Réplication optimiste et cohérence des données dans les environnements collaboratifs répartis

Gérald Oster

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Réplication optimiste et cohérence des données dans les environnements collaboratifs répartis

THÈSE

présentée et soutenue publiquement le 3 novembre 2005

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Gérald Oster

Composition du jury

Président :	Jens Gustedt	Directeur de Recherche, INRIA Lorraine
Rapporteurs :	Jean Ferrié	Professeur, Université Montpellier II
	Marc Shapiro	Directeur de Recherche, INRIA Rocquencourt
Invité :	Michel Raynal	Professeur, Université Rennes 1
Directeurs de thèse :	Claude Godart	Professeur, Université Henri Poincaré - Nancy 1
	Pascal Molli	Maître de Conférences, Université Henri Poincaré - Nancy 1

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

Mis en page avec la classe thloria.

Table des matières

Table des figures

Introduction

1 Contexte et objectifs de la thèse

Les éditeurs collaboratifs permettent à plusieurs utilisateurs d'éditer de manière collaborative des documents à travers un réseau informatique. Un éditeur collaboratif permet l'édition simultanée d'un même document par plusieurs utilisateurs distribués dans l'espace, le temps et à travers les organisations. En parallélisant les tâches d'édition, on réduit le temps d'écriture d'un document.

Les Wikiwikiwebs [Cun05], apparus en 1995, ont rendu les éditeurs collaboratifs extrêmement populaires. Le plus célèbre site Wiki est l'encyclopédie Wikipédia [Wik05]. Créée en 2000, elle compte aujourd'hui 1,5 millions d'articles dans plus de 200 langages. La Wikipédia est le fruit d'un effort collaboratif massif distribué sur l'ensemble de la planète.

Nous pensons que le succès de l'édition collaborative ouvre la porte à un nouveau domaine : l'édition collaborative massive. Il ne s'agit plus d'éditer à quelques uns mais à des milliers distribués sur l'ensemble de la planète. Ce qui a été réalisé dans le cadre d'une encyclopédie peut être reproduit pour des dictionnaires, des journaux etc.

Les éditeurs collaboratifs existants n'ont pas été conçus pour admettre un nombre si important d'utilisateurs. La Wikipédia rencontre actuellement des problèmes de passage à l'échelle. Ces problèmes ne sont pas d'ordre technologique, ils remettent en cause les algorithmes existants.

Le défi de cette thèse est de proposer des algorithmes adaptés à l'édition collaborative massive. Nous montrons qu'un tel algorithme doit concilier trois impératifs : assurer la convergence des données, préserver les intentions des opérations concurrentes et passer à l'échelle. Les algorithmes existants, soit convergent sans préserver les intentions, soit préservent les intentions mais ne passent pas à l'échelle.

Dans un premier temps, nous avons étudié les algorithmes basés sur l'approche des transformées opérationnelles. Malheureusement, nous montrons que cette approche nous ramène à un problème qui ne peut être résolu par les algorithmes actuels. Toutefois, les enseignements que nous avons pu tirer de cette expérience nous ont amené à imaginer une nouvelle approche qui assure la convergence des données, la préservation des intentions et passe à l'échelle. Elle ne nécessite ni site central, ni ordre global, ni vecteur d'horloges. Sa mise en œuvre est facile, et les premiers essais que nous avons réalisés sont très encourageants.

2 Résumé des contributions

Nos résultats sont les suivants :

1. Nous montrons que l'approche OT originellement conçue pour les éditeurs collaboratifs synchrones peut être utilisée pour réaliser des synchroniseurs [MSMOJ02, MOSMI03, OMISM04] ou des outils de gestion de configuration.
2. Nous utilisons une approche formelle pour concevoir et vérifier les fonctions de transformation. Nous mettons en place un environnement de conception [IMOR02, IMOU03, IROM05] et de vérification qui repose sur l'emploi d'un démonstrateur de théorème.
3. Cet environnement nous a permis de trouver des contre-exemples [IMOR03] pour toutes les fonctions de transformation existantes [RNRG96, SCF97, SJZ+98].
4. Nous montrons qu'il n'est pas possible avec juste des transformées en avant de construire un éditeur collaboratif qui converge et qui préserve les intentions.
5. Nous proposons un nouvel algorithme WOOT adapté à l'édition collaborative massive. Cet algorithme assure la convergence, préserve les intentions et passe l'échelle pour les structures linéaires. Il repose sur une fonction de linéarisation monotone des ordres partiels formés par les relations entre les différents éléments de la structure. Il ne nécessite ni serveur central, ni ordre global, ni vecteur d'horloges.

3 Organisation du document

Notre mémoire s'organise de la manière suivante.

Dans le chapitre 1, nous déterminons les caractéristiques d'un éditeur collaboratif massif. Un tel éditeur doit assurer la convergence, préserver les intentions et passer à l'échelle.

Dans le chapitre 2, nous nous intéressons aux différents systèmes permettant de réconcilier des copies divergentes. Nous évaluons chacune des propositions par rapport à nos trois critères : convergence, intention et passage à l'échelle. L'approche OT semble remplir ces trois critères s'il est possible de développer des fonctions de transformation respectant deux conditions dites C_1 et C_2.

Nous proposons au chapitre 3 une approche pour concevoir des fonctions de transformation et vérifier leur correction face aux conditions C_1 et C_2. Nous montrons, dans ce chapitre, qu'il n'est pas possible de construire des fonctions de transformation en avant pour les chaînes de caractères qui vérifient C_1, C_2 et qui préservent les intentions.

Dans le chapitre 4, nous présentons notre algorithme WOOT conçu pour l'édition collaborative massive.

Finalement, le dernier chapitre décrit le bilan de notre travail et présente les perspectives.

Chapitre 1

Problématique

Sommaire

L'édition collaborative massive soulève de nouvelles questions. Les éditeurs collaboratifs existants n'ont pas été conçus pour tolérer un nombre si important d'utilisateurs. Par exemple, la Wikipédia rencontre actuellement des problèmes de passage à l'échelle. Ces problèmes ne sont pas d'ordre technologique, ils remettent en cause les algorithmes existants.

Les algorithmes d'édition collaborative sont basés sur une réplication optimiste [SS05] des données. Chaque site dispose d'une copie qu'il peut modifier librement à tout moment. Les changements sont ensuite propagés aux autres sites pour y être ré-exécutés. Si deux sites modifient en parallèle leurs copies respectives, celles-ci divergent. Les algorithmes d'édition collaborative doivent alors assurer que le système va finir par converger. La convergence seule n'est pas suffisante, il est en effet possible de converger en ignorant certaines modifications. Un éditeur collaboratif doit converger vers un état où toutes les modifications concurrentes sont observables. On parle alors de préservation de l'intention.

L'objectif de cette thèse est de proposer les algorithmes nécessaires à la mise en place d'un environnement d'édition collaborative massive. L'algorithme idéal doit concilier convergence, préserver les intentions et passage à l'échelle.

1.1 Convergence

Un éditeur collaboratif est constitué d'un ensemble de sites interconnectés par un réseau. Chaque site possède une copie des objets partagés par exemple des documents textuels ou des images. Sur un site, une réplique peut être modifiée au moyen d'opérations. Quand une réplique est modifiée sur un site, l'opération correspondante est immédiatement exécutée sur ce site, puis propagée aux autres sites pour y être ré-exécutée.

Lorsque deux répliques de deux sites différents sont modifiées en parallèle, les répliques divergent. Il est donc possible d'observer au même moment une valeur sur un site et une autre valeur sur un autre site. Les algorithmes de réplication optimiste doivent assurer la convergence des répliques. Le système doit être convergeant quand le système est au repos, c'est-à-dire quand toutes les opérations ont été propagées à tous les sites.

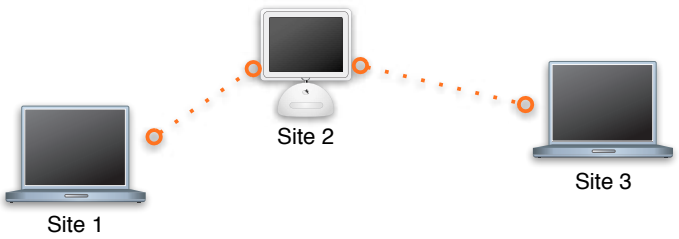


Fig. 1.1 – Système considéré.

Nous illustrons maintenant ces phases de divergence/convergence en utilisant la règle de Thomas [JT05]. Cette règle est utilisée pour assurer la convergence dans Usenet [Sal92].

Dans notre exemple, nous considérons trois sites interconnectés par un réseau informatique selon la topologie décrite à la figure 1.1. Les identifiants de chaque site sont totalement ordonnés. Chaque site réplique une chaîne de caractères s de valeur initiale "AB".

La règle de Thomas associe à chaque réplique sur un site i une estampille $T_i(h, s)$; h est une heure et s un numéro de site. Pour modifier sa réplique, un site génère une opération de mise à jour contenant la nouvelle valeur et une estampille. Cette estampille contient l'heure courante du site et le numéro de site. Cette opération est exécutée immédiatement sur le site puis propagée aux sites voisins. Par exemple, le site 1 peut générer à tout moment l'opération $op_1 = set(s, "AXB", (13h, 1))$, s est une chaine de caractère, "AXB" la nouvelle valeur, $(13h, 1)$ est l'estampille. Cette opération va ensuite être propagée aux sites voisins pour y être ré-exécutée.

Lorsqu'une opération est reçue sur un site, le site compare l'estampille de l'opération à celle de sa réplique. Si l'estampille de l'opération est plus récente alors il exécute l'opération et met à jour l'estampille, sinon, il ignore l'opération. Une estampille $T_1(h_1, s_1)$ est plus récente qu'une estampille $T_2(h_2, s_2)$ si : $(h_1 > h_2) ou ((h_1 = h_2) et (s_1 > s_2))$.

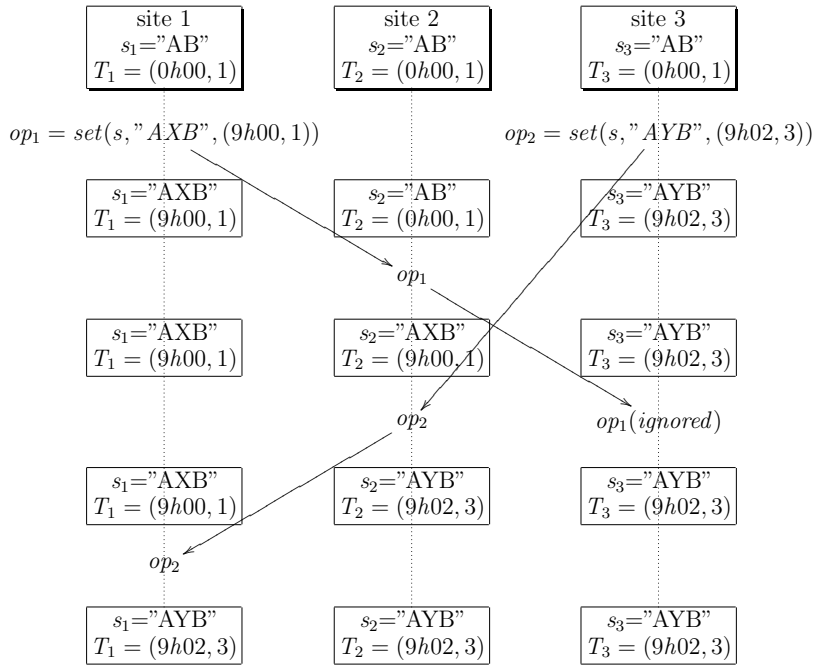


FIG. 1.2 – Utilisation de la règle de Thomas

Le scénario de la figure 1.2 illustre comment la règle de Thomas prend en charge deux opérations concurrentes.

Deux utilisateurs génèrent deux opérations concurrentes op_1 et op_2. Ils les exécutent localement et les propagent à leur voisin.

– Quand op_1 arrive sur le site 2, son estampille étant plus récente que celle de la réplique, op_1 est exécutée. L'état sur le site 2 devient "AXB" et l'estampille est mise à jour.

- À l'arrivée de op_2 sur le site 2, on compare l'estampille de op_2 avec celle la réplique. Comme l'estampille $(9h02, 3)$ est plus récente que l'estampille $(9h00, 1)$, op_2 est exécutée et l'état devient "AYB".
- Lorsque op_1 arrive sur le site 3, on compare son estampille $(9h00, 1)$ par rapport à l'estampille de la réplique $(9h02, 3)$. Puisque l'estampille de l'opération est plus ancienne, l'opération ignorée et l'état ne change pas.
- Quand op_2 arrive sur le site 1, elle est exécutée et l'état devient "AYB".

Au final, toutes les répliques ont bien convergé vers la même valeur "AYB". Cependant, dans un contexte collaboratif, le résultat obtenu n'est pas satisfaisant. En effet, la mise à jour apportée par le premier utilisateur, à savoir l'opération op_1, a été purement et simplement perdue. L'effet de op_1 n'est pas observable dans l'état final. Dans le domaine des éditeurs collaboratifs, ce problème est connu comme un problème de violation de l'intention de opérations.

En supposant que l'intention des deux utilisateurs était d'insérer un caractère entre les caractères 'A' et 'B', les états de convergence acceptables sont "AXYB" ou "AYXB" ; ces états préservent l'effet des deux insertions.

1.2 Intention

Sun [SJZ+98] définit l'intention comme suit :

Définition 1.1 (Intention) L'intention d'une opération *op* est l'effet observé lors de son exécution sur son état de génération.

Supposons un entier initialisé à 3. Une opération fait passer l'entier à 7. L'effet observé dépend du point de vue de l'observateur. On peut observer qu'il y un changement d'état de 3 vers 7 ou on peut observer une incrémentation de 4 de la valeur de l'entier. Ces deux observations sont différentes.

Considérons maintenant une chaîne de caractères contenant "AB". L'exécution d'une opération transforme la chaîne initiale en "AXB". On peut observer une insertion d'un caractère X à la position 2 $ins(2, X)$, ou l'insertion d'un caractère X entre A et B ($ins(A \prec X \prec B)$), ou l'insertion d'un caractère 'X' après 'A' ($ins(A \prec X)$) ou encore l'insertion d'un caractère 'X' avant 'B' ($ins(X \prec B)$).

Cette définition oblige donc le concepteur d'éditeur collaboratif à définir le type des objets partagés, ses opérations, et pour chaque opération ses intentions.

Sun définit ensuite la préservation de l'intention comme suit :

Définition 1.2 (Préservation de l'intention)

1. Pour toute opération *op*, les effets de l'exécution de *op* sur tous les sites doivent être les mêmes que l'intention de *op*.

2. L'effet d'exécuter une opération *op* ne doit pas changer les effets des opérations concurrentes.

En fonction de la définition des intentions des opérations, la préservation de l'intention est définie ou non. Bien évidemment, si elle ne l'est pas, il n'est pas possible de construire un éditeur collaboratif qui la préserve.

Par exemple, si on prend un objet répliqué de type entier. Cet objet dispose d'une opération d'affectation = avec pour intention de mettre à jour la valeur de l'entier. Il n'est pas possible de préserver l'intention dans ce cas. En effet, si un utilisateur génère une opération $x = 3$ pendant qu'un autre génère en parallèle $x = 7$, l'effet d'exécuter $x = 3$ change l'effet d'exécuter $x = 7$. La notion de préservation d'intention n'est pas définie pour ce type d'objet avec ce type d'opération.

Si on prend un objet de type entier avec une opération incrémenter $inc(x)$ qui incrémente l'entier x de 1. Si on définit l'intention comme, l'entier est incrémenté de 1 alors la notion de préservation de l'intention n'est pas définie. Par contre, si on définit l'intention comme, la valeur est croissante, alors la notion de préservation de l'intention est définie même si elle est sans grand intérêt.

On réplique maintenant un objet de type chaîne de caractères avec une opération d'insertion d'un caractère c à une position p : $ins(p, c)$. Si on définit l'intention de *ins* comme insérer le caractère c exactement à la position p alors la préservation de l'intention n'est pas définie. Si on définit l'intention de *ins* comme l'insertion entre le caractère précédent et le caractère suivant, alors la notion d'intention est définie.

En effet, prenons une chaîne de caractères avec comme valeur initiale "AB". Un utilisateur insère 'X' entre 'A' et 'B' ; $op_1 = ins(1, X)$ et l'intention de op_1 est de faire respecter la contrainte $A \prec X \prec B$, autrement dit 'X' est placé derrière 'A' et devant 'B'. Un autre utilisateur insère en parallèle 'Y' entre 'A' et 'B' ; $op_2 = ins(1, Y)$ et l'intention de op_2 est de préserver $A \prec Y \prec B$.

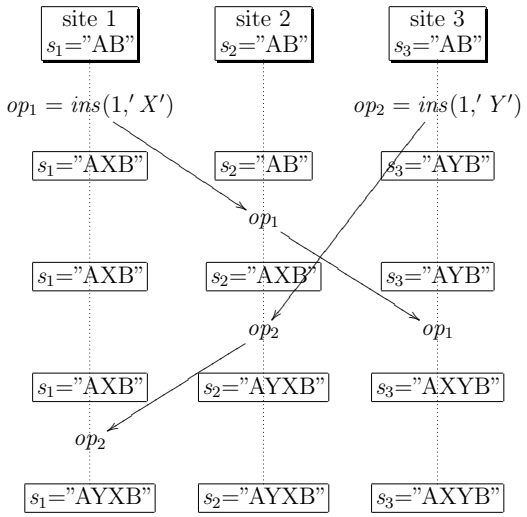


FIG. 1.3 – Préservation de l'intention

La figure 1.3 illustre la propagation de ces deux opérations sur les trois sites. Dès qu'une opération arrive sur un site, elle est exécutée telle quelle sans condition. Les états finaux "AXYB" et "AYXB" préservent les intentions de op_1 et op_2. En effet, les deux contraintes $A \prec X \prec B$ et $A \prec Y \prec B$ sont bien respectées. Enfin, l'effet de op_1 n'altère pas l'effet de op_2 et réciproquement. Malheureusement, les répliques ne convergent pas. Il est donc tout à fait possible de préserver les intentions et de ne pas converger.

1.3 Passage à l'échelle

Les éditeurs collaboratifs auxquels nous nous intéressons doivent supporter un grand nombre d'utilisateurs. Autrement dit, les mises à jour doivent être propagées à un grand nombre de répliques. Dans notre contexte, le nombre de sites n'est pas constant et la topologie d'interconnexion des sites est variable. Les participants à l'édition collaborative peuvent se connecter ou se déconnecter à tout moment, la collaboration ne doit pas être contrainte par une topologie rigide.

Les algorithmes de diffusion de groupes, tels que les algorithmes de diffusion totalement ordonnée [DSU04], sont connus pour passer difficilement l'échelle dans ces conditions. Seuls les algorithmes de diffusion à grande échelle tels que les algorithmes de propagation épidémique [DGH+87] sont adaptés au passage à l'échelle. Dans cette approche, chaque site participe à la diffusion. À intervalle régulier, chaque site choisit aléatoirement un ensemble de sites voisins vers lesquels il propage ses mises à jour. Il a été montré que ces algorithmes ont une forte probabilité à propager les mises à jour à toutes les sites.

Usenet [Sal92] a clairement montré sa capacité à passer à l'échelle. Il interconnecte aujourd'hui des milliers serveurs. Chaque serveur réplique tous les articles disponibles sur l'ensemble du réseau. Périodiquement, les nouveaux articles publiés sur un serveur sont propagés aux serveurs voisins. Chaque site stocke et transfère les articles reçus à ses propres voisins. Malheureusement, Usenet utilise la règle de Thomas pour assurer la convergence des répliques. Nous avons montré que cette règle ne préserve pas les intentions pour les opérations d'édition de texte.

Le système idéal pour l'édition collaborative massive serait donc semblable à Usenet mais avec une nouvelle règle qui, à la fois, assure la convergence et qui préserve les intentions.

1.4 Synthèse

Notre objectif est de concevoir un algorithme de réplication optimiste adapté à l'édition collaborative massive. Un tel algorithme devra préserver les intentions, assurer la convergence et passer à l'échelle.

Chapitre 2

État de l'art

Sommaire

Dans ce chapitre, nous nous intéressons aux systèmes d'édition collaborative et aux systèmes de réplication optimiste. Nous les évaluons par rapport à trois critères : convergence, préservation de l'intention et passage à l'échelle.

2.1 Wikiwikiweb

Les wikiwikiwebs [Cun05] ou Wikis constituent une famille d'éditeurs collaboratifs très populaires. Un wikiwikiweb est une application web permettant à un utilisateur d'éditer une page web depuis son navigateur internet.

Une page wiki peut être modifiée en cliquant sur le bouton "modifier" (voir figure 2.1)

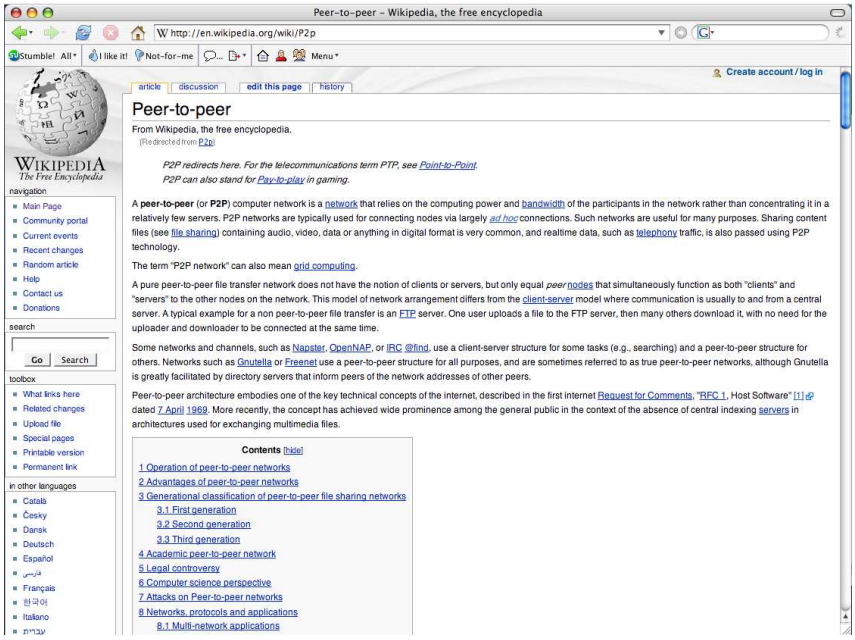


FIG. 2.1 – Une page wiki

Une page d'édition s'ouvre alors et permet l'édition selon la syntaxe wiki. Quand l'édition est terminée, il suffit de cliquer sur le bouton "sauvegarder" (cf figure 2.2). Chaque sauvegarde crée une nouvelle version de la page Wiki. Les utilisateurs voient seulement la dernière version de la page Wiki et peuvent éventuellement accéder aux versions antérieures si ils le désirent.

Deux utilisateurs peuvent modifier la même page en même temps. Deux sauvegardes seront donc effectuées et donc deux versions seront produites mais l'une après l'autre (voir figure 2.3). C'est donc le dernier utilisateur qui sauve qui "gagne". Sur notre figure 2.3, les changements de la version $v11$ masquent les changements de la version $v12$.

Si l'utilisateur de la version $v11$ veut rendre visible ses modifications, il doit produire une nouvelle version qui contiendra les changements de la version $v11$ et de la version $v12$.

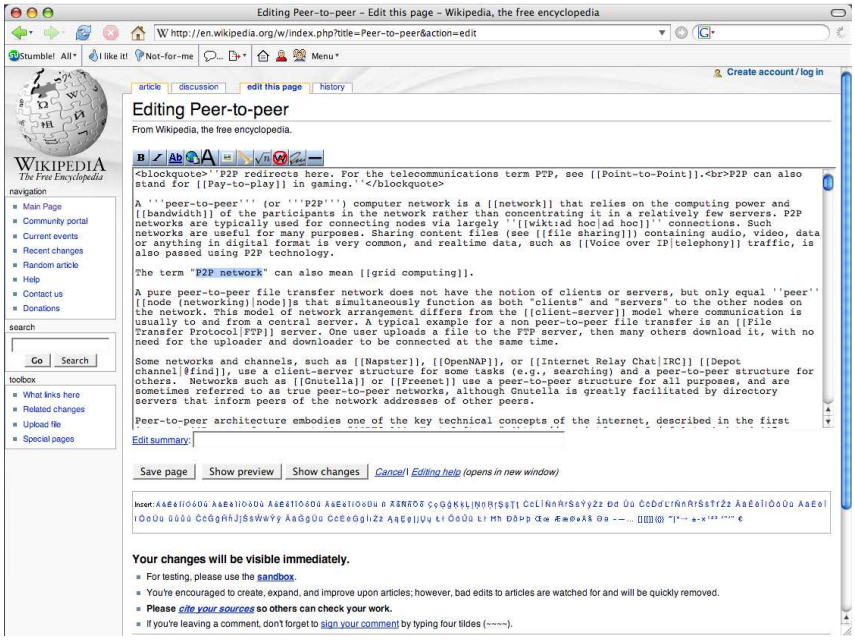


FIG. 2.2 – Editer une page wiki

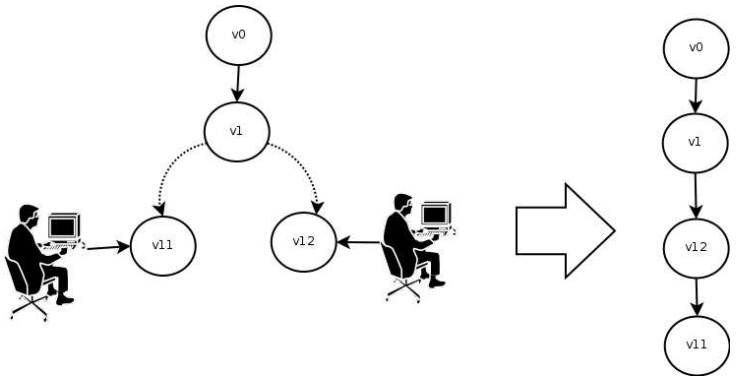


FIG. 2.3 – Edition concurrente d'une page Wiki

△ △ △

Nous évaluons un Wiki par rapport à nos critères :

Convergence On ne peut pas vraiment parler de réplication optimiste dans le cas du Wiki. Un serveur Wiki est une application centralisée sur un serveur web. Une page Wiki est tout de même dupliquée pendant une session d'édition sans mécanisme de verrouillage. En cas d'édition concurrente, la valeur finale est celle de celui qui sauvegarde en dernier. Dans ce cas, le système ne peut que converger.

Intention Bien évidemment, ce système ne préserve pas les intentions. Les effets de l'édition d'une page changent les effets de l'édition concurrente de la même page.

Passage à l'échelle Un wiki est une application web centralisée. Le serveur web hébergeant le service Wiki est un point de congestion.

2.2 Réplication optimiste et synchroniseurs

Les synchroniseurs de fichiers ou de données peuvent être considérés comme des systèmes de réplication optimiste. En effet, des outils comme ActiveSync [Mic05a], iSync [App05], HotSync [Pal05] permettent de dupliquer des données sur un appareil mobile et d'assurer la cohérence à terme avec une station maître. Les données peuvent être modifiées de manière concurrente sur la station maître et sur le mobile. Elles seront réconciliées plus tard. Dans ces outils, deux mises à jour sont dites conflictuelles si les effets de l'une interfèrent avec les effets de l'autre. Dans la plupart des cas, l'outil propage les mises à jour non conflictuelles et demande à l'utilisateur de trancher les mises à jour conflictuelles.

Tous les synchroniseurs reposent sur les même principes. Dans la suite, nous avons choisi de nous intéresser à l'un d'entre eux : le synchroniseur Unison.

Unison [BP98, PV04] est un synchroniseur de fichiers. Les auteurs définissent de façon formelle ce que sont des mises à jour conflictuelles et quel doit être le comportement du synchroniseur dans ce cas.

Le processus de réconciliation, mis en œuvre dans Unison, se déroule de la manière suivante.

- Sur chaque copie, Unison compare son fichier d'archive avec l'état courant de la copie pour déterminer quels sont les chemins qui ont été mis à jour. Le fichier d'archive conserve l'état de chaque chemin lors de la dernière synchronisation.
- Unison résoud, ensuite, les situations dites de *faux conflit*. Autrement dit, les chemins qui ont été modifiés sur les deux copies et qui possèdent actuellement des valeurs identiques. Ces chemins sont immédiatement marqués comme synchronisés dans le fichier d'archive.
- Unison présente à l'utilisateur la liste de chemins qui ont été mis à jour. Pour les mises à jour non conflictuelles, il propose une action par défaut, propager le nouveau

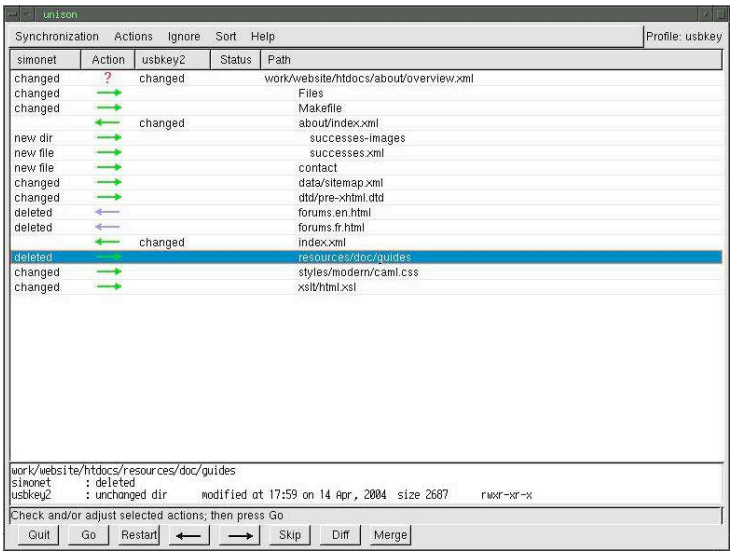


FIG. 2.4 – Unison : Résolution des conflits.

contenu de la copie modifiée vers l'autre copie. Les mises à jour conflictuelles sont uniquement affichées.

– L'utilisateur peut alors choisir (cf. figure 2.4) de ne pas propager certaines mises à jour et d'appliquer certaines procédures de résolution de conflit : forcer la recopie d'une valeur, appeler un outil de fusion ad-hoc.
– Unison exécute, une par une, les actions décidées par l'utilisateur et met à jour son fichier d'archive pour refleter le nouvel état de la copie.

Selon Unison, un synchroniseur sûr doit :

a) conserver les modifications faites par l'utilisateur. Cette propriété garantit donc qu'aucune modification ne sera perdue durant la synchronisation ;

b) propager uniquement des modifications effectuées par un utilisateur. Autrement dit, un synchroniseur ne doit pas générer de modifications autres que celles effectuées par l'utilisateur ;

c) s'arrêter lors d'un conflit.

△ △ △

Nous évaluons maintenant Unison par rapport à nos critères

Convergence Les auteurs d'Unison préfèrent ne pas converger plutôt que de converger vers un état où un conflit serait tranché de manière arbitraire. Unison n'assure donc pas la convergence dans le cas général.

Intention Unison définit un système de fichiers de façon formelle. Un système de fichiers est muni entre autres d'une opération $mkdir(path)$ pour créer un répertoire et $mkfile(path)$ pour créer un fichier. Si une opération $mkdir(/a)$ est concurrente à

une opération *mkfile*(/*a*), il est clair que l'exécution de l'une empêche l'exécution de l'autre. Les intentions de *mkdir* et *mkfile* ne peuvent être préservées.

Passage à l'échelle La convergence n'étant pas garantie pour deux répliques, évaluer les capacités de passage à l'échelle d'Unison est sans intérêt.

2.3 Réplication optimiste et gestionnaires de configuration

Les gestionnaires de configuration, tels que CVS [Ber90], Subversion [Col05], ou encore ClearCase [IBM05, AFK+95] permettent à des utilisateurs de partager des documents et de les éditer de manière collaborative. Ces environnements intègrent un système de gestion de versions qui repose sur un protocole optimiste de gestion des accès concurrents : le paradigme du *Copier-Modifier-Fusionner*.

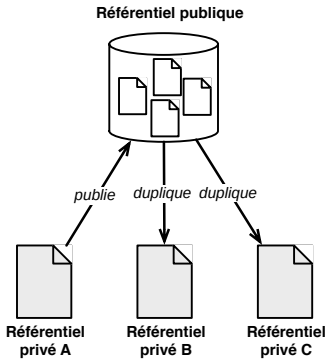


Fig. 2.5 – référentiel/espaces de travail

Ce modèle est un modèle centralisé. On distingue deux types d'espaces (cf. figure 2.5) :

un référentiel. Cet espace maintient un ensemble multi-versionné des objets partagés. C'est à dire que pour chaque objet partagé, il conserve l'ensemble des versions de cet objet dont la publication par les utilisateurs a réussi ;

des espaces de travail. Chaque utilisateur possède son propre espace de travail. Les modifications apportées dans cet espace restent confidentielles jusqu'à leur publication.

Pour modifier un objet, l'utilisateur en crée une réplique dans son espace de travail. Il peut ensuite librement la modifier. Une fois sa tâche terminée, il publie une nouvelle version dans le référentiel.

Deux utilisateurs peuvent modifier en parallèle deux répliques du même objet. Dans ce cas, le premier qui a fini peut publier sans problème. Le second est forcé d'intégrer les modifications du premier avant de publier à son tour.

La figure 2.6 illustre le comportement du modèle avec deux utilisateurs “foo” et “bar” :

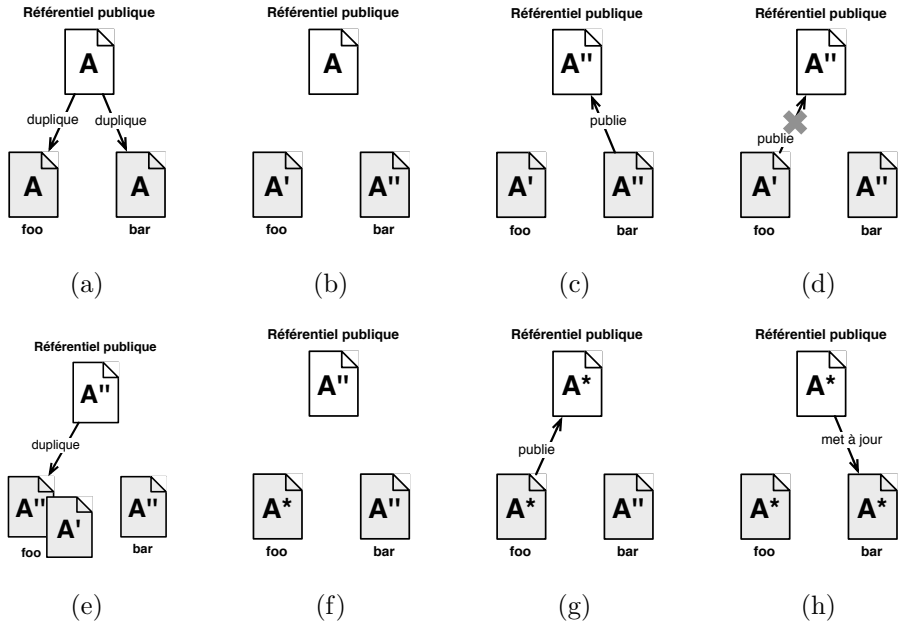


FIG. 2.6 – Le modèle “Copier-Modifier-Fusionner”

(a) les utilisateurs “foo” et “bar” créent des répliques dans leurs espaces de travail respectifs ;

(b) “foo” et “bar” travaillent librement sur leurs répliques ;

(c) “bar” valide ses modifications en créant des nouvelles versions des objets dans le référentiel ;

(d) si “foo” tente de publier ses modifications, il obtient une erreur lui signifiant qu'il doit obligatoirement consulter les dernières versions publiées.

