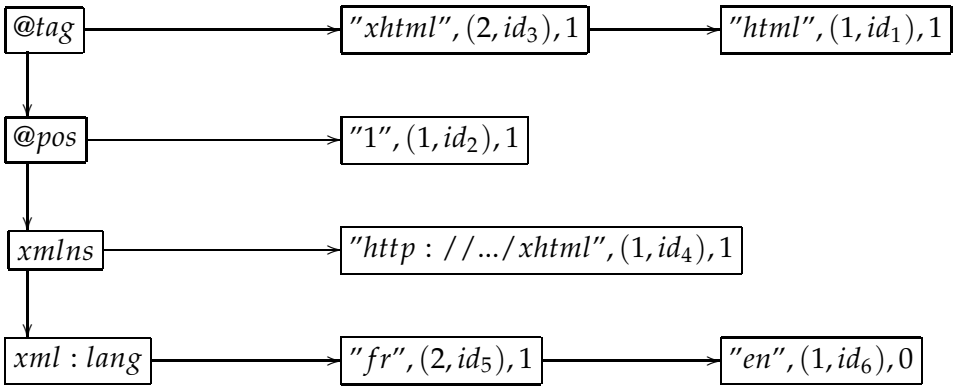


FIG. 6.2 – *Exemple d'une map d'attributs*

Exemple 6.1 *Cet exemple illustre les attributs que pourrait avoir le nœud tagué xhtml qui avant était nommé html. Ce tag possède deux attributs : xmlns et xml : lang le deuxième était en Anglais qui a été annulé pour le mettre en Français.*

L'intérêt d'une telle modélisation est de garder toutes les valeurs nécessaires à l'annulation et ne pose pas trop de problèmes au rendu, car les derniers éléments générés se situent au début. Même si certains ne sont pas visibles. Voici la définition formelle de cette map :

$$
\begin{aligned}
m ::= & \ \{\} \\
| & \ \{(Attr_1, Values_1), ..., (Attr_1, Values_1)\} \\
Values ::= & \ [] \\
| & \ [(value_1, vid_1, effect_1) :: ... :: (value_k, vid_k, effect_k)]
\end{aligned}
$$

Voici les opérations intermédiaires de modifications et de lecture de cette map.

Ajouter un attribut : $AddMap(Map, Attr, Value, vid)$ Cette fonction ajoute à la liste, un nouvel attribut ($Attr$), s'il n'a pas déjà été créé et une nouvelle valeur ($Value$). La valeur initiale du compteur d'effet est 1. Avant de la définir formellement, il faut définir une fonction ajoutant dans une liste une valeur avec le vid correspondant. $AddList : List \times \Sigma^* \times \Sigma^* \times VID \longmapsto List$

$$
\begin{aligned}
& AddList((value_1, vid_1, effect_1) :: q, value, vid) \\
& = \begin{cases} (value, vid, 1) :: (value_1, vid_1, effect_1) :: q \text{ si } vid_1 <_{vid} vid \\ (value_1, vid_1, effect_1) :: AddList(q, Value, vid) \text{ sinon} \end{cases}
\end{aligned}
$$

$$
AddList([], value, vid) = (valut, vid, 1) :: []
$$

Finalement nous écrivons formellement la fonction $AddMap$:

$$
\begin{aligned}
& AddMap((attr, List) :: q, nAttr, nValue, nVid) = \\
& \begin{cases} (attr, List) :: AddMap(q, nAttr, nValue, nVid) \text{ si } nAttr <_{lexi} attr \\ (attr, AddList(List, value, vid)) :: q \text{ si } nAttr = attr \\ (attr, List) :: q \text{ sinon} \end{cases}
\end{aligned}
$$

$$
AddMap([], Attr, nValue, nVid) = (Attr, AddList([], nvalue, nvid)) :: []
$$

Modifier la visibilité : $IncrementEffect(Map, Attr, id, delta)$ Additionne $delta$ à l'effet de la valeur id contenu dans l'attribut $Attr$. Pour définir cette fonction formellement nous devons d'abord définir une fonction intermédiaire. $incrementEffectList : List \times VID \times \mathcal{N} \longmapsto List$

$$
\begin{aligned}
& incrementEffectList((Value, id, effect) :: q, id', delta) \\
& = \begin{cases} (Value, (v, id), effect + delta) :: q \text{ si } id = id' \\ (Value, (v, id), effect) :: incrementEffectList(q, id', delta) \text{ si } id \neq id' \end{cases}
\end{aligned}
$$

$$
incrementEffectList([], id', delta) = []
$$

Nous définissons la fonction $IncrementEffect$ formellement par :

$$
\begin{aligned}
& IncrementEffect((attr, List) :: q, Attr', id', delta) = \\
& \begin{cases} (attr, List) :: IncrementEffect(q, Attr', id', delta) \text{ si } Attr' <_{lexi} attr \\ (attr, IncrementEffectList(List, Attr', id', delta)) :: q \text{ si } Attr' = attr \\ (attr, List) :: q \text{ sinon} \end{cases}
\end{aligned}
$$

$$
IncrementEffect([], Attr', id', delta) = []
$$

Lire la valeur d'un attribut : $ReadAttr(Map, Attr)$ permet de lire la valeur de l'attribut $Attr$. Nous la définissons formellement par :

$$
\begin{array}{l}
ReadAttr((Attr, (value, vid, effect) :: q_{values}) :: q, Attr') \\
= \left\{ \begin{array}{l} value \text{ si } Attr = Attr' \wedge effect > 0 \\ ReadAttr((Attr, q_{values}) :: q, Attr') \text{ si } Attr = Attr' \wedge effect \leq 0 \\ ReadAttr(q, Attr') \text{ sinon} \end{array} \right. \\
ReadAttr((Attr, []) :: q, Attr') = nil \\
ReadAttr([], Attr') = nil
\end{array}
$$

Faire une liste d'attributs : $ListAttr(Map)$ forme une liste de couple valeur attribut. Nous la définissons simplement par :

$$
\begin{array}{l}
ListAttr((Attr, List) :: q) \\
= \left\{ \begin{array}{l} (Attr, ReadAttr([(Attr, List)], Attr)) :: ListAttr(q) \\ \text{ si } ReadAttr([(Attr, List)], Attr) \notin \{nil, "@pos", "@tag", "@add", "@del"\} \\ ListAttr(q) \text{ Sinon} \end{array} \right. \\
ListAttr([]) = []
\end{array}
$$

L'annulation d'une opération est simplement la diminution d'un compteur d'effets. Lorsque qu'un *répéter* (annulation d'annulation) est reçu on incrémente le compteur d'effet de l'opération.

Opération d'édition

Correspondance avec les fonction d'éditions.

Nous rappelons la fonction de propagation : $Do : T \times Op \longmapsto T$

$$
\begin{array}{rcl}
Do(op, n(\{\ \})) & = & op(n(\{\ \})) \\
Do(op, n(\{n_1(t_1), \ldots, n_p(t_p)\})) & = & op(n(\{Do(op, n_1(t_1)), \ldots, Do(op, n_p(t_p))\}))
\end{array}
$$

Quelques fonctions :

$$
\begin{array}{rcl}
getId((id, map)) & = & id \\
getMap((id, map)) & = & map
\end{array}
$$

Ajouter une arête $Add(id, id_p)$ ajoute une arête sous l'arête id_p ne change pas trop elle ajoute en plus un attribut $@add$ ayant pour fonction de compter le nombre d'annulations et de répétitions. Formellement elle s'écrit :

$$
Add(id, id_p)(n(t)) = \left\{ \begin{array}{l} n(t \cup \{(id, map_{def})(\{\})\}) \text{ si } getId(n) = id_p \\ \text{Avec } map_{def} = [(@add, [("Add", (1, id), 1)])] \\ n(t) \text{ sinon} \end{array} \right.
$$

Mettre à jour un attribut $SetAttr(id, id_{op}, v, Attr, Value)$ modifie ou ajoute un attribut s'il n'existait pas déjà.

$$
\begin{array}{l}
SetAttr(id, id_{op}, v, Attr, Value)(n(t)) = \left\{ \begin{array}{l} n'(t) \text{ si } getId(n) = id \\ n(t) \text{ sinon} \end{array} \right. \\
\text{où } n' = (SetAttrMap(getMap(n), Attr, Value, (v, id_{op})), getId(n))
\end{array}
$$

Annuler la mise à jour d'un attribut $undo/redo(SetAttrMap(n, Attr, Value, (v, id_{op})))$ annule ou répète la mise à jour d'un attribut. En réalité il incrémente ou décrémente le compteur d'effet d'une mise à jour.

$$SetAttr(id, id_{op}, v, Attr, Value)(n(t)) = \begin{cases} n'(t) \text{ si } getId(n) = id \\ n(t) \text{ sinon} \end{cases}$$
$$\text{où } n' = (IncrementEffect(getMap(n), Attr, id_{op})), getId(n))$$

Annuler l'ajout d'une arête $Undo/Redo(Add(id, id_p))$ Annule ou répète l'opération d'ajout d'une arête. Comme nous l'avons précisé avant. Cette opération changera un compteur d'effet qui se trouve dans les attributs. $Undo/Redo(Add(id, id_p)) = Undo/Redo(SetAttr(id, id, 1, "@add", "Add"))$

Supprimer une arête $Del(id, id_{op})$ Nous allons ajouter une valeur à l'attribut *@del*. $Del(id, id_{op}) = SetAtt(id, id_{op}, 1, "@del", "Del")$

Annuler la suppression d'une arête $Undo/Redo(Del(id, id_{op}))$ Nous annulons l'opération du dessus. $Undo/Redo(Del(id, id_{op})) = Undo/Redo(SetAtt(id, id_{op}, 1, "@del", "Del"))$

Reprenons les mêmes modélisation pour les opérations *ChPos* et *ChLbl*.

$$ChLbl(id, id_{op}, v, lbl) = SetAttr(id, id_{op}, v, @tag, lbl) \text{ où } lbl \neq ""$$

$$ChPos(id, id_{op}, v, p) = SetAttr(id, id_{op}, v, @pos, p)$$

où p est une position définie au début, en section 6.1

La figure 6.3 représente l'application de nos opérations avec un scénario contenant des annulations et des répétitions concurrentes. A la fin, toutes les répliques sont les mêmes.

6.3.4 Génération du document XML

la structure de données utilisée contient des informations que les utilisateurs ne sont pas sensés voir. Dans ce modèle une arête est visible si le compteur d'effet de l'attribut "*@add*" est strictement positif et si tous les effets des valeurs dans l'attribut "*@del* ont un compteur d'effet négatif ou nul. La fonction *isVisible* exprimant la visibilité de chaque élément est définie de la manière suivante :

$$isVisible(map) = \begin{cases} & ReadAttr(map, "@del") = nil \\ True \text{ si : } & \wedge ReadAttr(map, "@add") \neq nil \\ & \wedge ReadAttr(map, "@tag") \neq nil \\ False \text{ sinon} \end{cases}$$

***Remarque* 6.4 —** Les attributs *@add* et *@tag* partagent les même conditions de visibilité, et auraient pu être fusionnés en gardant une valeur 'nil' spéciale dont le compteur d'effet sert pour la visibilité. Cette solution n'a pas été retenue, car cela complique l'écriture de la fonction *SetAttrMap*.

Nous pouvons maintenant écrire la fonction *model2XML(e)* pour exporter notre structure de donnée en XML. La fonction va d'abord vérifier que le

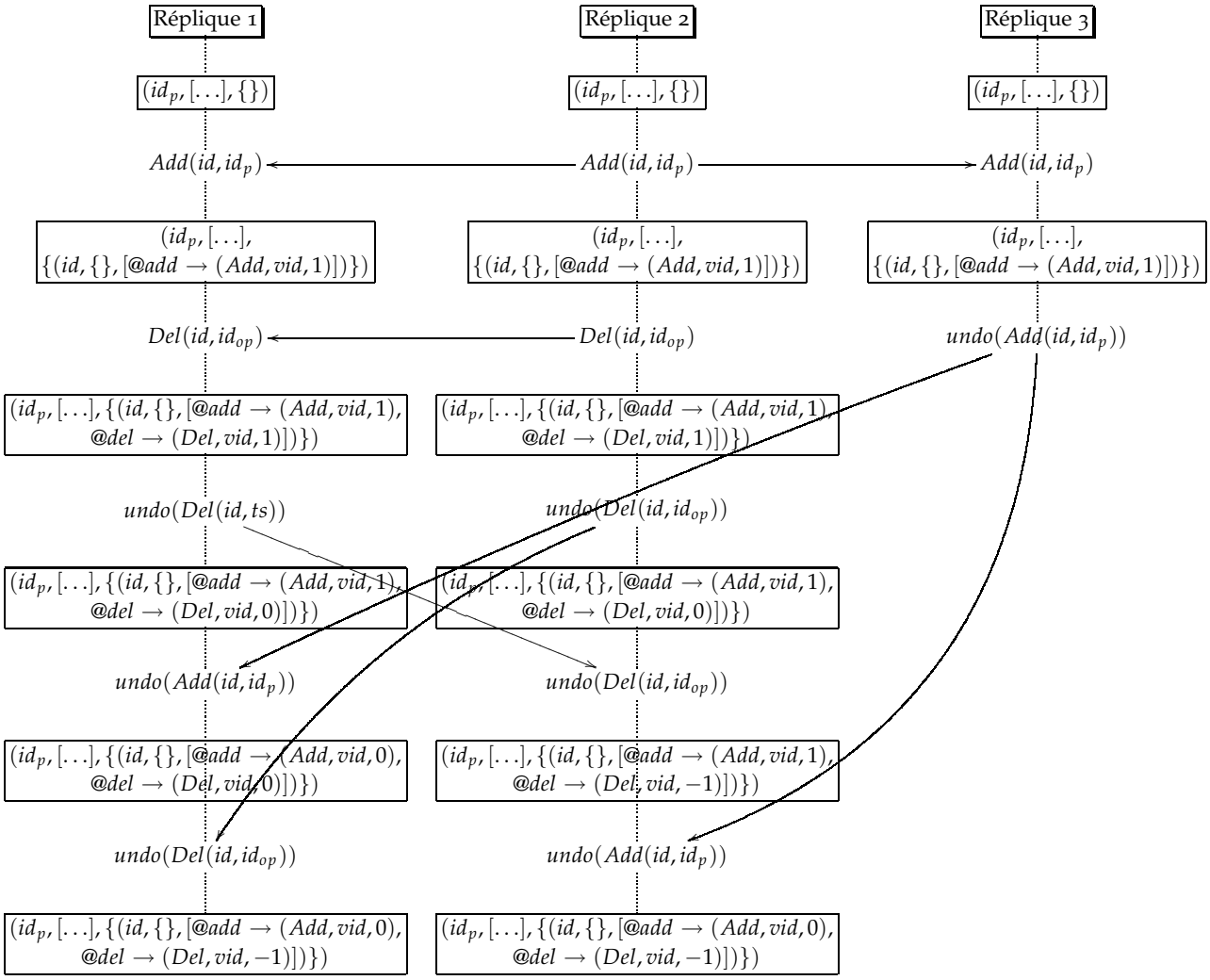


FIG. 6.3 – *Annulation concurrente avec les compteurs d'effets.*

nœud est visible. Si c'est le cas, elle affiche la balise ouvrante, les attributs et s'appelle récursivement sur les fils. Si notre arête n'a pas de fils, il se contentera d'une balise orpheline[2].

Nous définissons d'abord une fonction de mise en forme des attributs.

$$AttrPrint((Attr, Value) :: q) = "Attr =' Value' ".AttrPrint(q)$$
$$AttrPrint([]) = ""$$

Nous définissons une fonction donnant le rang de la première balise à imprimer. $GetFirst(\{(map_1, id_1)(t_1), ..., (map_n, id_n)(t_n)\})$ Revoie l'indice i tel qu'il n'existe pas de j avec $ReadAttr(map_j, "@pos") <_{lexi} ReadAttr(map_i, "@pos")$ ou

si $ReadAttr(map_i, "@pos") = ReadAttr(map_j, "@pos"), id_j <_{id} id_i$. Car nous comparons les positions, puis en cas d'égalité nous départageons avec l'identifiant.

[2]Nous pouvons aussi ajouter un élément dans les attributs pour générer une balise orpheline ou deux balises ouvrantes et fermantes.

Nous définissons notre fonction de rendu XML.

$$
\begin{aligned}
&model2XML(\{(map_1,id_1)(t_1),...,(map_i,id_i)(t_i),...,(map_n,id_n)(t_n)\})\\
&= \begin{cases}
"<".ReadAttr(map_i,"@tag")." ".AttrPrint(ListAttr(map_i))." > "\\
\qquad .model2XML(t_i).\\
" < /".ReadAttr(map_1,"@tag")." > "\\
.model2XML(\{(map_1,id_1)(t),...,(map_n,id_n)(t)\})\\
\text{Avec } i = GetFirst(\{(map_1,id_1)(t_1),...,(map_n,id_n)(t)\})\\
\text{Si } isVisible(map_i) = True \wedge t_i \neq \{\}\\
\\
" < ".ReadAttr(map_i,"@tag")." ".AttrPrint(ListAttr(map_i))."/ > "\\
.model2XML(\{(map_1,id_1)(t),...,(map_n,id_n)(t)\})\\
\text{Avec } i = GetFirst(\{(map_1,id_1)(t_1),...,(map_n,id_n)(t)\})\\
\text{Si } isVisible(map_i) = True \wedge t_i = \{\}\\
\\
"" \text{ Sinon}
\end{cases}\\
&model2XML(\{\}) = ""
\end{aligned}
$$

6.3.5 Dépendance Sémantique

Nous pouvons maintenant définir la dépendance sémantique de nos opérations.

- $Add(id', id_p) \succ'_s Add(id, id_p)$: ajouter une arête id sous l'arête id_p demande que l'arête id_p soit précédemment créée.
- $Add(id, id_p) \succ'_s SetAttr(id, Attr, Value,)$: Créer ou modifier un attribut sur une arête demande que cette dernière arête soit créée précédemment.
- $Add(id, id_p) \succ'_s Undo/Redo(Add(id, id_p))$: Annuler ou répéter la création d'un arête nécessite la création de l'arête.
- $SetAttr(id, Attr, Value, ts, id_{op}) \succ'_s Undo/Redo(SetAttr(id, Attr, Value, ts, id_{op}))$ Annuler ou répéter un changement d'attributs demande d'avoir fait le changement initial avant.
- par définition nous avons :
  - $Add(id, id_p) \succ'_s Del(id, id_{op})$
  - $Del(id, id_{op}) \succ'_s Redo/Undo(Del(id, id_{op}))$

Avec notre conception de l'annulation la dépendance sémantique n'est pas plus compliquée que dans les modélisations précédentes. Etant donné que les opération de modification ne changent que des valeurs dans la structure de données, il suffit que ces valeurs soient créées avant.

6.3.6 Correction des opérations

Dans cette section nous allons montrer que nos opérations sont $\succ_s$-indépendante.

Dans la version avec annulation $Del(id, id_{op})$ devient équivalent à l'opération $SetAttr(id, id_{op}, 1, @del, "Del")$

Théorème 6.2 *Soit $Op_2 = \{Add, SetAttr, Undo/Redo(SetAttr)\}$ avec annulation. l'ensemble $(Op_2, \succ'_s)$ est $\succ_s$-indépendent.*

Démonstration. Nous prouvons que si $op_1 \parallel^*_s op_2$ alors $[op_1, op_2](t) = [op_2, op_1](t)$ par analyse de cas.

1. $op_1 = Add(id_1, id_{p_1})$
   (a) $op_2 = Add(id_2, id_{p_2})$ L'opération ajoute juste un attribut spécial *@add*, la précédente preuve reste valide.
   (b) $op_2 = SetAttr(id_2, attr_2, val_2, ts_2)$
   - $id_2 = id_1$: Par définition, l'arête est créée avant la modification de l'attribut. Car $Add(id_1, id_{p_1}) \succ_s SetAttr(id_1, attr_2, val_2, ts_2)$.
   - Dans d'autre cas $id_1 \neq id_2$ l'opération *add* n'a pas d'effet sur le changement d'attributs d'autre arêtes et vice versa. ◇

   (c) $op_2 = Redo/Undo(SetAttr(id_2, attr_2, val_2, ts_2))$
   - if $id_1 = id_2$ par définition, $op_1 \not\parallel^*_s op_2$
   - sinon : La création d'un arête est indépendante de la modification d'une autre arête.

2. $op_1 = SetAttr(id_1, vid_1, attr_1, value_1)$
   (a) $op_2 = Add(id_2, id_{p_2})$: c'est le cas 1b.
   (b) $op_2 = SetAttr(id_2, vid_2, attr_2, value_2)$: Nous avons plusieurs cas possible :
   - $id_1 \neq id_2$: Les arêtes sont différentes (pas d'effet entre elles).
   - $attr_1 \neq attr_2$ Par définition la liste est la même, parce qu'elle est ordonné par les couples version, identifiant.
   - $id_1 = id_2 \wedge attr_1 = attr_2$: nous distinguons plusieurs cas :
     - $vid_1 \neq vid_2$: par définition nous ajoutons les deux dans la liste des valeurs. Cette même liste est triée par les $vid_{i \in \{1,2\}}$, résultat sera le même. Car l'ajout dans une liste ordonnée est indépendant de l'ordre dans lequel il se fait.
     - $ts_1 = ts_2$ par définition, nous savons que $vid_1 \neq vid_2$ ◇

   (c) $op_2 = undo/redo(SetAttr(id_2, attr_2, value_2, ts_2))$
   - si $id_1 \neq id_2$ ou $attr_1 \neq attr_2$: par définition, $op_1 \not\parallel^*_s op_2$
   - sinon op_1 crée une valeur dans une liste et op_2 incrémente ou décrémente le compteur d'effet d'un autre objet. C'est indépendant.

3. $op_1 = undo/redo(SetAttr(id_1, attr_1, value_1, ts_1))$
   (a) $op_2 = Add(id, id_p)$ c'est le cas 1c.
   (b) $op_2 = SetAttr(id_2, attr_2, value_2, ts_2)$ c'est le même cas que 2c
   (c) $op_2 = undo/redo(SetAttr(id_2, attr_2, value_2, ts_2))$: Nous observons plusieurs cas :
   - si $id_1 = id_2 \wedge attr_1 = attr2 \wedge ts_1 = ts_2$: alors par définition chaque opération incrémente ou décrémente le même compteur d'effet. Les opérations étant atomique, l'addition étant commutative, le résultat reste le même.
   - Sinon chaque opération incrémente ou décrémente deux compteur différents ce qui est strictement indépendant de l'ordre.

Les autres opérations – Del, $Undo/Redo(Add)$, $Undo/Redo(Del)$ – sont définies par composition avec l'opération $SetAttr$ et $Undo/Redo(SetAttr)$). Par extensions nous pouvons dire que l'ensemble d'opération $Add, Del, SetAttr, ChLbl, ChPos, Undo, Redo$ est $\succ_s$-indépendant. □

6.4 Discussion sur le passage à l'échelle

Dans cette section, nous discutons du passage à l'échelle de cette approche. Les approches XML utilisant l'algorithme *FCedit* avec ou sans annulation passent à l'échelle en terme du nombre d'utilisateurs (nombre de réplique). Il n'y a toujours pas de point centraux ou de vecteur d'état embarqué dans les messages.

Nous devons vérifier la complexité de nos fonctions pour qu'elles soient acceptables afin de faire de l'édition collaborative.

6.4.1 Complexité

Calculons la complexité en temps des fonctions utilisées.

- Opérations : Comme dit dans la section 5.8.1 le plus difficile est de trouver les arêtes correspondantes aux identifiants. Dans la modélisation sans undo, le pire des cas est la taille du document. Comme chaque opération doit le parcourir nous aurons une complexité de $\mathcal{O} = nb_e \times nb_{op}$ (nombre d'opérations fois le nombre d'arêtes). Le problème vient du fait qu'en ajoutant la fonctionnalité d'annulation sans ramasse-miette la taille de l'arbre sera égal au nombre d'opérations autres que des annulations. D'où la nécessité d'un ramasse-miette.
- model2XML : Cette fonction parcourt tous les nœuds une seule fois en les triant, sa complexité est de $\mathcal{O}(nb_e^2)$. En utilisant un bon algorithme de tri, on peut avoir une complexité de $\mathcal{O}(nb_e \log(nb_e))$. Evidemment à chaque opération l'arbre n'est pas entièrement revisité. Il est possible de ne changer à chaque fois uniquement les modifications apportées par les opérations et non en régénérant tout le fichier ; en intégrant le système d'édition directement dans un éditeur complet. Sans passer par des fichiers à chaque modification.

Finalement, concernant le passage à l'échelle en terme de nombre d'opérations, notre approche sans annulation demande d'avoir des "pierres tombales" pour les attributs comme pour les règles de thomas [JT05]. Cela entraine aussi un surcoût dû à la position des nœuds comme observés dans logout undo [WUM09].

6.4.2 Discussion pour la conception d'un ramasse-miettes

Pour réaliser les opérations d'annulation, nous devons garder les éléments supprimés et la liste des mises à jour précédentes des valeurs d'attributs ce qui représente un coût non négligeable.

Il est possible de mettre en place un mécanisme de ramasse-miettes comme celui décrit dans la RFC 667 [JT05]. Ce papier prend pour hypothèse qu'à partir d'un certain moment tous les participants ont reçu tous

les messages. Il crée alors un vecteur d'estampillages et déduit l'heure minimum à partir de laquelle aucune annulation n'est possible. Toute information d'annulation invisible, plus vieille que cette heure pourra être supprimée.

Si nous remplaçons les versions par simplement la date avec l'heure du site émetteur. La création d'un nouveau vecteur nous prend en espace $\mathcal{O}(s)$ avec s le nombre de sites suivit. En temps la complexité de nettoyage peut-être de $\mathcal{O}(n)$. Dans la RFC 667, il est nécessaire de connaitre le nombre de sites. Ce qui n'est pas envisageable pour du P2P. Mais compatible avec du cloud computing.

6.5 Conclusion & travaux futurs

Nous venons de voir qu'il est possible d'éditer des documents XML avec l'algorithme *FCedit* pour l'approche CRDT. Cette approche passe à l'échelle pour le nombre de répliques et la version sans annulation passe aussi à l'échelle pour le nombre d'opérations. Malheureusement le mécanisme d'annulation demande de garder les informations de l'opération que l'on veut annuler. Nous avons décrit une méthode de ramasse-miettes pour faire diminuer la taille du document. Un travail futur sera de formaliser cette approche et de l'implémenter dans notre prototype.

Traitement du Typage

7

7.1 Types de documents

Les documents semi-structurés sont comme leur nom l'indique, peu structurés : ce sont des arbres étiquetés d'arité non-bornée. Les *types de documents* sont introduits pour formater les documents manipulés de manière à offrir plus de structure sur ces objets ce qui permet ensuite de les traiter de manière plus sure et plus efficace. La manière usuelle de le faire est via des schémas de document qui permettent d'exprimer des contraintes régulières sur le document (au sens des langages réguliers d'arbres, voir [CDG+07]). Par exemple nous pouvons définir qu'un carnet d'adresse possède des noms uniques et pour chaque nom un type de renseignement (adresse, Téléphone portable, fixe, etc ...) et un seul numéro par type. Nous pouvons pour cela définir un schéma permettant de savoir si le document correspond à nos attentes ou pas. Un langage de contraintes très utilisé en pratique est celui des DTD pour : Document Type Definition. D'autres schémas plus expressifs ont également été proposés comme les schémas XML du W3C ou relax NG qui sont plus expressifs. Toutes ces classes définissent des sous-ensembles des langages réguliers d'arbres.

Travailler avec des documents bien typés est souvent un prérequis qu'exigent de nombreux logiciels et il est donc utile d'incorporer cette notion dans le cadre de l'édition collaborative. Dans ce cadre nous nous posons le problème d'éditer un document dont le type a été déterminé à l'avance, par exemple par une DTD. Par exemple un agenda commun devra respecter le type défini pour les agendas. Le problème de l'édition collaborative est donc d'éditer en mode pair-à-pair un document et d'assurer la convergence sur chaque site sur un document unique qui de plus devra avoir le type donné à l'avance.

Il est immédiat de constater que le processus d'édition ne peut pas garantir que le document a le bon type après chaque opération. Bien souvent, partant d'un document bien typé, une opération provenant d'un site extérieur peut donner un document mal-typé. Il faut donc pouvoir définir quand la vérification du typage est faite, et également comment retrouver un document bien typé à partir d'un document mal-typé. Pour remédier à ces problèmes, nous utiliserons une opération de synchronisation et des algorithmes de *réparation de documents*.

7.2 Réparation de document

7.2.1 Généralités

Le principe de la réparation de document est classique et lié à la notion de distance d'édition. Etant donné un langage L (ensemble de documents) et un ensemble d'opérations d'édition ayant chacune un coût (entier positif ou nul en général), la distance d'un document d au langage L est le coût minimal parmi les coûts des suites d'opérations $[op_1, \ldots, op_n]$ telle que le document $[op_1, \ldots, op_n](s)$ soit dans L. Usuellement, chaque opération a un coût unitaire mais cela n'est pas nécessaire. Un algorithme de réparation prend en argument un document d, un langage L, et renvoie une suite d'opérations correspondant à la distance d'édition.

Pour obtenir des algorithmes effectifs, la classe des langages considérée doit etre précisée et un langage doit se décrire de manière finie et le jeu d'opérations donné.

Par exemple, un algorithme polynomial en $O(n^2)$ existe pour un langage décrit par une DTD et les opérations d'insertion, suppression de feuilles, renommage, voir [SCo6]. D'autres algorithmes existent pour d'autres classes de langages et d'autres jeux d'opérations.

7.2.2 Prérequis

Pour éviter les aspects trop techniques liés au format XML, nous utilisons une définition simplifiée de document XML. Un document est un arbre d'arité non-bornée, ordonné et étiqueté sur les arêtes, défini par la grammaire :

$$\begin{array}{rcl} D \ni d := & \epsilon & \\ & | & l_1(d_1) \cdot \ldots \cdot l_m(d_m)\} \quad \text{avec } l_1, \ldots, l_m \in \mathcal{L}_{XML},\ d_1, \ldots, d_m \in D \end{array}$$

avec $\mathcal{L}_{XML}$ un ensemble de labels XML. Dans les exemples, nous utiliserons parfois la représentation d'un document en utilisant les balises ouvrantes et fermantes, ainsi $l(d_1 \cdot \ldots \cdot d_n)$ s'écrira $< l > (d_1 \cdot \ldots \cdot d_n) < /l >$

Une position est définie comme d'habitude par une suite d'entier et permet de déterminer une arête du document.

Les opérations d'édition sur ces documents sont :

1. $insert(p, i, l)$: si l'arête identifiée par p existe et a au moins $i-1$ fils, l'opération insère une arête fille étiquetée par l qui devient la i^{eme} arête.
2. $remove(p)$: si l'arête identifiée par p existe, l'opération supprime l'arête identifiée par p et ses descendantes.
3. $rename(p, l)$: renomme le label de l'arête identifiée par p en l.
4. $reorder(p, i)$: si l'arête identifiée par p existe et a au moins $i-1$ soeurs, l'opération réordonne ces arêtes de manière à ce que l'arête identifiée par p soit la i^{eme} arête. Les autres arêtes sont inchangées ou décalées de 1 ou -1 selon leur position.

Dans ce qui suit, nous supposons donné un algorithme de réparation *repare(d,S)* pour une classe de langage qui correspond à notre *schéma de type* (usuellement ceux définis par une DTD). Cet algorithme renvoie une liste d'opérations correspondant à la distance d'édition de d vis à vis de S. L'algorithme d'édition que nous définissons utilise cet algorithme de réparation comme une boite noire. De plus chaque site utilisera le même algorithme qui sera supposé déterministe.

7.3 La structure de données

La structure de données est similaire à celle du chapitre 5, mais avec quelques ajouts permettant de définir une vue *locale* et une vue *globale* de l'arbre.

7.3.1 Arbres étiquetés

Comme précedemment, chaque site a un numéro de site $SiteNb$ et chaque opération de ce site est numérotée incrementalement par $NbOp$. De plus nous identifions l'algorithme de réparation de ce site par un identifiant $RepSite$, et chaque opération effectuée par cet algorithme est numérotée incrémentalement par $NbRep$. L'ensemble ID des identifiants est l'ensemble des couples de la forme $(NbSite, NbOp)$ ou de la forme $(RepSite, NbRep)$.

La structure d'arbre non-ordonnées étiquetées est définie par

$$\begin{aligned} T \ni t := \quad & \{\} \\ | \quad & \{n_1(t_1), ..., n_m(t_m)\} \quad \text{avec } n_1, ..., n_m \in \Sigma,\ t_1, ..., t_m \in T \end{aligned}$$

Une étiquette $n \in \Sigma$ est un élément de la forme $(visible, id, order^{Rep}, l^{Rep}, order, l)$ tel que :

1. $visible \in \{0, 1\}$ est un marqueur indiquant que l'arête est visible localement (valeur 1) ou non (0),
2. $id \in ID$ est un identifiant qui n'apparait qu'une seule fois dans l'arbre (propriété d'unicité des identifiants),
3. $l \in \mathcal{L}$ est un label et $order \in \mathcal{L}_{Order}$ est l'information permettant de donner l'ordre de l'arête vis à vis des arêtes soeurs (voir chapitre 6). Ces labels sont définis comme au chapitre 5 (par exemple $l = (id', v, lbl)$ avec $id' \in ID$, v un numéro de version et $lbl \in \mathcal{L}_{XML}$ le label XML).
4. $l^{Rep} \in \mathcal{L}, order^{Rep} \in \mathcal{L}_{Order}$ sont les labels et l'information d'ordre de la vue locale (éventuellement leur valeur est non définie).