(e) “foo” recopie donc les nouvelles versions des objets validées par “bar”. A cet instant, “foo” ne peut toujours pas valider ses changements. En effet, il existe des nouvelles versions des objets dont il n'a pas pris connaissance ;

(f) “foo” intègre les modifications de “bar” en fusionnant ses copies locales avec les versions qu'il vient de rapatrier. Pour cela, il utilise un outil dédié à cette tâche comme Diff3 ;

(g) “foo” valide ses modifications, et, crée donc de nouvelles versions des objets intégrant les modifications des deux utilisateurs.

(h) “bar” intègre les nouvelles modifications publiées. Les deux copies ont convergé.

△ △ △

Nous évaluons les gestionnaires de configuration par rapport à nos critères :

Convergence CVS assure la convergence des répliques. Cette convergence a cependant un coût élevé. En effet, dans un système où n répliques sont modifiées en parallèle, il

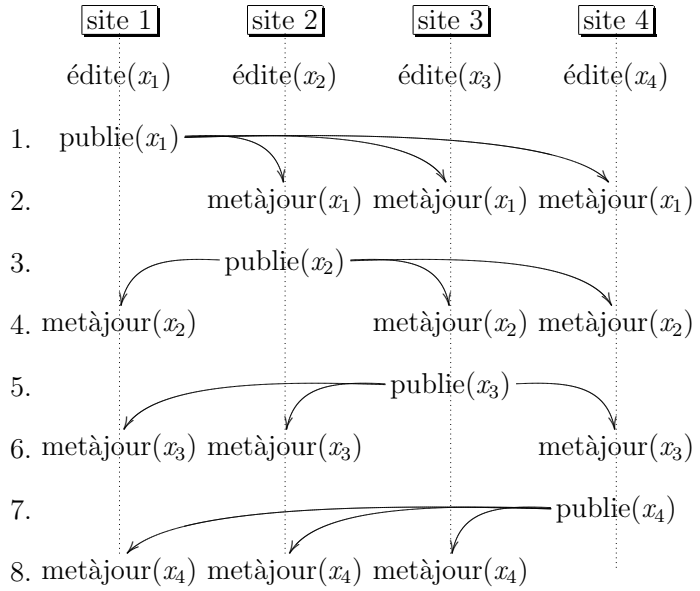


FIG. 2.7 – Convergence avec CVS

faut $2 * n$ étapes pour atteindre la convergence. La figure 2.7 présente les différentes étapes nécessaires à la convergence de quatre répliques.

Intention Le scénario de la figure 2.8 montre que CVS ne préserve pas les intentions. On considère deux répliques de la version `v0` d'un fichier texte. Ces deux répliques sont modifiées en parallèle. Sur la première réplique, on insère la ligne de texte `97. Enough [...]`. Tandis que, sur la seconde réplique, on supprime la ligne `35. Peace is [...]`. On obtient les états des répliques `v1` et `v2`. Si l'on réconcilie les deux répliques en utilisant CVS. On publie l'état `v2` de la seconde réplique, puis on met à jour la première réplique. Lors de cette mise à jour, les modifications concurrentes sont automatiquement fusionnées en utilisant la commande `diff3 --merge v2 v0 v1`.

Cet outil détecte un conflit, c'est-à-dire deux modifications dont les effets interfèrent, alors que les modifications n'étaient pas conflictuelles. La résolution de ce faux conflit conduit à un état qui ne respecte pas les intentions. Pourtant, ces deux répliques devraient converger vers le contenu suivant :

```
Rules of Acquisition
 97. Enough... is never enough.
242. More is good. All is better.
```

Passage à l'échelle Le paradigme Copier-Modifier-Fusionner associe à des outils de fusion déterministes un ordre total. Ainsi, chaque opération reçoit un numéro d'intégration avant d'être publiée. Une opération ne peut être publiée que si toutes les opérations précédentes ont été reçues. Cette stratégie fonctionne mais introduit un ordre total.

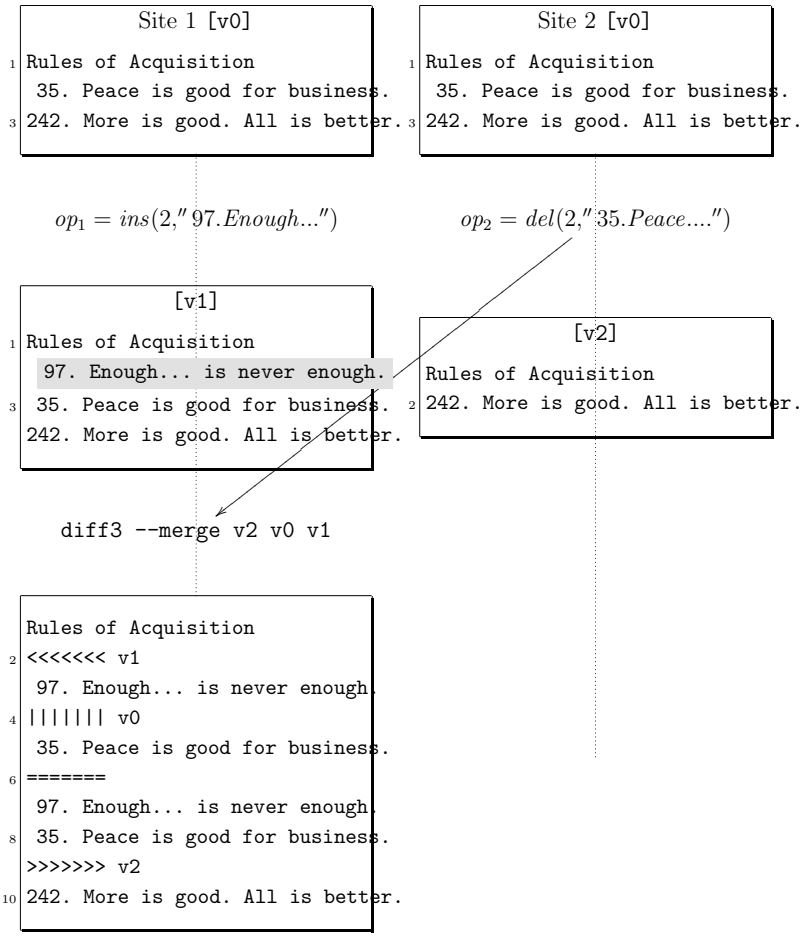


FIG. 2.8 – Violation de l'intention dans CVS

Cependant, maintenir un ordre total a un coût élevé. Soit cet ordre est maintenu par un site central avec les risques intrinsèques de pannes, soit il est maintenu avec des algorithmes de consensus distribués asynchrones. Dans ce cas, le système peut être tolérant aux pannes mais passe difficilement l'échelle [Lyn96].

2.4 Réplication optimiste et bases de données

Oracle [DDD+94, Ora01] permet de répliquer de manière optimiste un ensemble de tables sur plusieurs serveurs. Dans ce cas, des mises à jour concurrentes peuvent être faites sur chacun des serveurs et les répliques peuvent diverger. Les serveurs se réconcilient ensuite deux à deux.

Oracle détecte les mises à jour concurrentes et propose un ensemble de procédures de réconciliation. Le tableau 2.1 disponible dans la documentation Oracle [Ora01] détaille l'ensemble de ces procédures.

Type de conflit	Méthode de résolution	Converge avec 2 sites	Converge avec n sites
MISE À JOUR	valeur minimum	oui	oui, si toujours décroissante
	valeur maximum	oui	oui, si toujours croissante
	estampille la plus ancienne	oui	non
	estampille la plus récente	oui	oui, avec une méthode de secours
	priorité sur les valeurs	oui	oui, si toujours croissante
	priorité sur les sites	oui	non
	moyenne (numérique)	non	non
	addition (numérique)	oui	oui
INSERTION	concaténation du nom du site	non	non
	concaténation d'un numéro séquentiel	non	non
	ignore	non	non
SUPPRESSION	aucune		

Tab. 2.1 – Procédures de réconciliation dans Oracle

Oracle différencie les opérations de mises à jour, d'insertion et de destruction. Il propose des solutions assurant la convergence pour les mises à jour conflictuelles mais pas pour les insertions conflictuelles ou les destructions conflictuelles. La convergence n'est donc pas garantie dans tous les cas.

Si nous regardons maintenant le cas des mises à jour conflictuelles, la convergence à n sites n'est garantie de manière inconditionnelle qu'avec le + arithmétique sur les entiers. La documentation d'Oracle précise que la convergence à n sites n'est assurée que si la suite des mises à jour est croissante.

Ces restrictions viennent en fait de la façon dont Oracle détecte les conflits, c'est-à-dire deux mises à jour dont les effets interfèrent. Quand une opération de mise à jour est envoyée, l'ancienne valeur est jointe à la nouvelle. Oracle détecte un conflit si l'ancienne valeur de l'opération reçue ne correspond pas à la valeur actuelle. Dans ce cas, la procédure de réconciliation est appelée avec les deux valeurs en conflit.

Le tableau 2.2 illustre pourquoi *max* ne peut pas assurer la convergence si la séquence de mises à jour n'est pas croissante. Le scénario est basé sur trois sites répliquant un entier initialisé à 5. Deux opérations op_1 et op_2 sont générées. La panne [1] sur le site 3 force ce site à recevoir op_2 suivie de op_1 alors que op_2 a été produite après op_1. Dans l'étape 8,

[1] Cette panne n'est pas obligatoire, le site 3 pourrait tout à faire recevoir l'opération op_1 après op_2.

étape	Action	Site 1	Site 2	Site 3
0	état initial	5	5	5
1	Site 3 en panne	5	5	*HS*
2	Site 1 : $op_1 = 5 \rightarrow 7$	7	5	*HS*
3	Site 1 emet 7	7 $\xrightarrow{op_1=5\rightarrow 7}$	5	*HS*
4	Site 2 reçoit op_1	7	7	*HS*
5	Site 3 revit	7	7	5
6	Site 1 : $op_2 = 7 \rightarrow 2$	2	7	5
7	Site 1 émet op_2 vers sites 2 et 3	2 $\xrightarrow{op_2=7\rightarrow 2}$	7	$\overset{op_2=7\rightarrow 2}{\frown}$ 5
8	Site 3 reçoit op_2, détecte un conflit et applique $max(2,5) = 5$	2	2	$max(2,5) = 5$
9	Site 3 reçoit $op_1 = 5 \rightarrow 7$ (pas de conflit)	2	2	$\overset{op_1=5\rightarrow 7}{\frown}$ 7

TAB. 2.2 – détection et réconciliation dans Oracle avec la procédure de réconciliation *max*

le site 3 reçoit op_2, l'ancienne valeur de op_2 n'est pas égale à 5, Oracle détecte donc un conflit et appelle la procédure de réconciliation déclarée c'est-à-dire $max(2,5) = 5$. Puis le site 3 reçoit op_1, par coïncidence, il ne détecte pas de conflit donc applique directement 7. Les sites ont définitivement divergé.

△ △ △

Évaluons maintenant la réplication optimiste d'Oracle par rapport à nos critères :

Convergence Oracle n'assure pas la convergence dans le cas général.

Intention Une procédure de réconciliation peut être compatible avec l'intention des opérations utilisées dans une application base de données. Par exemple, nous répliquons

un entier muni d'une opération d'incrémentation. Si l'effet se limite à observer la croissance de cet entier, alors la procédure de réconciliation *max* ou + préserve les intentions.

Passage à l'échelle Oracle repose sur une propagation épidémique des mises à jour. Il ne requiert ni serveur central, ni ordre total. Le système de réplication optimiste d'Oracle passe donc à l'échelle même si il n'assure pas la convergence.

2.5 Réplication optimiste et systèmes répartis

2.5.1 Bayou

Bayou [PST+97, TTP+95] est un système de gestion de données pour des applications collaboratives dans un environnement mobile. Les données sont répliquées entre différents serveurs.

Dès le moment où un serveur reçoit une opération, il tente de l'exécuter. Deux sites peuvent donc recevoir et exécuter les même opérations dans un ordre différent. Pour assurer la convergence, les serveurs doivent exécuter toutes les opérations dans le même ordre. Dans Bayou, les serveurs vont continuellement défaire et rejouer les opérations au fur et à mesure qu'ils prennent connaissance de l'ordre final. Cet ordre final est décidé par un serveur principal désigné au lancement du système.

Ainsi, chaque serveur maintient un journal des opérations exécutées. Ce journal est scindé en deux parties. Un préfixe p qui contient les opérations validées par le serveur principal. Ces opérations sont ordonnées définitivement selon un ordre total introduit lors de la validation par le serveur principal. Le reste du journal, qualifié de "provisoire", est ordonné selon un ordre total qui peut être remis en cause au fur et à mesure que de nouvelles opérations sont reçues.

La figure 2.9 présente une opération d'écriture dans Bayou. Cette opération comprend la mise à jour à effectuer (`update`), la pré-condition de l'opération (`dependency check`) qui détermine si l'opération peut être exécutée sur l'état actuel, et la procédure de réconciliation (`mergeproc`). Quand cette opération s'exécute, soit la pré-condition est vraie et la mise à jour est effectuée telle quelle. Soit le pré-condition est fausse, et dans ce cas, la procédure de fusion est exécutée.

Dans cet exemple, cette opération réserve un créneau pour une réunion. Si la pré-condition détermine que ce n'est pas possible, c'est-à-dire qu'il existe d'autres réunions sur ce créneau, alors la procédure de réconciliation déplace la réunion à une autre dates. La pré-condition de cette opération est : "Il n'existe pas d'autres réunions à cette date". Si c'est faux, la procédure de réconciliation déplace la date de réunion à d'autres dates fournies par l'utilisateur. Si aucun créneau libre ne peut être trouvé, alors l'opération est consignée dans un journal dédié. Cette opération sera traitée ultérieurement par un administrateur.

```
1 Bayou_Write = {
    update = {insert, Meeting, 12/18/95, 1:30pm, 60mn, ‘‘Budget Meeting’’},
3
    dependency_check = {
5     query = ‘‘SELECT key FROM Meetings WHERE day = 12/18/95
                AND start < 2:30pm AND end > 1:30pm’’,
7     expected_result = EMPTY },

9   mergeproc = {
      alternates = {{12/18/95, 3:00pm}, {12/19/95, 9:30am}};
11    newupdate = {};
      FOREACH a IN alternates {
13      #check if there would be a conflict
        IF (NOT EMPTY(
15            SELECT key FROM Meetings WHERE day = a.date
              AND start < a.time + 60mn AND end > a.time))
17          CONTINUE;
        #no conflict, can shedule meeting at that time
19      newupdate = {insert, Meetings, a.date, a.time, 60mn, ‘‘Budget Meeting’’};
        BREAK;
21    }
      IF (newupdate = {}) # no alternate is acceptable
23        newupdate = {insert, ErrorLog, 12/18/95, 1:30pm, 60mn, ‘‘Budget Meeting’’};
      RETURN newupdate;
25  }
}
```

FIG. 2.9 – Une opération d'écriture dans Bayou.

△ △ △

Convergence Si l'on ne considère que les opérations dans le préfixe p géré par un ordre total, Bayou assure la convergence. Par contre, la partie du journal gérée selon un ordre total provisoire ne garantit rien. Elle permet cependant à l'utilisateur de voir des opérations qui pourront être validées par la suite. Cependant, ces opérations étant inéluctablement incluses dans le préfixe p, cette partie provisoire du journal n'a aucune incidence sur la convergence des répliques.

Intention Une opération de Bayou contient la déclaration explicite de ses intentions. Dans l'exemple présenté, les alternatives déclarées à la date initiale de réunion sont les effets que l'on accepte d'observer. L'ordre total couplé à ces opérations assure convergence et le respect des intentions.

Passage à l'échelle La mise en place du préfixe commun nécessite un ordre total. Cet ordre limite le passage à l'échelle de Bayou.

2.5.2 IceCube

IceCube [KRSD01, PSM03, SPO04] est un système de réconciliation générique. Ce système traite la réconciliation comme un problème d'optimisation : celui d'exécuter une combinaison optimale de mises à jour concurrentes.

À partir des différents journaux présents sur les différents sites, IceCube calcule un journal optimal unique contenant le nombre maximum de mises à jour non conflictuelles. Pour cela, IceCube utilise la sémantique, des mises à jour, exprimée sous forme de contraintes.

Dans IceCube, les mises à jour sont modélisées par des actions. Une action réifie une opération et est composée des éléments suivants :

une ou plusieurs cibles : elles identifient les objets accédés par l'action ;

une pré-condition : elle permet de détecter les conflits éventuels lors de l'exécution au cours de la phase de simulation. Les pré-conditions sont exprimées dans un langage de logique du premier ordre ;

une opération : un sous programme accédant aux objets partagés ;

des données privées : une liste de données propres à l'action. Il y a au moins les paramètres et le type de l'opération.

Le système permet de définir des dépendances sémantiques entre les actions sous forme de contraintes. Deux types de contraintes sont disponibles : les contraintes statiques et les contraintes dynamiques. Les contraintes statiques sont évaluées sans utiliser l'état des objets. Les contraintes dynamiques peuvent utiliser l'état des objets.

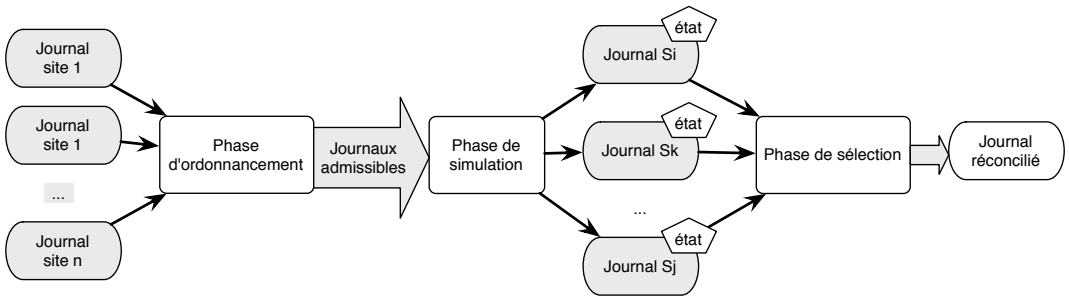


FIG. 2.10 – Processus de réconciliation dans IceCube.

La réconciliation se déroule en trois phases illustrées par la figure 2.10 :

une phase d'ordonnancement : durant cette phase, le système construit toutes les exécutions possibles en combinant les différentes actions issues des journaux des différents sites. L'espace de recherche est limité par les contraintes statiques.

une phase de simulation : on exécute les différents ordonnancements trouvés dans la phase précédente. Si une contrainte dynamique est violée alors l'ordonnancement en question est abandonné et rejeté. Après cette phase, il ne reste plus que les ordonnancements respectant les contraintes statiques et dynamiques.

une phase de sélection : la phase de sélection classe les ordonnancements restants et doit en choisir un. Le critère de sélection est choisi par un administrateur.

Cette description est une vision simplifiée du fonctionnement de IceCube. Dans IceCube, les trois phases décrites précédemment ne sont pas exécutées de manière séquentielle mais elles sont menées en parallèles.

Les contraintes statiques limitent la taille de l'espace de recherche, mais celui-ci reste trop grand pour effectuer une recherche exhaustive. C'est pourquoi, IceCube utilise une heuristique dont les solutions sont proches de l'optimum.

△ △ △

Nous évaluons maintenant IceCube selon nos critères :

Convergence IceCube assure la convergence.

Intention Si un type d'objet répliqué muni de ses opérations est défini, alors il est possible de déclarer les intentions des opérations à l'aide des contraintes statiques et dynamiques d' IceCube. Si ces intentions sont compatibles avec la préservation de l'intention, alors le moteur de résolution de contraintes d'IceCube va trouver toutes les ordonnancements préservant les intentions.

Passage à l'échelle Pour faire converger n sites, IceCube désigne un site comme responsable de la réconciliation. Tous les sites doivent envoyer les opérations effectuées depuis la dernière réconcilication vers ce site. L'algorithme principal d'IceCube peut alors calculer un journal reconcilié. Ce journal est ensuite renvoyé à tous les sites. Chaque site défait ses modifications locales avant réconciliation et rejoue le journal réconcilié. Cette façon de faire ne passe pas l'échelle.

2.6 Réplication optimiste et transformées opérationnelles

Le modèle des transformées opérationnelles a été développé par la communauté des éditeurs collaboratifs synchrones. La préservation de l'intention est au centre de la définition du modèle de cohérence de cette approche.

L'architecture générale du modèle des transformées opérationnelles distingue deux composants :

– un algorithme d'intégration. Il est responsable de la réception, de la diffusion et de l'exécution des opérations. Si nécessaire, il fait appel aux fonctions de transforma-

tion. Cet algorithme est indépendant du type des données manipulées (chaîne de caractères, document XML, système de fichiers).
– un ensemble de fonctions de transformation. Ces fonctions ont la charge de "fusionner" les mises à jour en sérialisant deux opérations concurrentes. Ces fonctions sont spécifiques à un type de données particulier.

L'algorithme d'intégration est défini comme correct si il assure la Causalité, la Convergence et la préservation des Intentions (CCI) [SCF97, SZJY97, SE98].

2.6.1 Respect de la causalité

Pour certaines opérations, il existe une relation de précédence causale qui doit être maintenue. On dit que l'opération op_1 *précède* l'opération op_2 (noté $op_1 \rightarrow op_2$) si et seulement si op_2 a été générée et exécutée sur une réplique après l'exécution de op_1 sur cette même réplique. Puisque op_2 a été générée après op_1, elle tient compte implicitement de ces effets, c'est-à-dire que l'on suppose que cette opération op_2 est dépendante des effets produits par l'opération op_1.

Le respect de la causalité garantit que pour les opérations pour lesquelles il existe une relation de causalité, celles-ci seront exécutées dans le même ordre sur toutes les répliques.

Par exemple, nous considérons le scénario présenté à la figure 2.11.

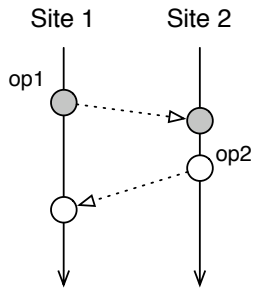


FIG. 2.11 – Exemple de relation de dépendance précédence.

Puisque l'opération op_2 a été générée sur le site 2 après que l'opération op_1 se soit exécutée, op_1 précède op_2. Si sur un troisième site, l'opération op_2 est reçue avant l'opération op_1 alors pour garantir l'exécution des opérations selon l'ordre de précédence, l'exécution de op_2 sera différée jusqu'à la réception et l'exécution de op_1.

Généralement, cette dépendance est maintenue en utilisant des vecteurs d'horloges [Mat89, Fid91]. Dans un système à n sites, un tel vecteur V possède n composantes. Chaque composante $V[i]$ compte le nombre d'opérations issues du site i qui ont été déjà exécutées sur le site. Lorsqu'une opération op est générée sur un site i, la composante $V[i]$ est incrémentée de 1. Une copie V_{op}, valeur de V après la génération de op, est alors associée à l'opération avant sa diffusion aux autres sites. Lors de la réception de cette opération sur un site j, si le vecteur V_{s_j} du site "domine" le vecteur V_{op} de l'opération, alors

l'opération est prête à être exécutée. Dans le cas contraire, son exécution doit être différée. On dit qu'un vecteur V_1 domine un vecteur V_2 si et seulement si on a $\forall\, i V_1[i] \geq V_2[i]$. Lors de l'exécution sur un site j d'une opération prête provenant d'un site i, pour maintenir le vecteur V_{s_j} d'horloges du site j, il faut incrémenter sa $i$ème composante de 1.

Deux opérations op_1 et op_2 qui ne sont pas liées par une relation de précédence (ni $op_1 \rightarrow op_2$, ni $op_2 \rightarrow op_1$) sont concurrentes (noté $op_1 \parallel op_2$).

Plus précisément, pour deux opérations quelconques op_1 et op_2, issues respectivement du site S_{op_1} et S_{op_2}, munies de leur vecteur d'horloges respectif V_{op_1} et V_{op_2}, on peut déduire :

- $op_1 \rightarrow op_2$ si et seulement si $V_{op_2}[S_{op_1}] > V_{op_1}[S_{op_1}]$.
- $op_1 \parallel op_2$ si et seulement si $V_{op_2}[S_{op_1}] \leq V_{op_1}[S_{op_1}] \wedge V_{op_2}[S_{op_2}] \geq V_{op_1}[S_{op_2}]$.

2.6.2 Préservation de l'intention

Deux opérations op_1 et op_2 qui ne sont pas liées par une relation de précédence sont concurrentes. De ce fait, ces deux opérations peuvent être exécutées sur les différentes répliques dans un ordre quelconque. Cependant, si l'on exécute l'opération op_1 avant l'opération op_2 alors il faudra lors de l'exécution de op_2 tenir compte des effets de op_1 de façon à respecter les effets de op_2.

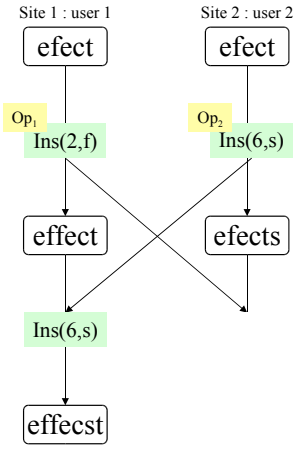


FIG. 2.12 – Intégration incorrecte

La figure 2.12 illustre un cas de violation de l'intention. Dans cet exemple, deux sites $site_1$ et $site_2$ partagent une chaîne de caractères dont la valeur initiale est `efect`. Les deux répliques sont modifiées en parallèle : sur le $site_1$, la chaîne est devenue `effect`, elle résulte de l'exécution de l'opération $op_1 = Ins(2, f)$ qui a inséré le caractère `f` en deuxième position dans la chaîne. Sur le $site_2$, la réplique a pour nouvelle valeur `efects` qui est le résultat de l'exécution de l'opération $op_2 = Ins(6, s)$.

Lorsque op_2 est reçue et exécutée sur le $site_1$, elle produit l'état `effecst` qui n'est pas l'état attendu. En effet, dans ce cas précis, l'effet de l'opération op_2 n'a pas été respecté : il s'agissait d'insérer le caractère `s` à la fin de la chaîne.

Dans le modèle des transformées opérationnelles, les opérations reçues doivent être transformées par rapport aux opérations locales concurrentes avant d'être exécutées.

Cette transformation est réalisée par des fonctions de transformation. Une fonction de transformation, notée T, prend en paramètre deux opérations concurrentes op_1 et op_2 définies *sur le même état* s et retourne en résultat une opération op'_1. op'_1 est une opération équivalente à op_1 mais définie sur l'état s'. s' étant l'état résultant de l'exécution de op_2 sur l'état s.

La fonction de transformation a été introduite par Ellis et al. [EG89] et reprise dans de nombreuses autres publications [RNRG96, SZJY97, SCF97] sous différentes appellations. Dans la suite du document, nous utiliserons la dénomination utilisée par Suleiman [SCF97] à savoir *transposition en avant*.

Définition 2.1 (Fonctions de transformation respectant l'intention) Soient S_i l'état d'une réplique, op_1 et op_2 deux opérations concurrentes définies sur cet état, l'état S_{i+1} résultant de l'exécution de op_2 sur S_i. On nomme *transposition en avant*, notée $T(op_1, op_2) = op_1{}^{op_2}$, la fonction qui calcule l'opération op'_1 définie sur l'état S_{i+1} qui a les mêmes effets sur cet état que ceux qu'avaient l'opération op_1 sur S_i.

Autrement dit, si l'on note $S \odot op$ l'exécution d'une opération op sur un état S, et $Intention(op, O)$ les effets de l'exécution d'une l'opération op sur un état O, on a :

$$Intention(op_1, O_i) = Intention(T(op_1, op_2), O_{i+1})$$

où $O_{i+1} = O_i \odot op_2$.

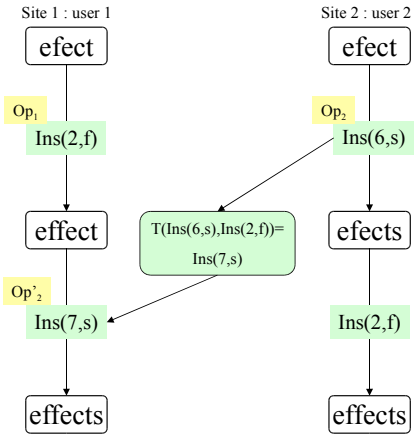


Fig. 2.13 – Intégration avec transformation

La figure 2.13 utilise la fonction de transformation de la figure 2.14.

```
T(Ins(p1,c1),Ins(p2,c2)):−
  if (p1 < p2) then
    return Ins(p1, c1)
  else
    return Ins(p1 + 1, c1)
  endif
```

FIG. 2.14 – Fonction de transformation naïve

Cette fois ci, lorsque op_2 est reçue par le $site_1$, elle doit être transformée par rapport à op_1. L'algorithme d'intégration applique donc la transformation suivante :

$$T(\overbrace{Ins(6,s)}^{op_2}, \overbrace{Ins(2,f)}^{op_1}) = \overbrace{Ins(7,s)}^{op_2'}$$

La position d'insertion de op_2 est alors incrémentée puisque l'opération op_1 a inséré le caractère `f` avant le point d'insertion de `s` sur l'état `efect`. Ensuite, op_2' est exécutée sur le site $site_1$. De la même façon, lorsque op_1 est reçue sur le site $site_2$, la transformation suivante est effectuée :

$$T(\overbrace{Ins(2,f)}^{op_1}, \overbrace{Ins(6,s)}^{op_2}) = \overbrace{Ins(2,f)}^{op_1'}$$

Dans ce cas, la fonction de transformation retourne l'opération $op_1' = op_1$ puisque, la lettre `f` est insérée avant la lettre `s`. Au final, sur les deux sites, les effets des deux opérations ont bien été conservés.

Dans le modèle OT, les intentions d'une opération ne sont pas exprimées de manière explicite. Elles apparaissent de manière implicite dans le code des fonctions de transformation. Le développeur de fonction de transformation doit écrire des fonctions qui assurent la préservation de l'intention comme dans la figure 2.14.

2.6.3 Convergence des répliques

Le fait de respecter les relations de causalité entre les opérations et de préserver les effets des opérations ne suffit pas pour garantir la convergence des répliques. La figure 2.15 donne un exemple de scénario où les deux critères sont respectés mais où les copies ne convergent pas.

Dans cet exemple, nous considérons deux répliques d'une chaîne de caractères de valeur initiale `ab`. Sur la première réplique, un utilisateur insère le caractère `x` entre les caractères `a` et `b`. En parallèle, sur la seconde réplique, un autre utilisateur insère au même emplacement le caractère `y`.

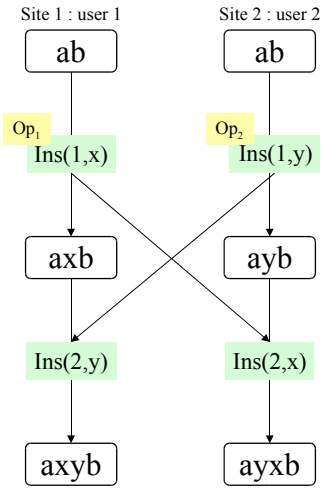


FIG. 2.15 – Divergence des copies due à la violation de C_1

Pour intégrer les opérations sur les différentes répliques, nous utilisons la fonction de transformation définie à la figure 2.14. Sur la première réplique, l'intégration de l'opération op_2 revient à décaler la position d'insertion d'une position. Nous obtenons alors la chaîne `axyb`. L'effet d'op_2 a été préservé puisque le caractère `y` est bien inséré entre les caractères `a` et `b`. En procédant de la même manière sur la seconde réplique, après avoir intégré l'opération op_1, nous obtenons l'état `ayxb`. L'effet de l'opération op_1 a également été préservé. Cependant, bien que les effets des deux opérations soient préservés, les deux répliques n'ont pas convergé.

Il a été montré [EG89] que pour garantir la convergence des copies, les fonctions de transformation doivent vérifier la condition suivante.

DÉFINITION 2.2 (CONDITION C_1) Soient op_1 et op_2 deux opérations concurrentes définies sur le même état. La transposée en avant T vérifie la condition C_1 si et seulement si :

$$op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$$

où $\circ$ dénote un opérateur concaténant deux opérations pour former une séquence d'opérations, et où $\equiv$ signifie que les deux séquences $op_1 \circ T(op_2, op_1)$ et $op_2 \circ T(op_1, op_2)$ produisent le même état.

Dans notre exemple, les opérations op_1 et op_2 sont concurrentes et définies sur le même état `ab`. Pourtant, l'exécution de la séquence d'opération $op_1 \circ T(op_2, op_1)$ sur cet état donne l'état `axyb` tandis que l'exécution de la séquence $op_2 \circ T(op_1, op_2)$ donne l'état `ayxb`. Les deux états ne sont pas égaux ce qui prouve la violation de la condition C_1.

Pour satisfaire cette condition, il faut que la fonction de transposition en avant prenne en compte le cas particulier de deux insertions à la même position dans la chaîne de caractères. Étant donné que rien ne nous permet de décider lequel des deux caractères

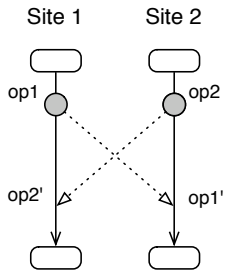


FIG. 2.16 – Condition C_1.

```
T(Ins(p1,c1),Ins(p2,c2)):−
  if (p1 < p2) then
    return Ins(p1, c1)
  else if (p1 > p2) then
    return Ins(p1 + 1, c1)
  else if (p1 = p2) then
      if (c1 < c2) then
         return Ins(p1, c1)
      else
         return Ins(p1 + 1, c1)
      endif
  endif
```

FIG. 2.17 – Fonction de transformation vérifiant la condition C_1.

doit être placé devant l'autre, nous devons prendre un choix arbitraire. Et, nous devons garantir que ce choix sera le même sur toutes les répliques.