De plus si $id = (RepSite : NbRep)$ alors $visible = 1$. Si $id = (SiteNb : NbOp)$ alors visible vaut 1 ou 0. Cette restriction impose que les arêtes crées par l'algorithme de réparation sont locales. Par contre, les arêtes crées par les utilisateurs (celui du site local ou celui d'un autre site) peuvent être ou non dans la vue locale.

Simplification d'écriture : nous identifierons *order* ou $order^{Rep}$ à l'information qu'ils representant c'est à dire l'entier qui indique la position de l'arêt. De même nous identifierons l à lbl si les autres informations de l ne sont pas nécessaires.

7.3.2 Vue locale et vue globale

Notre algorithme d'édition repose sur la notion de *vue* classique dans le cadre des bases de données. Chaque site a une vue globale qui est la partie du document commune à tous les sites et une vue locale du document qui correspond à la partie du document qu'il veut voir. La vue locale peut cacher des parties du document global et contenir des parties qui n'existent que localement.

La fonction $local : \Sigma \rightarrow Bool$ renvoie *True* si et seulement si le champ *visible* de l'étiquette n vaut 1.

La fonction $global : \Sigma \rightarrow Bool$ renvoie *True* si et seulement l'identifiant de l'étiquette n est de la forme $(SiteNb : NbOp)$ (et donc pas de la forme $(RepSite : NbRep)$).

La vue locale $loc(t)$ de l'arbre t est définie par :

$$loc(\{\ \}) = \{\ \}$$
$$loc(\{n_1(t_1), \ldots, n_p(t_p)\ \}) = \{n_{i_1}(loc(t_{i_1}), \ldots, n_{i_l}(loc(t_{i_l}))\}$$

où $n_{i_1}, \ldots, n_{i_l}$ sont les étiquettes n telles que $local(n)$ vaut *True*.

La vue globale $glob(t)$ de l'arbre t est définie par :

$$glob(\{\ \}) = \{\ \}$$
$$glob\{n_1(t_1), \ldots, n_p(t_p)\ \}) = \{n_{i_1}(glob(t_{i_1}), \ldots, n_{i_l}(glob(t_{i_l}))\}$$

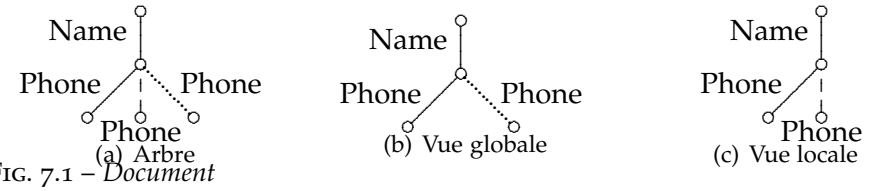
où $n_{i_1}, \ldots, n_{i_l}$ sont les étiquettes n telles que $global(n) = True$.

Le document XML associé à $t' = glob(t)$ consiste simplement à ne garder que la partie label XML d'une étiquette et à ordonner les arêtes selon l'information d'ordre. Il est définit par :

$$d = \epsilon \text{ si } t' = \{\ \}$$
$$d = l_{\sigma(1)}(d_{\sigma(1)}), \ldots, l_{\sigma(p)}(d_{\sigma(p)}) \text{ si } t' = \{n_1(t_1) \ldots, n_p(t_p)\}$$
$$\text{avec} \begin{cases} \sigma \text{ permutation de } \{1, \ldots, p\} \\ n_{\sigma(i)} = (\ldots, i, l_{\sigma(i)}) \\ d_{\sigma(i)} \text{ est le document associé à } t_{\sigma(i)} \end{cases}$$

Le document XML associé à $loc(t)$ est défini de manière similaire.

Exemple 7.1 *L'exemple suivant donne un arbre ainsi que sa vue locale et sa vue globale. Pour simplifier le dessin, une arête étiquetée par* $n = (visible, id, order^{Rep}, l^{Rep}, order, l)$ *avec* $id = (SiteNb : NbOp)$ *est représenté par une ligne continue étiquetée par* l *si* $visible = 1$ *et par une ligne pointillée ... étiquetée par* l *si* $visible = 0$. *Si* $id = (RepSite : NbOp)$, *elle est représentée par une ligne* $- -$ *étiquettée par* l^{Rep}.



(a) Arbre (b) Vue globale (c) Vue locale

FIG. 7.1 – *Document*

7.3.3 Opérations d'édition

Les opérations d'éditions sont les mêmes que celles du chapitre 5 avec une opération supplémentaire de réordonnancement d'arêtes.

- $Add(id, id_p)$: ajout d'une arête sous l'arête identifiée par id_p. Des valeurs par défaut sont données aux champs de l'étiquette non spécifiés.
- $Del(id)$: suppression de l'arête id et du sous-arbre correspondant.
- $ChLbl(id, l)$ (avec $l = (id', v, lbl)$) : remplace le label de l'arête identifiée par id par l (en conformité avec le numéro de version, voir chapitre 5).
- $ChOrder(id, order')$: cette opération est similaire à $ChLbl$ mais change l'information d'ordre de l'arête au lieu de changer le label.

Op dénote l'ensemble des opérations d''édition.

7.3.4 Dépendance entre les opérations

La dépendance sémantique $\succ_s$ entre opérations est définie comme dans les chapitres précédents.

$$Add(id_p, id) \succ_s Del(id)$$
$$Add(id', id_p) \succ_s Add(id_p, id)$$
$$Add(id_p, id) \succ_s ChLbl(id, l)$$
$$Add(id_p, id) \succ_s ChOrder(id, order)$$

Comme précédemment, $(Op, \succ_s)$ est indépendant. La fonction $DependancesOf$ est définie de la même manière ($ChOrder$ a les mêmes dépendances que $ChLbl$).

7.3.5 Opération Fix

L'opération *fix* est une opération interne à chaque site qui n'est pas diffusée aux autres sites, par contre elle sera invoquée par l'opération de synchronisation *sync* qui est décrite dans la section suivante. L'application de fix à t, notée $fix(t)$ se définit de la manière suivante. Dans un premier temps, l'algorithme de réparation *repare* est appelé sur l'arbre $glob(t)$[1], plus exactement sur d le document XML associé à $glob(t)$. Le résultat de cet algorithme est une suite d'opérations de la forme $insert(p, i, l)$, $remove(p)$, $rename(p, l)$ ou $reorder(p, i)$. Cette suite d'opérations est appliquée à l'arbre t pour obtenir une vue locale $loc(t)$ qui a le bon type de la manière suivante :

[1] pas sur $loc(t)$!

1. $t = glob(t)$ i.e. effacer toutes les arêtes générées par une réparation précédente et mettre les marqueurs *visible* à 1.
2. Appliquer chaque opération op de la liste générée par *repare* ainsi :
   - $op = insert(p, i, l)$: p est une position correspondant à id_p de l'arbre $loc(t)$, i est un entier, l est un label XML. Alors soit $id = (RepSite : NbRep)$, incrémenter $NbRep$ en $NbRep + 1$, et effectuer l'opération $Add(idp, id)$ en fixant l'étiquette de l'arête crée à $n = (1, id, i, l, order_\perp, l_\perp)$.
   - $op = remove(p)$: p est une position d'arête feuille de d qui correspond à une unique arête étiquetée $n = (\ldots, id_p, \ldots)$ dans t. L'application de $remove(p)$ remplace n par $n' = (0, id, order^{Rep}, l^{Rep}, order, l)$. L'arête n'est pas supprimée mais n'est plus dans la vue locale.
   - $rename(p, l')$: p est une position désignant une arête de d qui correspond à une unique arête de t étiquetée par $n = (visible, id_p, order^{Rep}, l^{Rep}, order, l)$. L'application de *rename* consiste à remplacer n par $n' = (1, id_p, order^{Rep}, l', order, l)$.
   - $reorder(p, i)$: p est une position désignant une arête de d qui correspond à une unique arête de t étiquetée par $n = (visible, id_p, order^{Rep}, l^{Rep}, order, l)$. Alors n est remplacé par $n' = (1, id_p, order^{Rep'}, l^{Rep}, order, l)$ tel que $order^{Rep'}$ établit que l'arête est la i^{th} arête.

Par définition de fix, nous avons la proposition suivante.

Proposition 7.1 *Après l'éxécution de $fix(t)$ sur un site s, la vue locale $loc(t)$ appartient au type S.*

7.3.6 L'opération sync

Nous étendons l'ensemble d'opération Op par une opération de synchronisation $sync(Dep)$ qui permet à un site de garantir qu'à un certain moment, la vue locale du document appartient au type voulu S. Comme il n'existe pas d'horloge globale dans notre modèle[2], elle est remplacée par un ensemble $Dep = \{op_1, \ldots, op_n\}$ d'opérations indépendantes (i.e. $op_i \parallel op_j$ for $i \neq j$) qui est l'ensemble des opérations minimales pour l'ordre de dépendance, calculé quand l'opération *sync* est invoqué. Cette opération n'a aucun effet sur le document, mais l'envoi aux autres sites permet à ceux-ci de savoir que le document est bien typé.

Par définition

$$sync(Dep)(t) = fix(t)$$

et l'ordre de dépendance est étendu par

$$op \succ_s sync(Dep) \quad \forall op \in Dep$$

Il faut noter que Dep peut contenir d'autres opérations *sync*.

[2] ce qui serait contradictoire avec l'aspect pair-à-pair

7.4 Modification de l'algorithme *FCedit*

7.4.1 Un exemple

Nous donnons un exemple qui permet de comprendre les interactions entre vue locale, globale, algorithme de réparation et synchronisation. Nous prenons comme type une DTD qui impose qu'il n'y a qu'une arête *Phone* sous une arête *Name*.

Un site (1) débute l'édition avec l'arbre global t_1. En invoquant l'algorithme *repare* par une opération *sync*, ce site obtient une vue locale $repare(t_1) \in S$

Deux sites (2) et (3) ajoutent chacun de manière concurrente une arête *Phone* sous l'arête *Name* ce qui donnera l'arbre t_2 sur le site (1) après réception et exécution de ces opérations. La vue globale partagée par tous les sites après réception de toutes les opérations d'édition est $glob(t_2)$ (par contre les vues locales de (1) et (2) après réception du *sync* du site (1) seront distinctes de cette vue globale).

Si le site (1) envoie aux autres sites une nouvelle opération *sync*, l'invocation de l'algorithme de réparation sur le site (1) via fix va supprimer l'arête locale *Phone* (provenant de la réparation initiale) et cacher -sans la détruire- une une arête *Phone* provenant du site (2) ou (3).

Si les autres sites n'ont fait aucune autre édition, alors à la réception du *sync*, chacun de ces sites effectuera l'opération locale fix qui invoque l'algorithme de réparation sur la vue globale $glob(t_2)$. Tous les sites auront la même vue locale (d) car l'algorithme de réparation est le même sur chaque site et déterministe et il est appelé sur la même vue globale.

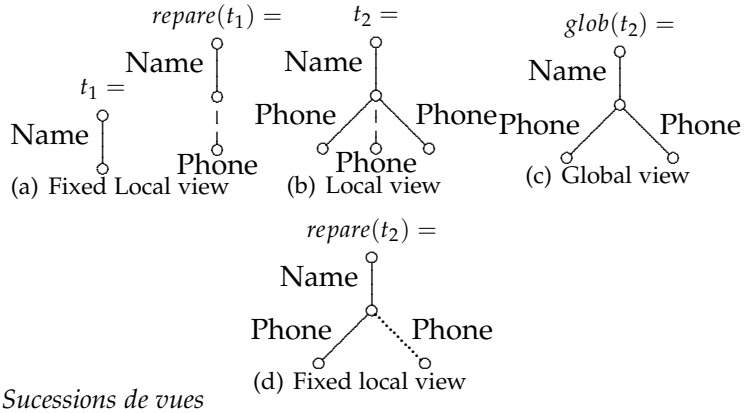


Fig. 7.2 – *Sucessions de vues*

7.4.2 Un algorithme d'édition avec typage

L'algorithme utilisé est une adaptation de l'algorithme *FCedit*, qui permet de prendre en compte les vues locales et d'effectuer des opérations de réparation si une opération de synchronisation est invoquée.

Les opérations *SendOp* **et** *GenerateRequest* Un site effectue l'édition à partir de sa vue locale. Il peut donc effectuer une opération *op* impossible pour les autres sites, par exemple ajouter une arête sous une arête locale crée par son algorithme de réparation. Dans ce cas il faut construire une suite d'opérations qui permette de définir *op* comme une opération

légale sur la vue globale. L'opération $SendOp(op)$ calcule la suite d'opérations d'édition nécessaires pour pouvoir effectuer op, complétée avec op. Dans la majorité des cas $SendOp(op)$ renvoie op, mais si op dépend d'une arête créée par une réparation, celle-ci devra être crée effectivement par une opération (qui sera envoyée à tous les sites). Par conséquent, le numéro d'opération attribué à op devra être modifié et l'identifant de l'arête modifié. Cette opération utilise deux fonctions auxiliaires :

- $getPere : ID \rightarrow ID$ telle que $getPere(id)$ renvoie l'identifiant de l'arête père de l'arête identifiée par id.
- $GenOpFix : ID \rightarrow Op$ telle que $genOpFix$ renvoie l'opération d'ajout permettant de créer l'arête identifiée par id. Elle est définie par

$$GenOpFix(id) = Letidp = getPere(id)inreturnAdd(idp, id).$$

$GenerateRequest(op)$ est l'opération qui génère une opération sur un site et l'envoie aux autres sites. Contrairement à l'algorithme *FCedit* original, il doit invoquer SendOp pour vérifier si d'autres opérations doivent être réalisées avant d'envoyer op.

Algorithme Les principales modifications de l'algorithme sont :

- *GenerateRequest* invoque *SendOp* pour garantir que l'opération op a un sens (et ces opérations sont mutuellement récursives).
- La réception d'une opération de synchronisation peut invoquer l'algorithme de réparation.

7.5 Convergence

Le théorème suivant établi que l'algorithme converge et que le document final est bien typé.

Théorème 7.1 *Si chaque site effectue une suite d'opération terminée par une opération* sync, *une fois que toutes les opérations ont été effectuées par tous les sites :*

1. *Tous les sites ont la même vue globale $glob(t)$,*
2. *Tous les sites ont la même vue locale $loc(t)$ qui est bien typée.*

Démonstration. Par définition des opérations et de l'algorithme :

- L'exécution d'un opération *sync* ne modifie pas $glob(t)$.
- L'éxécution d'une opération fix ne modifie pas $glob(t)$.
- Une exécution de l'algorithme modifié correspond à une exécution de l'algorithme *FCedit* original avec des invocations supplémentaires de fix et *sync*.

Par conséquent à la fin de l'exécution, la convergence de l'algorithme *FCedit* entraine que chaque site a le même arbre pour $glob(t)$.