La figure 2.17 présente un exemple de fonction de transformation vérifiant la condition C_1. Pour "départager" les deux opérations, lors de l'insertion à la même position, nous comparons les caractères. L'insertion dont le caractère est le plus petit selon l'ordre lexicographique s'effectue devant l'autre caractère.

La figure 2.18 illustre l'utilisation de cette nouvelle fonction de transformation sur le scénario précédent. Sur la première réplique, lors de l'intégration de op_2, l'opération voit sa position d'insertion incrémenter puisque le caractère `y` est "plus grand" selon l'ordre lexicographique que le caractère `x` inséré par op_1. Sur la seconde réplique, lors de l'intégration de op_1, le même choix est réalisé – placer `x` devant `y` – l'opération n'est donc pas modifiée. Au final, les deux répliques convergent vers le même état `axyb`.

La condition C_1 n'est pas suffisante pour garantir la convergence des copies en présence

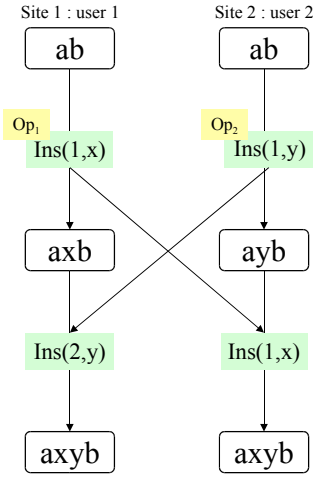


FIG. 2.18 – Respect de la condition C_1

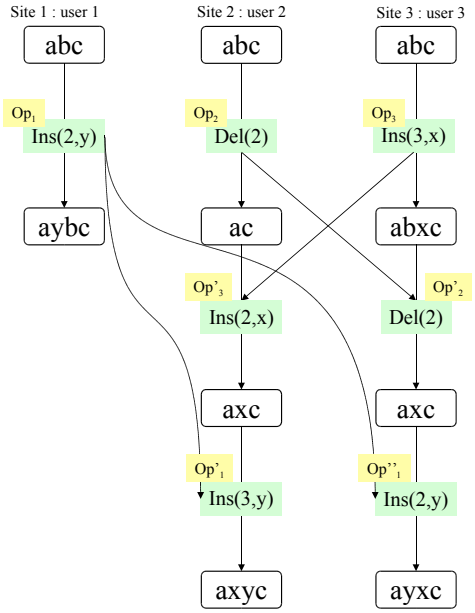


FIG. 2.19 – Insuffisance de la condition C_1.

```
T(Ins(p1,c1),Del(p2)):−
  if (p1 ≤ p2) then
    return Ins(p1, c1)
  else
    return Ins(p1 − 1, c1)
  endif
```

FIG. 2.20 – Une autre fonction vérifiant la condition C_1.

de plus de deux opérations concurrentes. La figure 2.19 illustre une telle situation. Dans ce scénario nous utilisons la fonction de transformation satisfaisant la condition C_1 que nous avons présentée précédemment. Nous utilisons également la fonction de transformation donnée par la figure 2.20.

Le respect de la condition C_1 nous garantit la convergence des copies après l'intégration de op_2 sur le site 2 et de op_3 sur le site 3. Cependant, un problème survient lors de l'intégration de op_1. Sur le site 2, on obtient l'opération $op'_1 = Ins(3, y)$ tandis que sur le site 3 on obtient l'opération $op''_1 = Ins(2, y)$. Ces deux opérations étant différentes lorsqu'elles sont exécutées sur le même état donnent forcément deux états différents. Les copies ne convergent donc pas.

Pour éviter cette situation un certain nombre d'algorithmes d'intégration, tels que GOT [], SOCT3, SOCT4 [VCFS00], SOCT5 [Vid02], imposent un ordre de sérialisation unique. Ainsi dans notre exemple, sur le site 3, op_2 est toujours transformée par rapport à op_3 et sur le site 2 op_3 est toujours transformée par rapport à op_2. Par contre, l'intégration sur ces deux sites de l'opération op_1 est réalisée en transformant par rapport à la même séquence $[op_2; T(op_3, op_2)] = [op_2; op'_3]$ et non plus par rapport à deux séquences équivalentes. Au final, l'équivalence entre les deux séquences exécutées sur les sites 2 et 3 est bien assurée. En effet, sur ces deux sites nous avons exécuté, tout d'abord, une des deux séquences équivalentes $[op_2; op'_3]$ ou $[op_3; op'_2]$, puis la même opération op'_1 ; cette opération résultant de la transformation de op_1 par rapport à la séquence $[op_2; op'_3]$.

Malheureusement, la construction d'un ordre de sérialisation unique dans un contexte réparti est coûteux et passe difficilement à l'échelle. C'est pourquoi, d'autres algorithmes d'intégration, tels que adOPTed [RNRG96], GOTO [SE98] SOCT2 [SCF98], ne requièrent pas d'ordre total, mais en contrepartie imposent aux fonctions de transformation de satisfaire une condition supplémentaire [RNRG96], la condition C_2.

DÉFINITION 2.3 (CONDITION C_2) Soient trois opérations op_1, op_2 et op_3 concurrentes définies sur le même état, la fonction de transposition en avant T vérifie la condition C_2

si et seulement si :

$$T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$$

On notera que par définition, l'expression $T(op_x, op_y \circ op_z)$ est égale à l'expression $T(T(op_x, op_y), op_z)$. Ainsi, l'égalité de la condition C_2 peut également s'écrire :

$$T(T(op_3, op_1), T(op_2, op_1)) = T(T(op_3, op_2), T(op_1, op_2))$$

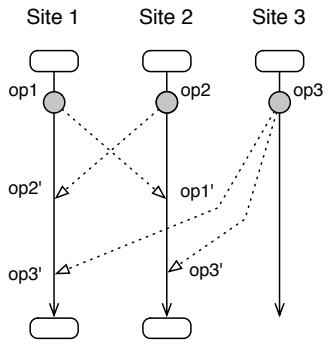


FIG. 2.21 – Condition C_2.

Contrairement à la condition C_1, il s'agit bien ici d'une égalité sur les opérations obtenues après transformation.

En généralisant cette dernière condition, on peut montrer [RNRG96] que si les fonctions de transformation vérifient la condition C_2 alors le résultat de la transformation d'une opération par rapport à une séquence d'opérations concurrentes ne dépend pas de l'ordre selon lequel les opérations de la séquence ont été transformées.

Enfin, il est possible de démontrer [SCF97, Sul98] que les conditions C_1 et C_2 sont deux conditions suffisantes pour assurer la convergence des copies. Autrement dit, si l'on considère n opérations $op_1, \ldots, op_n$ concurrentes et un ensemble de fonctions de transformation vérifiant les conditions C_1 et C_2, quel que soit l'ordre dans lequel on transforme ces opérations, les séquences obtenues sont équivalentes, c'est-à-dire qu'exécutées à partir du même état, elles produisent le même état.

△ △ △

Nous évaluons maintenant l'approche OT selon nos critères :

Convergence La convergence est assurée dans le cas général si les fonctions de transformation vérifient C_1 et C_2.

Intention Les intentions ne sont pas définies explicitement. Elles apparaissent implicitement et doivent être préservées au sein des fonctions de transformation.

Passage à l'échelle Avec des fonctions de transformation vérifiant C_1 et C_2, aucun ordre total, ni site central n'est requis. Par conséquent, l'approche OT doit passer à l'échelle.

2.7 SO6

SO6 est un gestionnaire de configuration reposant sur le modèle des transformées opérationnelles. Je l'ai développé pendant ma première année de doctorat. Cet outil est actuellement diffusé sous forme de logiciel libre et également intégré dans une offre commerciale.

SO6 montre qu'il est possible d'adapter les algorithmes OT, developpés pour des éditeurs collaboratifs temps-réels, afin de réaliser un outil comparable à CVS [Ced02] ou Subversion [Col05]. SO6 met en avant l'aspect générique de l'approche OT. En séparant l'algorithme d'intégration des fonctions de transformation, il est possible de construire de façon simple un gestionnaire de configuration. En démontrant que les fonctions de transformation vérifie la propriété dite C_1, on obtient immédiatement un outil assurant la convergence des copies dans tous les cas. Cet outil reste extensible à de nouveaux types de données répliquées en ajoutant les fonctions de transformation correspondantes.

Les fonctions de transformation nécessaires pour réconcilier un système de fichiers, des documents textuels et des documents XML sont présentées en annexe B.

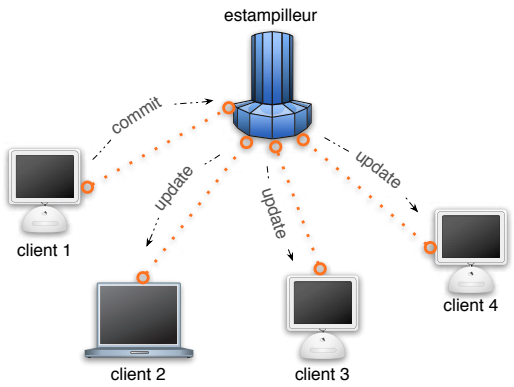


FIG. 2.22 – Architecture générale de SO6.

SO6 est basé sur l'algorithme SOCT4 [VCFS00]. Cet algorithme utilise un estampilleur pour ordonner totalement les opérations et requiert ainsi uniquement la vérification de la condition C_1. Les opérations sont estampillées selon un ordre total. Il se démarque des autres algorithmes d'intégration OT du même type en utilisant un mécanisme de diffusion différée. Pour envoyer une opération, il faut d'abord avoir reçu toutes les opérations estampillées et transformer les opérations locales en conséquence. Il est ensuite possible d'estampiller ses opérations locales et de les envoyer. Le fonctionnement de cet algorithme

s'apparente au paradigme "copier-modifier-fusionner" utilisé dans CVS où un utilisateur ne peut publier ses modifications que si il a pris en compte toutes les modifications déjà publiées.

L'architecture générale du système, illustrée à la figure 2.22, est centralisée autour d'un estampilleur qui délivre une estampille à chaque opération. Une application cliente s'exécute sur chaque machine des utilisateurs. Les applications clientes ne s'échangent pas directement les opérations, elles doivent les faire transiter par l'estampilleur.

SO6 met en œuvre l'estampilleur sous forme d'une file d'opérations. Cette file maintient, de manière persistante, les opérations publiées par les différents sites dans l'ordre des estampilles. Elle correspond à l'histoire commune des différentes copies.

Un estampilleur est constitué par :

une file d'opérations Q où les opérations sont rangées selon l'ordre des estampilles ;

une estampille *lastTicket* qui représente la dernière estampille délivrée ;

une méthode *publish*(*op*) qui attribue une nouvelle estampille à l'opération *op* et stocke cette opération dans la file Q ;

```
int publish(Operation op) {
2   lastTicket++
    Q[lastTicket] = op
4   return lastTicket
}
```

une méthode *retrieve*(*ticket*) qui retourne l'opération dont l'estampille a pour valeur *ticket* ;

```
1 Operation retrieve(int ticket) {
    return Q[ticket]
3 }
```

Nous avons developpé l'estampilleur de deux manière différentes : une version utilisant un disque partagé pour une utilisation sur un réseau local et une autre version utilisant un serveur J2EE [Mic05b] pour un déploiement à plus large échelle.

Un client SO6 doit détecter les modifications locales sous forme d'opérations, intégrer les opérations concurrentes, et publier les opérations résultant des modifications locales.

Un client est composé des éléments suivants :

une estampille *siteTicket* qui représente l'estampille de la dernière opération intégrée par le client. Autrement cette estampille est égale à celle de la dernière opération publiée par le site ou à celle de la dernière opération rappatriée par le site.

deux états *currentState* **et** *referenceState* . Ils permettent de calculer la séquence des opérations réalisées par l'utilisateur. *currentState* est l'état sur lequel l'utilisateur travaille. *referenceState* est l'état résultant de l'exécution de toutes les opérations qui ont été intégrées par le client ;

une séquence d'opérations *Hg* qui stocke les opérations qui ont été intégrées par l'application client. Elle contient donc d'une part les opérations publiées par les autres utilisateurs et qui ont été intégrées, et d'autre part, les opérations produites par l'utilisateur et qu'il a publiées. Les opérations sont rangées selon l'ordre de leur estampille.

L'exécution de cette séquence d'opérations *Hg* sur un état vide calcule toujours un état égal à l'état *referenceState* ;

une méthode *computeDifference*(*state*1, *state*2) qui calcule par différenciation la séquence d'opération permettant de générer l'état *state*2 à partir de l'état *state*1. Pour réaliser ce calcul sur les documents textuels nous utilisons l'algorithme Diff [MM85, Mye86]. Pour les documents XML, nous utilisons l'algorithme XyDiff [CAM02].

une méthode *commit*() qui publie les opérations locales vers l'estampilleur ;

```
1  commit() {
     if (estampilleur.lastTicket > siteTicket) then
3      abort "*** interdiction de publier sans être à jour ***"
     Operation[] locals = computeDifference(referenceState, currentState)
5    for (int i=0; i<locals.length; i++) {
       int ticket =estampilleur.publish(locals[i])
7      execute(locals[i], referenceState)
       Hg[ticket] = locals[i]
9    }
     siteTicket = estampilleur.lastTicket
11 }
```

Cette méthode commence par vérifier que le site a intégré toutes les opérations qui ont déjà été publiées et qui sont disponibles sur l'estampilleur. Ensuite, elle calcul la séquences des opérations locales qui ont été réalisées par l'utilisateur (et qui n'ont pas encore été publiées). Chaque opération est ensuite envoyée pour être estampillée, puis exécutée sur l'état de référence, et enfin, ajoutée à l'histoire *Hg* selon l'ordre des estampilles.

une méthode *update*() qui met à jour la réplique locale en rapatriant les opérations qui ont déjà été publiées ;

```
1  update() {
     Operation[] remotes
3    int i=0
     while (siteTicket < estampilleur.lastTicket) {
5      siteTicket++
       i++
7      remotes[i] = estampilleur.retrieve(siteTicket)
     }
9    Operation[] locals = computeDifference(referenceState, currentState)
```

```
    merge(remotes, locals)
11 }
```

Cette méthode commence par rapatrier toutes les opérations disponibles sur l'estampilleur et qui n'ont pas encore été intégrées. Elle calcule ensuite la séquence des opérations locales. Les deux séquences d'opérations obtenues, étant concurrentes, sont fusionnées.

une méthode *merge*(*Operation*[], *Operation*[]) qui "fusionne" deux séquences d'opérations concurrentes en utilisant la fonction de transformation T.

```
 1 merge(Operation[] remotes, Operation[] locals ) {
     for (int i=0; i<remotes.length; i++) {
 3       Operation opr = remotes[i]
         int ticket = remotes[i]. ticket
 5       for (int j=0; j<locals.length; j++) {
           Operation opl = locals[j]
 7         locals[j] = T(opl, opr)
           opr = T(opr, opl)
 9       }
         execute(remotes[i], referenceState)
11       Hg[ticket] = remotes[i]
         execute(opr, currentState)
13       siteTicket = ticket
     }
15 }
```

Cette méthode reprend le principe d'intégration développé dans l'algorithme SOCT4. Chaque opération à intégrer, *remote*[i] doit être transformée par rapport à la séquence des opérations locales *locals* en une opération *opr*. Cette opération peut être exécutée sur l'état actuel *currentState* sur lequel l'utilisateur travaille. L'opération *remote*[i] non transformée est ajoutée à la séquence *Hg*. Puis cette opération est exécutée sur l'état de référence *referenceState* pour maintenir la cohérence entre cet état et l'histoire *Hg*.

△ △ △

Convergence SO6 assure la convergence des répliques. Celle-ci est garantie d'une part par la preuve [Vid02] de correction de l'algorithme SOCT4, et d'autre part la preuve de la vérification de la condition C_1 par nos fonctions de transformation.

Intention Le scénario de la figure 2.23 conduit à une violation de l'intention dans SO6. Ce scénario considère un document textuel répliqué qui peut être modifié par deux opérations : *AddBlock*(l, b) qui insère le bloc de texte b à la ligne l du document ; *DelBlock*(l, b) qui détruit le bloc b situé à la ligne l. Dans ce scénario, il est clair

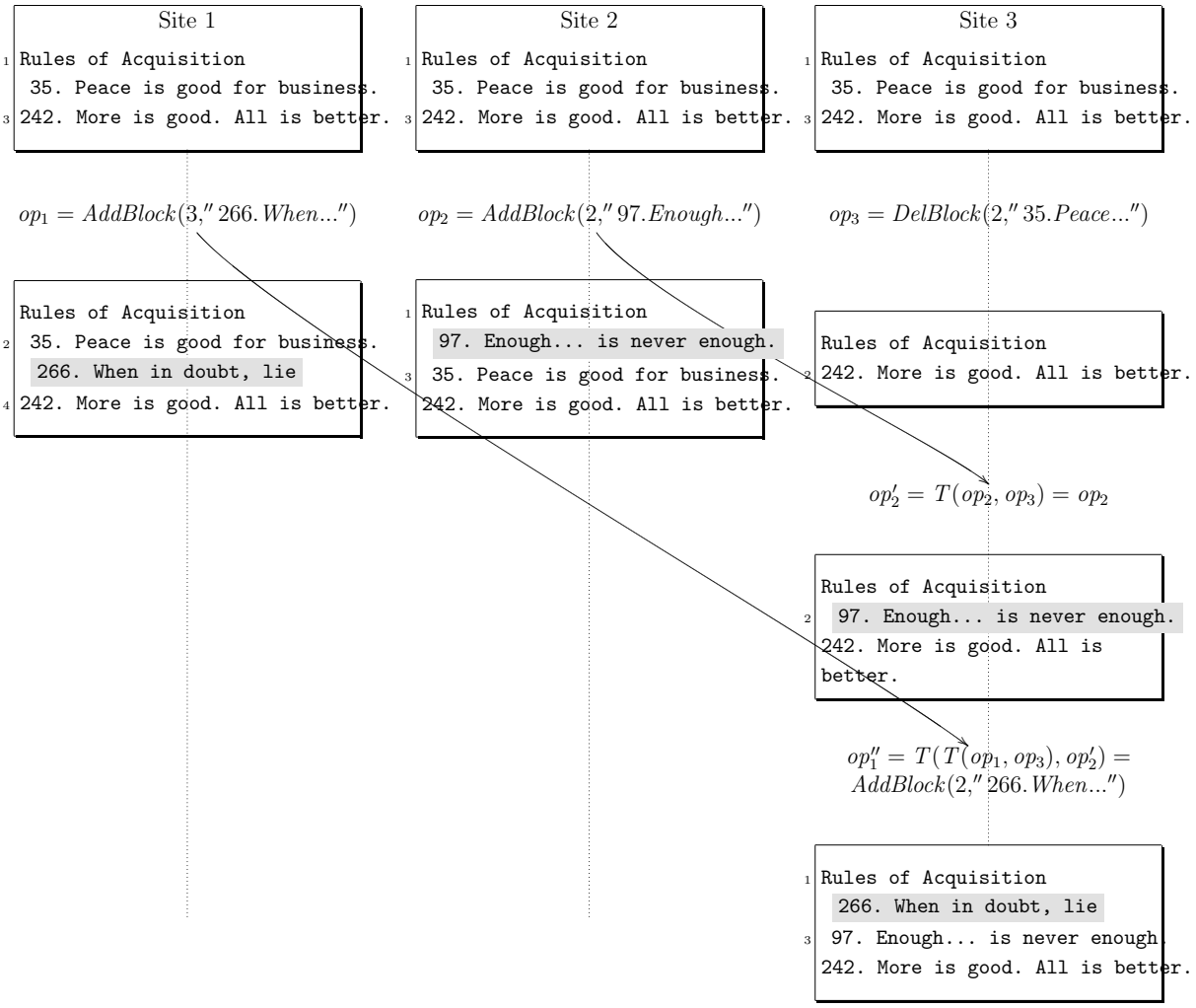


FIG. 2.23 – Violation de l'intention dans SO6

que sur l'état initial, op_1 insère sa ligne après celle insérée par op_2. Pourtant, si on suppose que l'estampilleur ordonnance les opérations dans l'ordre op_3, op_2, op_1, alors l'intégration de ces opérations conduit à une violation des intentions. Par exemple, on s'intéresse à l'intégration de ces opérations sur le site 3. On intègre op_2 en calculant [2] :

$$\begin{aligned} & T(op_2, op_3) \\ = \; & T(AddBlock(2,''97.Enough...''), DelBlock(2,''35.Peace...'')) \\ = \; & AddBlock(2,''97.Enough...'') \\ = \; & op_2' \end{aligned}$$

On intégre ensuite op_1 en calculant les transformées suivantes :

[2] la définition des fonctions de transformation T est disponible en annexe B

$T(T(op_1, op_3), op_2')$
$= T(T(AddBlock(3,''266.When...''), DelBlock(2,''35.Peace...'')), op_2')$
$= T(AddBlock(2,''266.When...''), op_2')$
$= T(AddBlock(2,''266.When...''), AddBlock(2,''97.Enough...''))$
$= AddBlock(2,''266.When...'')$
si l'on suppose que `code`$(''266.When...'')$ est inférieur à `code`$(''97.Enough...'')$

Au final, l'état obtenu ne respecte pas les intentions, puisque la ligne insérée par op_1 se retrouve devant ligne insérée par op_2. La vérification de la condition C_1, nous garantit quand même que toutes les répliques convergeront vers cet état.

SO6 ne préserve donc pas les intentions.

Passage à l'échelle SO6 utilise un estampilleur central, donc un ordre total. Par conséquent, SO6 ne passe pas l'échelle.

2.8 Synthèse générale

Nous avons évalué les systèmes potentiellement intéressants pour le domaine de l'édition collaborative selon trois critères : convergence, préservation des intentions et passage à l'échelle. Le tableau ci-dessous résume nos conclusions :

	Convergence	Intention	Passage à l'échelle
Wiki	Oui	Non	Non
Unison	Non	Non	Non
CVS	Oui	Non	Non
Oracle	Non	partiellement	Oui
Bayou	Oui	partiellement	Non
Icecube	Oui	Oui	Non
So6	Oui	Non	Non
OT	Oui si C_1 et C_2	Oui	Oui

Seule l'approche OT garantit la convergence, le respect des intentions et passe à l'échelle avec des fonctions de transformation vérifiant C_1 et C_2. Cependant, l'écriture de telles fonctions est un problème réputé difficile et il n'est même pas évident que de telles fonctions existent. Nous avons toutefois décidé de nous donner les moyens de les rechercher. Dans le chapitre suivant, nous utilisons un démonstrateur de théorème pour vérifier la correction des fonctions de transformation existantes et pour valider rapidement nos propositions.

Chapitre 3

Conception et validation de fonctions de transformation

Sommaire

Le modèle des transformées opérationnelles est un modèle de réplication optimiste adapté aux systèmes collaboratifs répartis. Il garantit la convergence des copies tout en assurant le respect des intentions. Ce dernier critère garantit que les différentes mises à jour seront préservées lors de la réconciliation des répliques.

La convergence et le passage à l'échelle sont garanties si les fonctions de transformation vérifient les deux propriétés C_1 et C_2. Bien que ces deux propriétés soient clairement identifiées, il est difficile de concevoir des fonctions de transformation les respectant.

Tout au long de ces quinze dernières années, il a été montré [IMOR03] que toutes les fonctions publiées concernant les chaînes de caractères sont fausses car elles violent la condition C_2. Ceci implique que toutes les propositions qui ont été faites pour d'autres structures de données linéaires et qui se sont basées sur ces propositions sont également fausses ; c'est le cas, par exemple, pour les fonctions de transformation développées [DSL02] pour réconcilier un document XML modélisé sous la forme d'un arbre ordonné.

Cette difficulté à proposer des fonctions de transformation correctes soulève plusieurs questions : existe-t-il des fonctions de transformation qui vérifient les conditions C_1 et C_2 tout en préservant l'intention ? Si oui, comment concevoir de telles fonctions ?

En effet, il existe des preuves de correction de certains algorithmes d'intégration [RNRG96, LC03, Sul98, Vid02] sous l'hypothèse que les fonctions vérifient les conditions. Mais il n'existe aucune preuve que de telles fonctions puissent exister. Et, si ces fonctions n'existent pas, ou que nous ne sommes pas capables de les concevoir, l'intérêt de l'approche dans son ensemble est remis en cause. Il n'est donc plus acceptable aujourd'hui de proposer des fonctions de transformation sans donner une preuve de correction de ces fonctions.

La conception et la vérification des fonctions de transformation sont donc cruciales pour l'approche. Malheureusement, au fur et à mesure des propositions, les fonctions de transformation sont devenues de plus en plus complexes. Aux opérations naïves munies uniquement des paramètres permettant de s'exécuter, ont été ajouté des paramètres supplémentaires permettant d'éviter les différents contre-exemples publiés. La correction de ses fonctions est devenue de plus en plus difficile à démontrer.

A la difficulté intrinsèque de la vérification s'ajoute un problème d'explosion combinatoire du nombre de cas à vérifier dû d'une part au nombre d'opérations et d'autre part à la complexité de la structure de données. Plus le nombre d'opérations permettant de modifier la donnée est élevé, plus le nombre de triplets d'opérations concurrentes à prendre en compte pour vérifier la condition C_2 est important. Et, plus la structure de données est complexe, plus le nombre de cas à vérifier pour chaque triplet d'opérations est important. Par exemple, pour une simple chaîne de caractères qui peut être modifiée par deux opérations – Insérer et Supprimer –, la vérification de la condition C_2 nécessite de traiter 8 triplets d'opérations. Et pour chaque triplet d'opérations, il faut s'intéresser aux relations possibles entre les positions où se réalisent les opérations ; ce qui donne, pour

notre exemple, 13 cas possibles. Ainsi, la preuve de la condition C_2 pour notre exemple nécessite de vérifier 104 cas.

À ce problème combinatoire vient s'ajouter le caractère itératif du processus de développement d'un jeu de fonctions de transformation. Pour concevoir des fonctions de transformation, il faut d'abord les imaginer, puis prouver qu'elles satisfont les conditions. Chaque violation nous apporte un nouveau contre-exemple qui nous permet de mieux cerner le problème, et, d'élaborer, ainsi, de nouvelles fonctions de transformation. Ces fonctions doivent être de nouveau vérifiées. Ce processus se répète jusqu'à ce que nous trouvions des fonctions correctes. Ainsi, la vérification des conditions sera réalisée de nombreuses fois. Le coût de cette vérification devient donc très rapidement prohibitif bien que cette vérification soit nécessaire pour assurer la correction et également trouver de nouvelles idées pour concevoir de nouvelles fonctions.

La preuve de C_1 et C_2 est déjà difficile pour une chaîne de caractères avec deux opérations : insert et delete. Il est évident que plus le type de donnée est complexe, plus le nombre d'opérations est élevé, plus la preuve est longue et difficile.

Dans ce chapitre, nous utilisons des méthodes formelles pour nous aider à concevoir des fonctions de transformation correctes.

Le modèle des transformées opérationnelles est bien adapté à l'utilisation de méthodes formelles. Les conditions à satisfaire sont déjà clairement exprimées. Les fonctions de transformation peuvent être apparentées à des règles de réécriture sur les opérations ; ce genre de règle étant très employé dans les méthodes formelles. Du point de vue "difficulté de la preuve", le problème de vérification semble facile. Seule l'explosion combinatoire du nombre de cas à vérifier demeure un problème. Ce dernier problème peut être résolu en utilisant un démonstrateur automatique qui va ainsi automatiser la majorité du déroulement de la preuve.

Dans ce chapitre, nous utilisons un démonstrateur automatique de théorème pour nous aider à concevoir des fonctions de transformation correctes. Le démonstrateur délivre ainsi une preuve formelle de la vérification des conditions. Cette preuve repose entièrement sur des règles syntaxiques maîtrisées et issues de la logique mathématique. De plus, son haut degré d'automatisation fiabilise le processus de validation et exclut toutes les "erreurs d'inattention".

Cette automatisation réduit le temps relatif à la phase de vérification. Nous disposons ainsi des moyens essentiels pour concevoir de nouvelles fonctions de transformation et vérifier rapidement si elles satisfont les conditions. Chaque violation est détectée plus rapidement, nous permettant de concevoir plus tôt de nouvelles fonctions de transformation ; ceci, dans un but précis : trouver des fonctions de transformation qui satisfassent les condition C_1 et C_2 et démontrer ainsi leur existence.

La figure 3.1 schématise le procédé de conception que nous mettons en place.

Ce procédé itératif se décompose en trois grandes étapes :

1. Une étape de modélisation où l'on formalise les fonctions de transformation que l'on

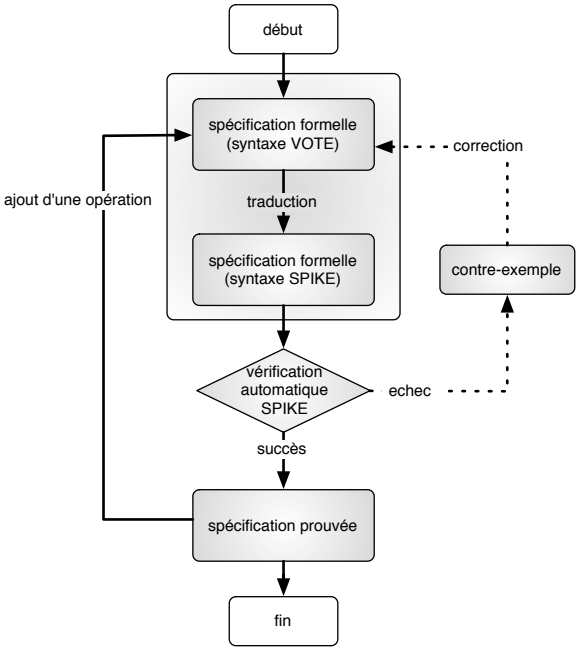


FIG. 3.1 – Vérification des fonctions de transformation en utilisant VOTE/SPIKE

souhaite prouver ;

2. une étape de vérification où l'on utilise un démonstrateur de théorème pour contrôler la correction des fonctions ;
3. une étape d'analyse où l'on interprète les résultats du démonstrateur.

Si le démonstrateur termine en ayant prouvé toutes les conjectures, alors les fonctions sont correctes.

Le procédé de conception des fonctions de transformation est incrémental. On commence par définir une opération, par exemple $Inserer(caractere, position)$. On définit l'unique fonction de transformation nécessaire $T(Inserer, Inserer)$. On vérifie sa correction. Puis, on ajoute une seconde opération, par exemple $Suppression(caractere, position)$. On définit les fonctions de transformation $T(Suppression, Suppression)$, $T(Suppression, Inserer)$ et $T(Inserer, Suppression)$. On vérifie leur correction. On procède ainsi jusqu'à ce que l'on ait défini toutes les opérations et toutes les fonctions de transformation.

Si le démonstrateur s'arrête sans avoir pu prouver les conjectures, alors cela peut avoir deux significations :

1. La spécification est incomplète et il manque des lemmes ou des définitions. Dans ce cas, il faut les rajouter.
2. Un contre-exemple a été trouvé. Dans ce cas, Il faut revoir les fonctions de transformation.

Dans la suite de ce chapitre, nous allons illustrer chaque étape du procédé sur un exemple. Cet exemple reprend la proposition originale d'Ellis [EG89]. On considère une

chaîne de caractères comme donnée répliquée. Cette donnée est vue comme une liste de caractères où la position dans la liste représente l'indice du caractère dans la chaîne.

Cette chaîne de caractères peut être modifiée par deux opérations :

- $Ins(p, c)$ insère le caractère c à la position p.
- $Del(p)$ détruit le caractère situé à la position p.

```
T(Ins(p1,c1,pr1), Ins(p2,c2,pr2)) :−
    if p1 < p2 return Ins(p1,c1,pr1)
    else if p1 > p2 return Ins(p1 + 1, c1,pr1)
    else if c1 == c2 return Id()
    else if pr1 > pr2 return Ins(p1 + 1,c1,pr1)
    else return Ins(p1,c1,pr1)

T(Ins(p1,c1,pr1), Del(p2,pr2)) :−
    if p1 < p2 return Ins(p1,c1,pr1)
   else return Ins(p1 − 1,c1,pr1)

T(Del(p1,pr1),Ins(p2,c2,pr2)) :−
    if p1 < p2 return Del(p1,pr1)
    else return Del(p1 + 1,pr1)

T(Del(p1,pr1),Del(p2,pr2)) :−
    if p1 < p2 return Del(p1,pr1)
    else if p1 > p2 return Del(p1 − 1,pr1)
    else return Id()
```

FIG. 3.2 – Fonctions de transformation proposées par Ellis et al. [EG89].

Les fonctions de transformation publiées par Ellis et Gibbs [EG89] sont détaillées dans la figure 3.2. Les auteurs ont étendu les profils des deux opérations *Ins* et *Del* en y ajoutant un paramètre supplémentaire pr. Ce paramètre représente une priorité reposant sur un identifiant [3].

Si on transforme une opération *Ins* par rapport à une autre opération *Ins* identique alors on obtient une opération identité (cf. ligne 4 de la figure 3.2).

Deux modifications concurrentes sont fusionnées de la manière suivante :

[3]Ce paramètre n'est pas uniquement constitué de l'identifiant du site. Il est géré à la manière d'une liste. Cependant, ceci n'a pas d'importance dans la suite car nous ne faisons aucune hypothèse sur ce paramètre.

- deux modifications portant sur deux positions différentes sont fusionnées en décalant la position la plus élevée ;
- deux suppressions concurrentes à la même position effacent le même caractère. La transformation inhibe une de ces deux opérations en la transformant en une opération identité ;
- il faut noter que, lorsque l'on considère une opération de suppression et une opération d'insertion portant sur la même position, l'effet de l'opération de suppression est en réalité situé "devant" l'effet de l'opération d'insertion. Par exemple, si nous considérons la chaîne 'abc' alors l'opération $del(2)$ détruit le caractère 'b', tandis que l'opération $ins(2, x)$ insère 'x' entre 'a' et 'b' c'est-à-dire devant 'b' ;
- le cas de deux opérations d'insertion à la même position constitue le cas problématique :
  - si les deux caractères sont les mêmes alors les deux opérations sont considérées comme identiques et l'effet de l'une d'elle est inhibé ;
  - si les deux caractères sont différents, il faut choisir un ordre pour les insérer l'un derrière l'autre. Dans ce cas, Ellis et Gibbs proposent de placer en premier le caractère dont l'opération est prioritaire, c'est-à-dire l'opération dont le paramètre pr est le plus petit. La position de l'autre opération est donc incrémentée.