La dernière opération générée sur chaque site est *sync* soit générée par le site, soit reçue d'un autre site, donc par définition la vue locale est bien typée. Comme elle est exécuté à partir de la même vue globale et que l'algorithme de réparation est déterministe, la même vue locale est calculée sur chaque site. □

```
1  SendOp(op):        // Vérifie et répare chaque dépendance d'un
       identifiant local
2  begin
3  |  forall id ∈ dependanceOf(op) do
4  |  |  if id = (RepSite : NbRep) then
5  |  |  |  Let OpFix = GenOpFix(id)
6  |  |  |  Let r = GenerateRequest(OpFix)
7  |  |  |  Let (op, #Site : #Op) = r
8  |  |  |  op = op[id := #Site : #Op]
9  |  return op
10 end
```

```
1  GenerateRequest(op):   // L'utilisateur envoie une opération
2  begin
3  |  op = SendOp(op)  // Peu s'appeler récursivement avec SendOp
4  |  Let r = (op, SiteId : OpCount)
5  |  OpCount = OpCount + 1
6  |  Execute(r)                          // Applique une opération
7  |  broadCast r vers les autres participants.
8  |  return r
9  end
```

```
1  Receive(r):   // Cette opération est appelé à chaque reception
2  begin
3  |  WaitingList = WaitingList ∪ r
4  |  forall r ∈ WaitingList|isExecutable(r) do
5  |  |  r = (op, #Site : #Op)
6  |  |  StateReceived[#Site] = #Op        // met à jour le vecteur
   |  |     d'état
7  |  |  Execute(r)
8  |  |  WaitingList = WaitingList/r     // Le supprime de la liste
   |  |     d'attente
9  end
```

Fig. 7.3 – *Algorithme* FCedit *avec réparation - partie 1*

```
1  INITIALIZE():
2  begin
3  |  ∀i, SReceived[i] = 0 // Le vecteur d'état des opérations reçus
4  |  (SiteId, t, OpCount, WaitingList) = (n, o, 1, {})
5  end
```

```
1  ISEXECUTABLE(r):     // Vérifie qu'une requête soit éxecutable
2  begin
3  |  Let r = (op, #Site : #Op)
   |  // Vérifie que la requête précédente soit éxecuté
4  |  if #Site ≠ SiteId ∧ SReceived[#Site] ≠ #Op − 1 then
5  |  |  return false
   |  // Vérifie chaque dépendance d'une opération
6  |  for (nSite : cSite) ∈ dependancesOf(r) do
7  |  |  if SReceived[nSite] < cSite then
8  |  |  |  return false
9  |  return true
10 end
```

```
1  EXECUTE(r):                                  // Execute la requête r
2  begin
3  |  r = (op, #Site : #Op)
4  |  if op = Sync(dep) then
5  |  |  t = fix(t)    // Répare l'arbre si l'opération est de type
   |  |     sync
6  |  else
7  |  |  t = op(t)                             // Applique l'opération
8  end
```

FIG. 7.4 – *Algorithme* FCedit *avec réparation - partie 2*

Édition de Graphes et l'opération Move

8

Nous avons vu dans la section 4.6 qu'il est possible de modéliser un graphe fonctionnel dans le modèle des transformées opérationnelles. Dans ce chapitre nous allons construire une structure de donnée graphe utilisable dans le modèle CRDT qui permet de modéliser l'opération de déplacement sur les arbres. Etonnamment, personne n'a modélisé une telle structure dans les modèles OT ou CRDT.

8.1 Une structure de donnée graphe

Un multi-graphe orienté $G(V, E, f)$ est un ensemble fini de nœuds V et d'arêtes E, f est une fonction liant l'arête à un couple de nœud $f : E \to V \times V$. Il peut y avoir plusieurs arêtes reliant deux nœuds identiques. La structure de donnée graphe que nous choisissons est un multi-graphe orienté étiqueté (V, E, f, λ, μ) avec une fonction d'étiquetage des nœuds $\lambda : V \to \mathcal{L}_{CRDT} \times ID$ telle que $\lambda(v) = (l, id)$,et $l = (id', v, lbl)$ et une fonction d'étiquetage des arêtes $\mu : E \to ID$. Le triplet (id', v, lbl) est la structure d'étiquette vue dans la section 5.6, où lbl est le label du nœud, v la version du label et id' permet de départager deux version identiques.

$VID = \mathbb{N} \times ID$ est identique à celui défini au chapitre 6 sur XML. Les ordres utilisés sont donnés par :

$$id_1 = (nsite_1; nbop_1) <_{id} id_2 = (nsite_2; nbop_2) \Leftrightarrow \begin{cases} nsite_1 < nsite_2 \text{ ou} \\ (nsite_1 = nsite_2 \text{ et } nbop_1 < nbop_2) \end{cases}$$

$$Vid_1 = (v_1; id_1) <_{vid} Vid_2 = (v_2; id_2) \Leftrightarrow \begin{cases} v_1 < v_2 \text{ ou} \\ (v_1 = v_2 \text{ et } id_1 <_{id} id_2) \end{cases}$$

8.1.1 Opérations

Les opérations d'édition correspondantes sur cette structure de données sont les suivantes.

Ajouter un nœud $AddNode(id)$ ajoute un nœud identifié par id et étiqueté par $NoValue$ avec l'identifiant de l'opération et la version 1.

$$\begin{aligned} AddNode(id)(V,E,f,\lambda,\mu) &= (V \cup \{v_{id}\}, E, f, \lambda', \mu) \\ &\quad \text{avec } v_{id} \text{ le nouveau nœud,} \\ &\quad \lambda'(v_{id}) = ((id, 0, NoValue), id) \\ &\quad \lambda'(v) = \lambda(v) \end{aligned}$$

Supprimer un nœud $DelNode(id)$ supprime un nœud identifié par id, ainsi que toutes les arêtes le reliant. Nous supprimons aussi les informations le concernant.

$DelNode(id)(V,E,f,\lambda) = (V', E/E', f', \lambda, \mu')$
Avec $V' = V/\{v|\lambda(v) = (l,id)\}$ (suppression du nœud)
et $E' = \{e \in E | f(e) = (v', v''), \lambda(v') = (l,id) \vee \lambda(v'') = (l,id)\}$

$$\lambda'(v^\bullet) = \begin{cases} \epsilon \text{ pour } v^\bullet = v \\ \lambda(v^\bullet) \text{ sinon} \end{cases}$$

$$\mu'(e) = \begin{cases} \epsilon \text{ pour } e \in E' \\ \mu(e) \text{ sinon} \end{cases}$$

$$f'(e) = \begin{cases} \epsilon \text{ pour } e \in E' \\ f(e) \text{ sinon} \end{cases}$$

(suppression des arêtes reliant le nœud).

Renommer un nœud $ChLbl(id, id'_{op}, ver', l')$ change l'étiquette du nœud id avec la version v l'identifiant de l'opération est id_{op}. Si le nœud n'existe pas [1] alors on ne fait rien.

Si le nœud existe, la définition formelle devient :

$ChLbl(id, id_{op}, v, l)(V,E,f,\lambda,\mu) = (V,E,f,\lambda',\mu)$

Soit $\lambda(v_{id}) = ((id_{op}, ver, l), id)$

$$\text{Avec } \lambda'(v_{id}) = \begin{cases} ((id'_{op}, ver', l'), id) \text{ Si } (ver', id'_{op}) >_{vid} (ver, id_{op}) \\ \lambda(v_{id}) \text{ sinon} \end{cases}$$

$\lambda'(v) = \lambda(v)$

Ajouter une arête $AddEdge(id, id_{dest}, id_{op})$ ajoute une arête liant le nœud identifié par id au nœud identifié par id_{dest}, Cette arête sera identifiée par id_{op}. L'identification de l'arête nous permet de lever l'ambiguité sur l'arête à supprimer et de garantir la convergence.

[1] par exemple, si on a eu une suppression et ré-étiquetage en concurrence

$$
\begin{aligned}
&AddEdge(id, id_{dest}, id_{op})(V, E, f, \lambda, \mu) \\
&= \begin{cases}
(V, E \cup \{e'\}, f', \lambda, \mu') \\
\qquad \text{Avec } \lambda(v) = (..., id) \text{ et } \lambda(v') = (..., id_{dest})\} \\
\qquad f'(e) = \begin{cases} (v, v') \text{ si } e = e' \\ f(e) \text{ sinon} \end{cases} \\
\qquad \mu'(e) = \begin{cases} id_{op} \text{ si } e = e' \\ \mu(e) \text{ sinon} \end{cases} \\
\qquad \text{on étiquette la nouvelle arête créée avec } id_{op}. \\
\quad \text{Si } v \text{ et } v' \text{ sont dans } V \\
\\
(V, E, f, \lambda, \mu) \text{ Sinon (un des nœuds a été supprimé -on ne fait rien-)}
\end{cases}
\end{aligned}
$$

Supprimer une arête $DelEdge(id)$ Supprime une arête identifiée par id.

$$
\begin{aligned}
DelEdge(id)(V, E, f, \lambda, \mu) \quad = \quad & (V, E/\{e|\mu(e) = id\}, f', \lambda, \mu') \\
& \text{avec } \mu'(e) = \begin{cases} \epsilon \text{ si } \mu(e) = id \\ \mu(e) \text{ sinon} \end{cases} \\
& f'(e) = \begin{cases} \epsilon \text{ si } \mu(e) = id \\ f(e) \text{ sinon} \end{cases}
\end{aligned}
$$

8.1.2 Dépendance

Nous pouvons définir la relation de dépendance entre ces opérations.
- $AddNode(id) \succ_s DelNode(id)$ la suppression d'un nœud nécessite sa création.
- $AddNode(id) \succ_s AddEdge(id, id_{dest}, id_{op})$ Ajouter une arête nécessite la création du nœud d'origine.
- $AddNode(id_{dest}) \succ_s AddEdge(id, id_{dest}, id_{op})$ Ainsi que du nœud de destination.
- $AddEdge(id, id_{dest}, id_{op}) \succ_s DelEdge(id)$ La suppression d'une arête est après sa création.

Nous définissons la fonction $dependancesOf$ comme suit :

$$
dependancesOf(Op) = \begin{cases}
\emptyset & \text{pour} \quad op = AddNode(id) \\
\{id\} & \text{pour} \quad op = DelNode(id) \\
\{id\} & \text{pour} \quad op = ChLbl(id, id_{op}, v, l) \\
\{id, id_{dest}\} & \text{pour} \quad op = AddEdge(id, id_{dest}, id_{op}) \\
\{id\} & \text{pour} \quad op = DelEdge(id)
\end{cases}
$$

8.1.3 Variantes

Cette structure de données nous permet de modéliser différents types de graphe autre que les multi-graphes orientés étiquetés par les nœuds. En fusionnant les arêtes, il est possible de faire un graphe "classique". L'éditeur supprimera toutes les occurrences des arêtes lors de sa suppression. Si l'éditeur n'autorise la création que d'une arête entre deux nœuds, le nombre d'arêtes doubles devrait être limité sauf dans le pire des cas où il

y aurait autant d'arêtes que de sites (ce qui n'arrive que si chaque site la crée concurrement).

Il est possible aussi de faire un étiquetage sur les arêtes en modifiant légèrement la fonction d'étiquetage des arêtes μ. En reprenant la structure d'étiquette et en créant les opérations nécessaires. Nous ne l'avons pas fait ici pour éviter de compliquer inutilement la modélisation et la preuve.

Evidemment, avec la méthode vue dans le chapitre 6, il est possible d'ordonner les arêtes issues d'un même noeud.

Dans la section 8.2, nous allons explorer la variante des graphes fonctionnels afin de réaliser l'opération *Move* sur les arbres dans le modèle $CRDT$. Nous verrons qu'il n'est pas nécessaire de fusionner les arêtes, car on les crée uniquement lors de la création d'un nœud. Du coup nous n'avons pas plus d'une arête sortante par nœud : la structure du graphe fonctionnel est stable par les opérations d'édition.

***Remarque* 8.1 —** La structure de donnée ci-dessus peut aussi être utilisée avec l'approche des transformées opérationnelles. Au vu de l'intérêt que nous portons à l'approche CRDT, nous n'allons pas le faire dans cette thèse.

8.1.4 Preuve de correction

Théorème 8.1 *$Op_{graph} = \{AddNode, DelNode, ChLbl, AddEdge, DelEdge\}$ est $\succ_s$-indépendant.*

Démonstration. Nous allons faire une preuve en analysant chaque cas. Pour g un multi-graphe défini plus haut, $\forall op_1, op_2 \in Op_{graph}, [op_1, op_2](g)^{(1)} = [op_2, op_1](g)^{(2)}$.

1. $op_1 = AddNode(id)$:
   (a) $op_2 = AddNode(id')$: par définition la fonction *add* ajoute un nœud à un ensemble de nœuds. CQFD.
   (b) $op_2 = DelNode(id')$: Il y a deux cas
      i. si $id = id'$: les deux opérations sont dépendantes.
      ii. Sinon les deux éléments sont différent toujours dans un ensemble. De plus l'opération op_1 ne modifie pas l'ensemble d'arêtes. Toutes deux ajoutent des étiquettes à des entrées différentes de λ. Donc on a bien le résultat escompté.
   (c) $op_2 = ChLbl(id', id'_{op}, v', l')$: Les deux opérations modifient des éléments différents et l'ordre n'intervient pas.
   (d) $op_2 = AddEdge(id', id'_{dest}, id'_{op})$: Si $id' = id$ ou $id = id'_{dest}$ les deux opérations on dépendantes. Sinon op_1 ajoute un nœud qui n'est pas utilisé dans op_2. Donc $^{(1)} =^{(2)}$.
   (e) $op_2 = DelEdge(id)$: Les deux opérations modifient des éléments différents.
2. $op_1 = DelNode(id)$

(a) $op_2 = AddNode(id')$: même cas que 1b.

(b) $op_2 = DelNode(id')$: Il y a deux cas de figure :

i. $id = id'$: Le nœud est supprimé deux fois. Comme les identifiants sont utilisés qu'une seule fois, la deuxième opération n'aura pas d'effet, quel que soit l'ordre d'exécution. Idem pour l'ensemble des arêtes enlevées.

ii. $id \neq id'$: les deux opérations enlèvent deux nœuds différents dans l'ensemble des nœuds et toutes les arêtes les reliants.

(c) $op_2 = ChLbl(id', id'_{op}, v', l')$: Si $id = id'$ dans le premier cas le nœud est renommé puis supprimé. Dans le deuxième le réétiquetage n'aura pas d'effet. Sinon les deux opérations modifient des entrés différente de λ.

(d) $op_2 = AddEdge(id', id'_{dest}, id'_{op})$: Il y a deux cas de figures.

i. $id = id'$ ou $id = id_{dest}$: Dans le cas $^{(1)}$ l'arête n'est pas créée, car l'un des deux nœuds a été supprimé. Dans le cas $^{(2)}$ l'arête est créée puis supprimé par le op_1, car l'opération *DelNode* supprime aussi les arêtes reliant le nœud.

ii. *Sinon* : *AddEdge* ajoute une arête hors de l'effet de *DelNode*. Le graphe sera donc le même, quel que soit l'ordre d'exécution.

(e) $op_2 = DelEdge(id')$: Il y a deux cas possible.

i. si $\mu((v', v'')) = id'$ et $\lambda(v''') = (..., id)$ et que $v' = v'''$ ou $v'' = v'''$ alors dans le cas $^{(1)}$ l'arête sera supprimée par l'opération op_1 et op_2 n'aura pas d'effet. Dans le cas $^{(2)}$, cette arête sera supprimée par l'opération op_2 et op_1 supprimera les autres.

Le résultat sera le même.

ii. *Sinon* : les deux opérations n'ont pas d'effets l'une envers l'autre.

3. $op_1 = ChLbl(id, id_{op}, v, l)$

(a) $op_2 = AddNode(id')$ est traité en 1c

(b) $op_2 = DelNode(id')$ est traité en 2c

(c) $op_2 = ChLbl(id', id'_{op}, v', l')$ Nous avons plusieurs cas :

i. Si $id = id'$ Les deux ChLbl concernent le même nœud.

- Si $(v', id') >_{vid} (v, id)$: Alors dans le cas $^{(1)}$ op_1 va modifier l'étiquette en l puis op_2 en l'. Dans l'autre cas $^{(2)}$: op_2 modifiera l'arête en l' puis d'après la définition, op_2 n'aura pas d'effet. Le résultat sera donc le même.
- Si $(v', id') <_{vid} (v, id)$: Même raisonnement que le précédent en inversant les termes.

ii. *Sinon* Les deux opérations changent le nom de deux nœuds différent. L'ordre n'a pas d'importance.

(d) $op_2 = AddEdge(id', id'_{dest}, id'_{op})$: Les deux opérations ne changent pas la même composante de la structure de données.

(e) $op_2 = DelEdge(id')$: Idem.

4. $op_1 = AddEdge(id, id_{dest}, id_{op})$:
   (a) $op_2 = AddNode(id')$: c'est le cas 1d.
   (b) $op_2 = DelNode(id')$: c'est le cas 2d
   (c) $op_2 = ChLbl(id', id'_{op}, v', l')$: c'est le cas 3d
   (d) $op_2 = AddEdge(id', id'_{dest}, id'_{op})$: Les deux ajoutent deux arêtes distinct dans un ensemble. Les entrées ajoutées dans les fonctions μ et f sont distincts.
   (e) $op_2 = DelEdge(id')$: Si $id = id'$ Les deux opérations sont dépendantes. Sinon, dans les deux cas, op_1 ajoute une arrête et op_2 en supprime une autre. Ce qui est indépendant de l'ordre d'exécution.

   ***Remarque* 8.2 —** L'identification des arêtes nous permet de rendre ce cas assez simple. Sinon nous aurions dû compter le nombre d'opérations d'ajout et de suppression.

5. $op_1 = DelEdge(id)$
   (a) $op_2 = AddNode(id')$: c'est le cas 1e.
   (b) $op_2 = DelNode(id')$: c'est le cas 2e
   (c) $op_2 = ChLbl(id', id'_{op}, v', l')$: c'est le cas 3e
   (d) $op_2 = AddEdge(id', id'_{dest}, id'_{op})$: c'est le cas 4e
   (e) $op_2 = DelEdge(id')$: Si $id = id'$ les deux opérations auront le même effet sur le graphe et la seconde dans l'ordre d'exécution ne fera rien, car l'arête sera déjà supprimée par l'autre, car les arêtes sont identifiées de manière unique.