3.1 Conception et modélisation formelle

Les démonstrateurs de théorème reposent sur un langage de spécification basé sur une logique. Ils apportent un support à la vérification de propriétés exprimées sous forme de formule logique. Les démonstrateurs de théorème peuvent être classifiés en deux grandes catégories :

Les assistants de preuve : des outils tels que Coq [Log05] et PVS [ORS92] interagissent avec l'utilisateur à différentes étapes de la preuve.

Les démonstrateurs automatiques : des outils tels que Spike [BR93] prouvent automatiquement en appliquant des stratégies préétablies. Cette catégorie d'outils est indiquée pour des preuves relativement simple mais comportant une forte combinatoire. C'est tout-à-fait notre cas.

Pour nos travaux, nous utilisons le démonstrateur automatique Spike [Bou96, Str01] issu des recherches du projet INRIA CASSIS. Spike est un démonstrateur de théorème construisant des preuves par récurrence. Il a été conçu pour vérifier des formules implicitement universellement quantifiées dans des théories construites à l'aide d'axiomes conditionnels du premier ordre. Les preuves réalisées par Spike reposent sur la couverture d'un ensemble d'induction.

Étant donnée une théorie, Spike calcule, dans un premier temps, les termes d'induction qui schématiquement représentent toutes les valeurs possibles que peuvent prendre les variables d'induction. Étant donnée une conjecture à vérifier, le démonstrateur sélectionne

les variables sur lesquelles il faut appliquer l'induction, il les substitue par les termes d'induction. Cette manipulation génère plusieurs instances de la conjecture qui sont alors simplifiées. Cette simplification s'effectue par réécriture conditionnelle en utilisant les axiomes, les lemmes ou encore les hypothèses d'induction.

Par exemple, dans l'arithmétique définie par Peano [Pea89], il est possible de construire tout entier naturel x en utilisant deux termes d'induction : la constante 0 et le terme successeur $s(x)$. L'addition peut être définie par les deux axiomes suivantes :

(R1) : $x + 0 = x$
(R2) : $x + s(y) = s(x + y)$

Nous souhaitons démontrer la conjecture suivante : $0 + x = x$. Tout d'abord, nous substituons la variable x par tous les termes de l'ensemble d'induction $(0, s(y))$. Nous obtenons les conjectures suivantes :

(1) : $0 + 0 = 0$
(2) : $0 + s(y) = s(y)$

Ensuite, nous simplifions ces deux instances en utilisant les règles et l'hypothèse d'induction. Pour la conjecture (1), nous appliquons la règle (R1) en substituant x par 0, nous obtenons l'instance $0 = 0$ qui est vraie. Pour la conjecture (2), nous appliquons la règle (R2) en substituant x par 0, nous obtenons l'instance $s(0+y) = s(y)$. Nous utilisons alors l'hypothèse d'induction, à savoir $0+X = X$, ce qui nous donne l'instance $s(y) = s(y)$ qui est également toujours vraie.

Nous venons ainsi de démontrer, comme le ferait le démonstrateur, que la conjecture est toujours vraie.

Les fonctions de transformation telles qu'elles sont présentées à la figure 3.2 sont déjà exprimées de manière formelle. Cependant, le formalisme utilisé n'est pas le même que celui qu'emploie notre démonstrateur de théorème. Ce dernier nécessite une spécification sous forme de règles de réécriture exprimées en logique équationnelle du premier ordre.

Nous devons donc effectuer une traduction de la spécification entre les deux formalismes. Cette traduction est réalisée de manière automatique par l'outil VOTE [IMOU03] qui a été développé spécifiquement pour cette tâche en coopération avec le projet INRIA CASSIS.

La figure 3.3 présente la spécification résultante de la traduction. Il nous était possible de spécifier directement les fonctions de transformation dans ce formalisme. Cependant, ce dernier formalisme étant clairement "moins lisible", nous avons privilégié cette approche de traduction automatique.

$(p_1 < p_2) = true \Rightarrow T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = Ins(p_1, c_1, pr_1);$
$(p_1 < p_2) = false, (p_1 > p_2) = true \Rightarrow$
$\quad T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = Ins(p_1 + s(0), c_1, pr_1);$
$(p_1 < p_2) = false, (p_1 > p_2) = false, c_1 = c_2 \Rightarrow$
$\quad T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = nop;$
$(p_1 < p_2) = false, (p_1 > p_2) = false, c_1 \neq c_2, (pr_1 > pr_2) = true \Rightarrow$
$\quad T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = Ins(p_1 + s(0), c_1, pr_1);$
$(p_1 < p_2) = false, (p_1 > p_2) = false, c_1 \neq c_2, (pr_1 > pr_2) = false \Rightarrow$
$\quad T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = Ins(p_1, c_1, pr_1);$

$(p_1 < p_2) = true \Rightarrow T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = Ins(p_1, c_1, pr_1);$
$(p_1 < p_2) = false \Rightarrow T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = Ins(p_1 - s(0), c_1, pr_1);$

$(p_1 < p_2) = true \Rightarrow T(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)) = Del(p_1, pr_1);$
$(p_1 < p_2) = false \Rightarrow T(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)) = Del(p_1 + s(0), pr_1);$

$(p_1 < p_2) = true \Rightarrow T(Del(p_1, pr_1), Del(p_2, pr_2)) = Del(p_1, pr_1);$
$(p_1 < p_2) = false, (p_1 > p_2) = true \Rightarrow T(Del(p_1, pr_1), Del(p_2, pr_2)) = Del(p_1 - s(0), pr_1);$
$(p_1 < p_2) = false, (p_1 > p_2) = false \Rightarrow T(Del(p_1, pr_1), Del(p_2, pr_2)) = nop;$

FIG. 3.3 – Résultat de la traduction par VOTE des fonctions de la figure 3.2.

3.1.1 Formalisation de la condition C_1

Nous rappelons tout d'abord la définition de la condition C_1 qui constitue, pour deux opérations concurrentes op_1 et op_2, l'*équivalence de séquence* suivante :

$$C_1 : \ op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$$

où l'opérateur $\circ$ construit une séquence d'opération à partir de deux opérations.

Deux séquences équivalentes sont deux séquences qui exécutées à partir d'un même état de départ donnent le même état d'arrivée. Nous pouvons exprimer plus formellement cette condition de la manière suivante :

$\forall op_i \in Opn, \forall op_j \in Opn, \forall st \in State,$
$Enabled(op_i, st) \wedge Enabled(op_j, st) \wedge Conc(op_i, op_j) \Rightarrow$
$\quad (st \odot op_i) \odot T(op_j, op_i) = (st \odot op_j) \odot T(op_i, op_j)$

Les symboles *Opn* dénote l'ensemble des opérations définies sur l'objet répliqué, *State* l'ensemble des états que peut prendre l'objet répliqué, $\odot$ l'opérateur d'exécution d'une opération sur un état.

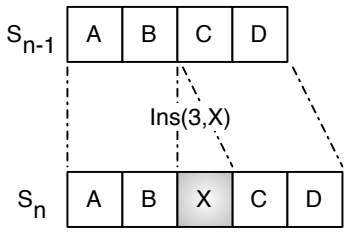


FIG. 3.4 – Un exemple d'observation.

Le prédicat $Enabled(op_i, st)$ teste la pré-condition de l'opération op_i, c'est à dire les conditions sous lesquelles l'opération op_i peut s'exécuter ou non sur un état st. Par exemple, l'opération $Ins(p_1, c_1, pr_1)$ possède la pré-condition suivante : $0 < p_1 \leq length(st)$. Ce qui signifie que l'insertion ne peut s'effectuer que si la position d'insertion est située dans la chaîne actuelle.

Le prédicat $Conc(op_i, op_j)$ permet d'introduire des conditions sous lesquelles les opérations op_i et op_j peuvent être concurrentes. Par exemple, pour deux opérations $op_1 = Ins(p_1, c_1, pr_1)$ et $op_2 = Ins(p_2, c_2, pr_2)$, le prédicat doit préciser que $pr_1 \neq pr_2$ puisque deux opérations concurrentes sont générées obligatoirement sur deux sites différents. Cette définition est bien évidemment incomplète puisque deux opérations générées sur deux sites différents peuvent être dépendantes (l'exécution de l'une précéde la génération de l'autre).

Enfin, pour pouvoir vérifier cette condition, nous devons donc être capable de caractériser l'état st et plus généralement tous les états envisageables de l'objet répliqué.

Une première solution consiste à définir cet état en utilisant des types abstraits. Cependant, la structure des objets partagés peut être très complexe comme par exemple un document XML modélisé par arbre ordonné où les nœuds sont étiquetés par des valeurs. La spécification algébrique devient plus compliquée. En conséquence, la preuve de C_1 nécessite la preuve de nombreuses propriétés sur la structure de l'objet manipulé ; ce qui ralentit considérablement la procédure de preuve.

Nous avons retenu une autre solution basée sur le calcul de situations [MH69]. L'état d'une réplique est représenté par une situation construite inductivement à partir des opérations affectant le système. L'ensemble des situations est noté Sit. L'effet d'une opération sur une situation est perçu à travers la modification d'un certain nombre de ses caractéristiques. Chaque caractéristique est modélisée sous la forme d'une fonction d'observation obs_n dont le dernier paramètre est une situation st.

Dans notre exemple, nous choisissons de définir un observateur $car(i, st)$ qui observe le i^{eme} caractère dans la situation st. Nous définissons ensuite comment cet observateur perçoit l'exécution des différentes opérations. Autrement dit, dans notre cas comment est perçu le changement apporté par une opération $Ins(p,c,pr)$ ou $Del(p, pr)$ sur la situation st.

La définition de l'observation d'une caractéristique s'effectue de manière récursive par rapport à la situation de la donnée. Par exemple, nous nous intéressons à l'exemple pré-

$n = p \Rightarrow car(n, xSt \odot Ins(p, c, pr)) = c;$
$n \neq p, (n > p) = true \Rightarrow car(n, xSt \odot Ins(p, c, pr)) = car(n - s(0), xSt);$
$n \neq p, (n > p) = false \Rightarrow car(n, xSt \odot Ins(p, c, pr)) = car(n, xSt);$

$(n \geq p) = true \Rightarrow car(n, xSt \odot Del(p, pr)) = car(n + s(0), xSt);$
$(n \geq p) = false \Rightarrow car(n, xSt \odot Del(p, pr)) = car(n, xSt);$

FIG. 3.5 – Traduction de la définition de l'observateur $car(pos, st)$.

$car'(n)/Ins(p, c, pr) =$
  if $(n == p)$ **then**
    return c
  else if $(n > p)$ **then**
    return $car(n - 1)$
  else
    return $car(n)$
  endif;

$car'(n)/Del(p, pr) =$
  if $(n \geq p)$ **then**
    return $car(n + 1)$
  else
    return $car(n)$
  endif;

FIG. 3.6 – Définition de l'observateur $car(pos, st)$.

senté à la figure 3.4. Dans cet exemple, nous souhaitons connaître le caractère situé à la position *pos* dans la situation S_n. Ceci revient à évaluer l'observation $car(pos, S_n)$. Supposons que la situation S_n résulte de l'exécution de l'opération $Ins(3, X)$ sur la situation S_{n-1}. Nous pouvons en déduire que $car(pos, S_n) = car(pos, S_{n-1} \odot Ins(3, X))$. Si la position *pos* est égale à 3, alors le caractère que l'on observe, à cette position dans la situation S_n, est le caractère `X`. Sinon, nous devons observer la situation précédent S_{n-1} à une position qui peut varier selon l'opération menant de S_{n-1} à S_n. Ainsi, si $pos > 3$ alors on doit observer le caractère à la position $pos - 1$ dans S_{n-1} ; et si $pos < 3$ alors la position d'observation ne varie pas.

La figure 3.6 contient la définition complète de l'observateur $car(pos, st)$ en fonction des deux opérations qui peuvent modifier la chaîne de caractères. $car'(n)/Ins(p, c, pr)$ dénote la fonction permettant de calculer la valeur $car'(n)$ de l'observateur sur une situation S_n en fonction de sa valeur $car(n)$ sur une situation S_{n-1}, la situation S_n résultant de l'exécution de l'opération $Ins(p, c, pr)$ sur la situation S_{n-1}. Cette définition doit être traduite dans le formalisme utilisé par notre démonstrateur de théorème (voir figure 3.5).

Finalement, en utilisant ces fonctions d'observation, nous pouvons définir une équiva-

lence entre deux situations.

DÉFINITION 3.1 (ÉQUIVALENCE DE SITUATION) Deux situations st_1 et st_2 sont dites équivalentes (noté $=_{Obs}$) si et seulement si :

$$\forall obs_n \in Obs, obs_n(st_1) = obs_n(st_2)$$

Nous pouvons ainsi donner la définition de deux séquences équivalentes.

DÉFINITION 3.2 (ÉQUIVALENCE DE SÉQUENCE) Deux séquences d'opérations seq_1 et seq_2 sont équivalentes si et seulement si :

$$\forall st_1, st_2 \in Sit, st_1 =_{Obs} st_2 \Rightarrow st_1 \odot seq_1 =_{Obs} st_2 \odot seq_2$$

Il nous est possible alors d'exprimer la condition C_1 par la conjecture à prouver suivante :

$$\begin{array}{l}\forall op_i \in Opn, \forall op_j \in Opn, \forall st \in Sit, \\ Enabled(op_i, st) \wedge Enabled(op_j, st) \wedge Conc(op_i, op_j) \Rightarrow \\ \qquad (st \odot op_i) \odot T(op_j, op_i) =_{Obs} (st \odot op_j) \odot T(op_i, op_j)\end{array}$$

Nous réécrivons cette définition en utilisant l'équivalence entre deux séquences :

$$\begin{array}{l}\forall op_i \in Opn, \forall op_j \in Opn, \forall st \in Sit, \forall obs_n \in Obs \\ Enabled(op_i, st) \wedge Enabled(op_j, st) \wedge Conc(op_i, op_j) \Rightarrow \\ \qquad obs_n((st \odot op_i) \odot T(op_j, op_i)) = obs_n((st \odot op_j) \odot T(op_i, op_j))\end{array}$$

Dans notre exemple, puisque nous avons défini un seul observateur, pour prouver la condition C_1, le démonstrateur de théorème doit prouver uniquement la conjecture suivante :

$$\begin{array}{l}conc(op_1, op_2) = true, enabled(op_1, xSt) = true, enabled(op_2, xSt) = true \Rightarrow \\ \qquad car(p_x, xSt \odot op_1 \odot T(op_2, op_1)) = car(p_x, xSt \odot op_2 \odot T(op_1, op_2));\end{array}$$

3.1.2 Formalisation de la condition C_2

La condition C_2 est plus facile à formaliser que la condition C_1. Elle ne nécessite pas de définir l'équivalence entre deux séquences et nous n'avons donc pas besoin d'utiliser d'observateurs.

Rappelons l'énoncé de la condition C_2 pour trois opérations concurrentes op_1, op_2 et op_3 :

$$C_2 : T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$$

Pour toutes opérations op_i, op_j et op_k la transformation $T(op_i, op_j \circ op_k)$ est égale à $T(T(op_i, op_j), op_k)$. La condition C_2 peut donc s'exprimer plus formellement de la sorte :

$\forall op_i \in Opn, \forall op_j \in Opn, \forall st \in Sit,$
$Enabled(op_i, st) \wedge Enabled(op_j, st) \wedge Enabled(op_k, st) \wedge$
$Conc(op_i, op_j, op_k) \Rightarrow$
$T(T(op_k, op_i), T(op_j, op_i)) = T(T(op_k, op_j), T(op_i, op_j))$

Pour vérifier la condition C_2, après traduction, le démonstrateur doit prouver la conjecture suivante :

$enabled(op_i, xSt) = true, enabled(op_j, xSt) = true, enabled(op_k, xSt) = true,$
$conc(op_i, op_j, op_k) = true \Rightarrow$
$T(T(op_k, op_i), T(op_j, op_i)) = T(T(op_k, op_j), T(op_i, op_j));$

3.2 Vérification

Nous avons développé un outil permettant de vérifier le caractère correct des fonctions de transformation. En cas de violation, cet outil nous fournit des contre-exemples permettant ainsi d'accélérer le développement de nouvelles fonctions.

Dans cette section, nous présentons les différents contre-exemples que nous avons trouvés et nous explicitons les problèmes soulevés sur ces différentes propositions.

3.2.1 Proposition d'Ellis et al. [EG89]

La première proposition que nous abordons est la proposition d'Ellis présentée précédemment. Nous détaillons précisément la tentative de preuve de la condition C_1 telle que le démonstrateur de théorème l'effectue.

Étape 1 : Génération des conjectures

Nous avons vu à la section 3.1.1 que pour vérifier la condition C_1, le démonstrateur doit prouver la conjecture suivante :

$conc(op_1, op_2) = true, enabled(op_1, xSt) = true, enabled(op_2, xSt) = true \Rightarrow$
$car(p_x, xSt \odot op_1 \odot T(op_2, op_1)) = car(p_x, xSt \odot op_2 \odot T(op_1, op_2));$

Spike commence par générer les différentes conjectures à vérifier en remplaçant dans la conjecture initiale les variables d'induction par les différents termes d'induction. Les variables d'induction sont : op_1, op_2, et p_x.

Les variables op_1 et op_2 sont substituées par les termes d'induction $Ins(p_i, c_i, pr_i)$ et $Del(p_i, pr_i)$.

En générant toutes les combinaisons possibles, le démonstrateur se retrouve avec quatre nouvelles conjectures à prouver.

(G1) $conc(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Ins(p_2, c_2, pr_2), xSt) = true \Rightarrow$

$$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot T(Ins(p_2, c_2, pr_2), Ins(p_1, c_1, pr_1)))$$
$$= car(p_x, xSt \odot Ins(p_2, c_2, pr_2) \odot T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)));$$

(G2) $conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot T(Del(p_2, pr_2), Ins(p_1, c_1, pr_1)))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)));$

(G3) $conc(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)) = true,$
$enabled(Del(p_1, pr_1), xSt) = true, enabled(Ins(p_2, c_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Del(p_1, pr_1) \odot T(Ins(p_2, c_2, pr_2), Del(p_1, pr_1)))$
$= car(p_x, xSt \odot Ins(p_2, c_2, pr_2) \odot T(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)));$

(G4) $conc(Del(p_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Del(p_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Del(p_1, pr_1) \odot T(Del(p_2, pr_2), Del(p_1, pr_1)))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot T(Del(p_1, pr_1), Del(p_2, pr_2)));$

Nous limitons notre étude à la preuve de la conjecture (G3).

Étapes 2 : Réécriture et simplification

Au cours de cette étape, Spike utilise les différentes règles de réécriture issues de la définition des fonctions de transformation et des fonctions d'observation pour simplifier les conjectures courantes et dériver de nouvelles conjectures. Le résultat final est indépendant de l'ordre d'application des règles de réécriture. Spike peut donc choisir librement la règle qu'il souhaite appliquer à chaque étape de réécriture.

Par exemple, nous utilisons les règles de réécriture issues de la définition de la fonction de transformation $T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2))$. Notre conjecture est réécrite en deux nouvelles conjectures :

(G5) $(p_1 < p_2) = true$, $conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot T(Del(p_2, pr_2), Ins(p_1, c_1, pr_1)))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot$ $Ins(p_1, c_1, pr_1)$) ;

(G6) $(p_1 < p_2) = false$, $conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot T(Del(p_2, pr_2), Ins(p_1, c_1, pr_1)))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot$ $Ins(p_1 - s(0), c_1, pr_1)$) ;

Puis, nous utilisons les règles de réécriture issues de la définition de la fonction de transformation $T(Del(p_2, pr_2), Ins(p_1, c_1, pr_1))$, les deux conjectures précédentes sont réécrites en quatre nouvelles conjectures :

(G7) $(p_2 < p_1) = true$, $(p_1 < p_2) = true, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot Del(p_2, pr_2))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1, c_1, pr_1));$

(G8) $(p_2 < p_1) = false$, $(p_1 < p_2) = true, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot Del(p_2 + s(0), pr_2))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1, c_1, pr_1));$

(G9) $(p_2 < p_1) = true$, $(p_1 < p_2) = false, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot Del(p_2, pr_2))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1));$

(G10) $(p_2 < p_1) = false$, $(p_1 < p_2) = false, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot Del(p_2 + s(0), pr_2))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1));$

Après chaque étape de réécriture, Spike exécute une phase de simplification afin d'éliminer certaines conjectures dont les hypothèses sont contradictoires. Par exemple, durant cette phase, la conjecture (G7) est éliminée puisqu'il est impossible de satisfaire les deux hypothèses $(p_2 < p_1) = true$ et $(p_1 < p_2) = true$.

Le processus de preuve se poursuit. Nous limitons notre explication à la preuve de la dernière conjecture (G10).

Supposons que le démonstrateur choisisse de réécrire le terme $car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot Del(p_2 + s(0), pr_2))$ en utilisant les règles relatives à la fonction d'observation.

(G11) $(p_x \geq p_2 + s(0)) = true$, $(p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x + s(0), xSt \odot Ins(p_1, c_1, pr_1))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1));$

(G12) $(p_x \geq p_2 + s(0)) = false$, $(p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1));$

Puis, il continue ainsi en réécrivant ensuite les autres termes faisant intervenir la fonction d'observation. Nous nous limitons à l'étude de la preuve de la conjecture (G12).

En réécrivant le terme $car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1))$, nous obtenons :

(G13) $p_x = p_1 - s(0)$, $(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) =$ c_1 ;

(G14) $p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true$, $(p_x \geq p_2 + s(0)) = false,$
$(p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) =$ $car(p_x - s(0), xSt \odot Del(p_2, pr_2))$;

(G15) $p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = false$, $(p_x \geq p_2 + s(0)) = false,$
$(p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) =$ $car(p_x, xSt \odot Del(p_2, pr_2))$;

Nous continuons en nous intéressant à la preuve de la seconde conjecture (G14). Nous supposons que Spike réécrit le terme $car(p_x - s(0), xSt \odot Del(p_2, pr_2))$.

(G16) $(p_x - s(0) \geq p_2) = true$, $p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true,$
$(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) =$ $car(p_x - s(0) + s(0), xSt)$;

(G17) $(p_x - s(0) \geq p_2) = false$, $p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true,$
$(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) =$ $car(p_x - s(0), xSt)$;

Nous supposons que Spike réécrit le terme $car(p_x, xSt \odot Ins(p_1, c_1, pr_1))$.

(G18) $p_x = p_1$, $(p_x - s(0) \geq p_2) = true, p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true,$
$(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
c_1 $= car(p_x - s(0) + s(0), xSt);$

(G19) $p_x \neq p_1, (p_x > p_1) = true$, $(p_x - s(0) \geq p_2) = true, p_x \neq p_1 - s(0)$,
$(p_x > p_1 - s(0)) = true, (p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false$,
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x - s(0), xSt) = car(p_x - s(0) + s(0), xSt)$;

(G20) $p_x \neq p_1, (p_x > p_1) = false$, $(p_x - s(0) \geq p_2) = true, p_x \neq p_1 - s(0)$,
$(p_x > p_1 - s(0)) = true, (p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false$,
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt) = car(p_x - s(0) + s(0), xSt)$;

(G21) $p_x = p_1$, $(p_x - s(0) \geq p_2) = false, p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true$,
$(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false$,
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$c_1 = car(p_x - s(0), xSt)$;

(G22) $p_x \neq p_1, (p_x > p_1) = true$, $(p_x - s(0) \geq p_2) = false$,
$p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true, (p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false$,
$(p_1 < p_2) = false, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x - s(0), xSt) = car(p_x - s(0), xSt)$;

(G23) $p_x \neq p_1, (p_x > p_1) = false$, $(p_x - s(0) \geq p_2) = false$,
$p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true, (p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false$,
$(p_1 < p_2) = false, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt) = car(p_x - s(0), xSt)$;

Les conjectures (G19), (G20), (G22) et (G23) sont éliminées durant la phase de simplification – puisqu'elles possèdent des hypothèses contradictoires – . Les deux conjectures restantes (G18) et (G21) vont être réécrites en utilisant les règles relatives aux termes $enabled(op, xSt)$ et $conc(op_i, op_j)$. Ainsi, par exemple la conjecture (G17) donne :

(G24) $p_x = p_1, (p_x - s(0) \geq p_2) = true, p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true$,
$(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false$,
$pr_1 \neq pr_2, 0 < p_1, 0 < p_2 \Rightarrow$
$c_1 = car(p_x - s(0) + s(0), xSt)$;

Cette conjecture ne peut plus être réécrite, ni même simplifiée. Quand le démonstrateur ne possède plus que des conjectures de ce type, il s'arrête.

Étape 3 : Analyse

L'arrêt du démonstrateur de théorème peut avoir plusieurs significations :

- La preuve se termine par un succès. Dans ce cas, le démonstrateur n'a trouvé aucune réfutation de la conjecture à prouver. Nous pouvons en déduire que la conjecture est correcte.
- La preuve se termine sur un échec. Ce résultat peut avoir deux interprétations :
  - une ou plusieurs réfutations ont été trouvées. Normalement, cela signifie qu'un contre-exemple a été trouvé et que les fonctions de transformation sont incorrectes. Cependant, nous avons constaté que certains contre-exemples ne sont pas atteignables. Ces scénarios ne peuvent pas être construits par l'algorithme d'intégration. Il est très difficile d'exprimer complètement le prédicat *Conc*. Il faudrait spécifier l'algorithme d'intégration. Nous avons choisi de ne pas spécifier cet algorithme car le processus de preuve deviendrait beaucoup plus long et complexe.
    Ce compromis reste tout à fait acceptable. Le démonstrateur de théorème automatise la majeure partie de la preuve qui est fastidieuse, pour nous déléguer le traitement de quelques contre-exemples pertinents. La réfutation des contre-exemples est extrêmement intéressante car elle permet de dégager des propriétés supplémentaires sur les fonctions de transformation ou les algorithmes d'intégration.
  - aucune réfutation n'a été trouvée, mais le démonstrateur n'est plus capable de simplifier les conjectures restantes. Dans ce cas, nous devons ajouter des définitions ou des lemmes au système.

Par exemple, si nous reprenons la conjecture (G24) obtenue à la section précédente, il existe une simplification que le démonstrateur n'a pas été capable d'effectuer : $p_x - s(0) + s(0) = p_x$. Si nous réalisons cette simplification, nous obtenons :

$p_x = p_1, (p_x - s(0) \geq p_2) = \mathit{true}, p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = \mathit{true},$
$(p_x \geq p_2 + s(0)) = \mathit{false}, (p_2 < p_1) = \mathit{false}, (p_1 < p_2) = \mathit{false},$
$pr_1 \neq pr_2, 0 < p_1, 0 < p_2 \Rightarrow$
$\quad c_1 = \mathit{car}(\ p_x\ , xSt);$

Sur cette conjecture, on ne peut plus effectuer de simplification. En outre, il n'existe aucune contradiction entre les différentes hypothèses. Cette conjecture est un contre-exemple menant à la violation de la condition C_1.

Pour obtenir un scénario, il nous suffit :

- de reprendre les deux opérations sur lesquelles nous avons effectué la vérification, à savoir les deux opérations concurrentes $\mathit{Ins}(p_1, c_1, pr_1)$ et $\mathit{Del}(p_2, pr_2)$;
- d'instancier les variables avec des valeurs respectant les hypothèses du contre-exemple. Dans notre cas, nous posons $p_1 = p_2 = 2$, $pr_1 = 1$, $pr_2 = 2$, et $c_1 = x$;
- d'exécuter le scénario obtenu.

Nous venons de retrouver le contre-exemple connu [RNRG96] décrit à la figure 3.7. Ce contre-exemple nécessite deux utilisateurs $user_1$ et $user_2$ possédant chacun une réplique

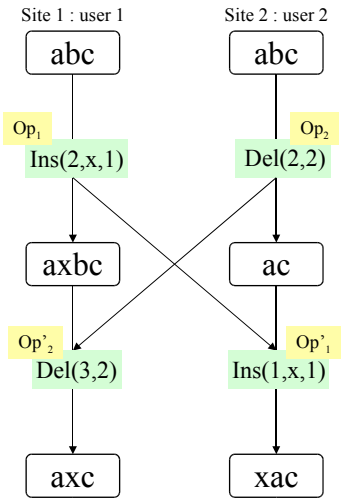


FIG. 3.7 – Contre-exemple violant la condition C_1.

de la chaîne de caractère `'abc'`.

1. L'utilisateur $user_1$ insère le caractère `'x'` à la position 2, produisant ainsi l'opération op_1. En parallèle, l'utilisateur $user_2$ détruit le caractère `'b'` se situant à la position 2 (ce qui génère l'opération op_2).
2. Lorsque l'opération op_2 est reçue par l'utilisateur $user_1$, elle doit être transformée par rapport à l'opération op_1 qui lui est concurrente. Ainsi, la transformation $T(Del(2,2), Ins(2,'x',1))$ est évaluée en $Del(3,2)$. L'exécution de cette opération donne la chaîne de caractères `'axc'`
3. De la même manière, à la réception de op_1 sur le site 2, cette opération doit être transformée par rapport à op_2. L'appel de la fonction $T(Ins(2,x,1), Del(2,2))$ est résolu en $Ins(1,x,1)$ qui après exécution donne la chaîne de caractère `'xac'`.

La condition C_1 est violée. En conséquence, les deux répliques n'ont pas convergé.

3.2.2 Proposition de Ressel et al. [RNRG96]

Nous nous intéressons maintenant à la vérification de la condition C_2. Pour cela nous allons étudier le jeu de fonctions de transformation proposé dans Ressel et al. [RNRG96].

Présentation

Les auteurs ont modifié la proposition originale de Ellis [EG89] afin que leurs fonctions de transformation vérifient les conditions C_1 et C_2. Ils ont remplacé le paramètre pr par le paramètre u_i. Ce paramètre représente le numéro du site qui a généré l'opération. Il est utilisé pour définir un ordre entre les opérations lorsque c'est nécessaire.

Dans l'article originel présentant cette solution, les fonctions de transformation pour deux opérations $Ins()$ et $Del()$ concurrentes sont identiques à celles proposées par Ellis [EG89]. Ces fonctions ne vérifiant pas la condition C_1, nous supposons que les auteurs font référence à une version corrigée de la fonction $T(Ins, Del)$ où l'inégalité stricte entre p_1 et p_2 a été remplacée par une inégalité non stricte($p_1 \leq p_2$). La figure 3.8 montre que cette retouche permet de corriger le contre-exemple violant la condition C_1 pour la proposition d'Ellis.

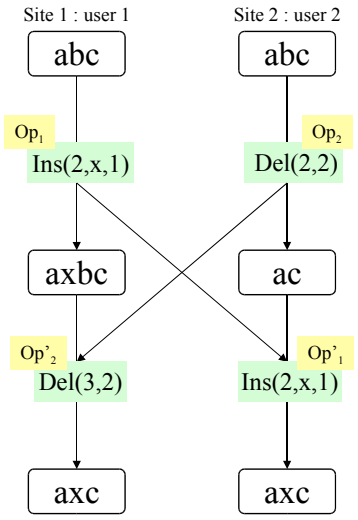


FIG. 3.8 – Exécution correcte du contre-exemple pour Ellis [EG89].

La seconde modification apportée par les auteurs est la définition de la fonction T de telle sorte que : lorsque deux opérations concurrentes insèrent un caractère à la même position p, le caractère produit par le site de plus petit identifiant est inséré à cette position p ; l'autre caractère étant inséré à la position $p + 1$.

On notera également que contrairement à la proposition d'Ellis, lorsque deux opérations concurrentes insèrent le même caractère à la même position, au final deux caractères sont insérés. Le choix originel consistait à n'insérer qu'un seul exemplaire du caractère en considérant que les deux utilisateurs effectuaient la même insertion.

La définition complète des fonctions de transformation est donnée par la figure 3.9.

Vérification et contre-exemple

Nous avons vérifié cette proposition avec SPIKE. Ces fonctions de transformation vérifient la condition C_1 mais il existe un contre-exemple pour C_2.

La conjecture initiale que nous avons cherché à vérifier est la suivante :

$$enabled(Ins(p_3, c_2, u_3), xSt) = true, enabled(Del(p_2, u2), xSt) = true,$$
$$enabled(Ins(p_1, c_1, u_1), xSt), conc(Del(p_2, u2), Ins(p_3, c_2, u_3)) = true,$$
$$conc(Ins(p_3, c_2, u_3), Ins(p_1, c_1, u_1)) = true, conc(Del(p_2, u2), Ins(p_1, c_1, u_1)) = true \Rightarrow$$

```
T(Ins(p1,c1,u1),Ins(p2,c2,u2)) :−
    if p1<p2 or (p1=p2 and u1<u2) return Ins(p1,c1,u1)
    else return Ins(p1+1,c1,u1)

T(Ins(p1,c1,u1), Del(p2,u2)) :−
    if p1 ≤ p2 return Ins(p1,c1,u1)
    else return Ins(p1 − 1,c1,u1)

T(Del(p1,u1),Ins(p2,c2,u2)) :−
    if p1 < p2 return Del(p1,u1)
    else return Del(p1 + 1,u1)

T(Del(p1,u1),Del(p2,u2)) :−
    if p1<p2 return Del(p1,u1)
    else if p1>p2 return Del(p1 − 1,u1)
    else return Id()
```

FIG. 3.9 – Fonctions de transformation proposées par Ressel et al. [RNRG96].

$$\begin{aligned} & T(\,T(Ins(p_1,\,c_1,\,u_1),\,Del(p_2,\,u2)),\,T(Ins(p_3,\,c_2,\,u_3),\,Del(p_2,\,u2))) \\ = {} & T(\,T(Ins(p_1,\,c_1,\,u_1),\,Ins(p_3,\,c_2,\,u_3)),\,T(Del(p_2,\,u2),\,Ins(p_3,\,c_2,\,u_3)));\end{aligned}$$

Spike a trouvé le contre-exemple suivant :

$(p_1 \leq p_2) = false, (p_1 + s(0) \leq p_2 + s(0)) = false,$
$(p_1 < p_3) = false, p_1 \neq p_3, p_1 - s(0) = p_3, (p_1 - s(0) < p_3) = false,$
$(p_2 < p_3) = false, (p_3 \leq p_2) = true,$
$u_1 < u_3,$
$u_1 \neq u_2, u_2 \neq u_3, u_1 \neq u_3, 0 < p_1, 0 < p_2, 0 < p_3 \Rightarrow$

Nous instancions les variables avec des valeurs tout en respectant les hypothèses. Ainsi, nous posons $p_1 = 3$, $p_2 = p_3 = 2$, et $u_1 < u_3$.