□

Le résultat permet donc de réaliser de l'édition collaborative sur des graphes avec l'algorithme *FCedit*.

Théorème 8.2 *L'algorithme* FCedit *est un algorithme d'édition collaborative sur les graphes étiquetés.*

8.2 Édition collaborative d'arbre avec l'opération Move

Nous allons utiliser la structure de graphe vue dans la section 4.6.3 en reprenant le méthode utilisé pour le *ChLbl* dans la section 5.6. C'est à dire qu'on ajoute l'identifiant de l'opération et un numéro de version à l'étiquetage des arêtes afin de pouvoir instaurer des priorités dans le déplacement d'un sous arbre. Cette structure va permettre d'une part de modéliser les

arbres (par un graphe fonctionnel) et d'autre part de définir l'opération Move.

Plus formellement nous avons la structure de graphe suivante : (V, E, λ, μ) où V est un ensemble de nœud, E est un ensemble d'arête reliant deux nœuds, λ est une fonction de d'étiquetage de nœud et μ une fonction d'étiquetage d'arêtes. Ces deux dernières fonctions sont définit comme suit :$\lambda : V \rightarrow ID \times \mathcal{L}_{CRDT}$ telle que $\lambda(v) = (l, id)$,et $l = (id', v, lbl)$. et $\mu : E \rightarrow VID$ telle que $\mu(e) = (v, id)$.

8.2.1 Opérations

Voici les opérations sur cette structure :

Ajouter un nœud $Add(id, id_p)$ ajoute un nœud étiqueté par $((id, 0, NoValue), id)$ et s'il existe un nœud étiqueté par id_p, l'opération ajoute une arête le reliant à son père. Si le père n'existe plus, le nœud est quand même créé, mais pas l'arête. Plus formellement :

$$
\begin{aligned}
Add(id, id_p)(V, E, \lambda, \mu) \quad = \quad & (V \cup \{v_{id}\}, E', \lambda', \mu') \\
& \text{avec } v_{id} \text{ le nouveau nœud,} \\
& v_{id_p} \text{ est tel que } \lambda(v_{id_p}) = (l, id_p) \\
& E' = \begin{cases} E \cup (v_{id_p}, v_{id}) \text{ si } v_{id_p} \in V \text{ et } \lambda(v_{id_p}) = (l, id_p) \\ E \text{ sinon} \end{cases} \\
& \lambda'(v_{id}) = ((id, 0, NoValue), id) \\
& \lambda'(v) = \lambda(v) \\
& \mu((v_{id_p}, v_{id})) = (0, id)
\end{aligned}
$$

Supprimer un nœud $Del(id)$ supprime un nœud du graphe et les arêtes le contenant comme extrémité. Le sous arbre n'est pas supprimé, mais il n'est plus connexe à la racine. Ce qui donne formellement :

$$
\begin{aligned}
& Del(id)(V, E, \lambda) = (V', E', \lambda, \mu) \\
& \quad \text{Avec } V' = V / \{v | \lambda(v) = (l, id)\} \text{ (suppression du nœud).} \\
& \quad \text{et } E / \{(v', v'') | \lambda(v') = (l, id) \vee \lambda(v'') = (l, id)\} \\
& \quad \text{(suppression des arêtes contenant le noeud).}
\end{aligned}
$$

Changer le label $ChLbl(id, id'_{op}, v', l')$ modifie la fonction d'étiquetage pour changer la valeur.

$$
\begin{aligned}
& ChLbl(id, id'_{op}, v', l')(V, E, \lambda) = (V, E, \lambda', \mu) \\
& \text{Soit } \lambda(v_{id}) = ((id_{op}, ver, l), id) \\
& \text{Avec } \lambda'(v_{id}) = \begin{cases} ((id'_{op}, ver', l'), id) \text{ Si } (ver', id'_{op}) >_{vid} (ver, id_{op}) \\ \lambda(v_{id}) \text{ sinon} \end{cases} \\
& \lambda'(v) = \lambda(v)
\end{aligned}
$$

Déplacer un nœud $Mv(id, id_p, id_{op}, ver)$ supprime l'arête entrante du nœud id et la remplace par l'arête reliant entre id et id' si id' existe encore. Si le nouveau nœud père n'existe pas c'est qu'il a été supprimé, nous supprimons les arêtes entrantes, mais nous ne créons pas de nouvelles arêtes.

$$
\begin{aligned}
&Mv(id, id_p, id_{op}, ver)(V, E, \lambda, \mu) \\
&= \begin{cases}
(V, E'', \lambda, \mu') \\
\quad \text{soit } \lambda(v_{id}) = (l, id), \mu((v, v_{id})) = vid \\
\quad \text{avec } E' = E / \{(v, v_{id}) | v \in V\} \\
\quad E'' = \begin{cases} E' \cup (v_{id_p}, v_{id}) \text{ si } v_{id_p} \in V \text{ et } \lambda(v_{id_p}) = (l, id_p) \\ E' \text{ sinon} \end{cases} \\
\quad \mu'((v_{id_p}, v_{id})) = (ver, id_{op}) \\
\quad \mu'(e) = \mu(e) \\
\quad \text{si } \exists v_{id} \in V \text{ et si } \nexists (v, v_{id}) \in E \text{ ou si } (ver, id_{op}) <_{vid} vid \\
\\
(V, E, \lambda, \mu) \text{ sinon}
\end{cases}
\end{aligned}
$$

***Remarque* 8.3 —** Dans l'implémentation, les arêtes et les nœuds sont généralement des objets ou enregistrement. Il est évident qu'il faudra aussi y mettre les informations d'étiquetage. Ainsi lors de la suppression de l'objet ou de l'enregistrement l'information contenue dans λ ou μ seront aussi supprimées.

8.3 Propriétés de stabilité et de convergence

La proposition suivante montre la stabilité de la structure de données par les opérations d'édition.

Proposition 8.1 *Soit $G = (V, E, \lambda, \mu)$ un graphe fonctionnel. L'application d'une opération $op \in \{Add, Del, ChLbl, Mv\}$ à G retourne un graphe fonctionnel.*

Démonstration. Nous allons étudier chaque opération :

- $Add(id, id_p)(V, E, \lambda, \mu)$ crée un nœud et ajoute une seule arête entrante sur ce nœud avec un nouvel identifiant. Le graphe reste fonctionnel.
- $Del(id)(V, E, \lambda, \mu)$ supprime des arêtes donc n'augmente pas le degré sortant des nœuds du graphe.
- $ChLbl(id, id'_{op}, v', l')(V, E, \lambda, \mu)$ ne change que l'étiquetage et ne modifie pas les arêtes.
- $Mv(id, id_p, id_{op}, v)(V, E, \lambda, \mu)$. En reprenant les notations de la définition, l'arête $e_{id} = (v, v')$ est supprimée, et une arête (w', v') est ajoutée. Le degré entrant du nœud v' reste donc 1, celui des autres nœuds n'est pas modifié.

□

8.3.1 La relation de dépendance

Définissons la relation de dépendance sémantique.

$$\begin{array}{rcl} Add(id_p, id') & \succ_{sMv} & Add(id, id_p) \\ Add(id, id_p) & \succ_{sMv} & Del(id) \succ_s Add(id_p{}', id) \\ Add(id, id_p) & \succ_{sMv} & ChLbl(id, id_{op}, ver, lbl) \\ Add(id, id_p{}') & \succ_{sMv} & Mv(id, id_p, id_{op}, v) \\ Add(id_p, id_p{}'') & \succ_{sMv} & Mv(id, id_p, id_{op}, v) \end{array}$$

La fonction $dependancesOf$ devient donc :

$$dependancesOf(op) = \begin{cases} \{id_p\} & \text{pour} \quad op = add(id, id_p) \\ \{id\} & \text{pour} \quad op = Del(id) \\ \{id\} & \text{pour} \quad op = ChLbl(id, id_{op}, v, l)) \\ \{id, id_p\} & \text{pour} \quad op = Mv(id, id_p, id_{op}, v) \end{cases}$$

***Remarque* 8.4 —** La proposition 4.6 établit que "La composante connexe d'un graphe fonctionnel contenant la racine est un arbre." Nous pouvons à partir d'une racine définie à l'avance représenter un arbre.

Dans ce modèle là, il est aussi possible d'avoir un arbre ordonné en ajoutant les informations nécessaires dans les nœuds(voir le début du chapitre 6).

8.3.2 Résultat de convergence

Proposition 8.2 *$(Op_{Mv}, \succ_{sMv})$ est indépendant.*

Démonstration. Montrons que $(Op_{Mv}, \succ_{sMv})$ est indépendant. Pour cela nous allons montrer par étude de cas que $\forall op_1, op_2 \in Op_{mv}$,
$op_1 \not\succ_{sMv} op_2 \wedge op_2 \not\succ_{sMv} op_1 \Rightarrow [op_1, op_2](s)^{(1)} = [op_2, op_1](s)^{(2)}$

1. $op_1 = Add(id_p, id)$
   (a) $op_2 = Add(id', id_p{}')$: Il y a deux possibilités :
      i. Si $id = id_p{}'$ ou $id' = id_p$, les deux opérations sont dépendantes.
      ii. Sinon l'une ajoute son arête et son nœud dans un ensemble et l'autre fait de même. Toutes deux ajoutes des étiquettes à des entrées différentes de μ et λ. L'ordre d'exécution ne change pas le résultat.
   (b) $op_2 = Del(id')$
   Nous savons que $id \neq id'$ car elle sont indépendantes. Il nous reste deux possibilitées.
   – si $id = id_p$ Dans $^{(1)}$ l'arête (id_p, id) sera rajoutée puis supprimée par le Del. Dans $^{(2)}$ l'arête (id_p, id) ne sera pas ajoutée car le nœud père n'existe plus par définition de l'opération Add. L'ensemble des nœuds est le même et les fonctions d'étiquetage aussi. Donc $^{(1)} =^{(2)}$

- sinon, op_1 ajoute et op_2 supprime des éléments distincts. Les deux opérations modifient des entrées distincts dans les fonction d'étiquetage. et donc $^{(1)} =^{(2)}$.

(c) $op_2 = ChLbl(id', id'_{op}, ver', lbl')$
Par définition $id' \neq id$, sinon, elle seraient dépendantes. L'opération op_1 et op_2 ne modifient pas les mêmes entrées de l'opération λ et la cette dernière opération ne modifie que cette fonction. donc $^{(1)} =^{(2)}$.

(d) $op_2 = Mv(id', {id_p}', id'_{op}, v')$ Par définition $id \neq {id_p}'$ et $id \neq id'$
Nous savons que l'opération Mv ne modifie uniquement les arêtes et la fonction μ. L'opération Op_1 va ajouter une arête qui ne sera pas supprimé par Mv et les deux opérations modifieront des entrées différentes de la fonction μ.

2. $op_1 = Del(id'))$

(a) $op_2 = Add(id', {id_p}')$ Même cas que 1b

(b) $op_2 = Del(id')$ Nous avons deux cas :

i. Si $id = id'$, op_1 va supprimer le nœud étiqueté id et les arêtes entrante ou sortante liant ce nœud. La deuxième opération va rien faire car tout sera déjà supprimé et vice versa.

ii. Sinon, les deux nœuds seront supprimés de E, quel que soit l'ordre d'exécution. S'il existe des arêtes communes au deux, elles seront supprimées aussi. Et ce, quel que soit l'ordre d'exécution.

(c) $op_2 = ChLbl(id', id'_{op}, ver', lbl')$:
Les deux opérations ne changent pas la même composante de la structure de données. Donc quel que soit l'ordre d'exécution, le résultat sera le même.

(d) $op_2 = Mv(id', {id_p}', id'_{op}, v')$: La fonction Mv ne modifie que l'ensemble d'arête et la fonction μ. Tant dit que l'opération Del Ne touche qu'à l'ensemble des arêtes et l'ensemble des nœuds. Nous avons plusieurs cas :

i. $id = id'$: Dans le premier cas, par définition, le déplacement n'aura pas lieu car le nœud à déplacer n'existera plus. Dans le deuxième cas, l'arête originale pointant vers le père sera déplacée ou pas (selon l'existence du nouveau père) et sera supprimée par l'opération Del. Le résultat sera donc le même.

ii. $id = {id_p}'$: Dans le premier cas, le nœud id_p n'existera plus donc l'opération Mv supprimera uniquement l'arête entrante du nœud à déplacer. Dans le second cas, l'opération Mv va déplacer l'arête pointant vers le nœud étiqueté par id_p et le supprimera ensuite ainsi que la nouvelle arête. Nous avons donc le même graphe à la fin.

iii. sinon : L'opération Mv va faire son déplacement sans interférence de la part de Del. Car les deux ensembles d'arêtes modifiées et supprimées sont distinct.

3. $op_1 = ChLbl(id', id'_{op}, ver', lbl')$
   (a) $op_2 = Add(id', id_p{}')$ C'est le même cas que 1c
   (b) $op_2 = Del(id'))$ C'est le même cas que 2c
   (c) $op_2 = ChLbl(id', id'_{op}, ver', lbl')$Nous avons plusieurs cas :
      i. Si $id = id'$ Les deux ChLbl essayent de changer le même nœud.
         – Si $(ver', id') >_{vid} (ver, id)$: Alors dans le cas $^{(1)}$ op_1 va modifier l'étiquette en l puis op_2 en l'. Dans l'autre cas $^{(2)}$: op_2 modifiera l'arête en l' puis d'après la définition, op_2 n'aura pas d'effet. Le résultat sera donc le même.
         – Si $(v', id') <_{vid} (v, id)$: Même raisonnement que le précédent en inversant les termes.
      ii. Si $id \neq id'$ Les deux entrées de la fonction λ modifiés sont distinct, l'ordre n'importe pas.
   (d) $op_2 = Mv(id', id_p{}', id'_{op}, v')$ Trivial car Mv ne modifie que l'ensemble d'arête et la fonction μ, tant dit que $ChLbl$ que la fonction λ.

4. $op_1 = Mv(id, id_p, id_{op}, v)$
   (a) $op_2 = Add(id', id_p{}')$ C'est le même cas que 1d
   (b) $op_2 = Del(id'))$ C'est le même cas que 2d
   (c) $op_2 = ChLbl(id', id'_{op}, ver', lbl')$ C'est le même cas que 3d
   (d) $op_2 = Mv(id', id_p{}', id'_{op}, v')$
      i. Si $id \neq id'$ Les deux éléments modifiés sont distinct, l'ordre n'importe pas.
      ii. Si $id = id'$: Nous avons deux cas car les VID sont uniques :
         – Si $(v, id_{op}) <_{vid} (v', id'_{op})$ Dans le premier cas, le déplacement aura lieu et l'arête sera remplacée par celle donnant comme père au nœud étiqueté par id le nœud étiqueté par id_p. Dans un second temps elle sera remplacée par une arête donnant $id_p{}'$ comme père à id.
           Dans le second cas, le premier déplacement donne comme père au nœud étiqueté par id le nœud étiqueté par $id_p{}'$. Dans un second temps, par définition l'opération Mv ne fera rien. Car l'arête existe elle est prioritaire. Nous savons que les deux opérations changent différentes entrées dans la fonction μ. Nous avons donc le même résultat à la fin.
         – Sinon : Les deux opérations suppriment et créent différentes arête. Elles n'ont pas d'effet l'une sur l'autre. Nous savons que les deux opérations changent différentes entrées dans la fonction μ. L'ordre d'exécution n'a pas d'impact sur le résultat final.

□

Un graphe fonctionnel permettant de modéliser une structure de donnée arborescente avec étiquetage sur les nœuds, le résultat précédent permet d'énoncer le résultat final :

Théorème 8.3 *L'algorithme* FCedit*est un algorithme d'édition collaborative convergent pour une structure de donnée arborescente avec les opérations d'édition* ajout, suppression, réétiquetage de nœud *et* déplacement de sous-arbres.

8.4 Conclusion

Dans ce chapitre, nous avons montré que nous pouvons appliquer notre algorithme à des structures de graphes. Il est étonnant que personne n'ait exploré ces structures de données pour l'édition collaborative. Pourtant, elles sont utiles par exemple pour l'édition cartographique de manière collaborative.

Le déplacement d'une partie d'un document est une opération classique et utile pour l'édition de document. Il est donc pertinent de vouloir l'intégrer dans un éditeur collaboratif. Cela implique qu'un participant peut modifier une partie de document (paragraphe, section, ...) alors même qu'un autre participant le déplace. Notre structure de données nous permet de faire cette opération dans le modèle CRDT, ce qui n'avait pas été réalisé jusqu'à présent.

Ces deux approches seront bientôt implémentées dans le prototype.

Prototype

9

Dans ce chapitre, nous décrivons un prototype limité d'éditeur collaboratif qui a permis de faire une approche expérimentale du problème en implémentant plusieurs des méthodes vues dans cette thèse.

9.1 Descriptif du prototype

Nous avons réalisé un prototype en java 1.6 qui nous a permis d'expérimenter plusieurs des algorithmes et méthodes proposées dans un cadre de simulation d'édition collaborative. Ce prototype est organisé en différentes couches afin de rendre l'implémentation des différentes modélisations aisée et modulable. Dans ce prototype nous avons implémenté la structure de données du chapitre 5 qui définit les arbres ordonnés ou non-ordonnés avec les opérations d'édition $Add, Del_1, ChLbl$ et de plus une opération pour modifier l'ordre des arêtes issues d'un même père dans le cas des arbres ordonnés.

L'implémentation est formée de plusieurs couches reliées entre elles via des interfaces java.

9.1.1 Couche *structure de données*

Les structures de données arborescentes sont implémentées à l'aide d'objets appelé *Edges* possédant des attributs *id*, *label* (lui même formé d'un *id* et d'un numéro de *version*), et *position* (qui est un objet similaire à label ayant de plus un fonction pour comparer les positions et une fonction pour en générer une). L'objet *Edges* possède aussi une liste de fils (LinkedList) et la référence du père (lui-même un objet du type *Edges*). L'accès à une arête se fait directement et de manière efficace avec une table de hachage (*HashMap*).

9.1.2 Couche *opérations*

Chaque opération est un objet qui comporte une méthode *Do* (défini par une interface) qui permet de l'appliquer sur un arbre. La génération d'une opération par un site est faite via l'interface utilisateur, ou le scénario de tests.

9.1.3 Couche *algorithme*

L'algorithme gère uniquement des objets opérations.

Il génère un identifiant frais pour chaque opération et l'envoie à la couche réseau qui se charge de la propagation. Une opération venant du réseau est traitée selon l'algorithme et envoyée à l'objet collaboratif pour y être exécutée après traitement. Cette approche générique nous a permis d'implémenter aussi bien l'algorithme de Ressel que l'approche CRDT.

9.1.4 Interface

Pour l'instant l'interface est assez basique. Nous ne pouvons pour l'instant que dessiner des arbres comme définis dans le chapitre 5. Les différentes couleurs représentent les différents utilisateurs. L'exemple suivant donne une illustration d'un tel arbre, construit par quatre utilisateurs différents.

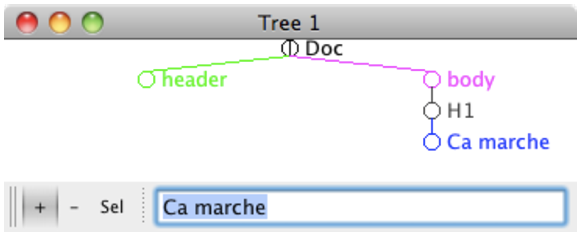


FIG. 9.1 – *Gui minimaliste*

9.1.5 Problématique réseau

Nous n'avons pas réalisé une vraie implémentation de la partie échange de messages. Cela pourrait se faire en utilisant un algorithme de propagation de type inondation. Chaque émetteur envoie ses messages à k sites qui eux-mêmes relaient ces messages vers k autres sites en mode connecté (tcp). L'utilisation d'accusé de réception permet de gérer les pertes de messages. De nombreux travaux à ce sujet ont été publiés et la thématique reste d'actualité, [RM ; SNG10]. Ces problèmes seront traités dans le cadre de l'amélioration du prototype.

Actuellement, le prototype contient un objet qui prend les messages délivrés par l'algorithme pour un site et les envoie aux autres sites en modifiant de façon aléatoire l'ordre de délivrance des messages. Cet ordre peut-être visualisé via une Message Sequence Chart [wikc] dessinée en temps réel. Selon le type de réseau, le coût du broadcast sera soit de $\mathcal{O}(1)$ (cas d'un réseau local), soit de $\mathcal{O}(n)$ (cas général).

9.2 Expérimentations

Nous avons utilisé le prototype pour tester certains de nos algorithmes.

9.2.1 Protocole d'expérimentation

Nous avons simulé la concurrence sur les messages de la manière suivante : Chaque site reçoit un certain nombre de messages (100 dans notre implémentation), puis effectue une permutation sur cette liste de messages. Cette liste simule l'ordre de réception des messages d'un vrai éditeur collaboratif. Le processus continue ensuite avec les 100 messages suivants etc...

Nous avons mesuré le temps global du processus d'édition en fonction de trois paramètres :

- le nombre d'utilisateurs
- la taille du document (fonction du nombre d'opérations d'édition).
- le nombre de modifications d'un document sans faire varier la taille du document en tirant de façon aléatoire des opérations *Add* et *Del*. La taille du document était de 1000 arêtes.

Il faut remarquer que le temps global mesuré n'est pas le temps nécessaire pour chaque utilisateur pour effectuer son processus d'édition (qui est moindre).

9.2.2 Résultats

L'approche transformée opérationnelle sur le modèle d'arbre du chapitre 4 (avec ajout de l'ordre entre arête) a été implémentée avec l'algorithme de Ressel. Nous avons utilisé une liste chainée des opérations avec leur vecteur d'état et l'algorithme utilisait ce vecteur pour calculer les transformations nécessaires. Les résultats obtenus pour un nombre de sites limité (20 sites) et un nombre d'opérations de l'ordre de 2000 montrent que la méthode est couteuse. Par exemple, au bout d'un certain temps une opération pouvait nécessiter de l'ordre d'une seconde pour être effectuée. Cela peut être du à une mauvaise optimisation de cet algorithme, mais cela reflète également le caractère quadratique de celui-ci. Ces premiers résultats ont justifié notre passage à l'approche CRDT, et nous n'avons pas effectué plus d 'expérimentation sur ce modèle.

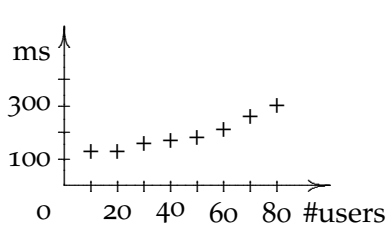
Les résultats sur l'approche CRDT sont donnés dans ce qui suit.

Le graphique 9.2.a montre qu'il y a une augmentation linéaire par rapport au nombre d'utilisateurs ce qui est normal étant donné que pour l'instant le "broadcast" envoie à tous les utilisateurs et que tous les utilisateurs sont sur le même nœud réseau.

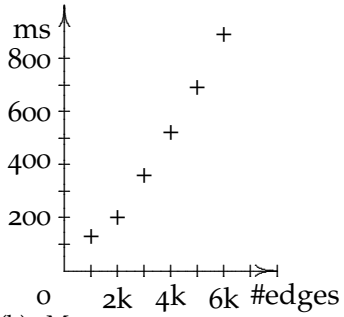
Le graphique 9.2.b montre bien la complexité quadratique de l'algorithme par rapport à la taille du document.

Le graphique 9.2.c montre l'indépendance vis-à-vis de l'historique. Ainsi faire 10000 opérations équilibrées entre l'ajout et la suppression laissant le document à 1000 arêtes n'a pas d'impact sur le temps de réponse.

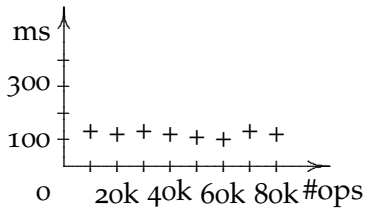
A titre de comparaison, une page web possède en moyenne actuellement 90 objets[wso].



(a) Moyenne en temps en fonction du nombre d'utilisateur pour 1000 opérations



(b) Moyenne temps pour remplir un document avec 20 utilisateur.



(c) Moyenne du temps mis pour exécuter 10 000 opérations en fonction du nombre totale d'action.

FIG. 9.2 – *Performances du Prototype*

9.3 Travaux futurs

Nous prévoyons d'implémenter la partie réseau de manière plus réaliste, et non plus de la simuler. Un autre travail sera de réaliser une interface ou d'en implémenter une via un greffon d'éditeur. Cela permettra de rendre l'éditeur utilisable dans les conditions réelles. La partie expérimentation sera approfondie afin d'étudier plus en détail l'approche CRDT en utilisation réelle sur des documents XML.

Conclusion 10

10.1 Bilan

Dans cette thèse, nous avons proposé différentes modélisations de structures de données pour les arbres, que nous avons adaptées pour les documents XML. La première proposition donne un modèle d'arbres qui permet d'obtenir des propriétés suffisantes ($TP1$ et $TP2$) pour la convergence dans le cadre des transformées opérationnelle. Nous proposons notamment une opération inédite qui est le déplacement de sous-arbres. Ensuite, nous avons proposé l'algorithme *FCedit*, un algorithme CRDT utilisant un nouveau type de dépendance décrite en fonction des opérations et de la structure de données. Cette dépendance permet de ne pas envoyer de messages contenant le vecteur d'état complet tout en ayant un ordre partiel permettant d'assurer une cohérence. Ce nouvel algorithme passe à l'échelle, supporte un changement de la topologie du réseau et est totalement symétrique (entre sites). Il nous a aussi permis de définir des opérations sur les arbres et des documents XML. Nous avons proposé une modélisation des multi-graphes et graphes. Nous proposons aussi faire le déplacement de sous-arbres dans le CRDT.

Les différentes modélisations et l'algorithme proposés dans cette thèse permettent de faire de l'édition collaborative en pair-à-pair sur différents documents écrits dans le format XML et d'explorer une généralisation originale grâce à l'édition de graphes. Cette édition se fait sans contraintes, sans mise à jour faite par l'utilisateur et les conflits sont réglés de façon automatique avec une convergence des copies.

Même si nous ne l'avons pas encore testé dans les conditions réelles, nous pensons qu'il est une solution pour répondre à une utilisation massive, car il répond à certaines propriétés des réseaux pair-à-pair.

10.2 Perspectives

A l'issue de cette thèse, plusieurs points s'avèrent intéressants à explorer et développer :

Optimisation des réparateurs d'arbre pour l'édition collaborative Nous avons vu que dans le chapitre 7 nous utilisons un réparateur arbre pour qu'il soit dans un schéma prédéfini. Il serait intéressant d'étudier plus

profondément ce genre de réparateur pour l'adapter au contexte de l'édition collaborative afin de pouvoir créer des interactions plus fines entre l'éditeur et le réparateur. Un autre but serait d'enrichir le schéma afin de proposer des scénarios de réparation, toujours dans le but de se rapprocher des intentions de l'auteur.

Sécurité du document Une autre piste est de proposer des Access Control List (ACL [wikb]) ou des Role-Based Access Control (RBAC [FK92]) pour affecter des droits de lecture et, ou d'écriture sur différentes parties. L'objectif est d'intégrer ces propriétés à l'algorithme d'édition et de démontrer que ces rôles ou droits restent stables durant l'édition.

On pourrait imaginer plusieurs groupes avec différents rôles qui travaillent sur différentes parties du document. Des parties pourraient être invisibles ou partiellement visibles pour les autres groupes.

Il serait intéressant de pouvoir changer dynamiquement les droits d'accès pendant l'édition et de définir un formellement une propriété de maintien de cohérence.

Prototype Notre prototype nous a servi à tester la faisabilité de l'approche, mais il reste très insuffisant et demande à être étendu pour devenir un vrai prototype d'éditeur collaboratif, qui pourrait être testé dans des conditions réelles d'utilisation. L'ajout de l'opération de déplacement de sous-arbres et du typage (avec la réparation) permettrait d'obtenir un logiciel comportant des caractéristiques inédites et utiles.

Annexes

A

Sommaire

A.1 Preuves du Chapitre 4

A.1.1 Preuve de la propriété TP1 pour le modèle Harmony

Nous prouvons que :
$\forall op_1, op_2 \in Op, t \in T[op_1; IT(op_2, op_1)](t) = [op_2; IT(op_1, op_2)](t)$

Nous allons énumérer tous les cas :

1. $op_1 = Add(p, n)$ et $op_2 = Add(p', n')$
   $[Add(p, n); IT(Add(p', n'), Add(p, n))] = [Add(p, n); Add(p', n')]$
   $[Add(p', n'); IT(Add(p, n), Add(p', n'))] = [Add(p', n'); Add(p, n)]$

   $$[Add(p, n); Add(p', n')](t) = [Add(p', n'); Add(p, n)](t)$$

   en notant $^{(1)}$ la première expression et $^{(2)}$ la seconde.

   (a) p n'est pas préfixe de p' et réciproquement, donc $p = \bar{p}.p_1$ et $p' = \bar{p}p'_1$ avec p_1 et p'_1 incomparables. Alors les sous-arbres $t_{|_p}$ et $t_{|_{p'}}$ sont distincts et $Add(p, n)$ modifie $t_{|_p}$ par ajout éventuel de $n(\{\})$ et symétriquement pour $Add(p', n')$. L'ordre d'ajout n'a pas d'importance et donc $^{(1)} =^{(2)}$

   (b) p est prefixe de p' avec $p' = p.p''$.

i. Si $p'' = n.\bar{p}$. Alors $Add(p,n)(t) = t$ et $^{(1)} = Add(p',n')(t)$. $^{(2)} = Add(p,n)(Add(p',n')(t))$. Le chemin $p.n$ existe dans $Add(p',n')$ et donc $Add(p,n)(Add(p',n')(t)) = Add(p',n')(t)$. Donc $^{(1)} =^{(2)}$

ii. $p'' = m.\bar{p}$. Si $t_{|_p} = \{n_1(t_1), \ldots, n_q(t_q), m(t')\}$ alors $Add(p,n)(t)$ ajoute un terme $n(\{\})$ à ce sous-arbre et $Add(p',n')$ modifie le terme m(t') en $m(Add(\bar{p},n')(t'))$. L'arbre résultant est le même quel que soit l'ordre d'exécution des deux opérations d'ajout, donc $^{(1)} =^{(2)}$.

2. $op_1 = Add(p,n)$ et $op_2 = Del(p',n')$
Montrons que

$$[Add(p,n); IT(Del(p',n'), Add(p,n))](t) = [Del(p',n'); IT(Add(p,n), Del(p',n'))](t)$$

en notant $^{(1)}$ la première expression et $^{(2)}$ la seconde.

- $p = p'$ et $n = n'$
  $^{(1)} = [Add(p,n), Del(p,n)](t)$ et $^{(2)} = [Del(p,n); Nop()](t)$
  - si $n \in Dom(t_{|_p})$ alors, dans $^{(1)}$,Dans $Add(p,n)(t)$ ne fait rien, puis l'arête $p.n$ et le sous-arbre sont supprimés. supprimé. Dans $^{(2)}$, $Nop()$ ne modifie pas t puis l'arête $p.n$ et le sous-arbre sont supprimés.
    Donc $^{(1)} =^{(2)}$
  - Si $n \notin Dom(t_{|_p})$ alors, dans $^{(1)}$, $Add(p,n)(t)$ crée un nœud qui sera supprimé par $Del(n,p)$. et $Del(n,p)$ ne fait rien dans $^{(2)}$
    Donc $^{(1)} =^{(2)}$
- $p'.n' \lhd p$
  $^{(1)} = [Add(p,n); Del(p',n')](t)$ et $^{(2)} = [Del(p',n'); Nop()](t)$
  Si p=p'.n'.p''

  Soit $t_{|_{p'}} = \{n_1(t_1), ..., n_q(t_q), n'(t)\}$ alors :

  Pour $^{(1)}$, après l'exécution de $Add(p,n)$ nous avons $Add(p,n)(t)_{|_p} = \{n_1(t_1), ..., n_q(t_q), n'(Add(p'',n)(t))\}$ et donc après l'exécution de $Del(p',n')$, le sous-arbre à la position p' est $\{n_1(t_1), \ldots, n_q(t_q)\}$

  Pour $^{(2)}$, le $Nop()$ ne modifie pas t, et après l'exécution de $Del(p',n')$, le sous-arbre à la position p' est $\{n_1(t_1), \ldots, n_q(t_q)\}$
  Donc $^{(1)} =^{(2)}$.
- sinon : même raisonnement que pour 1a.

3. Même cas que $op_1 = Del(p,n)$ et $op_2 = Add(p',n')$
4. $op_1 = Del(p,n)$ et $op_2 = Del(p',n')$

Montrons que

$$[Del(p,n); IT(Del(p',n'), Del(p,n))](t) = [Del(p',n'); IT(Del(p,n), Del(p',n'))](t)$$

en notant $^{(1)}$ la première expression et $^{(2)}$ la seconde.