Nous obtenons le scénario illustré par la figure 3.10. Ce scénario requiert trois opérations concurrentes effectuées par trois utilisateurs différents : $op_1 = Ins(3, x)$, $op_2 = Del(2)$ et $op_3 = Ins(2, y)$. Dans un soucis de lisibilité, nous omettons le paramètre de priorité. Nous supposons qu'il a pour valeur le numéro de l'opération.

1. Tout d'abord, op_2 est reçue sur le site de l'utilisateur $user_3$ pour y être intégrée. op_2 est donc transformée par rapport à op_3 ce qui donne $op_2' = T(Del(2), Ins(2, y)) = Del(3)$.
2. Lorsque op_3 est intégrée sur le site 2, nous évaluons $op_3' = T(Ins(2, y), Del(2)) =$

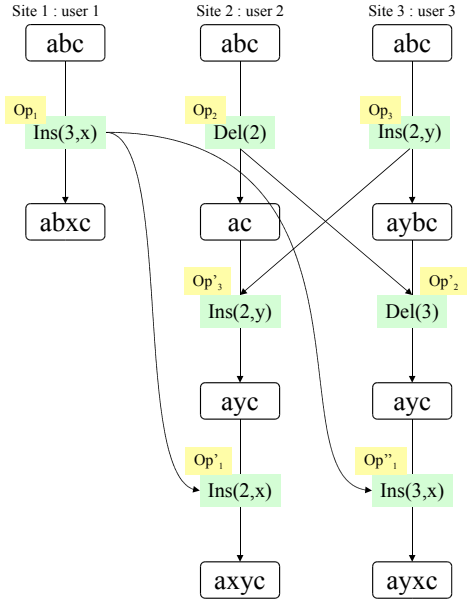


FIG. 3.10 – Contre-exemple violant la condition C_2 pour les fonctions de la figure 3.9.

$Ins(2, y)$.

3. Ensuite, nous intégrons op_1 sur le site 2 : op_1 doit être transformée par rapport à op_2, ce résultat étant lui-même transformé au regard de op'_3.

$$T(T(\overbrace{Ins(3,x)}^{op_1}, \overbrace{Del(2)}^{op_2}), \overbrace{Ins(2,y)}^{op'_3}) = \overbrace{Ins(2,x)}^{op'_1}$$

4. De la même manière nous intégrons op_1 sur le site 3.

$$T(T(\overbrace{Ins(3,x)}^{op_1}, \overbrace{Ins(2,y)}^{op_3}), \overbrace{Del(3)}^{op'_2}) = \overbrace{Ins(3,x)}^{op''_1}$$

Au final, nous obtenons sur les sites 2 et 3 deux opérations op'_1 et op''_1 différentes, ce qui contredit la condition C_2. En conséquence les états des copies sur les sites 2 et 3 n'ont pas convergé. Ce contre-exemple est un scénario déjà connu [SCF97] dans le domaine, nous y ferons référence dans la suite de ce document sous la dénomination "situation problématique TP2" [4].

Mise en évidence du problème

Lors de leur intégration sur le site 2, les deux opérations op_1 et op_3 sont considérées comme insérant un caractère différent à la même position. Or sur leur état de génération, clairement op_1 insère son caractère après celui de op_3.

[4] la dénomination anglaise étant "TP2 puzzle" où TP2 est une autre appellation de la condition C_2.

En effet, après transformation par rapport à op_2, op_1 donne l'opération $op_1^2 = Ins(2, x)$, et l'opération op_3 donne $op_3^2 = Ins(2, y)$. À partir de là, il n'est plus possible de distinguer les positions d'insertion de ces deux opérations.

Par conséquent, lors du calcul suivant, à savoir la transformation $T(op_1^2, op_3^2)$, les deux opérations sont considérées comme insérant des caractères différents à la même position. La priorité du site u_i est alors utilisé à mauvais escient pour décider lequel des deux caractères doit être placé devant l'autre.

3.2.3 Proposition de Suleiman et al. [SCF97]

Suleiman et al. [SCF97] proposent un jeu de fonctions de transformation pour un objet répliqué de type chaîne de caractères. Ils ajoutent deux paramètres supplémentaires à l'opération d'insertion. Ces paramètres *av* et *ap* conservent l'ensemble des opérations de suppression concurrentes à l'opération d'insertion. L'ensemble *av* contient les opérations de suppression qui ont effacé un caractère devant la position d'insertion. L'ensemble *ap* contient les opérations qui ont effacé un caractère derrière la position d'insertion.

Ainsi, pour deux opérations concurrentes $op_1 = Ins(p, c_1, av_1, ap_1)$ et $op_2 = Ins(p, c_2, av_2, ap_2)$ définies sur le même état et insérant un caractère à la même position, trois cas sont envisageables :

- si $(av_1 \cap ap_2) \neq \emptyset$ alors le caractère c_2 est inséré avant le caractère c_1,
- si $(ap_1 \cap av_2) \neq \emptyset$ alors le caractère c_2 est inséré après le caractère c_1,
- si $(av_1 \cap ap_2) = (ap_1 \cap av_2) = \emptyset$ alors une fonction $code(c_i)$ qui calcule un ordre total sur les caractères (tel que l'ordre lexicographique) est utilisée pour choisir lequel des deux caractères (c_1 ou c_2) doit être inséré à la position p (l'autre caractère étant quant à lui inséré à la position $p + 1$). La fonction $code(c_i)$ remplace le système de priorité utilisé dans les autres propositions.

Le jeu complet des fonctions de transformation est donné par la figure 3.11.

L'utilisation des deux paramètres additionnels *av* et *ap* dans les fonctions de transformation permet de corriger le scénario qui viole la condition C_2 pour la proposition de Ressel. L'exécution correcte de ce scénario est illustrée par la figure 3.12.

Le problème était situé sur le site 2. Nous rejouons donc l'intégration des différentes opérations sur ce site.

- Lorsque l'opération op_3 est reçue sur le site 2, elle est transformée par rapport à l'opération op_2. L'opération résultante de cette transformation est l'opération $op_3' = Ins(2, x, \{\}, \{Del(2)\})$. L'opération op_2 a été placée dans l'ensemble *ap* de op_3' car le caractère qu'elle détruit est situé derrière le caractère qu'op_3 insère.
- Lorsque l'opération op_1 est reçue sur le site 2, elle doit être transformée par rapport aux opérations op_2 et op_3'. La première transformation donne l'opération $op_1^2 = Ins(2, y, \{Del(2)\}, \{\})$. Cette fois ci, op_2 est placée dans l'ensemble *av* de op_1^2 car son effet se situe devant l'effet de op_1.
- Lors de la transformation de op_1^2 par rapport à op_3', les positions ne permettent pas

```
1  T(Ins(p1,c1,b1,a1),Ins(p2,c2,b2,a2)) :−
       if p1 < p2 return Ins(p1,c1,b1,a1)
3      else if p1 > p2 return Ins(p1 + 1,c1,b1,a1)
       else if (b1 ∩ a2) ≠ ∅ return Ins(p1 + 1,c1,b1,a1)
5      else if (a1 ∩ b2) ≠ ∅ return Ins(p1,c1,b1,a1)
       else if code(c1)>code(c2) return Ins(p1,c1,b1,a1)
7      else if code(c1)<code(c2) return Ins(p1 + 1,c1,b1,a1)
       else return Id()
9
   T(Ins(p1,c1,b1,a1),Del(p2)) :−
11     if p1 > p2 return Ins(p1 − 1,c1,b1+Del(p2),a1)
       else  return Ins(p1,c1,b1,a1+Del(p2))
13
   T(Del(p1,pr1),Ins(x2,p2,b2,a2)) :−
15     if p1 < p2 return Del(p1)
       else return Del(p1 + 1)
17
   T(Del(p1),Del(p2)) :−
19     if p1 < p2 return Del(p1)
       else if p1 > p2 return Del(p1 − 1)
21     else return Id()
```

FIG. 3.11 – Fonctions de transformation proposées par Suleiman et al. [SCF97].

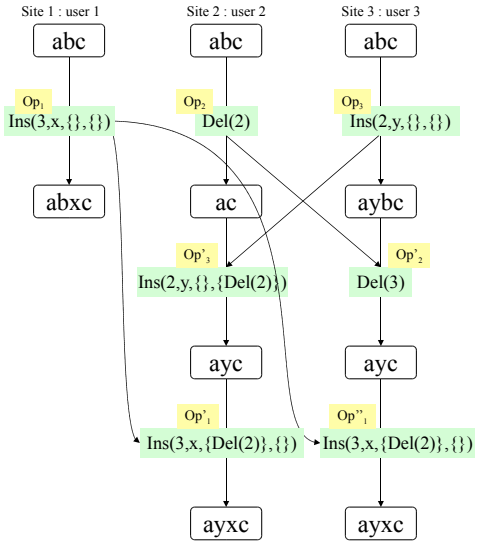


FIG. 3.12 – Exécution correcte du contre-exemple pour Ressel et al. [RNRG96].

de savoir laquelle des deux opérations à son effet devant l'autre. Dans la proposition de Ressel, nous utilisions la priorité du site u_i pour décider arbitrairement. En utilisant les fonctions de Suleiman, nous utilisons les ensembles supplémentaires av et ap. Comme op_2 est dans l'ensemble av de op_1^2, on en déduit op_1^2 "est derrière" op_2. Comme op_2 est dans l'ensemble ap de op_3', on en déduit que op_3' "est devant" op_2. Par transitivité, on en déduit que l'effet de op_1^2 est derrière l'effet de op_2^2. Par conséquent, la position de op_1^2 est incrémentée. Le résultat de la transformation est l'opération $op_1' = Ins(3, x, \{Del(2)\}, \{\})$.

Le contre-exemple est ainsi résolu et ne viole plus la condition C_2.

Vérification et contre-exemple

Le résultat de la preuve "à la main" de la vérification de la condition C_2 par ce jeu étant disponible dans la Thèse de Doctorat de Suleiman [Sul98], nous nous attendions à corroborer ses dires. En utilisant notre système de vérification, nous avons trouvé un contre-exemple illustré par la figure 3.13.

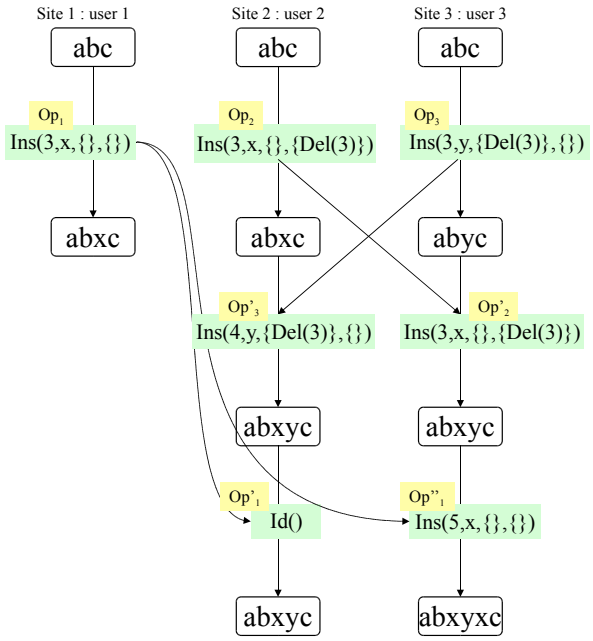


FIG. 3.13 – Contre-exemple pour Suleiman et al. [SCF97] violant la condition C_2.

Ce scénario fait intervenir trois opérations concurrentes : $op_1 = Ins(3, x, \{\}, \{\})$, $op_2 = Ins(3, x, \{\}, Del(3))$ et $op_3 = Ins(3, y, \{Del(3)\}, \{\})$. Le scénario se déroule de la manière suivante :

1. Le site 3 intègre l'opération op_2 en calculant :

$$T(\overbrace{Ins(3, x, \{\}, \{Del(3)\})}^{op_2}, \overbrace{Ins(3, y, \{Del(3)\}, \{\})}^{op_3}) = \overbrace{Ins(3, x, \{\}, \{Del(3)\})}^{op_2'}$$

2. Tandis que le site 2 intègre op_3 :

$$T(\overbrace{Ins(3, y, \{Del(3)\}, \{\})}^{op_3}, \overbrace{Ins(3, x, \{\}, \{Del(3)\})}^{op_2}) = \overbrace{Ins(4, y, \{Del(3)\}, \{\})}^{op'_3}$$

3. Ensuite, le site 2 intègre op_1 :

$$T(T(\overbrace{Ins(3, x, \{\}, \{\})}^{op_1}, \overbrace{Ins(3, x, \{\}, \{Del(3)\})}^{op_2}), \overbrace{Ins(4, y, \{Del(3)\}, \{\})}^{op'_3}) = \overbrace{Id()}^{op'_1}$$

4. Finalement, le site 3 intègre op_1 :

$$T(T(\overbrace{Ins(3, x, \{\}, \{\})}^{op_1}, \overbrace{Ins(3, y, \{Del(3)\}, \{\})}^{op_3}), \overbrace{Ins(3, x, \{\}, \{Del(3)\})}^{op'_2}) = \overbrace{Ins(5, x, \{\}, \{\})}^{op''_1}$$

Effectivement, nous constatons que les résultats finaux sur les sites 2 et 3 sont différents. La condition C_2 n'est donc pas satisfaite puisque :

$$\overbrace{Id()}^{op'_1} = T(op_1, op_2 \circ T(op_3, op_2)) \neq \overbrace{Ins(5, x, \{\}, \{\})}^{op''_1} = T(op_1, op_3 \circ T(op_2, op_3))$$

Nous devons nous assurer que ce scénario est atteignable. Autrement dit qu'il peut exister une exécution où les opérations op_1, op_2 et op_3, telles qu'elles sont définies, soient concurrentes.

Contrairement aux autres contre-exemples du domaine, les opérations incriminées ont déjà subi au moins une transformation. Leurs ensembles *av* et *ap* ne sont pas vides. Notre système de vérification ne nous fournit pas ces opérations supplémentaires. Mais, nous pouvons les déduire en étudiant les paramètres des différentes opérations.

- $op_1 = Ins(3, x, \{\}, \{\})$. Les deux ensembles *av* et *ap* sont vides. Nous en déduisons que l'opération op_1 n'a pas été transformée par rapport à une opération de suppression.
- $op_2 = Ins(3, x, \{\}, \{Del(3)\})$. L'ensemble *ap* contient une opération $Del(3)$. L'opération op_2 a donc subi une transformation par rapport à une opération $Del(3)$. Comme cette opération $Del(3)$ a été placée dans l'ensemble *ap*, nous en déduisons que la position d'insertion de l'opération op_2 était inférieure ou égale à la position 3 avant transformation. Nous pouvons ainsi supposer que l'opération op_2 résulte de la transformation de l'opération $op_2^{\#} = Ins(3, x, \{\}, \{\})$ par rapport à $Del(3)$.
- En effectuant le même raisonnement, nous déduisons que l'opération op_3 est une opération résultante de la transformation d'une opération $op_3^{\#} = Ins(4, y, \{\}, \{\})$ par rapport à une opération $Del(3)$.

En tenant compte de ces différentes hypothèses additionnelles, nous pouvons construire le scénario de la figure 3.14. Ce scénario englobe le scénario découvert par notre système de vérification automatique.

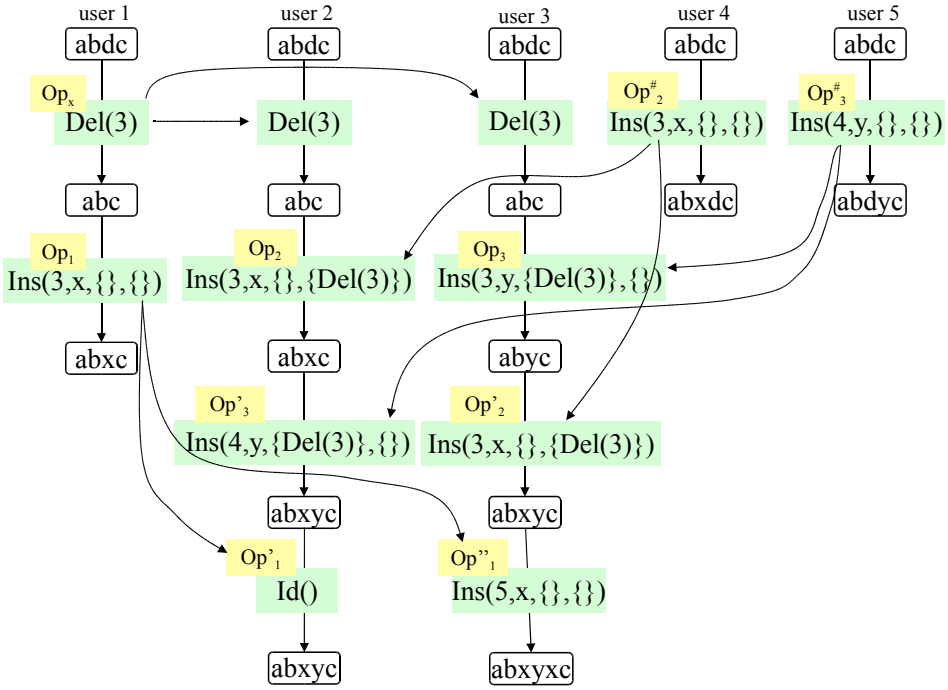


FIG. 3.14 – Contre-exemple complet pour Suleiman et al. [SCF97] violant la condition C_2.

Analyse

Lors de son intégration sur le site 2, l'opération op_1 est détectée comme équivalente à l'opération op_2. Elle est donc transformée en une opération identité. Tandis que sur le site 3, lors de leur intégration, les opérations op_1 et op_2 sont détectées comme différentes si `code('x')` est différent de `code('y')`.

Ce problème est dû à une perte d'information au cours du processus de transformation.

3.2.4 Proposition de Imine et al. [IMOR03]

En 2003, la proposition de Suleiman [SCF97] était encore considérée comme correcte. Cependant, nous avons proposé d'autres fonctions de transformation [IMOR03] plus simples qui résolvaient le problème détecté chez Ressel.

Nous pensions à cette époque que le problème survenait uniquement lors de la transformation de deux opérations d'insertion concurrentes. Pour résoudre ce problème, nous avons supprimé les ensembles *av* et *ap* et ajouté un paramètre supplémentaire *ip* qui mémorise la position initiale d'insertion de l'opération. Ainsi, lors de la transformation de deux opérations d'insertion dont les positions actuelles sont égales, nous comparons leur positions initiales. Ces positions nous permettent de savoir si les deux insertions à leur génération inséraient effectivement à la même position. Si tel est le cas, nous comparons les "codes des caractères" comme dans Suleiman [SCF97]. Sinon, nous décalons les positions en fonction de la relation entre les positions initiales.

```
1  T(Ins(p1,ip1,c1),Ins(p2,ip2,c2)) :−
     if (p1<p2) return Ins(p1,ip1,c1)
3    else if (p1>p2) return Ins(p1+1,ip1,c1)
     else if (ip1<ip2) return Ins(p1,ip1,c1)
5    else if (ip1>ip2) return Ins(p1+1,ip1,c1)
     else if (code(c1) < code(c2)) return Ins(p1,ip1,c1)
7    else if (code(c1) > code(c2)) return Ins(p1+1,ip1,c1)
     else return Id()
9
   T(Ins(p1,ip1,c1),Del(p2)) :−
11   if (p1>p2) return Ins(p1 − 1,ip1,c1)
     else return Ins(p1,ip1,c1)
13
   T(Del(p1),Del(p2)) :−
15   if (p1<p2) return Del(p1)
     else if (p1>p2) return Del(p1 − 1)
17   else return Id()

19 T(Del(p1,pr1),Ins(p2,ip2,c2)) :−
     if (p1<p2) return Del(p1)
21   else return Del(p1 + 1)
```

FIG. 3.15 – Fonctions de transformation conservant la position initiale [IMOR03].

À la génération d'une opération d'insertion, les paramètres p et ip sont égaux. Par exemple, si un utilisateur insère un caractère à la position 3 d'une chaîne, cette modification génère l'opération $Ins(3, 3,' x')$. Si cette opération est transformée, seule la position varie, la position initiale reste toujours identique.

La figure 3.15 donne la définition complète des fonctions de transformation.

Vérification et contre-exemple

En 2003, nous avons affirmé [IMOR03] que ces fonctions de transformation étaient correctes. Elles semblaient satisfaire les conditions C_1 et C_2. Par exemple, la figure 3.16 illustre la résolution du problème détecté dans la proposition de Ressel.

Nous rejouons l'intégration des différentes opérations sur le site 2.

– Lorsque l'opération op_3 est reçue sur le site 2, elle est transformée par rapport à l'opération op_2. L'opération résultante de cette transformation est l'opération $op'_3 = Ins(2, 2, y)$.

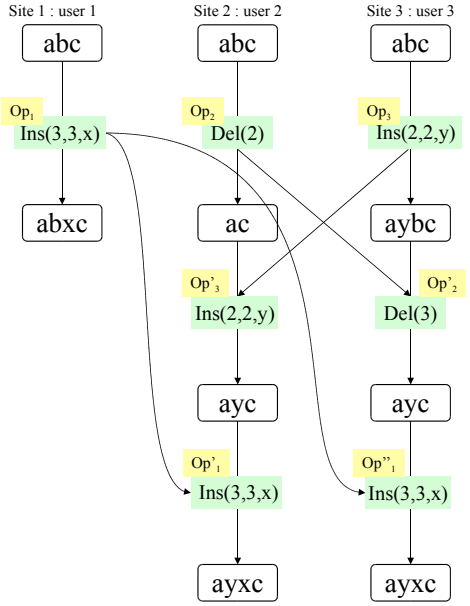


FIG. 3.16 – Exécution correcte du contre-exemple pour Ressel et al. [RNRG96].

- Lorsque l'opération op_1 est reçue sur le site 2, elle doit être transformée par rapport aux opérations op_2 et op'_3. La première transformation donne l'opération $op_1^2 = Ins(2, 3, x)$. La position d'insertion est décrémentée et la position initiale ne change pas.
- Lors de la transformation de op_1^2 par rapport à op'_3, les positions d'insertion ne permettent pas de savoir laquelle des deux opérations à son effet devant l'autre. Nous comparons donc les positions initiales des deux opérations. La position initiale de op_1^2 est égale à 3, tandis que celle de op'_3 à pour valeur 2. Nous incrémentons donc la position de op_1^2 pour obtenir l'opération $op'_1 = Ins(3, 3, x)$.

Le démonstrateur ne trouvant pas d'autre contre-exemple, nous pensions détenir des fonctions de transformation vérifiant la condition C_2.

Malheureusement, nous avons commis une erreur dans notre spécification et plus particulièrement dans la conjecture à vérifier. En effet, nous avions supposé que les trois opérations intervenant dans la condition C_2 n'avaient subi aucune transformation auparavant. Nous avions donc spécifié que ces trois opérations possédaient leurs positions égales à leur positions initiales. Or, pour que la condition C_2 puisse être généralisée à n opérations concurrentes, cette hypothèse ne doit pas être posée.

Nous avons corrigé notre spécification en autorisant cette fois ci des positions d'insertion initiales quelconques. Spike trouve alors un contre-exemple présenté par la figure 3.17. La figure 3.18 donne le scénario complet qui mène au contre-exemple précédent.

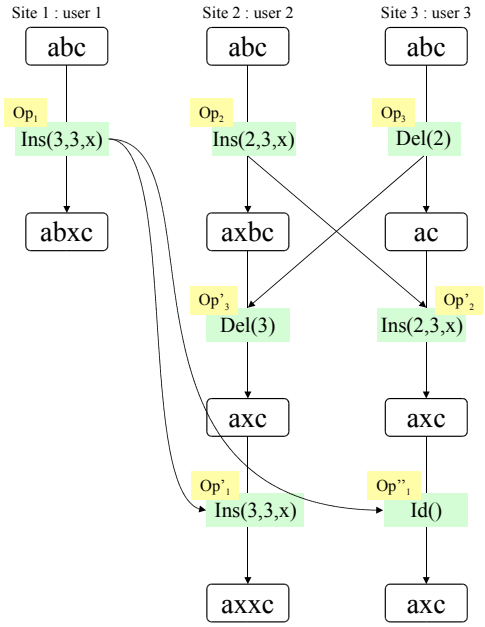


FIG. 3.17 – Contre-exemple pour les fonctions de la figure 3.15 violant la condition C_2.

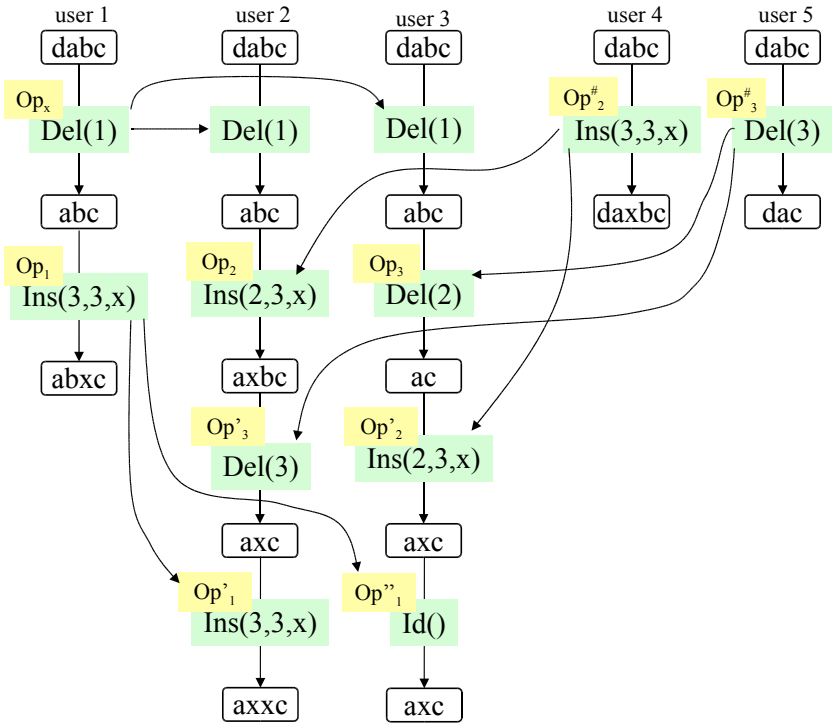


FIG. 3.18 – Contre-exemple complet pour les fonctions de la figure 3.15 violant la condition C_2.

Analyse

Ce dernier contre-exemple soulève un nouveau problème : les positions initiales ne permettent pas de départager convenablement deux opérations transformées qui n'ont pas été générées sur le même état initial. En effet, dans cet exemple, lors de leur intégration sur le site 3, les opérations op_1 et $op_2^{\#}$ sont considérées comme équivalentes. Or, si nous comparons les effets de ces opérations à leur génération, clairement elles ne sont pas équivalentes : op_1 insère le caractère `x` entre `b` et `c`, tandis que $op_2^{\#}$ insère le caractère `x` entre les caractères `a` et `b`. Nous pouvons même affirmer que l'effet de l'opération $op_2^{\#}$ est situé devant l'effet de l'opération op_1. Cette relation doit être préservée au cours des transformations. Sinon, les intentions des deux opérations sont violées.

Nous avions commis la même erreur pour les fonctions de Suleiman. Les ensembles av et ap des trois opérations intervenant dans la condition C_2 ne sont pas forcément vides.

3.2.5 Proposition de Imine et al. [IMOR04]

Pour résoudre le défaut de notre proposition précédente [IMOR03], nous avons proposé en 2004 [IMOR04] de ne pas conserver uniquement la position d'insertion initiale, mais l'ensemble des positions que prend une opération au cours de sa transformation. Nous appelons ce nouveau paramètre un p-word. Il peut être considéré comme une pile de positions. À sa génération, une opération d'insertion possède un p-word vide. À chaque transformation, on empile l'ancienne position d'insertion. Quand deux opérations insèrent à la même position, la comparaison de leur deux p-word selon l'ordre lexicographique donne la relation originelle entre ces deux opérations.

La figure 3.19 donne la définition des fonctions de transformation.

Nous utilisons la fonction $prepend(p, w)$ pour ajouter en tête la position p au p-word w. Le symbole $\prec$ dénote l'opérateur de comparaison selon l'ordre lexicographique. Par exemple, on a $[1;2;3] \prec [2;4]$.

Vérification

Ces fonctions de transformation vérifient la condition C_1.

La figure 3.20 montre comment la convergence est garantie sur le contre-exemple pour la proposition de Ressel. Sur leurs états de génération, l'effet de l'opération op_1 est derrière l'effet de l'opération op_2. Cette relation doit être préservée au cours des transformations. Ainsi, quand op_1 est intégrée sur le site 2, elle est transformée par rapport à op_2 et produit l'opération $Ins(2, x[3])$. Puis cette opération est transformée par rapport à $op'_3 = Ins(2, y, [2])$. En comparant les p-word, nous avons la relation $[2;2] \prec [2;3]$, l'opération résultante de la transformation est $op'_1 = Ins(3, x, [2;3])$. La relation a bien été préservée.

Bien que cette solution assure la convergence dans ce scénario, la condition C_2 est violée. Le p-word de l'opération $op'_1 = Ins(3, x, [2;3])$ n'est pas identique à celui de l'opération $op''_1 = Ins(3, x, [4;3])$. En fait, lorsqu'une opération est transformée par rapport à

$T(Ins(p_1, c_1, w_1), Ins(p_2, c_2, w_2)) =$
  let $\alpha_1 = prepend(p_1, w_1)$ and
    $\alpha_2 = prepend(p_2, w_2)$
  if $(\alpha_1 \prec \alpha_2$ or $(\alpha_1 = \alpha_2$ and $code(c_1) < code(c_2)))$ **return** $Ins(p_1, c_1, w_1)$
  else if $(\alpha_1 \succ \alpha_2$ or $(\alpha_1 = \alpha_2$ and $code(c_1) > code(c_2)))$ **return** $Ins(p_1 + 1, c_1, \alpha_1)$
  else return $Id()$

$T(Ins(p_1, c_1, w_1), Del(p_2)) =$
  if $(p_1 > p_2)$ **return** $Ins(p_1 - 1, c_1, prepend(p_1, w_1))$
  else if $(p_1 < p_2)$ **return** $Ins(p_1, c_1, w_1)$
  else return $Ins(p_1, c_1, prepend(p_1, w_1))$

$T(Del(p_1), Del(p_2)) =$
  if $(p_1 < p_2)$ **return** $Del(p_1)$
  else if $(p_1 > p_2)$ **return** $Del(p_1 - 1)$
  else return $Id()$

$T(Del(p_1), Ins(p_2, c_2, w_2)) =$
  if $(p_1 < p_2)$ **return** $Del(p_1)$
  else return $Del(p_1 + 1)$

FIG. 3.19 – Fonctions de transformation proposées utilisant des p-word [IMOR04].

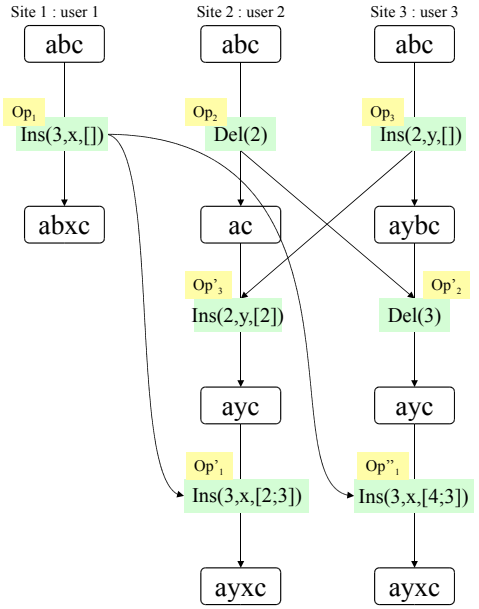


FIG. 3.20 – Exécution correcte du contre-exemple pour Ressel en utilisant des p-words

deux séquences équivalentes les deux p-word obtenus peuvent être différents.

Heureusement, nous pouvons montrer qu'ils sont équivalents si les deux positions d'insertion sont égales et si les deux dernières valeurs des p-word sont égales. Les autres valeurs contenues stipulant que les deux opérations ont été transformées par rapport à des chemins différents.

Définition 3.3 (Équivalence de deux opérations munies de p-word) Soient deux opérations op_1 et op_2, on a : $op_1 \equiv_{\mathcal{P}} op_2$ si et seulement si l'une des deux conditions suivantes est vraie :

1. $op_1 = op_2$;
2. $op_1 = Ins(p_1, c_1, w_1)$, $op_2 = Ins(p_2, c_2, w_2)$, $p_1 = p_2$, $c_1 = c_2$ et $last(w_1) = last(w_2)$.

où la fonction $last(w)$ retourne la dernière valeur de liste w, ou la valeur -1 si la liste est vide.

Nous définissons une nouvelle condition C_2' qui repose sur cette nouvelle équivalence.

Définition 3.4 (Condition C_2') Soient trois opérations op_1, op_2 et op_3 concurrentes définies sur le même état, la fonction de transposition en avant T vérifie la condition C_2' si et seulement si :

$$T(op_3, op_1 \circ T(op_2, op_1)) \equiv_{\mathcal{P}} T(op_3, op_2 \circ T(op_1, op_2))$$

Nous avons vérifié que notre jeu de fonctions de transformation vérifie la condition C_2'.

Contre-exemple

Malheureusement, les fonctions de transformation proposées ne permettent pas de préserver les relations entre tous les effets de n opérations concurrentes. La figure 3.21 illustre une telle situation.

- l'effet de l'opération op_1 est clairement situé derrière l'effet de l'opération op_4. Cette relation est bien préservée sur les sites 2 et 3, puisque le caractère `x` se retrouve derrière le caractère `y` ;
- de même, la relation entre l'effet de op_1 et l'effet de op_5 est également préservée. `x` se retrouve derrière `z` ;
- malheureusement, il n'existe pas de relation entre op_5 et op_4. Il n'y a donc pas de relation à préserver entre les caractères `z` et `y`. Ce qui peut conduire à une divergence des deux copies : l'une avec `y` devant `z` et l'autre avec `z` devant `y`.

3.3 Synthèse

Toutes les fonctions de transformation que nous avons vérifiées sont fausses. En fait à chaque nouvelle tentative, soit on déplace la "situation problématique TP2", soit on tombe sur la nouvelle situation décrite à la figure 3.22.

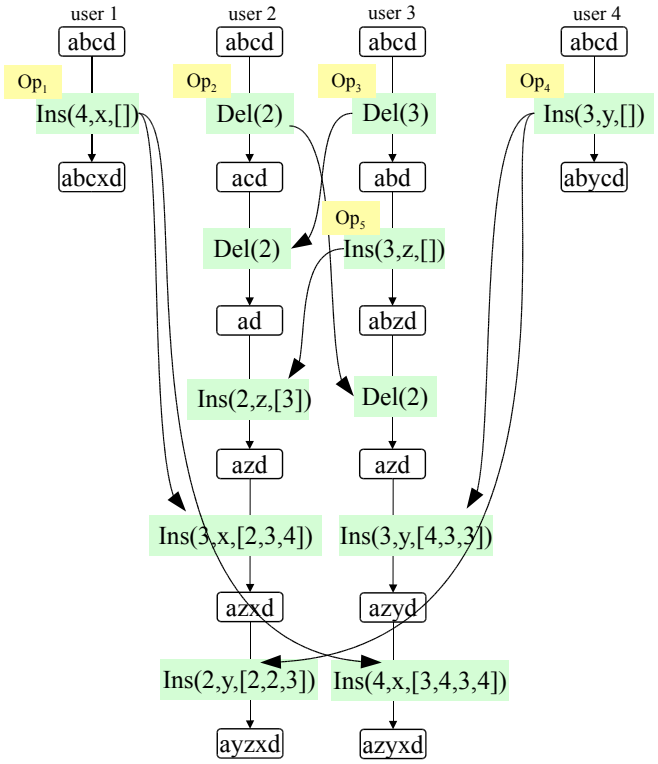


FIG. 3.21 – Contre-exemple pour les p-words violant la convergence.

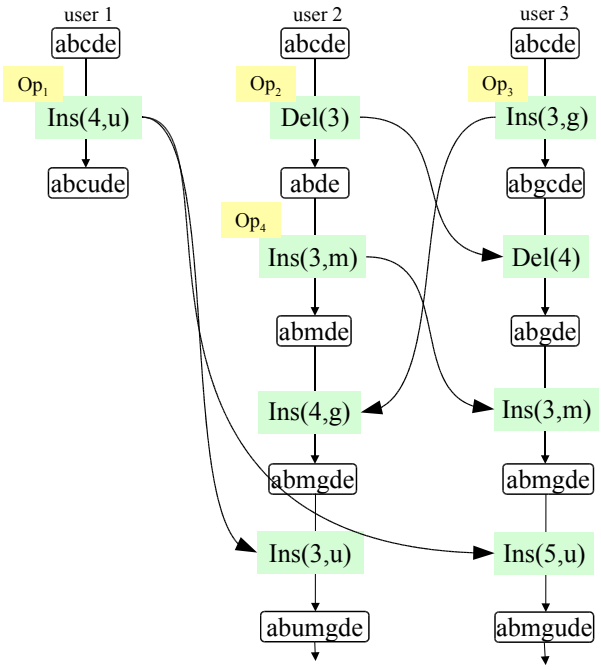


FIG. 3.22 – Nouvelle situtation problématique à concurrence partielle.

Cette nouvelle situation pose de nouveaux problèmes. Il est clair, dans cet exemple, que tous les sites doivent converger vers un état où `'u'` est après `'g'`. Effectivement, sur l'état initial, `'g'` précède `'c'` et `'c'` précède `'u'`. On a donc $g \prec c \prec u$. Il y a bien convergence vers l'état attendu sur le site 3 mais pas sur le site 2.

Si on analyse le problème, l'erreur est commise quand op_1 est transformée par rapport à op_4. À ce moment là, il faut calculer $T(Ins(3,u), Ins(3,m))$. Il n'existe aucune relation entre ces deux opérations. Par exemple, on peut ordonner les caractères selon l'ordre des sites.`'u'` est donc placé devant `'m'`. C'est bien ce qu'il aurait fallu faire si op_4 était directement concurrente à op_1. Mais dans cet exemple, op_4 est partiellement concurrente à op_1. C'est cette concurrence partielle qui rend le problème impossible à résoudre avec juste les informations collectées lors des transformations.

En fait, tout ce que l'on sait sur `'m'` c'est qu'il est entre `'b'` et `'d'`. On ne sait pas comment il est placé par rapport à `'c'` et c'est normal car `'c'` a été détruit avant la génération de `'m'`. Pour résoudre le problème il faut décider arbitrairement si `'c'` est devant ou derrière `'m'`.

Très concrètement, pour résoudre le problème, il faudrait, avec l'approche de Suleiman, quand une opération $ins()$ est générée remplir les ensembles en tenant compte de toutes les opérations $del()$ qui la précède dans l'histoire locale. Cela revient à transformer en arrière l'opération générée par rapport à l'histoire locale de manière à remplir les ensembles de manière adéquate.

L'algorithme SDT [LL04a, LL04b] propose finalement cette solution sans donner d'exemple pour la motiver. Quand SDT transforme deux opérations d'insertion concurrentes o_1 et o_2 définies sur le même état s, SDT calcule $\beta_{lsp}(o_1)$ et $\beta_{lsp}(o_2)$ qui sont les positions de o_1 et o_2 sur un état appelé LSP[5]. Cet état est identifié par le vecteur d'horloge $V_{min} = min(v(o_1), v(o_2))$. $v(o_1), v(o_2)$ sont les vecteurs d'horloge de o_1 et o_2. V_{min} est construit avec la valeur minimale de chaque composant des vecteurs d'horloge de o_1, o_2. SDT commence par calculer la séquence SQ d'opération qui génère l'état s en partant de l'état LSP. Puis il calcule une séquence SD équivalente à SQ mais minimale. Enfin, SDT enlève l'effet de SD sur o_1 et o_2 et obtient donc les positions de o_1 et o_2 sur l'état LSP. La comparaison de ces deux positions permet de trancher en respectant les intentions.

Sur notre exemple de la figure 3.22 , quand SDT transforme $op_1 = ins(4,u)$ par rapport à $op_4 = ins(3,m)$, V_{min} référence l'état initial. Il doit donc comparer les positions de op_1 et op_4 sur cet état. Mais op_4 n'existe pas sur cet état. SDT va alors soustraire l'effet de op_2 sur op_4. op_4 peut alors être réécrite en $ins(3,m)$ ou $ins(4,m)$. SDT décide toujours dans ce cas de trancher en faveur de $ins(4,m)$. Même sur l'état LSP, les deux opérations ne peuvent être départagées, SDT prend alors l'ordre des sites et place `'u'` devant `'m'`. SDT va faire la même chose entre $op_3 = ins(3,g)$ et op_4. Il va placer `'g'` devant `'m'`. Quand op_1 est transformée par rapport à op_3, `'g'` est devant `'u'`. Donc l'état final sera $g \prec u \prec m$. Ce résultat préserve bien les intentions.

[5]Last Synchronization Point

Malgré tout, SDT parait démesurément complexe pour résoudre un problème qui semble simple. Il nous a semblé qu'à ce stade, le principal obstacle pour assurer la convergence, tout en préservant les intentions, était l'approche des transformées opérationnelles elle-même. Si au départ l'approche semble simplifier le problème de réconciliation en le ramenant à l'écriture de fonctions respectant C_1et C_2, les problèmes de concurrence partielle font perdre cet avantage.

En analysant SDT, nous avons conclu que l'algorithme déploie un effort colossal pour calculer et recalculer les relations de précédence entre les caractères. Or, nous disposons de cette information dès la génération des opérations . Nous avons eu alors l'idée de changer le profil des opérations. Et si au lieu d'envoyer $ins(3, m)$, nous envoyons $ins(b \prec m \prec d)$? Ce point de départ a conduit à l'algorithme WOOT détaillé dans le chapitre 4.

Chapitre 4

WOOT : une nouvelle approche pour réconcilier des structures linéaires divergentes

Sommaire

Dans ce chapitre nous présentons les travaux que nous menons actuellement concernant la conception d'un nouvel algorithme de réplication optimiste d'une structure linéaire. Nous ne possèdons pas actuellement de preuve formelle de la convergence de cet algorithme. Cet algorithme ne repose pas sur le modèle des transformées opérationnelles.

L'idée de notre nouvelle approche est simple. Elle consiste à diffuser les relations avec les opérations au lieu de recalculer celles-ci lors de l'intégration. En effet, lorsque nous générons une opération, les relations sont connues, alors qu'à la réception les algorithmes OT doivent les recalculer.

Quand un utilisateur observe la chaîne de caractères `"ABCDE"` et qu'il insère `"12"` à la position 2, en réalité, il insère `"12"` entre `A` et `B`. Respecter l'intention des effets de cette opération consiste à préserver la relation $'A' \prec$ "12" $\prec' B'$ sur tous les états futurs de la chaîne. Comme nous venons de le voir au chapitre précédent, les algorithmes reposant sur le modèle des transformées opérationnelles peinent à préserver ces relations. Cette déficience entraîne le plus souvent une divergence des copies.

Nous devons donc changer le profil des opérations. Dans le modèle des transformées opérationnelles, nous exécutons et nous diffusons l'opération $insert(2, "12")$. Dans notre approche, nous exécutons l'opération $insert(2, "12")$, mais nous diffusons l'opération $insert('A' \prec "12" \prec' B')$.

Le premier problème qui se pose est comment exécuter l'opération $insert('A' \prec "12" \prec' B')$ si localement le caractère `'A'` est supprimé ? Notre solution est très simple : nous ne détruisons pas les caractères, nous les marquons seulement comme invisibles. Ainsi, `'A'` existera toujours. Bien sûr, si nous ne détruisons pas les caractères, notre approche consomme plus de mémoire et génère des fichiers plus volumineux. Nous montrons dans la section 4.2.2 que la complexité en mémoire de notre algorithme est inférieure ou égale à celle des algorithmes OT. En effet, nous ne conservons pas le journal des opérations et nous n'utilisons pas de vecteurs d'horloges.

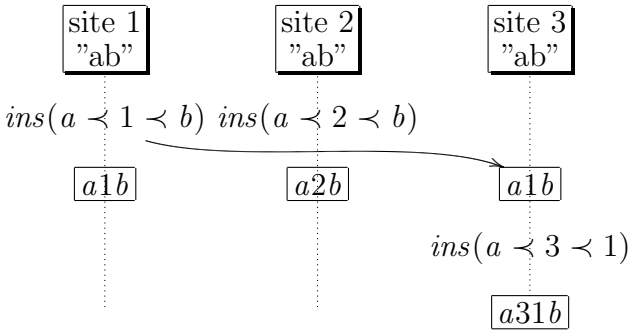


FIG. 4.1 – Exemple de génération d'un ordre partiel.

Chaque opération d'insertion génère de nouvelles relations. L'ensemble des relations ne constitue pas un ordre total, mais uniquement un ordre partiel. Par exemple, trois sites génèrent chacun une opération comme décrit dans la figure 4.1. Le diagramme de Hasse, présenté à la figure 4.2, représente graphiquement l'ensemble des relations entre

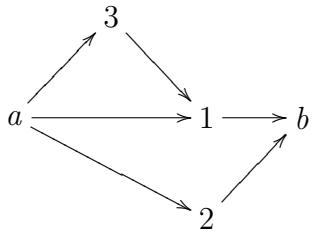


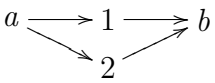
FIG. 4.2 – Diagramme de Hasse des relations d'ordre entre les caractères.

les caractères.

Les états de convergence où les intentions sont préservées correspondent à l'ensemble des extensions linéaires de cet ordre partiel, c'est-à-dire `a312b`, `a321b`, `a231b`. Cependant, pour assurer la convergence de l'ensemble des copies, tous les sites doivent trouver la même extension linéaire.

Généralement, on utilise un tri topologique pour trouver une extension linéaire d'un ensemble partiellement ordonné. Pourtant, dans notre cas, nous ne pouvons pas utiliser un tri topologique quelconque. En effet, chaque fois qu'une opération est reçue, nous effectuons une linéarisation et l'utilisateur observe un nouvel état. La nouvelle linéarisation doit être compatible avec toutes les linéarisations précédentes. Autrement dit, si une linéarisation a déterminé que le caractère `'a'` est devant le caractère `'b'`, les prochaines linéarisations ne peuvent remettre en cause ce choix.

Considérons l'exemple suivant sur lequel nous appliquons un tri topologique ne respectant pas la contrainte évoquée ci-dessus :



- Premièrement, nous choisissons un nœud sans prédécesseur. Il en existe un seul : `'a'`. Nous l'empilons sur la pile de résultat ;
- nous supprimons ce nœud et les arcs partant de ce nœud ;
- il y a maintenant deux nœuds qui ne possèdent pas de prédécesseur : `'1'` et `'2'`. Nous avons besoin d'un axiome de choix. Nous choisissons de traiter le nœud de valeur inférieure en premier. Nous empilons donc le nœud `'1'`, nous le supprimons du graphe ainsi que ses arcs ;
- nous répétons ces manipulations jusqu'à obtenir un graphe vide.
- le résultat final de notre tri topologique est `"a12b"`.

Nous supposons, maintenant, que l'opération $insert(a \prec 3 \prec 1)$ est reçue sur le même site. Nous ré-exécutons le tri topologique sur le diagramme de la figure 4.2. Nous obtenons `"a231b"` qui n'est pas compatible avec la linéarisation précédente, en effet `2` est maintenant devant `1`. Ce tri topologique n'est donc pas adapté à notre approche.

L'objectif de cette nouvelle approche est donc de garantir la convergence en utilisant une fonction de linéarisation *monotone*. Dans la suite de ce chapitre, nous manipulons une chaîne de caractères, mais cette approche peut être adaptée pour n'importe quelle

structure linéaire. Cette structure linéaire peut être complexe comme par exemple un document XML modélisé sous la forme un arbre ordonné.

4.1 WOOT : le modèle

Dans cette section, nous présentons le modèle WOOT et ses algorithmes. Nous commençons par définir formellement les structures de données utilisées, ainsi que les relations d'ordre prises en compte pour linéariser les caractères. Puis, nous explicitons les algorithmes. Enfin, nous évaluons le modèle en terme de correction et de complexité.

4.1.1 Modèle de cohérence PCI

Notre mécanisme de réplication optimiste garantit le modèle de cohérence PCI que nous définissons par les trois critères suivants :

Respect des Pré-conditions Les opérations reçues sont intégrées si leurs pré-conditions sont satisfaites. Les pré-conditions d'une opération exprime sous quelles conditions cette opération peut s'exécutée.

Convergence des Copies Lorsque tous les sites ont exécuté les mêmes opérations, les répliques sont identiques.

Respect de l'Intention Pour toute opération *op*, les effets de l'exécution de *op* sur tous les sites sont les mêmes que les effets de l'exécution de l'opération *op* sur son état de génération.

Contrairement au modèle de cohérence originel des transformées opérationnelles [EG89], le respect de la causalité n'est pas requis. Une opération peut être intégrée dès que sa pré-condition est vraie. Par exemple, un utilisateur exécute localement $o_1 = ins(a \prec' 1' \prec b)$ et diffuse cette opération. Puis, il exécute $o_2 = ins(c \prec' 2' \prec d)$ et diffuse également cette opération. Il se peut qu'un autre site reçoive o_2 avant o_1. Clairement, si le respect de la causalité est requis, o_2 doit être exécutée après o_1. Dans notre approche, nous autorisons l'intégration de o_2 dès que sa pré-condition est vraie ; autrement dit, pour une opération $ins(c \prec' 2' \prec d)$ dès que les caractères `'c'` et `'d'` existent sur la copie locale.

D'un côté, en relâchant la causalité, nous autorisons une concurrence accrue. D'un autre côté, si il existait une dépendance cachée entre o_1 et o_2, celle-ci risque d'être violée. Nous avons préférée améliorer la concurrence, les intentions étant préservées.

Notre modèle de cohérence PCI est similaire au modèle de cohérence CCI [SJZ+98]. Nous avons juste remplacé le respect de la causalité par le respect des pré-conditions.

4.1.2 Structures de données

Chaque site s possède un identifiant unique $numSite_s$, une horloge logique H_s, une séquence $string_s$ de W-caractères, et un ensemble $pool_s$ d'opérations en attente.

DÉFINITION 4.1 Un *W-caractère* c est un quintuplet $< id, v, \alpha, id_{cp}, id_{cn} >$ où
- id est l'identifiant unique du caractère ;
- $v \in \{Vrai, Faux\}$ indique si le caractère est visible ou non ;
- α est la valeur alphabétique du caractère ;
- id_{cp} est l'identifiant du W-caractère précédent c lors de la génération de c ;
- id_{cn} est l'identifiant du W-caractère suivant c lors de la génération de c ;

DÉFINITION 4.2 Le W-caractère précédent d'un W-caractère c est noté $C_P(c)$. Le W-caractère suivant d'un W-caractère c est noté $C_N(c)$.

DÉFINITION 4.3 Un *identifiant de caractère* est un couple (ns, ng) où ns est l'identifiant du site de génération et ng est un entier naturel. Quand un caractère est généré sur un site s, son identifiant est fixé à la valeur $(numSite_s, H_S)$ où $numSite_s$ dénote l'identifiant unique du site s et H_s la valeur actuelle de l'horloge logique de s. Les identifiants sont donc ordonnés.

Chaque fois qu'un W-caractère est généré sur un site s, l'horloge logique de ce site H_s est incrémentée. Puisque *numSite* est unique, le couple $(numSite_s, H_s)$ forme un identifiant unique pour le caractère.

$string_s$ est une W-chaîne. Elle contient tous les caractères qui ont été intégrés sur le site s.

DÉFINITION 4.4 Une *W-chaîne* est une séquence ordonnée de W-caractères $C_b\, c_1\, c_2 \ldots c_n\, C_e$ où C_b et C_e sont des W-caractères spéciaux (possédant des identifiants spéciaux) marquant le début et la fin de la séquence.

Sur une séquence S, nous définissons les fonctions suivantes :
- $|S|$ dénote la longueur de la séquence S (C_b et C_e sont pris en compte) ;
- $S[p]$ dénote l'élément situé à la position $p \in \mathbb{N}$ dans la séquence S. Nous considérons que le premier élément a comme position 0 ; le dernier élément a donc pour position $|S| - 1$;
- $pos(S, c)$ retourne la position de l'élément c dans la séquence S ;
- $insert(S, c, p)$ insère l'élément c à la position p dans S ;
- $subseq(S, c, d)$ retourne la sous-séquence de S située entre les éléments c et d ; ces deux éléments exclus ;
- $contains(S, c)$ retourne vrai si s contient l'élément c.

Nous avons besoin également des fonctions suivantes afin de faire le lien entre une W-chaîne et la chaîne de caractère qu'un utilisateur observe en réalité.
- $value(S)$ retourne la représentation de S, c'est-à-dire la séquence des valeurs alphabétiques des W-caractères visibles ;
- $ithVisible(S, i)$ retourne le ième W-caractère visible dans S.

Une W-chaîne S peut être modifiée par deux opérations :

ins(c) insère le W-caractère c entre son précédent et son suivant. La pré-condition de cette opération stipule que les caractères précédent et suivant doivent être dans S.

del(c) supprime le W-caractère c. La pré-condition stipule que le W-caractère c doit être dans S.

4.1.3 Relations d'ordre

DÉFINITION 4.5 (RELATION DE PRÉCÉDENCE $\prec$) Soient deux W-caractères a et b. On a $a \prec b$ si et seulement si il existe une séquence de W-caractères $c_0, c_1, ...c_i$ telle que $a = c_0, b = c_i$ et $C_N(c_j) = c_{j+1}$ et $c_j = C_P(c_{j+1})$ pour tout j $0 \leq j \leq i$.

$\prec$ est une relation binaire sur l'ensemble des W-caractères d'une W-chaîne. $\prec$ est non réflexive, transitive et asymétrique. $\prec$ est une relation d'ordre partielle stricte.

(i) $\forall a$ un W-caractère, $\neg(a \prec a)$ (non réflexive)

(ii) $\forall a, b$ deux W-caractères, $\neg((a \prec b) \wedge (b \prec a))$ (non symétrique)

(iii) $\forall a, b, c$ trois W-caractères, $(a \prec b) \wedge (b \prec c) \Rightarrow (a \prec c)$ (transitive)

Pour obtenir une chaîne de caractères à partir de cet ordre partiel, nous devons trouver une extension linéaire, autrement dit un ordre total.

DÉFINITION 4.6 (RELATION $\leq_S$) Soient une séquence S, deux W-caractères a et b appartenant à S, la relation $a \leq_S b$ est définie si et seulement si $pos(S, a) \leq pos(S, b)$

Lorsque nous ne pouvons pas établir de relation de précédence entre deux caractères, nous devons les ordonner. Pour assurer la convergence des copies, cet ordonnancement doit être indépendant de l'état du site. Nous avons choisi de comparer les identifiants des caractères.

DÉFINITION 4.7 (RELATION $<_{id}$) Soient deux W-caractères a et b munis de leur identifiant respectif (ns_a, ng_a) et (ns_b, ng_b). La relation $a <_{id} b$ est définie si et seulement si (1) $ns_a < ns_b$ ou (2) $ns_a = ns_b$ et $ng_a < ng_b$.

La relation $<_{id}$ est une relation d'ordre totale.

4.1.4 Algorithmes

Quand un utilisateur demande à modifier une réplique, l'opération correspondante est générée. Cette opération est (i) immédiatement intégrée en local, puis (ii) diffusée aux autres sites. Elle est ensuite (iii) reçue par les autres sites, pour être y enfin (iv) intégrées.

L'opération doit être intégrée sur la copie locale afin que le caractère inséré soit positionné par rapport aux caractères invisibles, qui ont été détruits, comme il le sera sur les autres sites.

```
GenerateIns(pos, α)
    H_s := H_s + 1
    let c_p := ithVisible(string_S, pos),
        c_n := ithVisible(string_S, pos + 1),
        wchar :=< (numSite_s, H_s), True, α, c_p.id, c_n.id >
    IntegrateIns(wchar, c_p, c_n)
    broadcast ins(wchar)

GenerateDel(pos)
    let wchar := ithVisible(string_s, pos)
    IntegrateDel(wchar)
    broadcast del(wchar)
```

FIG. 4.3 – Algorithme de *GenerateIns* et *GenerateDel*.

Génération d'une opération

Du point de vue de l'utilisateur, la chaîne ne contient que les caractères visibles. Elle a donc pour valeur $value(S)$. Ainsi, lorsque l'interface utilisateur génère une opération d'insertion, cette opération ne contient que la valeur alphabétique du caractère à insérer et sa position dans la chaîne visible. Cette opération doit donc être traduite en une opération mémorisant les relations d'ordre.

Par exemple, l'opération $ins(2, a)$ effectuée dans la chaîne de caractères `'xyz'` sera traduite en l'opération $ins(y \prec a \prec z)$. De même, lors de la génération d'une opération de suppression, nous devons retrouver le W-caractère qui doit être supprimé.

Les deux procédures réalisant cette traduction sont présentées à la figure 4.3.

Réception d'une opération

La figure 4.4 illustre la boucle principale de réception des opérations sur un site. L'ordre de réception des opérations peut être différent de l'ordre de génération. Ainsi, une opération peut être reçue sur site, mais ne pas pouvoir y être exécutée car ses pré-conditions ne sont pas vraies. Par exemple, un site peut exécuter l'opération $del(c)$ si et seulement si le caractère c est présent dans la chaîne. Si c n'est pas présent, l'intégration de cette opération doit être retardée.

Pour gérer les opérations en attente, chaque site maintient un ensemble d'opérations en attente $pool_s$.

Supposons que le système se stabilise, c'est-à-dire que plus aucune opération ne soit produite et que toutes les opérations produites finissent par arriver sur tous les sites. Dans ce cas, on peut affirmer que toutes les opérations finiront par être exécutées. En effet, les

```
Main()
  loop
    find op in pool_s s.t isExecutable(op)
    let c := char(op)
    if type(op) = del then
      IntegrateDel(c)
    else
      IntegrateIns(c, C_P(c), C_N(c))
    endif
  endloop
```

FIG. 4.4 – Boucle principale de l'algorithme de réception.

```
isExecutable(op)
  let c := char(op)
  if type(op) = del then
    return contains(string_s, c)
  else
    return contains(string_s, C_P(c)) and contains(string_S, C_N(c))
  endif
```

FIG. 4.5 – Algorithme de *isExecutable*.

pré-conditions de ces opérations ne dépendent que de l'exécution d'autres opérations. Ces dépendances n'étant pas cyclique, il est inéluctable que les pré-conditions seront vraies et donc que toutes les opérations seront inéluctablement exécutées.

```
Reception(op)
  add op to pool_s
```

La figure 4.5 donne la définition de la fonction *isExecutable* qui vérifie si les pré-conditions d'une opération sont satisfaites. Dans cette définition, la fonction *type*(*op*) retourne le type de l'opération *op* à savoir *ins* ou *del* ; et la fonction *char*(*op*) retourne le W-caractère manipulé par l'opération *op*.

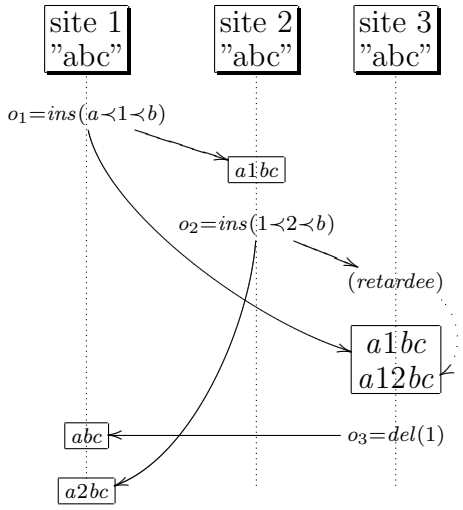


FIG. 4.6 – Réception non causale.

```
1 IntegrateDel(c)
    c.v := False
```

FIG. 4.7 – Procédure d'intégration d'une opération de suppression.

Exemple Dans le scénario de la figure 4.6, le site 3 doit retarder l'intégration de l'opération o_2 jusqu'à l'intégration de l'opération o_1. Le site 1 peut tout à fait exécuter l'opération o_3 avant de recevoir l'opération o_2 ; Et ce, même si l'opération o_2 a été produite avant l'opération o_3. Dans les approches traditionnelles qui reposent sur l'utilisation de vecteur d'horloges, l'exécution des opérations serait retardée dans les deux cas.

Intégration d'une opération

Comme nous ne détruisons pas les caractères, l'intégration d'une opération $del(c)$ est triviale. Nous rendons simplement le caractère invisible qu'il le soit déjà ou non. La figure 4.7 décrit la procédure utilisée.

L'intégration d'une opération $ins(c_p \prec c \prec c_n)$ dans la chaîne $string_s$ est plus délicate. Nous devons placer c parmi les caractères situés entre c_p et c_n. Ces caractères peuvent être des caractères qui ont été supprimés précédemment ou des caractères qui ont été insérés en concurrence. La procédure $IntegrateIns(c, c_p, c_n)$ décrite à la figure 4.8 réalise cet ordonnancement.

L'algorithme doit ordonner les caractères, qui n'ont pas de relation selon $\prec$ à respecter, selon la relation $<_{id}$. L'algorithme filtre tous les caractères de S' qui possèdent

```
IntegrateIns(c, c_p, c_n)
  let S := string_s
  let S' := subseq(S, c_p, c_n)
  if S' = ∅ then
     insert(S, c, pos(S, c_n))
  else
     let  L := c_p d_0 d_1 ... d_m c_n où d_0 ... d_m sont les
          W−caractères de S' tels que C_P(d_i) ≤_S c_p et c_n ≤_S C_N(d_i)
     let i := 1
     while (i < |L| − 1) and (L[i] <_id c) do
          i := i + 1
     IntegrateIns(c, L[i − 1], L[i])
  endif
```

FIG. 4.8 – Procédure d'intégration d'une opération d'insertion.

leur caractère prédécesseur ou leur caractère successeur dans S' ; car, ces caractères sont ordonnés selon la relation de précédence et non selon la relation $<_{id}$. Ainsi, tous les caractères présents dans $d_0 d_1 ... d_m$ sont triés selon la relation $<_{id}$. L'algorithme n'a donc plus qu'à insérer le caractère c à la bonne position selon cette relation. Des caractères peuvent exister entre $L[i-1]$ et $L[i]$ dans la chaîne S. Le caractère c doit être ordonné par rapport aux caractères situés entre ces deux caractères. C'est pourquoi l'algorithme effectue un appel récursif.

La construction de l'ordre $\leq_S$ est monotone. L'insertion d'un nouveau caractère dans une chaîne ne remet jamais en cause l'ordonnancement des autres caractères de cette chaîne.

4.1.5 Exemples

Dans cette section, nous donnons deux exemples de fonctionnement de WOOT.

Exemple 1

Soient trois sites site 1, site 2 et site 3 possédant chacun une réplique d'une chaîne vide qui contient donc uniquement "$c_b c_e$". Nous considérons le scénario de la figure 4.9.

Une fois toutes les opérations reçues, le scénario produit le diagramme de Hasse illustré par la figure 4.10.

Nous supposons que la relation $<_{id}$ ordonne les caractères de la manière suivante : '1' $<_{id}$ '2' $<_{id}$ '3' $<_{id}$ '4'. Nous supposons que le site 2 reçoit les opérations o_1, o_3, o_4 dans cet

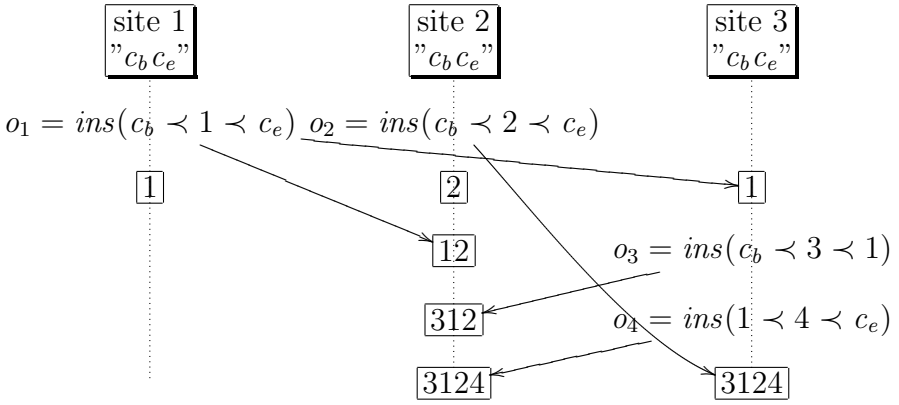


FIG. 4.9 – Scénario d'exemple 1.

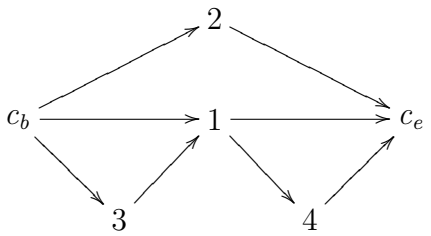


FIG. 4.10 – Diagramme de Hasse résultant du scénario d'exemple 1.

ordre. L'intégration se déroule de la manière suivante :

1. *IntegrateIns*$(1, c_b, c_e)$: $S' = L =$ "2" et $1 <_{id} 2$, WOOT intègre '1' entre 'c_b' et '2'. Au cours de l'appel récursif *IntegrateIns*$(1, c_b, 2)$, nous obtenons $S' = \varnothing$. Et donc, le résultat obtenu est la chaîne "$c_b 12 c_e$".
2. *IntegrateIns*$(3, c_b, 1)$: $S' = \varnothing$, WOOT insère '3' entre c_b et '1'.
3. *IntegrateIns*$(4, 1, c_e)$: $S' = L =$ "2" et $2 <_{id} 4$, WOOT intègre '4' entre '2' et 'c_e' . Nous obtenons au final la chaîne "$c_b 3124 c_e$".

Lorsque le site 3 intègre o_2, son état se résume par le diagramme de la figure 4.11.

L'appel *IntegrateIns*$(2, c_b, c_e)$ est résolu : $S' =$ "314" et $C_N(3) = C_P(4) = 1$, nous avons $L =$ "1". Comme $1 <_{id} 2$, WOOT intègre '2' entre $'1'$ et c_e. Au cours de l'appel

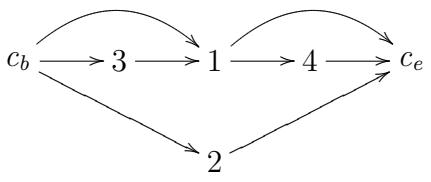


FIG. 4.11 – Diagramme de Hasse de l'état du site 2.

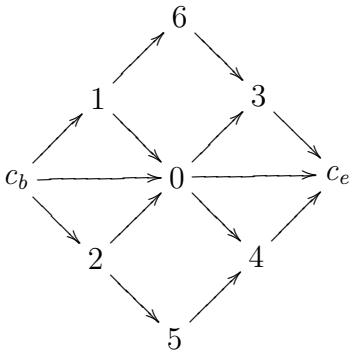


FIG. 4.12 – Scénario d'intégration combinant 7 sites.

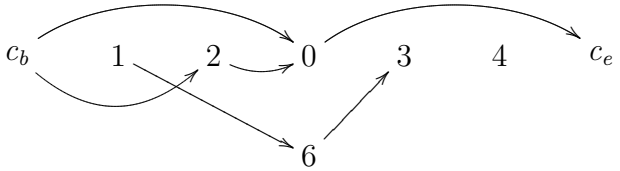


FIG. 4.13 – Diagramme de Hasse du scénario combinant 7 sites

récursif *IntegrateIns*$(2, 1, c_e)$: nous avons $S' = L =$ "4" et $2 <_{id} 4$, WOOT intègre '2' entre '1' et '4'. Sur le site 3, nous obtenons la chaîne "c_b3124c_e" comme résultat final.

Sur le site 1, quelque soit l'ordre d'arrivée des opérations o_2, o_3, o_4, nous obtenons la chaîne "c_b3124c_e".

Ainsi, dans tous les cas et sur tous les sites, nous avons obtenu l'état final "3124". Les répliques ont convergé et les intentions ont bien été respectées.

Exemple 2

Il est possible de construire des scénarios plus complexes à résoudre. Par exemple, nous produisons un scénario avec sept sites. Chaque site génère un caractère contenant son numéro de site. Le diagramme de Hasse est donné par la figure 4.12.

Nous supposons que les caractères '0', '1', '2', '3' et '4' ont déjà été reçus sur un site. L'intégration de ces cinq premiers caractères a produit la chaîne "12034". '1' a été linéarisé devant '2' car $1 <_{id} 2$, et '3' a été linéarisé devant '4' car $3 <_{id} 4$. De la même manière, '2' a été placé devant '0' et '3' derrière '0'.

Nous simulons la réception et l'intégration du caractère '6' puis '5'. Ensuite, nous simulons la réception et l'intégration de '5' puis '6'. Enfin, nous vérifions que nous obtenons le même résultat dans les deux cas.