- p.n=p′.n′ :
  $^{(1)} = [Del(p,n), Nop()](t)$ et $^{(2)} = [Del(p,n), Nop()](t)$
- $p.n \triangleleft p'$
  Nous avons p′=p.n.p″ ;
  Soit $t_{|_p} = \{n_1(t_1), .., n_q(t_q), n(t)\}$
  $^{(1)} = [Del(p,n); Nop()](t)$ et $^{(2)} = [Del(p',n'); Del(p,n)](t)$

  $^{(1)}{}_{|_p} = \{n_1(t_1), ..., n_q(t_q)\}$
  Pour $^{(2)}$, après l'exécution de $Del(p',n')$, nous avons : $Del(p,n)(t)_{|_p} = \{n_1(t_1), \ldots, n_q(t_q), n(Del(p'',n')(t))\}$
  alors $Del(p,n)(Del(p',n')(t))_{|_p} = \{n_1(t_1), ..., n_q(t_q)\}$
- idem pour $p'.n' \triangleleft p'$
- cas similaire au cas 1a.

5. Le cas op_1 ou $op_2 = Nop()$ est trivial. □

A.1.2 Preuve de la propriété TP2

Nous devons prouver que

$$IT(IT(op, op_1), IT(op_2, op_1) = IT(IT(op, op_2), IT(op_1, op_2))$$

La preuve se fait par énumération des cas possibles : En notons $^{(1)}$ la première expression et $^{(2)}$ la seconde.
- $op = Add(p,n)$, $op_1 = Add(p_1,n_1)$ et $op_2 = Add(p_2,n_2)$ alors $^{(1)} = Add(p,n)$ et $^{(2)} = Add(p,n)$
- $op = Add(p,n), op_1 = Add(p_1,n_1)$ et $op_2 = Del(p_2,n_2)$
  $IT(IT(Add(p,n), Add(p_1,n_1)), IT(Del(p_2,n_2), Add(p_1,n_1)))^{(1)}$

  $IT^{(b)}(IT(Add(p,n), Del(p_2,n_2), IT^{(a)}(Add(p_1,n_1), Del(p_2,n_2)))^{(2)}$

  $^{(1)} = IT(Add(p,n), Del(p_2,n_2))$
  $^{(2)} = IT^{(b)}(IT(Add(p,n), Del(p_2,n_2)), Add(X,n_1)) = IT(Add(p,n), Del(p_2,n_2))$
  ou $^{(2)} = IT^{(b)}(IT(Add(p,n), Del(p_2,n_2)), Nop()) =$
  $IT(Add(p,n), Del(p_2,n_2))$
  parce que $^{(a)}$ donne un Add() ou un Nop() le second argument de $^{(b)}$ est un Add ou un Nop.
- Idem pour $op = Add(p,n), op_1 = Del(p_1,n_1)$ et $op_2 = Add(p_2,n_2)$
- $op = Add(p,n), op_1 = Del(p_1,n_1)$ et $op_2 = Del(p_2,n_2)$
  $IT(IT(Add(p,n), Del(p_1,n_1)), IT(Del(p_2,n_2), Del(p_1,n_1)))^{(1)}$
  $IT(IT(Add(p,n), Del(p_2,n_2)), IT(Del(p_1,n_1), Del(p_2,n_2)))^{(2)}$
  - Si $p_1 = p_2$ et $n_1 = n_2$
    $^{(1)} = IT(IT(Add(p,n), Del(p_1,n_1)), Nop())$
    $^{(2)} = IT(IT(Add(p,n), Del(p_1,n_1)), Nop())$
  - Si $p_2.n_2 \triangleleft p_1$
    $^{(1)} = IT(IT(Add(p,n), Del(p_1,n_1)), Del(p_2,n_2)))$
    car $p_1.n_1 \neq p_2.n_2$
    $^{(2)} = IT(IT(Add(p,n), Del(p_2,n_2)), Nop())$
    - Si $p_2.n_2 \triangleleft p$
      $^{(1)} = IT(IT(Add(p,n), Del(p_1,n_1)), Del(p_2,n_2)))$
      $^{(2)} = Nop()$

- Si $p_1.n_1 \lhd p$
  alors $^{(1)} = Nop()$ et $^{(2)} = Nop()$
- sinon :
  $^{(1)} = IT(Add(p,n), Del(p_2,n_2)) = Nop()$
  $^{(2)} = Nop()$
- idem pour $p_1.n_1 \lhd p$
- sinon :
  $^{(1)} = IT(Add(p,n), Del(p_2,n_2)) = Add(p,n)$ car
  $p_2.n_2 \not\lhd p \wedge p_1.n_1 \not\lhd p$
  $^{(2)} = IT(IT(Add(p,n), Del(p_2,n_2)), Nop())) =^{(1)}$

- idem si $p_1.n_1 \lhd p_2$
- Sinon :
  $^{(1)} = IT(IT(Add(p,n), Del(p_1,n_1)), Del(p_2,n_2)))$
  $^{(2)} = IT(IT(Add(p,n), Del(p_2,n_2)), Del(p_1,n_1)))$

- si $p = p_1 \wedge n = n_1$
  $^{(1)} = IT(Nop(), Del(p_2,n_2))) = Nop()$
  $^{(2)} = IT(IT(Add(p_1,n_1), Del(p_2,n_2)), Del(p_1,n_1)))$
  Nous en déduisons :
  $^{(2)} = IT(Add(p_1,n_1), Del(p_1,n_1))) = Nop()$
- idem si $p = p_2 \wedge n = n_2$
- si $p_1.n_1 \lhd p$ $^{(1)} = IT(Nop(), Del(p_2,n_2)))$
  $^{(2)} = IT(IT(Add(p,n), Del(p_2,n_2)), Del(p_1,n_1)))$
  Nous savons que $p_2 \neq p \vee n_2 \neq n$ et $p_2.n_2 \not\lhd p$ car $p_2.n_2 \lhd p \wedge p_1.n_1 \lhd p \Rightarrow p_1.n_1 \lhd p_2.n_2 \vee p_2.n_2 \lhd p_1.n_1$
  $^{(2)} = IT(Add(p,n), Del(p_1,n_1))) = Nop() =^{(1)}$
- idem si $p_2.n_2 \lhd p$
- sinon :
  $^{(1)} = IT(Add(p,n))$ et $^{(2)} = IT(Add(p,n))$

- C'est trivial pour $op = Del(p,n), op_1 = Add(p_1,n_1)$ et $op_2 = Add(p_2,n_2)$
- Si $op = Del(p,n), op_1 = Del(p_1,n_1)$ et $op_2 = Add(p_2,n_2)$
  $^{(1)} = IT(IT(Del(p,n), Del(p_1,n_1)), IT(Add(p_2,n_2), Del(p_1,n_1))$
  $^{(1)} = IT(Del(p,n), Del(p_1,n_1))$ parce que le premier argument va être '*Del*' et le second sera '*Add*'.

  $^{(2)} = IT(IT(Del(p,n), Add(p_2,n_2)), IT(Del(p_1,n_1), Add(p_2,n_2))$
  $^{(2)} = IT(Del(p,n), Dell(p_1,n_1))$
- Même raisonnement pour $op = Del(p,n), op_1 = Add(p_1,n_1)$ et $op_2 = Del(p_2,n_2)$
- Si $op = Del(p,n), op_1 = Del(p_1,n_1)$ et $op_2 = Del(p_2,n_2)$ Il y a plusieurs cas :
  - Si n est le plus haut dans l'arbre. Alors $op = IT(IT(op,op_1), IT(op_2,op_1))$ et $op = IT(IT(op,op_2), IT(op_1,op_2))$.
  - Si n est entre n_1 et n_2 et n_1 plus haut que n_2. Alors $Nop = IT(Nop(), IT(op_2,op_1)) = IT(IT(op,op_1), IT(op_2,op_1))$ et $Nop() = IT(IT(op,op_1) = IT(IT(op,op_2), IT(op_1,op_2))$.
  - idem si n est entre n_1 et n_2 et n_2 plus haut que n_1
  - Si n est dans un sous-arbre différent de n_1 et n_2 Trivial.
  - Si n est dans un sous-arbre différent de n_1 et n_2 plus haut.

$Nop = IT(op, op_2) = IT(IT(op, op_1), IT(op_2, op_1))$ et $Nop = IT(Nop(), IT(op_1, op_2)) = IT(IT(op, op_2), IT(op_1, op_2))$.
  - Si n est dans un sous-arbre différent de n_1 et n_2 plus bas : $op = IT(IT(op, op_1), IT(op_2, op_1))$ et $op = IT(IT(op, op_2), IT(op_1, op_2))$.
  - Idem pour le cas symétriques.
- si $op = Nop()$. Evident.
- si $op = X(n, p), op_1 = Nop()$ et $op_2 = X'(n_2, p_2)$
  $^{(1)} = IT(IT(X(p, n), Nop()), IT(X'(p_2, n_2), Nop())$
  $= IT(X(p, n), X'(p_2, n_2))$
  $^{(2)} = IT(IT(X(p, n), X'(p_2, n_2)), IT(Nop(), X'(p_2, n_2))$
  $= IT(X(p, n), X'(p_2, n_2))$
- idem pour $op = X(n, p), op_1 = X(p_1, n_1)$ et $op_2 = Nop()$
- si $op = X(n, p), op_1 = Nop()$ et $op_2 = Nop()$. Evident.

□

A.1.3 Fichiers Vote

TP1 et TP2 pour les opérations $Add, Del_1, ChLbl$

```
type node,lbl(novalue),nat;

observator
%test l'existance d'un noeud
 bool exist(node);
%relation liant le pere d'un fils
 bool childof(node, node);
%donnant la valeur du label
 lbl getLbl(node);

auxiliary
%dit si il existe un chemin entre le pere et le fils
% bool childofst(node, node);

operation
%ajoute un noeud n si il n'existe pas, il deviendra fils de p qui doit exister
 not(exist(n)) and exist(p) and (p!=n)     : Add(node p,node n);
%supprime un noeud autre que mem et data
 exist(n)     : Del(node n);
%renome ou defini un label.
 exist(n)     : ChLbl(node n,lbl l,nat t);

transform
%On definir
T(Add(p1,n1),Del(n2))= if(n1==n2) then
  return nop
 else
  return Add(p1,n1)
 endif;
T(ChLbl(n1,l1,s1),ChLbl(n2,l2,s2))= if (n1==n2 and s1 > s2) then
  return nop
 else
  return ChLbl(n1,l1,s1)
 endif;
T(Del(n1),Del(n2)) = if (n1==n2) then
  return nop
 else
```

```
38   return Del(n1)
39  endif;
40 definition
41 %On definit les observatuer d'existance
42 exist'(n1)/Add(p2,n2) = if (n1 == n2) then return true
43     else return exist(n1)
44     endif;
45 exist'(n1)/Del(n2) =  if (n1 == n2)  then return false
46     else return exist(n1)
47     endif;
48
49
50 %Observateur de structure
51  childof'(n1,p1)/Add(p2,n2) = if (n1 == n2 and p2==p1)
52        then return true
53      else
54       return childof(n1,p1)
55      endif;
56
57 childof'(n1,p1)/Del(n2) = if (n2 == n1) then
58      return false
59   %  elseif (not(exist(n1)) or not(exist(p1))) then
60   %   return false
61      else
62      return childof(n1,p1)
63     endif;
64
65
66 %Observateur du label.
67  getLbl'(n1)/Add(p2,n2) = if (n1==n2) then return novalue
68      else
69      return getLbl(n1)
70      endif;
71  getLbl'(n1)/Del(n2) = if (n2 == n1) then return novalue
72      else
73      return getLbl(n1)
74     endif;
75 getLbl'(n1)/ChLbl(n2,l2,s2) =if (n1==n2) then
76      if (exist(n1)) then
77       return l2
78      else
79       return novalue
80      endif
81      else
82      return getLbl(n1)
83     endif;
84 lemma
85
86
87 %Je prend pour hypothese qu'il n'y a pas de concurence sur un
       meme site.
88 s1>=s2 and s2>=s1 =>;
89 not( s1>s2) and not(s2>s1) =>;
90 childof(x,x)=>;
91 childof(x,y) and childof(x,z) and (z!=y) =>;
92 %childof(x,y) and childof(y,z)=>childofst(x,z);
93 %childofst(x,x)=>;
```

TP1 et TP2 pour les opérations $Add, Del_2, ChLbl$

```
type node(none),lbl(novalue),nat;
observator
%test l'existance d'un noeud
 bool exist(node);
%relation liant le pere d'un fils
 bool childof(node, node);
%Observateur du label
 lbl getLbl(node);

operation
%ajoute un noeud n si il n'existe pas, il deviendra fils de p qui doit exister
 not(exist(n)) and exist(p)     : Add(node p,node n);
%supprime un noeud et fait remonter le sous arbre
 exist(n) and  exist(p) and childof(n,p)   : Del(node n,node p);
%Le noeud doit exister
 exist(n)        : ChgLbl(node n,lbl l,nat t);

transform

 T(Add(p1,n1),Del(n2,p2)) = if (n1==n2) then
  return nop
 elseif (p1==n2) then
  return Add(p2,n1)
 else
  return Add(p1, n1)
 endif;
 T(Del(n1,p1),Del(n2,p2)) = if (n1==n2) then
  return nop
 elseif (p1==n2) then
  return Del(n1,p2)
 else
  return Del(n1,p1)
 endif;

T(ChgLbl(n1,l1,s1),ChgLbl(n2,l2,s2))= if (n1==n2 and s1 > s2) then
  return nop
 else
  return ChgLbl(n1,l1,s1)
 endif;
definition
exist'(n1)/Add(p2,n2) = if (n1 == n2) then return true
    else return exist(n1)
    endif;
 exist'(n1)/Del(n2,p2) =  if (n1 == n2) then return false
    else return exist(n1)
    endif;
childof'(n1,p1)/Add(p2,n2) = if (n1 == n2 and p2==p1) then return true
     else return childof(n1,p1)
     endif;
childof'(n1,p1)/Del(n2,p2) = if (n2 == n1) then
     return false
    elseif (p1==p2 and childof(n1,n2)) then
     return true
    elseif (p1==n2) then
     return false
     else
     return childof(n1,p1)
    endif;
```

```
59
60   getLbl'(n1)/Add(p2,n2) = if (n1==n2) then return novalue
61      else
62      return getLbl(n1)
63      endif;
64  getLbl'(n1)/Del(n2,p2) = if (n2 == n1) then return novalue
65      else
66      return getLbl(n1)
67     endif;
68 getLbl'(n1)/ChgLbl(n2,l2,s2) =if (n1==n2) then
69      if (exist(n1)) then
70       return l2
71      else
72       return novalue
73      endif
74      else
75      return getLbl(n1)
76     endif;
77
78 lemma
79 %Je prend pour hypothese qu'il n'y a pas de concurence sur un
       meme site.
80 s1>s2 and s2>s1 =>;
81 s1>=s2 and s2>=s1 =>;
82 not( s1>s2) and not(s2>s1) =>;
83 childof(x,x)=>;
84 childof(x,y) and childof(x,z) and (z!=y) =>;
85 childof(x,y) and childof(y,x) =>;
```

Bibliographie

[ABGM90] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina.
Data caching issues in an information retrieval system.
ACM Trans. Database Syst., 15 :359–384, September 1990.
(Cité pages 12 et 13.)

[AD] Karl Aberer and Zoran Despotovic.
On the convergence of structured search, information retrieval and trust management in distributed systems.
(Cité page 12.)

[ARS02] Alessandro Armando, Michaël Rusinowitch, and Sorin Stratulat.
Incorporating decision procedures in implicit induction.
J. Symb. Comput., 34 :241–258, October 2002.
(Cité page 26.)

[Ber90] Brian Berliner.
CVS II : Parallelizing software development.
In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, Californie, États-Unis, 1990. USENIX Association.
(Cité pages 2 et 6.)

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman.
Concurrency Control and Recovery in Database Systems.
Addison-Wesley, 1987.
(Cité page 9.)

[BKR92] Adel Bouhoula, Emmanuel Kounalis, and Michaël Rusinowitch.
Automated mathematical induction.
Research Report RR-1663, INRIA, 1992.
Projet EURECA.
(Cité page 26.)

[BP98] S. Balasubramaniam and Benjamin C. Pierce.
What is a file synchronizer ?
In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '98, pages 98–108, New York, NY, USA, 1998. ACM.
(Cité page 28.)

[BSR95] Adel Bouhoula, Sorin Stratulat, and Michael Rusinowitch, 1995.
`http://www.loria.fr/equipes/cassis/softwares/spike/`.
(Cité page 26.)

[CD95] Rajiv Choudhary and Prasun Dewan.
A general multi-user undo/redo model.
In *ECSCW'95 : Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, pages 231–246, Norwell, MA, USA, 1995. Kluwer Academic Publishers.
(Cité page 77.)

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi.
Tree automata techniques and applications.
Available on : `http://www.grappa.univ-lille3.fr/tata`, 2007.
release October, 12th 2007.
(Cité page 89.)

[CF07] Michelle Cart and Jean Ferrie.
Asynchronous reconciliation based on operational transformation for p2p collaborative environments.
In *Proceedings of the 2007 International Conference on Collaborative Computing : Networking, Applications and Worksharing*, pages 127–138, Washington, DC, USA, 2007. IEEE Computer Society.
(Cité pages 25 et 31.)

[Cov] Coverpages.
`http://xml.coverpages.org/xmlApplications.html`.
(Cité pages 1 et 20.)

[DSL02] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu.
Generalizing operational transformation to the Standard General Markup Language.
In *Proceedings of the ACM conference on Computer supported cooperative work - CSCW'02*, pages 58–67, New York, New York, États-Unis, 2002. ACM Press.
(Cité page 26.)

[DSU03] Xavier Défago, André Schiper, and Péter Urbán.
Total order broadcast and multicast algorithms : Taxonomy and survey.
ACM Computing Surveys, 36 :2004, 2003.
(Cité page 10.)

[EG89] C. A. Ellis and S. J. Gibbs.
Concurrency control in groupware systems.
SIGMOD Rec., 18 :399–407, June 1989.
(Cité pages 7 et 21.)

[EG09] Ami Eyal and Avigdor Gal.
Self organizing semantic topologies in p2p data integration systems.
In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1159–1162, Washington, DC, USA, 2009. IEEE Computer Society.
(Cité page 11.)

[EGmKM04] P. T. Eugster, R. Guerraoui, A. m. Kermarrec, and L. Massoulié.
From epidemics to distributed computing.
IEEE Computer, 37 :60–67, 2004.
(Cité page 10.)

[FK92] David Ferraiolo and Richard Kuhn.
Role-based access control.
In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.
(Cité page 116.)

[git] Git wikipedia.
`http://en.wikipedia.org/wiki/Git_(software)`.
(Cité page 7.)

[HKP+05] J. Hromkovic, R. Klasing, A. Pelc, P. Ruzicka, and W. Unger.
Dissemination of Information in Communication Networks : Broadcasting, Gossiping, Leader Election, and Fault-Tolerance (Texts in Theoretical Computer Science. An EATCS Series).
Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
(Cité page 11.)

[Ign06] C. Ignat.
Maintaining Consistency in Collaboration over Hierarchical Documents.
Dissertation, ETH Zurich, Switzerland, July 2006.
ETH No. 16766.
(Cité page 26.)

[Imi06] A. Imine.
Conception Formelle d'Algorithmes de Réplication Optimiste. Vers l'Edition Collaborative dans les Réseaux Pair-à-Pair.
PhD thesis, Université Henri Poincaré, Nancy, décembre 2006.
(Cité pages 16, 24, 25, 26, 27 et 31.)

[IMOU03] Abdessamad Imine, Pascal Molli, Gérald Oster, and Pascal Urso.
VOTE : Group Editors Analyzing Tool.
Electronic Notes Theoretical Computer Science, 86(1), June 2003.
(Cité page 26.)

[INo3] Claudia-Lavinia Ignat and Moira C. Norrie.
Customizable Collaborative Editor Relying on treeOPT Algorithm.
In *Proceedings of the 8th European Conference on Computer-supported Cooperative Work (ECSCW'03)*, pages 315–334. Kluwer Academic Publishers, September 2003.
(Cité page 26.)

[IOo8] C. Ignat and G. Oster.
Peer-to-peer Collaboration over XML Documents.
In *Proceedings of the 5th International Conference on Cooperative Design, Visualization and Engineering - CDVE 2008*, pages 66–73, Mallorca, Spain, September 2008. Springer.
(Cité page 26.)

[Jaco9] Judah Jacobson.
A formalization of darcs patch theory using inverse semigroups, 2009.
`http://www.math.ucla.edu/~jjacobson/patch-theory/patch-semigro`
(Cité page 8.)

[JTo5] Paul R. Johnson and Robert H. Thomas.
RFC 677 : Maintenance of duplicate databases, 1975, (Septembre 2005).
`http://www.ietf.org/rfc/rfc677.txt`.
(Cité page 86.)

[Ka] Eric Kow and all.
Darcs theory.
`http://wiki.darcs.net/Theory`.
(Cité page 8.)

[KRSDo1] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel.
The icecube approach to the reconciliation of divergent replicas.
In *PODC '01 : Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 210–218, New York, NY, USA, 2001. ACM.
(Cité pages 2 et 7.)

[Lam78] Leslie Lamport.
Ti clocks, and the ordering of events in a distributed system.
Commun. ACM, 21 :558–565, July 1978.
(Cité pages 13, 14 et 58.)

[LCo3] B. Lushman and G.V. Cormack.
Proof of correctness of ressel's adopted algorithm.
Inf. Process. Lett., 86(6) :303–310, 2003.
(Cité page 25.)

[LLo5] Rui Li and Du Li.
Commutativity-based concurrency control in groupware.
In *Collaborative Computing*, 2005.
(Cité page 25.)

[LLS04] Rui Li, Du Li, and Chengzheng Sun.
A time interval based consistency control algorithm for interactive groupware applications.
In *Proceedings of the Parallel and Distributed Systems, Tenth International Conference*, ICPADS '04, pages 429–, Washington, DC, USA, 2004. IEEE Computer Society.
(Cité page 23.)

[MOSmI03] Pascal Molli, Gérald Oster, Hala Skaf-molli, and Abdessamad Imine.
Using the transformational approach to build a safe and generic data synchronizer.
In *In Proceedings of ACM Group 2003 Conference, November 9–12 2003*, pages 212–220, 2003.
(Cité pages 2 et 25.)

[MUW10] Stéphane Martin, Pascal Urso, and Stéphane Weiss.
Scalable xml collaborative editing with undo - (short paper).
In *OTM Conferences (1)*, pages 507–514, 2010.
(Cité page 73.)

[NCDL95] 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.
(Cité pages 9 et 25.)

[Ora] Oracle.
`http://download.oracle.com/javase/6/docs/api/java/util/HashMap`
(Cité pages 41 et 68.)

[OSMMNJ07] G. Oster, H. Skaf-Molli, P. Molli, and H. Naja-Jazzar.
Supporting collaborative writing of xml documents.
unpublished, 2007.
(Cité pages 26 et 31.)

[OUMaI06] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessam ad Imine.
Data Consistency for P2P Collaborative Editing.
In *Proceedings of the ACM Conference on Computer-Supported Coo perative Work - CSCW 2006*, pages 259–267, Banff, Alberta, Canada, nov 2006. ACM Press.
(Cité pages 2 et 28.)

[PMSL09] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia.
A commutative replicated data type for cooperative editing.
In *ICDCS*, Montreal, Quebec, Canada, June 2009. IEEE Computer Society.

(Cité pages 2, 29 et 57.)

[PRS97] Ravi Prakash, Michel Raynal, and Mukesh Singhal.
An adaptive causal ordering algorithm suited to mobile computing environments.
Journal of Parallel and Distributed Computing, 41 :190–204, 1997.
(Cité page 30.)

[PSG03] B.C. Pierce, A. Schmitt, and M.B. Greenwald.
Bringing Harmony to optimism : A synchronization framework for heterogeneous tree-structured data.
Technical Report MS-CIS-03-42, University of Pennsylvania, 2003.
Superseded by MS-CIS-05-02.
(Cité pages 2 et 32.)

[RC01] Norman Ramsey and Előd Csirmaz.
An algebraic approach to file synchronization.
In *9th Foundations of Softw. Eng.*, pages 175–185, Austria, September 2001.
(Cité page 28.)

[RG99] Matthias Ressel and Rul Gunzenhäuser.
Reducing the problems of group undo.
In *GROUP '99 : Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 131–139, New York, NY, USA, 1999. ACM.
(Cité page 77.)

[RM] J. Risson and T. Moors.
Rfc 4981 survey of research on p2p search september 2007.
`http://www.normes-internet.com/normes.php?rfc=rfc4981`.
(Cité page 112.)

[RNRG96] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser.
An integrating, transformation-oriented approach to concurrency control and undo in group editors.
In *CSCW '96 : Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 288–297, New York, NY, USA, 1996. ACM.
(Cité pages 2, 16, 22, 24 et 58.)

[Roc75] M.J. Rochkind.
The source code control system.
IEEE Trans. Software Eng., 1(4) :364–370, 1975.
(Cité page 6.)

[Sal92] Rich Salz.
InterNetNews : Usenet transport for Internet sites.
In *USENIX conference proceedings*, pages 93–98, San Antonio, Texas, États-Unis, Été 1992. USENIX.
(Cité page 6.)

[SC06] S. Staworko and J. Chomicki.
Validity-sensitive querying of XML databases.
In *EDBT Workshops (dataX)*, pages 164–177. Springer LNCS 4254, 2006.
(Cité page 90.)

[SCF97] M. Suleiman, M. Cart, and J. Ferrié.
Serialization of concurrent operations in a distributed collaborative environment.
In *GROUP '97 : Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 435–445, New York, NY, USA, 1997. ACM.
(Cité page 24.)

[SCF98] M. Suleiman, M. Cart, and J. Ferrié.
Concurrent operations in a distributed and mobile collaborative environment.
In *ICDE '98 : Proceedings of the Fourteenth International Conference on Data Engineering*, pages 36–45, Washington, DC, USA, 1998. IEEE Computer Society.
(Cité page 26.)

[SE98] C. Sun and C. Ellis.
Operational transformation in real-time group editors : issues, algorithms, and achievements.
In *CSCW '98 : Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68, New York, NY, USA, 1998. ACM.
(Cité page 26.)

[sf11] Reporters sans frontières.
Journée mondiale contre la censure., 12 Mars 2011.
`http://12mars.rsf.org/`.
(Cité page 11.)

[SJZ+98] 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 Trans. Comput.-Hum. Interact., 5 :63–108, March 1998.
(Cité pages 15 et 24.)

[SLG10] Bin Shao, Du Li, and Ning Gu.
A fast operational transformation algorithm for mobile and asynchronous collaboration.
IEEE Trans. Parallel Distrib. Syst., 21 :1707–1720, December 2010.
(Cité pages 25, 31 et 58.)

[SNG10] Idrissa Sarr, Hubert Naacke, and Stéphane Gançarski.
Routage décentralisé de transactions avec gestion des pannes dans un réseau à large échelle.

Ingénierie des Systèmes d'Information, 15(1) :87–111, 2010.
(Cité page 112.)

[SS05] Yasushi Saito and Marc Shapiro.
Optimistic replication.
ACM Comput. Surv., 37 :42–81, March 2005.
(Cité page 9.)

[SS06] David Sun and Chengzheng Sun.
Operation Context and Context-based Operational Transformation.
In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 279–288, Banff, Alberta, Canada, November 2006. ACM Press.
(Cité page 25.)

[SS09] David Sun and Chengzheng Sun.
Context-based operational transformation in distributed collaborative editing systems.
IEEE Trans. Parallel Distrib. Syst., 20 :1454–1470, October 2009.
(Cité page 25.)

[Sun02] Chengzheng Sun.
Undo as concurrent inverse in group editors.
ACM Trans. Comput.-Hum. Interact., 9 :309–361, December 2002.
(Cité page 15.)

[Tho79] Robert H. Thomas.
A majority consensus approach to concurrency control for multiple copy databases.
ACM Trans. Database Syst., 4(2) :180–209, 1979.
(Cité page 6.)

[Tic85] Walter F. Tichy.
Rcs–a system for version control.
Software–Practice and Experience, 15 :637–654, July 1985.
(Cité page 6.)

[TKDP+08] Mounir Tlili, William Kokou Dedzoe, Esther Pacitti, Patrick Valduriez, Reza Akbarinia, Pascal Molli, Gérôme Canals, Julien Maire, and Stéphane Laurière.
Data Reconciliation in P2P Collaborative Applications.
In *24ème Journées "Bases de Données Avancées" - BDA 2008*, Guilherand-Granges, France, 2008.
(Cité page 12.)

[Tor05] Linus Torvalds.
git, (April 2005).
http ://git.or.cz/.
(Cité page 7.)

[VCFS00] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman.
Copies convergence in a distributed real-time collaborative environment.
In *CSCW '00 : Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 171–180, New York, NY, USA, 2000. ACM.
(Cité page 25.)

[Wei10] Stéphane Weiss.
Edition collaborative massive sur reseaux Pair-a-Pair.
PhD thesis, l'université Henri Poincaré Nancy1 France, 2010.
(Cité pages 15, 30 et 68.)

[Wika] Wikipedia.
`http://fr.wikipedia.org/wiki/Fonction_de_hachage`.
(Cité pages 41 et 68.)

[wikb] wikipedia.
Access control list.
`http://fr.wikipedia.org/wiki/Access_Control_List`.
(Cité page 116.)

[wikc] wikipedia.
Message sequence chart.
`http://en.wikipedia.org/wiki/Message_Sequence_Chart`.
(Cité page 112.)

[Wikd] Wikipedia.
Unicode.
`http://fr.wikipedia.org/wiki/Unicode`.
(Cité page 20.)

[WML10] David Wang, Alex Mah, and Soren Lassen.
Google wave protocol, july 2010.
`http://wave-protocol.googlecode.com/hg/whitepapers/operational`
(Cité pages 2, 9 et 25.)

[wso] web site optimization.
Average web page size septuples since 2003.
`http://www.websiteoptimization.com/speed/tweak/average-web-pag`
(Cité page 113.)

[WUM08] Stéphane Weiss, Pascal Urso, and Pascal Molli.
An Undo Framework for P2P Collaborative Editing .
In *CollaborateCom*, Orlando, USA, November 2008.
(Cité page 77.)

[WUM09] Stephane Weiss, Pascal Urso, and Pascal Molli.
Logoot-undo : Distributed collaborative editing system on p2p networks.
IEEE Transactions on Parallel and Distributed Systems, 99(PrePrints), 2009.
(Cité pages 2, 29 et 86.)

[YV01] Haifeng Yu and Amin Vahdat.
The costs and limits of availability for replicated services.
SIGOPS Oper. Syst. Rev., 35 :29–42, October 2001.
(Cité page 13.)

Titre Édition collaborative des documents semi-structurés

Résumé Les éditeurs collaboratifs permettent à des utilisateurs éloignés de collaborer à une tâche commune qui va de l'utilisation d'un agenda partagé à la réalisation de logiciels. Ce concept est né avec SCCS en 1972 et connait un engouement récent (ex : Wikipedia). L'absence de centralisation et l'asynchronisme sont des aspects essentiels de cette approche qui relève d'un modèle pair-à-pair (P2P). D'un autre côté, le format XML est devenu une référence pour la manipulation et l'échange de documents. Notre travail vise à la réalisation d'un éditeur collaboratif P2P pour l'édition de documents semi-structurés qui sont une abstraction du format XML. Le problème est difficile et de nombreuses propositions se sont révélées erronées ou ne passant pas à l'échelle. Nous rappelons les concepts et l'état de l'art sur l'édition collaborative, les modèles centralisés et le P2P. Ensuite, nous explorons deux approches différentes : les transformées opérationnelles et le CRDT (Commutative Replicated Data Type) avec différentes structures de données arborescentes. L'objectif est de réaliser les opérations de base (ajout, suppression et ré-étiquetage) tout en garantissant la convergence du processus d'édition. Nous proposons un algorithme générique pour l'approche CRDT basée sur une notion d'indépendance dans la structure de données. Nous avons étendu nos travaux afin de réaliser l'opération de déplacement d'un sous-arbre et de prendre en compte le typage XML. Peu de travaux abordent ces deux points qui sont très utiles pour l'édition de documents. Finalement, nous donnons les résultats expérimentaux obtenus avec un prototype permettant de valider notre approche.

Mots-clés Edition collaborative, Pair-à-Pair, Documents semi-structurés, Opération d'édition commutatives (CRDT), Transformées opérationnelles, Type pour XML

Title Collaborative edition on semi-structured documents

Abstract Collaborative editors allow different users to work together on a common task. Such tasks range from using a shared calendar to realizing software programmed by users located at distant sites. This concept was invented in 1972 with SCCS. In the last years, this paradigm became popular (ex. Wikipedia). Decentralization and asynchronicity are essential in this approach, leading to peer-to-peer (P2P) models. Meanwhile, the XML format has arrived as the de facto standard for editing and exchanging documents. Our work aims at defining a collaborative editor for semi-structured documents, which provide an abstraction of the XML format. The problem is difficult since many previous approaches are flawed or not scalable. Firstly, we describe the basic concepts on collaborative edition and network models and we give the state of the art of this topic. Then, we investigate two different approaches : the operational transformation (OT) approach and the Commutative Replicated Data Type (CRDT) approach for different (tree-like) data structures. Our goal is to ensure the

convergence of the editing process with the basic operations (Add, Del and rename a node. We have proposed a new generic algorithm based on semantic independence in data structure for CRDT approach. We have extended our results by dealing with the operation that moves a subtree and with XML schema compliance. Few works have been devoted to these extensions which are useful in collaborative edition. Finally, we provide experimental results obtained from our implementation that validate our approach.

Keywords Collaborative Editing, Peer-to-Peer, Semistructured document, Commutative Replicated Data Type (CRDT), Operational Transformation, Types for XML