Nous intégrons l'opération $ins(1 \prec 6 \prec 3)$ dans la chaîne "12034". $S' =$ "20" et $L =$ "0". Car, comme le confirme le diagramme de la figure 4.13, seul le caractère '0' possède les relations $C_P(0) \leq_S C_P(6)$ et $C_N(6) \leq_S C_N(0)$.

Comme nous avons la relation $0 <_{id} 6$, '6' est linéarisé entre '0' et '3'. Nous obtenons

ainsi la chaîne "120634".

Nous intégrons maintenant l'opération $ins(2 \prec 5 \prec 4)$ dans la chaîne "120634". $S' =$ "063" et $L =$ "0". Comme $0 <_{id} 5$, nous intégrons '5' récursivement entre '0' et '4'. $S' =$ "63" et $L =$ "3". Comme $3 <_{id} 5$, '5' est inséré entre '3' et '4'. Le résultat final est donc "1206354".

Si nous rejouons le même scénario à partir de la chaîne "12034", mais cette fois ci en intégrant d'abord l'opération $ins(2 \prec 5 \prec 4)$, puis l'opération $ins(1 \prec 6 \prec 3)$.

- Intégration de $ins(2 \prec 5 \prec 4)$. $S' =$ "03" et $L =$ "0". Comme $0 <_{id} 5$, nous intégrons récursivement '5' entre '0' et '4'. $S' = L =$ "3" et $3 <_{id} 5$, nous obtenons la chaîne"120354".
- Intégration de $ins(1 \prec 6 \prec 3)$ sur la chaîne "120354". $S' =$ "20" et $L =$ "0". Comme $0 <_{id} 6$, '6' est inséré entre '0' et '3'. Nous obtenons l'état final "1206354" qui est bien l'état attendu.

4.2 Correction et complexité

Nous commençons par montrer que notre algorithme garantit notre modèle de cohérence PCI. Ensuite, nous évaluons sa complexité en espace et en temps.

4.2.1 Correction

THÉORÈME 1 Les algorithmes d'intégration de WOOT terminent.

Preuve

Clairement l'algorithme *IntegrateDel* termine.

Nous démontrons par l'absurde que l'algorithme d'intégration *IntegrateIns* termine. *IntegrateIns* ne termine pas si et seulement si l'appel récursif ne s'effectue pas sur une séquence strictement plus petite. Ce situation peut arriver seulement si nous obtenons une séquence S' non vide et une séquence $L = c_p\, c_n$. Si $L = c_p\, c_n$ alors tous les caractères de S' possèdent leur prédécesseur ou leur successeur dans S'. Étant donné que les caractères ont été générés dans un ordre strict, il existe au moins un caractère, qui a été intégré en premier entre c_p et c_n, dont son prédécesseur et son successeur sont en dehors de S'.

Il y a donc au moins trois caractères dans la séquence L. Ce qui contredit l'hypothèse $L = c_p\, c_n$. Et donc l'appel récursif s'effectue toujours sur une séquence strictement plus petite. En conséquence, l'algorithme d'intégration termine toujours.

Respect des pré-conditions

Une opération *op* est intégrée uniquement si la fonction *isExecutable*(*op*) retourne vrai. Cette fonction retourne vrai lorsque les pré-conditions de l'opération sont satisfaites. Clairement, les pré-conditions des opérations sont respectées.

Respect des intentions

La linéarisation calculée respecte les intentions des opérations. Autrement dit, la relation de précédence $\prec$ définie sur les caractères à leur génération doit être préservée.

THÉORÈME 2 WOOT construit sur chaque site une relation $\leq_s$ qui est une extension linéaire de la relation de précédence $\prec$.

Preuve

Après génération d'une opération, les relations $\prec$ ne sont jamais modifiées. Seule la procédure *IntegrateIns* modifie la linéarisation $\leq_s$. L'intégration d'un caractère c est toujours réalisée en insérant ce caractère entre $C_P(c)$ et $C_N(c)$. Donc $\leq_s$ est une extension linéaire de la relation $\prec$.

Cependant, respecter les intentions n'est pas suffisant. Par exemple, si deux sites insèrent respectivement 'x' et 'y' entre les même caractères 'a' et 'b', nous pouvons calculer deux chaînes différentes sur les deux sites ; respectivement "axyb" et "ayxb". Ces deux extensions linéaires préservent les intentions, pourtant les deux copies ne convergent pas.

Convergence des copies

Pour l'instant, nous ne possédons pas de preuve formelle de la convergence de WOOT. Nous avons vérifié sa correction en utilisant le système de vérification de modèle [6] TLC [YML99] sur une spécification modélisée en TLA+. Les techniques de vérification de modèle sont particulièrement bien adaptées pour vérifier des systèmes concurrents.

Malheureusement, ces techniques souffrent d'un défaut majeur : un problème d'explosion combinatoire sur le nombre d'états à vérifier. Il est impossible de vérifier notre système pour un nombre important de sites et d'opérations. Nous avons donc effectué la vérification pour 4 sites et 5 opérations concurrentes. Cette démonstration a nécessité environ deux semaines de calcul sur un poste de travail standard. La spécification utilisée est disponible en annexe A.

Pour montrer que la convergence est garantie, nous devons démontrer, que sur deux sites, deux caractères sont toujours linéarisés dans le même ordre. Étant donné que tous les caractères générés seront insérés sur tous les sites, la conjecture suivante garantit la convergence :

CONJECTURE 1 Soient deux W-chaînes S_1 et S_2 maintenues par deux sites distincts. Pour chaque paire de W-caractères $\{c, d\}$ qui appartient aux deux chaînes S_1 et S_2, on a :

$$c \leq_{S_1} d \Leftrightarrow c \leq_{S_2} d$$

Afin de simplifier la spécification et ainsi accélérer le processus de vérification, nous avons effectué deux généralisations :

[6] model-checker

1. nous n'avons pas modélisé l'opération *del* et sa procédure d'intégration. Celles-ci n'ont aucun impact sur le calcul de l'extension linéaire. Pour simuler des caractères supprimés, nous nous autorisons à générer une opération d'insertion $ins(c_p \prec c \prec c_n)$ dont les caractères c_n et c_p ne sont pas contiguës. Normalement, ce genre d'opération résulte de l'insertion entre des caractères détruits ;

2. nous avons représenté les caractères uniquement par leur identifiant.

En utilisant, le système de vérification de modèle TLC, nous avons trouvé une erreur dans une version précédente de notre algorithme. Dans cette version naïve, on ne filtrait pas l'ensemble S' pour obtenir l'ensemble L. À cette époque, nous pensions que tous les caractères entre c_p et c_n étaient concurrents, ils devaient être ordonnés uniquement par la relation $<_{id}$. Le système de vérification avait alors trouvé le scénario illustré par la figure 4.2 à la page 77. Ce contre-exemple nous a permis de concevoir l'algorithme d'intégration actuel.

4.2.2 Analyse de complexité

Nous évaluons maintenant la complexité de WOOT en fonction du nombre total n d'opérations qui ont été générées par tous les sites.

Complexité en espace

THÉORÈME 3 (COMPLEXITÉ EN ESPACE) WOOT a une complexité en espace de l'ordre de $O(n)$.

Autrement dit, WOOT possède une complexité en espace proportionnelle au nombre d'opérations présentes dans le système.

Preuve

- la taille de l'horloge logique et de l'identifiant du site sont constants. ($O(1)$)
- la taille du tampon $pool_s$ contenant les opérations en attente d'intégration est de k, $k \leq n$.
- un W-caractère a une taille constante ($O(1)$). Soit m le nombre d'opérations d'insertion déjà intégrées, $m \leq (n - k)$. La W-chaîne d'un site contient au départ deux W-caractères c_b et c_e. Une opération de suppression ne change pas la taille de la W-chaîne. Une opération d'insertion ajoute un W-caractère. Donc la taille de la W-chaîne maintenue sur un site est proportionnelle à $m + 2$.

Ainsi, la complexité en mémoire de WOOT est proportionnelle à $k + m$, et puisque $k + m \leq n$, nous pouvons affirmer que notre algorithme a une complexité mémoire en $O(n)$.

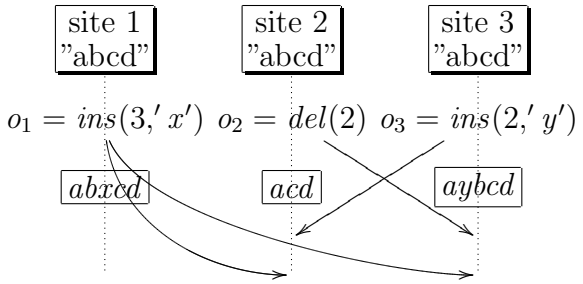


FIG. 4.14 – Situation problématique TP2.

Complexité en temps

THÉORÈME 4 Dans le pire des cas, WOOT a une complexité d'exécution en temps en $O(n^3)$

Preuve

Soit m la taille de la W-chaîne $string_s$.

L'intégration d'une opération $del(c)$ prend un temps linéaire $O(m)$. Nous devons parcourir la W-chaîne pour trouver l'identifiant du W-caractère c.

Concernant une opération $ins(c)$, nous devons évaluer le coût du pire cas d'exécution de la procédure d'intégration qui exécute les instructions suivantes :

- $S' = subseq(\ldots)$ $(O(m))$
- soit $insert(\ldots)$ $(O(m))$
- ou
  - filtrer S' pour obtenir L. Puisque $\leq_S$ est en $O(m)$, l'ensemble du filtre à un coût de $O(m^2)$.
  - `while` la boucle pour trouver i $(O(m))$
  - un appel récursif.

Dans le pire cas, il y a au plus m appels récursifs à la procédure *IntegrateIns*. Donc, la complexité en temps est $O(m(m + max(m, m^2 + m))) = O(m^3) = O(n^3)$ puisque $m \leq n$.

Nous avons supposé que la recherche d'un W-caractère et la comparaison des positions de deux W-caractères dans une W-chaîne s'effectue en $O(n)$. Si nous maintenons un index sur les identifiants dans une W-chaîne, la complexité du cas moyen d'exécution devient $O(n^2)$. L'index consommera de la mémoire, mais la complexité en espace de l'algorithme sera toujours en $O(n)$.

4.3 Comparaison avec les approches existantes

Dans le domaine des transformées opérationnelles, peu d'algorithmes garantissent le respect de l'intention tout en assurant la convergence des copies. La plupart échouent face à la situation problématique TP2 illustrée par la figure 4.14. Ce scénario considère trois sites qui possèdent chacun une réplique d'une chaîne `"abcd"`. Le site 1 exécute l'opération

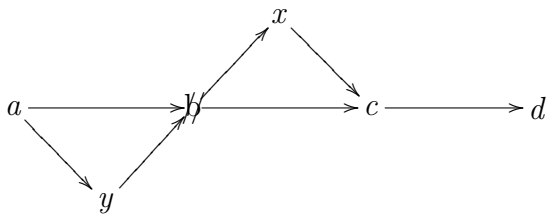


FIG. 4.15 – Diagramme de Hasse de la situation problématique TP2.

$o_1 = ins(3,'x')$, le site 2 exécute l'opération $o_2 = del(2)$, et le site 3 exécute l'opération $o_3 = ins(2,'y')$. Clairement, le caractère `'b'` est supprimé, le caractère `'x'` est inséré après le caractère `'b'` alors que le caractère `'y'` est inséré devant `'b'`. On peut donc penser que l'état de convergence doit être `"ayxcd"`.

Pourquoi ce problème est-il si difficile à résoudre pour les algorithmes OT ? En fait, ce problème semble simple parce nous observons tous les sites simultanément. Or, un algorithme OT doit construire les relations entre les opérations uniquement durant le processus de transformation, et donc uniquement à partir du journal des opérations. Quand o_1 est reçue sur le site 2, cette opération ne contient pas l'information "x doit être après b". Les algorithmes OT doivent recalculer cette information. Pour cela, les algorithmes se retrouvent à résoudre des problèmes de plus en plus difficiles en proposant des solutions de plus en plus complexes.

Si nous déroulons ce scénario en utilisant WOOT, le problème se résout simplement. Nous construisons le diagramme de Hasse présenté par la figure 4.15. À partir de ce graphe représentant les relations précédence entre les caractères, nous ne pouvons construire qu'une seule extension linéaire : `"aybxcd"`. Le caractère `'b'` est invisible, la chaîne finale observée par l'utilisateur est `"ayxcd"`. Cette chaîne correspond bien au résultat que nous attendions.

La figure 4.16 illustre un des scénarios contre-exemples pour la proposition de Suleiman [SCF97]. Ce scénario viole la condition C_2 et ne préserve pas les intentions. Le scénario produit le diagramme de Hasse illustré par la figure 4.17. Sur le site 2, lorsque $ins(b \prec x \prec d)$ est reçue, nous devons intégrer le caractère `'x'` dans l'état courant "abxyd". Nous calculons $S' =$ "xcy" et $L =$ "c". Nous supposons que $c <_{id} x$. Un appel récursif est nécessaire pour insérer `'x'` entre `'c'` et `'d'`. Au cours de cet appel, $S' = L =$ "y" et $x <_{id} y$, nous obtenons la chaîne "abxcyd". L'intégration des opérations sur les site 1 et site 3 conduit au même résultat. Les copies ont convergé et les intentions ont bien été respectées.

Si nous comparons WOOT et SDT, WOOT est plus simple. SDT recalcule a posteriori les relations d'ordre entre les caractères. WOOT conserve les relations et les diffuse avec

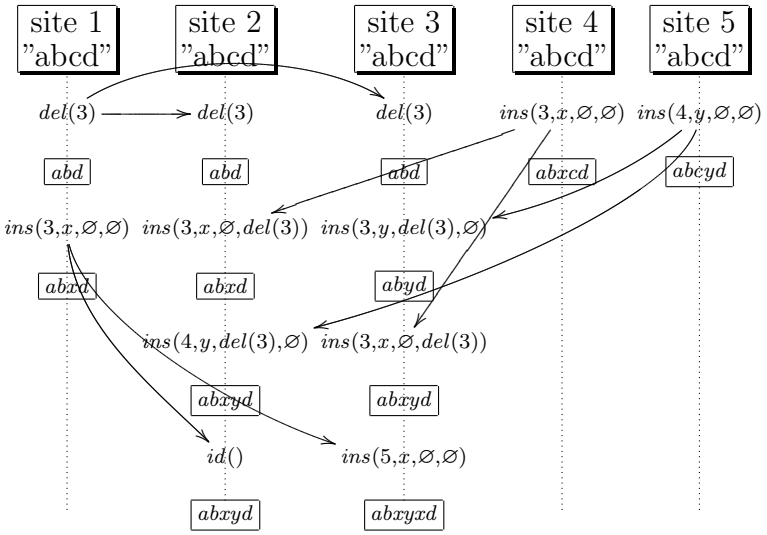


FIG. 4.16 – Contre-exemple pour la proposition de Suleiman [SCF97].

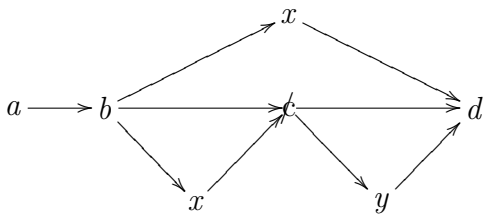


FIG. 4.17 – Diagramme de Hasse produit par le contre-exemple de la figure 4.16.

les opérations. Les deux approches ont une complexité en temps du même ordre. La complexité en mémoire de WOOT est moins élevée que celle de SDT. Enfin, contrairement à SDT, WOOT ne requiert pas de vecteurs d'horloges.

Notre approche repose sur le calcul incrémental d'une extension linéaire monotone de l'ordre partiel formé par la relation de précédence $\prec$ entre les caractères. Ce problème peut être considéré comme un problème de satisfaction de contraintes. Il doit donc être possible à IceCube de calculer une telle extension linéaire. La relation de précédence peut s'exprimer comme une contrainte statique sur les opérations. Les ordonnancements construits par IceCube correspondent à toutes les extensions linéaires de l'ordre partiel.

Pourtant, un problème subsiste : comment exprimer qu'une linéarisation future doit être compatible avec les linéarisations passées ? en ajoutant des contraintes supplémentaires ? Même si nous trouvons une réponse à cette question, notre environnement est plus adapté au calcul de cette extension linéaire. D'une part, si nous utilisons IceCube alors le calcul sera sujet au problème d'explosion combinatoire. L'apparition de ce phénomène, bien qu'il puisse être restreint, reste non négligeable. D'autre part, le processus de réconciliation utilisé dans IceCube repose sur une copie maître. Contrairement à notre environnement, le calcul de l'extension linéaire serait donc centralisé.

4.4 Conclusion

Hormis l'algorithme SDT qui repose sur le modèle OT, notre environnement WOOT est le seul algorithme à garantir la convergence des copies tout en préservant les intentions pour des données structurées linéaires. Il ne nécessite ni site central, ni ordre global, ni vecteur d'horloges. Il est donc parfaitement adapté pour des applications déployées sur un réseau pair à pair à large échelle.

Nous avons réalisé un premier éditeur collaboratif synchrone [7] reposant sur le modèle WOOT utilisant un réseau multicast.

Malheureusement, WOOT se limite à la réconciliation de structure linéaire. Il est difficilement extensible à d'autre type de structures de données.

[7] http ://www.loria.fr/~ molli/woot/

Chapitre 5

Bilan et perspectives

La Wikipédia est un formidable exemple d'édition collaborative massive. Ce qui a été fait pour une encyclopédie peut être reproduit pour un dictionnaire, des journaux etc.

Actuellement, les outils d'édition collaborative ne sont pas à la hauteur de ces enjeux. Il nous faut produire des outils d'édition collaborative massive.

Dans cette thèse, nous avons voulu concevoir un algorithme supportant l'édition collaborative massive. Il s'agit d'un algorithme de réplication optimiste devant concilier convergence, intentions et passage à l'échelle.

Nous avons, dans un premier temps, étudié l'approche transformationnelle. Malheureusement, nos efforts dans cette voie nous ont amené face à un problème sans solution.

Nous avons tout de même pu observer que le problème essentiel résidait dans la préservation de l'intention. Dans l'approche transformationnelle, curieusement, les intentions d'une opération ne sont pas exprimées explicitement. Elles apparaissent implicitement dans les fonctions de transformation.

Pour construire notre algorithme WOOT, nous avons repris le problème autrement. Il nous a semblé que le plus important était d'abord de préserver les intentions et ensuite de converger. Nous avons donc mis la préservation des intentions au coeur de WOOT. Les intentions des opérations sont explicites et nous avons prouvé que WOOT les préserve. Il faut ensuite assurer la convergence en supportant la propagation épidémique. Nous avons montré que cette propriété peut être atteinte en utilisant une fonction de linéarisation monotone qui est la procédure d'intégration de WOOT.

En conclusion, WOOT est un algorithme parfaitement adapté à l'édition collaborative massive.

5.1 Contributions

5.1.1 SO6

Afin de démontrer le potentiel de l'approche OT comme système de réplication optimiste, nous avons réalisé en 2003 un gestionnaire de configuration [MSMOJ02, MOSMI03, OMISM04] basé sur SOCT4. Cet outil est capable, avec le même algorithme, de réconcilier un système de fichiers et le contenu de ses fichiers que ce soient des documents textes, XML ou binaires.

Grâce au modèle OT, cet outil reste extensible. Il est possible d'intégrer de nouvelles fonctions de transformation pour lui permettre de réconcilier de nouveaux types de document. La conception de ces fonctions peut être effectuée en utilisant l'environnement de développement VOTE.

Cet outil garantit la convergence des répliques, ne préserve pas les intentions, et ne passe pas à l'échelle.

Cet outil constitue un transfert industriel aboutit. Il est utilisé comme outil de gestion de configuration dans la plateforme RNTL LibreSource (`http://www.libresource.org/`) qui est actuellement diffusée sous forme de logiciel libre.

5.1.2 Spike+VOTE

Comme nous l'avons souligné dans notre problématique, l'intérêt de l'approche des transformées opérationnelles repose sur les fonctions de transformation et les conditions qu'elles doivent vérifier. Malheureusement, si il n'existe pas de telles fonctions, l'approche des transformées est sans intérêt.

Pour concevoir et vérifier de telles fonctions de transformation, nous avons proposé un environnement de conception [IMOR02, IMOU03, IROM05, IMOR05] reposant sur une approche formelle et sur l'utilisation d'un démonstrateur automatique de théorème. En utilisant, cet environnement nous avons contrôlé toutes les fonctions déjà connues. Il résulte de cette analyse que toutes les propositions qui ont été faites jusqu'ici sont fausses [IMOR03]. Aucune d'entre elles ne parvient à satisfaire les deux conditions. Nous avons exprimé ces erreurs sous forme de scénarios contre-exemples. Nous avons également utilisé cet environnement de conception afin de réaliser des fonctions de transformation, vérifiant la condition C_1, pour différents types de données répliquées.

Malheureusement, l'environnement proposé ne peut pas vérifier si les fonctions de transformation préservent les intentions des opérations.

5.1.3 WOOT

WOOT est un nouvel algorithme pour réconcilier des structures linéraires divergentes. Nous avons constaté que les fonctions de transformation ne préservent pas les relations

entre différents éléments d'une même structure. Ce problème est principalement dû au choix réalisé lors de la définition du profil des opérations de mises à jour.

Nous avons ainsi modifié le profil des opérations. Ce profil ne contient plus la position d'insertion. Il maintient deux références vers les éléments entre lesquels l'insertion dans la structure linéaire s'effectue.

Notre approche ne repose pas sur le modèle OT. Elle repose sur une fonction de linéarisation monotone des ordres partiels constitués par les références précédent/suivant.

Notre approche ne nécessite ni site central, ni ordre global, ni vecteur d'horloges. Cette approche est donc parfaitement adaptée à l'édition collaborative massive.

5.2 Perspectives

À court terme, il est nécessaire de terminer la preuve de convergence de l'algorithme d'intégration WOOT. Actuellement, la correction de notre algorithme repose sur une validation par model-checking. Cette validation bien que déjà satisfaisante ne peut être considérée comme une démonstration à part entière.

Au cours de la conception de nos fonctions de transformation pour XML, nous avons constaté que les problèmes étaient pratiquement identiques à ceux rencontrés sur une structure linéaire. Il nous semble donc que l'extension de WOOT pour des structures arborescentes linéaires constitue une tâche aisée qui pourrait être réalisée rapidement.

La plupart des approches de réconciliation proposent un mécanisme d'annulation [RNRG96, Sun02, FVC04]. Ce mécanisme permet aux utilisateurs d'annuler certaines de leurs actions. Nous avons proposé un nouvel algorithme de réconciliation, nous voulons intégrer cette fonctionnalité dans WOOT.

Annexe A

Spécification TLA de WOOT

MODULE *woot*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

CONSTANTS

Sites,	set of sites
MAXGEN,	maximun generated number
CB, *CE*	

VARIABLES

string,	array of W-string
pool,	array of pools of messages
ccp, *ccn*,	orginial previous-next
gens	generated chars

$Chars \triangleq 1 \,..\, MAXGEN \cup \{CB,\ CE\}$
$Perms \triangleq Permutations(Sites)$
$less(c,\ d) \triangleq (c < d)$
$find(seq,\ c) \triangleq$
  CHOOSE $i \in 1 \,..\, Len(seq) : seq[i] = c$

$integrate(c,\ icp,\ icn,\ seq) \triangleq$
  LET $int[cp,\ cn \in Chars] \triangleq$
    LET $i1 \triangleq find(seq,\ cp)$
      $i2 \triangleq find(seq,\ cn)$
      $comp(d) \triangleq (find(seq,\ ccp[d]) \leq i1) \wedge (i2 \leq find(seq,\ ccn[d]))$
    IN

```
      IF (i2 − i1) = 1 THEN
        [i ∈ 1 .. (Len(seq) + 1) ↦ IF (i < i2) THEN seq[i] ELSE
          IF (i = i2) THEN c ELSE seq[i − 1]]
       ELSE
        LET ic ≜ CHOOSE i ∈ i1 .. i2 − 1 :
          (i = i1 ∨ (comp(seq[i]) ∧ less(seq[i], c))) ∧
          ∀ j ∈ i + 1 .. i2 − 1 : comp(seq[j]) ⇒ less(c, seq[j])
            id ≜ CHOOSE i ∈ ic + 1 .. i2 :
          (i = i2 ∨ (comp(seq[i]) ∧ less(c, seq[i]))) ∧
          ∀ j ∈ i1 + 1 .. i − 1 : comp(seq[j]) ⇒ less(seq[j], c)
        IN
          int[seq[ic], seq[id]]
    IN
      int[icp, icn]

init ≜
    ∧ pool = [s ∈ Sites ↦ {}]
    ∧ ccp = [c ∈ 1 .. MAXGEN ↦ ⟨⟩]
    ∧ ccn = [c ∈ 1 .. MAXGEN ↦ ⟨⟩]
    ∧ string = [s ∈ Sites      ↦ ⟨CB, CE⟩]
    ∧ gens = {}

sendIns(s, c, cp, cn) ≜
  LET n ≜ Len(string[s])
  IN
    ∧ ∃ i ∈ 1 .. n − 1 : (cp = string[s][i] ∧
                          ∃ j ∈ i + 1 .. n : cn = string[s][j])
    ∧ ∀ i ∈ 1 .. n : string[s][i] ≠ c
    ∧ c ∉ gens
    ∧ pool′ = [i ∈ Sites ↦ IF i = s THEN pool[i]
                           ELSE pool[i] ∪ {⟨"Ins", c, cp, cn⟩}]
    ∧ string′ = [string EXCEPT ![s] = integrate(c, cp, cn, string[s])]
    ∧ ccp′ = [ccp EXCEPT ![c] = cp]
    ∧ ccn′ = [ccn EXCEPT ![c] = cn]
    ∧ gens′ = gens ∪ {c}
    ∧ UNCHANGED ⟨⟩
```

$getIns(s,\ c,\ cp,\ cn) \triangleq$
LET $msg \triangleq \langle$"Ins", $c,\ cp,\ cn\rangle$
$n \triangleq Len(string[s])$
IN
$\land\ msg \in pool[s]$
$\land\ \exists\, i \in 1\,..\,n : string[s][i] = cp$
$\land\ \exists\, i \in 1\,..\,n : string[s][i] = cn$
$\land\ pool' = [pool \text{ EXCEPT } ![s] = pool[s] \setminus \{msg\}]$
$\land\ string' = [string \text{ EXCEPT } ![s] = integrate(c,\ cp,\ cn,\ string[s])]$
$\land$ UNCHANGED $\langle ccp,\ ccn,\ gens\rangle$

$raz \triangleq$
$\land\ Cardinality(gens) = MAXGEN$
$\land\ \forall\, s \in Sites : pool[s] = \{\}$
$\land\ ccp' = [c \in 1\,..\,MAXGEN \mapsto \langle\rangle]$
$\land\ ccn' = [c \in 1\,..\,MAXGEN \mapsto \langle\rangle]$
$\land\ string' = [s \in Sites \mapsto \langle CB,\ CE\rangle]$
$\land\ gens' = \{\}$
$\land$ UNCHANGED $\langle pool\rangle$

$wtNext \triangleq$
$\lor\ \exists\, s \in Sites : \exists\, c \in Chars : \exists\, cp \in Chars : \exists\, cn \in Chars :$
$sendIns(s,\ c,\ cp,\ cn)$
$\lor\ \exists\, s \in Sites : \exists\, c \in Chars : \exists\, cp \in Chars : \exists\, cn \in Chars :$
$getIns(s,\ c,\ cp,\ cn)$
$\lor\ raz$

$Conv \triangleq$
$\forall\, S1 \in Sites : \forall\, S2 \in Sites \setminus \{S1\} :$
$\forall\, i1 \in 1\,..\,Len(string[S1]) : \forall\, i2 \in 1\,..\,Len(string[S2]) :$
$\forall\, j1 \in 1\,..\,Len(string[S1]) : \forall\, j2 \in 1\,..\,Len(string[S2]) :$
$string[S1][i1] = string[S2][i2] \land string[S1][j1] = string[S2][j2]$
$\Rightarrow ((i1 < j1) \equiv (i2 < j2))$

$vars \triangleq \langle pool,\ string,\ gens,\ ccp,\ ccn\rangle$
$spec \triangleq init \land \quad \Box[wtNext]_{vars}$

THEOREM $spec \Rightarrow \Box Conv$

Annexe B

SO6 informations additionnelles

B.1 Fonctions de transformation utilisées dans SO6

L'algorithme d'intégration utilisé dans SO6 repose sur l'algorithme SOCT4. Les fonctions de transformation doivent donc satisfaire uniquement la condition C_1. Elles n'ont pas à vérifier la condition C_2. Toutes les fonctions, que nous présentons dans cette section, ont été conçues et validées selon l'approche décrite au chapitre 3. Elles vérifient la condition C_1 mais pas les conditions C_2.

B.1.1 Document textuel

Dans cette section, nous décrivons les fonctions de transformation nécessaires pour réconcilier un document textuel. Nous modélisons ce document sous la forme d'une liste de blocs de lignes de texte.
Deux opérations sont utilisées pour modifier le document :

AddBlock(p,v) insère le bloc de lignes v à la ligne p du document ;

DelBlock(p,ov) détruit le bloc de lignes ov débutant à la ligne p du document.

Nous définissons également une troisième opération nommée $Id()$ correspondant à la fonction identité. Cette opération n'a aucun effet sur l'état sur lequel elle est exécutée. Elle peut être obtenue par transformation de deux opérations identiques, comme par exemple deux insertions à la même ligne du même contenu.

Cas de deux insertions concurrentes

La fonction de transformation correspondante est la suivante :

```
1 T(AddBlock(p1,v1), AddBlock(p2,v2)) =
    if (p1 < p2) then
3       return AddBlock(p1,v1)
```

```
    else if (p2 < p1) then
5       return AddBlock(p1 + sizeof(v2),v1)
    else if (v1 = v2) then
7       return Id()
    else if (code(v1) < code(v2)) then
9       return AddBlock(p1,v1)
    else
11      return AddBlock(p1 + sizeof(v2),v1)
    endif;
```

Au cours de la transformation, trois cas peuvent se produire :

1. soit les deux insertions ne se font pas à la même ligne. Dans ce cas, l'une des deux insertions est décalée afin de prendre en compte l'autre insertion qui la précède selon l'ordre des numéros de ligne. Nous utilisons La fonction $sizeof(v)$ pour calculer le nombre de ligne du bloc v afin de s'en servir comme pas de décalage ;
2. soit les deux insertions se font à la même ligne et insèrent le même bloc de lignes. Dans ce cas, nous inhibons une des deux opérations d'insertion. Cette opération est transformée en une opération identité.
3. soit les deux insertions se font à la même ligne, mais les deux blocs sont différents. Dans ce cas, nous utilisons la fonction $code(v)$ pour calculer une valeur entière à partir du contenu du bloc v. Le bloc dont la valeur retournée par la fonction $code(v)$ est la moins élevée est placé devant l'autre bloc.

Cas de deux suppressions concurrentes

La fonction de transformation suivante concerne deux suppressions concurrentes :

```
T(DelBlock(p1,ov1), DelBlock(p2,ov2)) =
2   if (p1 < p2) then
      if (p1 + sizeof(ov1) − 1 < p2) then
4         return DelBlock(p1,ov1)
      else if (p1 + sizeof(ov1) − 1 < p2 + sizeof(ov2) − 1) then
6         return DelBlock(p1,subset(ov1,1,p2 − p1))
      else
8         return [  DelBlock(p1,subset(ov1,1,p2 − p1))
                  ; Id()
10                ; DelBlock(p1,subset(ov1,p2 + (sizeof(ov2) − p1) + 1,sizeof(ov1))) ]
      endif
12  else if (p2 + sizeof(ov2) − 1 < p1) then
      return DelBlock(p1 − sizeof(ov2),ov1)
```

```
14    else if (p1 + sizeof(ov1) − 1 < p2 + sizeof(ov2) − 1) then
        return Id()
16    else if (p1 + sizeof(ov1) − 1 = p2 + sizeof(ov2) − 1) then
        return Id()
18    else
        return DelBlock(p2,subset(ov1,p2 + (sizeof(ov2) − p1) + 1,sizeof(ov1)))
20    endif;
```

Si l'on considère deux opérations de suppression concurrentes, deux cas sont à envisager :

1. soit les deux suppressions sont disjointes. Autrement dit, leurs effets ne se chevauchent pas. Par exemple, l'une des opérations détruit un bloc de 3 lignes débutant à la ligne 2 du document, tandis que l'autre opération détruit un bloc de 5 lignes situé à la ligne 17. Dans ce cas, la deuxième suppression doit être décalée de 3 lignes vers le début du fichier ;
2. Soit il existe un chevauchement entre les effets des deux opérations : l'effet de l'une englobe l'autre ou bien l'une a déjà détruit une partie que l'autre doit détruire. Dans ce cas, la transformation modifie l'opération pour qu'elle détruise tout ce qu'elle devait détruire et qui ne l'a pas encore été. Nous utilisons une fonction $subset(v, p, l)$ qui extrait le sous-bloc du bloc v débutant à la position p de longueur l afin de calculer le contenu du bloc à détruire.

On notera que cette transformation est un peu spéciale. Elle ne retourne pas une opération en résultat mais une séquence d'opérations.

Cas d'une suppression et d'une insertion concurrente

Le cas d'une suppression et d'une insertion concurrente constitue le cas le plus problématique pour cet objet partagé.

```
T(AddBlock(p1,v1), DelBlock(p2,ov2)) =
2   if (p1 ≤ p2) then
      return AddBlock(p1,v1)
4   else if (p2 + sizeof(ov2) − 1 < p1) then
      return AddBlock(p1 − sizeof(ov2),v1)
6   else
      return AddBlock(p2,append(ov2,v1))
8   endif;

10 T(DelBlock(p1,ov1), AddBlock(p2,v2)) =
    if (p2 ≤ p1) then
12    return DelBlock(p1 + sizeof(v2),ov1)
```

```
else if (p1 + sizeof(ov1) − 1 < p2) then
    return DelBlock(p1,ov1)
else
    return [ DelBlock(p2,v2)
           ; DelBlock(p1,ov1)
           ; AddBlock(p1,append(ov1,v2)) ]
endif;
```

De la même manière que pour la fonction de transformation précédente, deux cas sont à envisager selon que les effets des deux opérations se chevauchent ou non :

1. Si les effets ne se chevauchent pas alors la seconde opération selon l'ordre défini sur les numéros de ligne est décalée pour prendre en compte la première opération. Le décalage a une longueur égale à la taille du bloc inséré ou supprimé par la première opération.
2. Un seul cas de chevauchement est possible. Il s'agit du cas où l'opération d'insertion s'effectue à une ligne n tandis que l'opération de suppression détruit un bloc contenant cette ligne. Les effets des deux opérations sont totalement en contradiction. Il nous est impossible de réconcilier tout en conservant les effets de deux opérations. C'est pourquoi nous avons décidé de représenter ce conflit afin de le montrer à l'utilisateur. Pour cela, l'intégration de ces deux opérations conflictuelles se solde par l'insertion d'un bloc de texte contenant un délimiteur de début `<<<`, le contenu du bloc qu'une des deux opérations souhaitait détruire, un délimiteur central `===`, le bloc que l'autre opération souhaitait insérer et enfin un délimiteur de fin `>>>`. Pour générer un tel bloc, nous utilisons la fonction $append(ov, v)$ qui prend comme paramètre le bloc ov qui devait être détruit et le bloc v qui devait être inséré.

La figure suivante donne un exemple de document textuel résultant de la réconciliation d'un conflit entre l'opération de suppression d'un bloc de deux lignes débutant à la seconde ligne (son contenu correspondant aux lignes `34. War [...]` et `97. [...]`) et l'opération d'insertion ajoutant une ligne à la troisième ligne (le contenu de cette insertion étant `35. [...]`) :

```
Rules of Acquisition
<<<<<<< removed
 34. War is good for business.
 97. Enough... is never enough.
=======
 35. Peace is good for business.
>>>>>>> inserted
242. More is good. All is better.
```

B.1.2 Système de fichiers

Dans cette section, nous décrivons brièvement les fonctions de transformation permettant de réconcilier un système de fichiers partagé. Sur ce système de fichiers nous définissons quatre opérations :

AddFile(p, t) ajoute un fichier ayant pour chemin d'accès p. Par exemple, l'opération *AddFile*$('/a/b', t)$ ajoute un fichier b dans le répertoire $'/a/'$;

AddDir(p, t) ajoute un répertoire ayant pour chemin d'accès p. Dans la suite, nous ne donnons pas la définition des fonctions de transformation faisant intervenir cette opération. Celle-ci se comporte de la même manière que l'opération *AddFile*(p, t). Autrement dit, pour obtenir les fonctions relatives à cette opération, il suffit de dupliquer la définition des fonctions de transformation faisant intervenir l'opération *AddFile*(p, t) est de substituer cette opération par l'opération *AddDir*(p, t) ;

Move(p, np, t) permet de déplacer un objet ayant pour chemin p vers un nouveau chemin np. L'opération de suppression d'un fichier ou répertoire est assimilée à une opération de déplacement sous un chemin particulier représentant une corbeille. Par exemple, si un utilisateur supprime le fichier de chemin $'/a/file'$, l'opération correspondante à cette modification sera *Move*$('/a/file', '/corbeille/a/file', t)$;

UpdateFile(p, c, t) . Cette opération nous permet de décrire le comportement général d'une opération de mise à jour d'un objet par rapport aux autres opérations *AddFile*(p, t), *AddDir*(p, t) et *Move*(n, np, t). Il faut donc imaginer que la valeur c est une méta-donnée de l'objet. Cette valeur peut aussi bien être une propriété d'un répertoire que le contenu d'un fichier binaire. Nous n'avons pas besoin de différencier ces différentes mises à jour pour écrire la plupart des fonctions de transformation. Seule la fonction de transformation de deux opérations *UpdateFile*(p, c, t) concurrentes nécessite des informations supplémentaires.

Comme dans toutes nos autres propositions nous définissons l'opération identité *Id*(). Nous nous autorisons également à ce que la fonction de transformation retourne non pas seulement une opération mais une séquence d'opérations.
Nous ajoutons un paramètre supplémentaire t à toutes les opérations. Ce paramètre correspond à l'identifiant du site sur lequel l'opération a été générée.

Cas de deux ajouts concurrents

Il ne peut se produire qu'un seul cas de conflit. Ce conflit correspond à l'ajout en concurrence du même fichier avec un même nom. Dans ce cas, nous utilisons le second paramètre (l'identifiant du site de génération) pour choisir lequel des deux ajouts s'exécute sous un autre nom. Ce choix permet de conserver les deux ajouts de fichiers. L'autre nom est calculé par la fonction *uniquePath*(). Cette fonction peut être réalisée, par exemple, en concaténant le numéro du site à l'ancien nom du fichier.

```
T(AddFile(p1,t1), AddFile(p2,t2)) =
  if (p1 = p2) then
    if (t1 < t2) then
      return AddFile(uniquePath(p1),t1)
    else
      return [ Move(p1,uniquePath(p1),t1)
             ; AddFile(p1,t1) ]
  endif
else
    return AddFile(p1,t1)
endif;
```

Ainsi, si l'on considère les deux opérations concurrentes $AddFile('/a/b', 1)$ et $AddFile('/a/b', 2)$ alors après intégration des deux opérations, le répertoire $/a$ contiendra deux fichiers : l'un nommé $b.1$ correspondant au fichier ajouté par la première opération ; et l'autre nommé b correspondant à l'ajout effectué par la seconde opération.

Cas d'un ajout et d'un déplacement concurrent

Nous devons prêter attention à deux cas en particulier :

1. L'ajout d'un fichier et le déplacement vers un même chemin. Dans ce cas, l'ajout du fichier se fait avec un autre nom généré par la fonction *uniquePath*() ;
2. L'ajout d'un fichier dans un répertoire qui est déplacé par l'autre opération. Dans ce cas, il faut modifier en conséquence le chemin de l'opération d'ajout du fichier ;

Nous définissons trois fonctions supplémentaires :
- le prédicat $childOf(p_1, p_2)$ qui permet de savoir si le chemin p_1 est fils du chemin p_2 ;
- la fonction $tail(p_1, p_2)$ qui extrait la queue du chemin p_1 par rapport à son chemin père p_2. Autrement dit, $tail('/a/b/c/d', '/a/b')$ retourne le morceau de chemin c/d ;
- la fonction $concat(p_1, p_2)$ qui concatène deux chemins. Par exemple, $concat('/a/b', 'c/d')$ calcule le chemin $/a/b/c/d$.

```
T(AddFile(p1,t1), Move(p2,np2,t2)) =
  if (p1 = np2) then
    if (t1 < t2) then
      return AddFile(uniquePath(p1),t1)
    else
      return [ Move(p1,uniquePath(p1),t1)
             ; AddFile(p1,t1) ]
```

```
    endif
  else if (childOf(p1,p2)) then
    return AddFile(concat(np2,tail(p1,p2)),t1)
  else
    return AddFile(p1,t1)
  endif;

T(Move(p1,np1,t1), AddFile(p2,t2)) =
  if (np1 = p2) then
    if (t1 < t2) then
      return Move(p1,uniquePath(np1),t1)
    else
      return [ Move(np1,uniquePath(np1),t1)
             ; Move(p1,np1,t1) ]
    endif
  else
    return Move(p1,np1,t1)
  endif;
```

Cas de deux déplacements concurrents

De nombreux cas différents sont à envisager :
- déplacement de deux fichiers vers la même destination : $Move('/a','/c')$ et $Move('/b','/c')$;
- déplacement d'un fichier vers une destination qui est déplacée par l'autre opération : $Move('/a','/b/c')$ et $Move('/b','/d')$;
- déplacement du même fichier vers deux destinations différentes : $Move('/a/b','/c')$ et $Move('/a/b','/d')$
- ...

Nous avons traité tous les cas possibles, nous avons obtenu la fonction de transformation suivante ;

```
T(Move(p1,np1,t1), Move(p2,np2,t2)) =
  if (np1 = np2) then
    if (p1 = p2) then
      return Id()
    else if (childOf(p1,p2)) then
      return Move(concat(np1,tail(p1,p2)), uniquePath(np1), t1)
    else if (childOf(p2,p1)) then
      return [ Move(np1,uniquePath(np1),t1)
             ; Move(p1,np1,t1) ]
```

```
    else if (t1 < t2) then
11      return Move(p1,uniquePath(np1),t1)
    else
13      return [ Move(np1,uniquePath(np1),t1)
                 ; Move(p1,np1,t1) ]
15  endif
  else if (p1 = p2) then
17  if (t1 < t2) then
      return Id()
19  else
      return Move(np2,np1,t1)
21  endif
  else if (childOf(np1,p2)) then
23  if (childOf(np2,p1)) then
      if (t1 < t2) then
25      return Id()
      else
27      return [ Move(np2,p2,t1)
                 ; Move(p1,np1,t1) ]
29    endif
    else if (childOf(p1,p2)) then
31    return Move(concat(np2,tail(p1,p2)),concat(np2,tail(np1,p2)),t1)
    else if (childOf(p2,p1)) then
33    return Move(concat(np2,tail(p1,p2)),concat(np2,tail(np1,p2)),t1)
    else
35    return Move(p1,concat(np2,tail(np1,p2)),t1)
    endif
37  else if (childOf(np2,p1)) then
    if (childOf(p2,p1)) then
39    return Move(p1,np1,t1)
    else if (childOf(p1,p2)) then
41    return Move(p1,np1,t1)
    else
43    return Move(p1,np1,t1)
    endif
45  else if (childOf(p1,p2)) then
     return Move(concat(np2,tail(p1,p2)),np1,t1)
47   else if (childOf(p2,p1)) then
     return Move(p1,np1,t1)
49   else
```

```
        return Move(p1,np1,t1)
51  endif;
```

Cas d'un ajout/déplacement et d'une une mise à jour de contenu concurrente

Ces cas ne posent pas de problèmes particuliers.

```
1   T(AddFile(p1,t1), UpdateFile(p2,c2,t2)) =
      return AddFile(p1,t1);
3
    T(UpdateFile(p1,c1,t1), AddFile(p2,t2)) =
5     return UpdateFile(p1,c1,t1);

7   T(Move(p1,np1,t1), UpdateFile(p2,c2,t2)) =
      return Move(p1,np1,t1);
9
    T(UpdateFile(p1,c1,t1), Move(p2,np2,t2)) =
11    if (childOf(p1,p2)) then
        return UpdateFile(concat(np2,tail(p1,p2)),c1,t1)
13    else
        return UpdateFile(p1,c1,t1)
15    endif;
```

Cas de deux mises à jour de contenu concurrentes

Comme nous l'avons expliqué auparavant, les fonctions de transformation précédentes ne nécessitent pas savoir quel genre de mise à jour est effectuée sur le contenu d'un fichier. En effet, jusqu'à présent, le fait que la modification soit l'ajout d'un bloc de texte dans un fichier textuel ou le remplacement entier du contenu d'un fichier binaire n'a aucune importance. Nous nous sommes intéressés uniquement au chemin du fichier sur lequel la mise doit être appliquée.
Si les mises à jour de contenu s'effectuent sur un fichier textuel, nous pouvons réutiliser les fonctions de transformation que nous avons présentées à la section B.1.1. Dans un tel cas, les mises à jour sont "fusionnées" dans le fichier.
Dans l'hypothèse où les mises à jour concurrentes s'effectuent sur le même fichier de contenu binaire, il n'est pas satisfaisant de fusionner ces modifications à l'intérieur même du fichier. Il est plus raisonnable de dupliquer le fichier, et d'appliquer chacune des deux mises à jour sur une copie. La fonction de transformation suivante implémente cette stratégie :

```
1  T(UpdateFile(p1,c1,t1), UpdateFile(p2,c2,t2)) =
     if (p1 = p2) then
3       if (c1 = c2) then
           return Id()
5       else if (t1 < t2) then
           return [ AddFile(uniquePath(p1),t1)
7                 ; UpdateFile(uniquePath(p1),c1,t1) ]
        else
9          return [ Move(p1,uniquePath(p1),t1)
                  ; AddFile(p1,t1)
11                ; UpdateFile(p1,c1,t1) ]
        endif
13   else
         return UpdateFile(p1,c1,t1)
15   endif;
```

B.1.3 Document XML

Nous présentons dans cette partie les fonctions de transformation nécessaires à la réconciliation d'un arbre ordonné. Un tel objet peut être utilisé afin de modéliser un document XML. Dans un arbre ordonné, chaque nœud est identifié de manière unique par son chemin. Ce chemin est la séquence des valeurs entières menant à ce nœud en parcourant de père en fils l'arbre depuis la racine. La figure suivante illustre un tel arbre où nous avons indiqué les valeurs des chemins de chaque nœud.

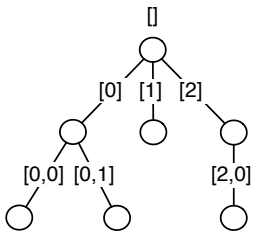


FIG. B.1 – Un exemple d'arbre ordonné

Sur un arbre ordonné, nous définissons deux opérations permettant de le modifier :

InsNode$(parent, n, val)$ insère un nouveau nœud fils du nœud identifié par le chemin *parent*. Ce nouveau nœud est inséré en tant que le $n$ème fils et a pour valeur *val*.

DelNode$(parent, n)$ détruit le $n$ème fils du nœud identifié par le chemin *parent*.

Afin de pouvoir manipuler un arbre ordonné, nous définissons quelques fonctions supplémentaires sur les chemins :

length(*path*) retourne la longueur du chemin *path*,

childOf(p_1, p_2) retourne vrai si le nœud identifié par le chemin p_1 est un descendant du nœud identifié par le chemin p_2.

getPos(p, n) retourne la n+1ème composante du chemin p. Par exemple, *getPos*$([3, 2, 1, 4], 2) = 1$.

incPos(p, n) retourne le chemin p auquel on incrémente sa $n + 1$ème composante. Par exemple, *incPos*$([3, 2, 1, 4], 2) = [3, 2, 2, 4]$.

decPos(p, n) retourne le chemin p auquel on décrémente sa $n + 1$ème composante. Par exemple, *decPos*$([3, 2, 1, 4], 2) = [3, 2, 0, 4]$.

Enfin, nous définissons la fonction *codeInf*(val_1, val_2) qui nous permet de comparer les deux valeurs val_1 et val_2 en définissant une relation d'ordre. Il est toujours possible de réaliser une telle fonction. Par exemple, pour des nœuds dont le contenu est du texte, la fonction *codeInf*() se résume à évaluer la relation qu'il existe entre les deux valeurs selon l'ordre lexicographique.

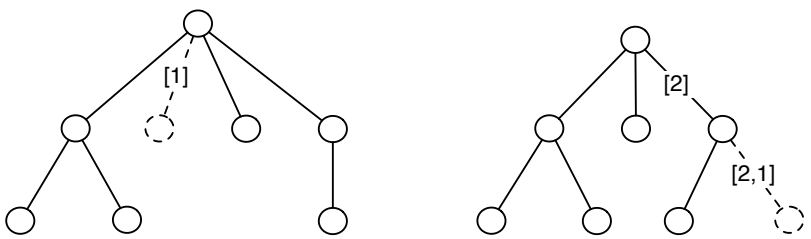
Les fonctions que nous présentons maintenant, bien qu'elles paraissent complexes, sont très similaires aux fonction définies sur les chaînes de caractères.

Si nous considérons le cas de deux opérations d'insertion de nœuds concurrentes qui s'effectuent sur le même nœud parent, alors le problème à résoudre est identique à celui de chaînes de caractères :

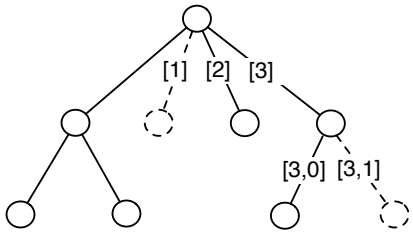
– il faut incrémenter la position d'insertion, le numéro de fils, de l'opération qui s'exécute derrière l'autre opération en terme de position d'insertion ;
– dans le cas où les deux insertions se font à la même position, il faut décider laquelle nous plaçons devant l'autre.

Si nous examinons le cas de deux insertions concurrentes sur des nœuds parents différents alors nous constatons que le problème est similaire : l'insertion la "plus en haut à gauche" décale l'insertion "la plus en bas à droite".

Par exemple, l'exécution en parallèle des deux opérations $op_1 = InsNode([], 1, X)$ et $op_2 = InsNode([2], 1, Y)$ donne les deux arbres suivants :



Après réconciliation, nous devons obtenir l'arbre suivant :



L'exécution de l'opération op_1 déplace le nœud parent sur lequel l'opération op_2 doit s'exécuter. Autrement dit, la transformation de l'opération op_2 par rapport à l'opération op_1 doit calculer l'opération $op_2' = InsNode([2+1], 1, Y)$.

Cas des deux insertions concurrentes

```
1  T(InsNode(p1, n1, v1), InsNode(p2, n2, v2) =
     if (p1 = p2) then
3       if (n1 < n2) then
           return InsNode(p1, n1, v1)
5       else if (n2 < n1) then
           return InsNode(p1, n1 + 1, v1)
7       else if (codeInf(v1, v2) = true) then
           return InsNode(p1, n1, v1)
9       else if (codeInf(v2, v1) = true) then
           return InsNode(p1, n1 + 1, v1)
11      else
           return Id()
13      endif
     else if (childOf(p1,p2)) then
15      if (n2 < getPos(p1, length(p2))) then
           return InsNode(incPos(p1,length(p2)), n1, v1)
17      else if (n2 = getPos(p1, length(p2))) then
              return InsNode(incPos(p1,length(p2)), n1, v1)
19      else
           return InsNode(p1, n1, v1)
21      endif
     else
23      return InsNode(p1, n1, v1)
     endif;
```

Cas de deux suppressions concurrentes

```
T(DelNode(p1, n1), DelNode(p2, n2)) =
  if (p1 = p2) then
     if (n1 < n2) then
       return DelNode(p1, n1)
     else if (n2 < n1) then
       return DelNode(p1, n1 − 1)
     else
       return Id()
     endif
  else if (childOf(p1,p2)) then
     if (n2 < getPos(p1, length(p2))) then
        return DelNode(decPos(p1,length(p2)),n1)
     else if (n2 = getPos(p1, length(p2))) then
           return Id()
     else
        return DelNode(p1,n1)
     endif
  else
     return DelNode(p1,n1)
  endif;
```

Cas d'une insertion et d'une suppression concurrente

```
T(InsNode(p1,n1,v1), DelNode(p2,n2)) =
  if (p1 = p2) then
     if (n1 < n2) then
       return InsNode(p1,n1,v1)
     else if (n2 < n1) then
       return InsNode(p1,n1 − 1,v1)
     else
       return InsNode(p1,n1,v1)
     endif
  else if (childOf(p1,p2)) then
     if (n2 < getPos(p1, length(p2))) then
        return InsNode(decPos(p1,length(p2)),n1,v1)
```

```
    else if (n2 = getPos(p1, length(p2))) then
14          return Id()
    else
16      return InsNode(p1,n1,v1)
    endif
18  else
    return InsNode(p1,n1,v1)
20  endif;

22 T(DelNode(p1,n1), InsNode(p2,n2,v2)) =
   if (p1 = p2) then
24    if (n1 < n2) then
        return DelNode(p1,n1)
26    else if (n2 < n1) then
        return DelNode(p1,n1 + 1)
28    else
        return DelNode(p1,n1 + 1)
30    endif
   else if (childOf(p1,p2)) then
32    if (n2 < getPos(p1, length(p2))) then
        return DelNode(incPos(p1,length(p2)),n1)
34    else if (n2 = getPos(p1, length(p2))) then
        return DelNode(incPos(p1,length(p2)),n1)
36    else
        return DelNode(p1,n1)
38    endif
   else
40    return DelNode(p1,n1)
   endif;
```

Bibliographie

[AFK+95] Larry Allen, Gary Fernandez, Kenneth Kane, David Leblang, Debra Minard, et John Posner. ClearCase MultiSite : Supporting geographically-distributed software development. Dans *Software Configuration Management : selected papers of the ICSE SCM-4 and SCM-5 Workshops*, numéro 1005 dans Lecture Notes in Computer Science, pages 194–214. Springer-Verlag, Octobre 1995.

[App05] Apple. Apple iSync. More ways to sync your digital life, (Septembre 2005). `http://www.apple.com/macosx/features/isync/`.

[Ber90] Brian Berliner. CVS II : Parallelizing software development. Dans *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, Californie, États-Unis, 1990. USENIX Association.

[Bou96] Adel Bouhoula. Using induction and rewriting to verify and complete parameterized specifications. *Theoretical Computer Science*, 170(1-2), pages 245–276, 1996.

[BP98] Sundar Balasubramaniam et Benjamin C. Pierce. What is a file synchronizer ? Dans *fourth annual ACM/IEEE International Conference on Mobile Computing and Networking - MobiCom'98*, Octobre 1998.

[BR93] Adel Bouhoula et Michaël Rusinowitch. Automatic case analysis in proof by induction. Dans *Proceedings of the thirteenth International Joint Conference on Artificial Intelligence - IJCAI'93*, pages 88–94, Septembre 1993.

[CAM02] Gregory Cobena, Serge Abiteboul, et Amélie Marian. Detecting changes in XML documents. Dans *Proceedings of the eighteenth International Conference on Data Engineering - ICDE'02*, pages 41–52, San Jose, Californie, États-Unis, Février 2002. IEEE Computer Society.

[Ced02] Per Cederqvist. *Version Management with CVS*. Network Theory Ltd., Décembre 2002.

[Col05] CollabNet. Subversion, (Septembre 2005). `http://subversion.tigris.org/`.

[Cun05] Ward Cunningham. Wikiwikiweb history, (Septembre 2005). `http://c2.com/cgi/wiki?WikiHistory`.

[DDD+94] Dean Daniels, Lip Boon Doo, Alan Downing, Curtis Elsbernd, Gary Hallmark, Sandeep Jain, Bob Jenkins, Peter Lim, Gordon Smith, Benny Souder, et Jim Stamos. Oracle's symmetric replication technology and implications for application design. Dans *Proceedings of the 1994 ACM SIGMOD International Conference on Management of data - SIGMOD'94*, page 467, New York, New York, États-Unis, 1994. ACM Press.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenkcr, Howard Sturgis, Dan Swinehart, et Doug Terry. Epidemic algorithms for replicated database maintenance. Dans *Proceedings of the sixteenth annual ACM Symposium on Principles of Distributed Computing - PODC'87*, pages 1–12. ACM Press, 1987.

[DSL02] Aguido Horatio Davis, Chengzheng Sun, et Junwei Lu. Generalizing operational transformation to the Standard General Markup Language. Dans *Proceedings of the ACM conference on Computer supported cooperative work - CSCW'02*, pages 58–67, New York, New York, États-Unis, 2002. ACM Press.

[DSU04] Xavier Défago, André Schiper, et Péter Urbán. Total order broadcast and multicast algorithms : Taxonomy and survey. *ACM Computing Surveys*, 36(4), pages 372–421, 2004.

[EG89] Clarence A. Ellis et Simon J. Gibbs. Concurrency control in groupware systems. Dans *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8), pages 28–33, Août 1991.

[FVC04] Jean Ferrié, Nicolas Vidot, et Michèle Cart. Concurrent undo operations in collaborative environments using operational transformation. Dans *On the Move to Meaningful Internet Systems 2004 : CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, ODBASE 2004*, volume 3290 de *Lecture Notes in Computer Science*, pages 155–173. Springer, Novembre 2004.

[IBM05] IBM. Rational ClearCase, (Septembre 2005). `http://www.ibm.com/software/awdtools/clearcase/`.

[IMOR02] Abdessamad Imine, Pascal Molli, Gérald Oster, et Michaël Rusinowitch. Development of transformation functions assisted by a theorem prover. Dans *fourth International Collaborative Editing Workshop - ICEW 2002*, Nouvelle Orléans, Louisiane, États-Unis, Novembre 2002.

[IMOR03] Abdessamad Imine, Pascal Molli, Gérald Oster, et Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. Dans *eighth European Conference of Computer-Supported Cooperative Work - ECSCW 03*, Helsinki, Finlande, Septembre 2003.

[IMOR04] Abdessamad Imine, Pascal Molli, Gérald Oster, et Michaël Rusinowitch. Achieving convergence with operational transformation in distributed groupware systems. Rapport de recherche 5188, LORIA–INRIA Lorraine, Mai 2004.

[IMOR05] Abdessamad Imine, Pascal Molli, Gérald Oster, et Michaël Rusinowitch. Towards synchronizing linear collaborative objects with operational transformation. Dans *Twenty-Fifth IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems - FORTE'2005*, volume 3731 de *Lecture Notes in Computer Science (LNCS)*, pages 411–427, Taiwan, Octobre 2005. Springer-Verlag.

[IMOU03] Abdessamad Imine, Pascal Molli, Gérald Oster, et Pascal Urso. VOTE : Group editors analyzing tool. Dans *Fourth Internation Workshop on First-Order Theroem Proving - FTP 2003*, volume 86 de *ENTCS*, Valence, Espagne, Juin 2003. Elsevier.

[IROM05] Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, et Pascal Molli. An algebraic framework for designing operational transformation algorithms. *Theoretical Computer Science : selected papers of the tenth International Conference on Algebraic Methodology of Software Technology - AMAST 2004*, 2005.

[JT05] Paul R. Johnson et Robert H. Thomas. RFC 677 : Maintenance of duplicate databases, Janvier 1975, (Septembre 2005). `http://www.faqs.org/rfcs/rfc677.html`.

[KRSD01] Anne-Marie Kermarrec, Antony I. T. Rowstron, Marc Shapiro, et Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. Dans *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing - PODC'01*, pages 210–218. ACM Press, 2001.

[LC03] Brad Lushman et Gordon V. Cormack. Proof of correctness of Ressel's adOPTed algorithm. *Information Processing Letters*, 86(6), pages 303–310, 2003.

[LL04a] Du Li et Rui Li. Ensuring content and intention consistency in real-time group editors. Dans *Proceedings of the twenty fourth International Conference on Distributed Computing Systems - ICDCS'04*, pages 748–755, Hachioji, Tokyo, Japon, Mars 2004. IEEE Computer Society.

[LL04b] Du Li et Rui Li. Preserving operation effects relation in group editors. Dans *Proceedings of the ACM conference on Computer Supported Cooperative Work - CSCW '04*, pages 457–466. ACM Press, 2004.

[Log05] Projet INRIA LogiCal. *The Coq Proof Assistant Reference Manual*, (Septembre 2005). `http://coq.inria.fr/doc/main.html`.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, Californie, États-Unis, 1996.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. Dans Michel Cosnard et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Château de Bonas, France, Octobre 1989. Elsevier Science Publishers.

[MH69] John McCarthy et Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. Dans *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[Mic05a] Microsoft. Microsoft ActiveSync, (Septembre 2005). `http://www.microsoft.com/windowsmobile/downloads/activesync38.mspx`.

[Mic05b] Sun Microsystems. Java 2 Platform Enterprise Edition, (Septembre 2005). `http://java.sun.com/j2ee/`.

[MM85] Webb Miller et Eugene W. Myers. A file comparison program. *Software—Practice and Experience*, 15(11), pages 1025–1040, 1985.

[MOSMI03] Pascal Molli, Gérald Oster, Hala Skaf-Molli, et Abdessamad Imine. Using the transformational approach to build a safe and generic data synchronizer. Dans *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work - GROUP'03*, Sanibel Island, Floride, États-Unis, Novembre 2003. ACM Press.

[MSMOJ02] Pascal Molli, Hala Skaf-Molli, Gérald Oster, et Sébastien Jourdain. SAMS : Synchronous, Asynchronous, Multi-Synchronous Environments. Dans *Seventh International Conference on Computer Supported Cooperative Work in Design - CSCWD 2002*, Rio de Janeiro, Brésil, Septembre 2002.

[Mye86] Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2), pages 251–266, 1986.

[OMISM04] Gérald Oster, Pascal Molli, Abdessamad Imine, et Hala Skaf-Molli. Un modèle sûr et générique pour la synchronisation de données divergentes. Dans *Premières Journées Francophones : Mobilité et Ubiquité*, Nice, France, Juin 2004.

[Ora01] Oracle. *Oracle 9i Replication*. Oracle Corporation, Juin 2001.

[ORS92] Sam Owre, John M. Rushby, et Natarajan Shankar. PVS : A prototype verification system. Dans *Proceedings of the eleventh International Conference on Automated Deduction - CADE'11*, pages 748–752, Londres, Royaume Unis, 1992. Springer-Verlag.

[Pal05] Palm. Palm HotSync, (Septembre 2005). `http://www.palm.com/us/support/hotsync.html`.

[Pea89] Giuseppe Peano. *Arithmetices principia novo methodo exposita*. Turin, 1889.

[PSM03] Nuno M. Preguiça, Marc Shapiro, et Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. Dans *On The Move to Meaningful Internet Systems 2003 : CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003*, volume 2888 de *Lecture Notes in Computer Science*, pages 38–55. Springer, Novembre 2003.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, et Alan J. Demers. Flexible update propagation for weakly-consistent replication. Dans *Proceedings of the sixteenth ACM symposium on Operating systems principles - SOSP'97*, pages 288–301. ACM Press, 1997.

[PV04] Benjamin C. Pierce et Jérôme Vouillon. What's in Unison ? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Department of Computer and Information Science, University of Pennsylvania, 2004.

[RNRG96] Matthias Ressel, Doris Nitsche-Ruhland, et Rul Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. Dans *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW'96*, pages 288–297, Boston, Massachusetts, États-Unis, Novembre 1996.

[Sal92] Rich Salz. InterNetNews : Usenet transport for Internet sites. Dans *USENIX conference proceedings*, pages 93–98, San Antonio, Texas, États-Unis, Été 1992. USENIX.

[SCF97] Maher Suleiman, Michèle Cart, et Jean Ferrié. Serialization of concurrent operations in a distributed collaborative environment. Dans *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work : The Integration Challenge - GROUP'97*, pages 435–445. ACM Press, Novembre 1997.

[SCF98] Maher Suleiman, Michèle Cart, et Jean Ferrié. Concurrent operations in a distributed and mobile collaborative environment. Dans *Proceedings of the fourteenth International Conference on Data Engineering - ICDE'98*, pages 36–45, Orlando, Floride, États-Unis, Février 1998. IEEE Computer Society.

[SE98] Chengzheng Sun et Clarence A. Ellis. Operational transformation in real-time group editors : Issues, algorithms, and achievements. Dans *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW'98*, pages 59–68, New York, New York, États-Unis, Novembre 1998. ACM Press.

[SJZ+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, et David Chen. Achieving convergence, causality preservation, and intention preservation in

real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1), pages 63–108, Mars 1998.

[SPO04] Marc Shapiro, Nuno M. Preguiça, et James O'Brien. Rufis : Mobile data sharing using a generic constraint-oriented reconciler. Dans *Proceedings of the fifth IEEE International Conference on Mobile Data Management - MDM'04*, pages 146–151. IEEE Computer Society, Janvier 2004.

[SS05] Yasushi Saito et Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1), pages 42–81, 2005.

[Str01] Sorin Stratulat. A general framework to build contextual cover set. *Journal of Symbolic Computation*, 32(4), pages 403–445, 2001.

[Sul98] Maher Suleiman. *Sérialisation des Opérations Concurrentes dans les Systèmes Collaboratifs Répartis*. Thèse de Doctorat, Université de Montpellier II, 1998.

[Sun02] Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(4), pages 309–361, Décembre 2002.

[SZJY97] Chengzheng Sun, Yanchun Zhang, Xiahua Jia, et Yun Yang. A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems. Dans *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work : The Integration Challenge - GROUP'97*, pages 425–434. ACM Press, Novembre 1997.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, et Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. Dans *Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP'95*, pages 172–182. ACM Press, 1995.

[VCFS00] Nicolas Vidot, Michèle Cart, Jean Ferrié, et Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. Dans *Proceedings of the ACM conference on Computer supported cooperative work - CSCW'00*, pages 171–180, New York, New York, États-Unis, 2000. ACM Press.

[Vid02] Nicolas Vidot. *Convergence des Copies dans un Environnements Collaboratifs Répartis*. Thèse de Doctorat, Université de Montpellier II, 2002.

[Wik05] Wikimedia. Wikipedia. The free encyclopedia that anyone can edit, (Septembre 2005).

[YML99] Yuan Yu, Panagiotis Manolios, et Leslie Lamport. Model checking TLA+ specifications. Dans *Proceedings of Correct Hardware Design and Verification Methods - CHARME'99*, pages 54–66, 1999.

Résumé

Les systèmes d'édition collaborative permettent à plusieurs utilisateurs d'éditer simultanément un document. Aujourd'hui, l'édition collaborative massive est une réalité. Il ne s'agit plus d'éditer à quelques utilisateurs mais à des milliers d'utilisateurs répartis dans le monde. Les éditeurs collaboratifs actuels n'ont pas été conçus pour supporter un nombre si important d'utilisateurs. Les problèmes soulevés ne sont pas d'ordre technologique, ils remettent en cause les fondements algorithmiques des éditeurs.
L'objectif de cette thèse est de proposer des algorithmes adaptés à l'édition collaborative massive. Nous montrons qu'un tel algorithme doit assurer trois critères : convergence des données, préservation des intentions et passage à l'échelle. Au regard de l'état de l'art, seul le modèle des transformées opérationnelles (OT) peut concilier ces trois critères.
La première contribution de cette thèse montre que l'approche OT conçue pour des éditeurs temps-réel peut être utilisé pour réaliser des outils asynchrones. Nous avons réalisé un gestionnaire de configurations nommé SO6.
La seconde contribution est une approche formelle à la conception et à la vérification de fonctions de transformation pour le modèle OT. Cette approche repose sur un démonstrateur automatique de théorème. Avec cette approche, nous montrons que toutes les fonctions de transformation proposées jusqu'ici sont fausses.
La troisième et dernière contribution de ce travail est un nouvel algorithme de réplication optimiste (WOOT) adapté à l'édition collaborative massive de structures linéaires. Ce modèle repose sur le calcul monotone d'une extension linéaire des ordres partiels formés par les relations entre les différents éléments de la structure.

Abstract

Collaborative editing systems (CES) allow multiple users to edit the same document. Today, massive collaborative editing becomes reality. CES are not limited to a small amount of users, they are used by thousands users. Unfortunately, current collaborative editing systems were not designed to support such collaboration. Arisen issues are not only technological difficulties, they bring algorithmic foundations of editors into question.
The aim of this thesis is to propose new algorithms suitable for massive collaborative editing. We show that theses algorithms must ensure three criteria : copies convergence, intention preservation and scalability. As regards related work, only operational transformation approach (OT) ensures these three criteria.
The first contribution of this dissertation shows that OT approach designed to build realtime collaborative systems can also be used to build asynchronous systems. Using OT, we developed a configuration management tool called SO6.
The second contribution is a formal approach to design and verify transformation functions for OT model. This approach relies on an automatic theorem prover. Using this approach, we show that all previously published functions are wrong.
The third and last contribution of this research is a new optimistic algorithm (WOOT) that is suitable for massive collaborative editing of linear structures. This algorithm relies on a monotonic computation of a linear extension of partial orders built from relations between elements of the structure.