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

Conception Formelle d'Algorithmes de Réplication Optimiste Vers l'Edition Collaborative dans les Réseaux Pair-à-Pair

THÈSE

présentée et soutenue publiquement le 11 décembre 2006

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Abdessamad IMINE

Composition du jury

Président :	Jacques Julliand	Professeur, Université de Franche-Comté
Rapporteurs :	Jean Ferrié	Professeur, Université Montpellier II
	Jean-François Monin	Professeur, Université Joseph Fourier, Grenoble I
Examinateurs :	Dominique Méry	Professeur, Université Henri Poincaré, Nancy 1
	Pascal Molli	Maître de Conférences, Université Henri Poincaré, Nancy 1
Directeur :	Michaël Rusinowitch	Directeur de recherche, INRIA, Nancy

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

Mis en page avec la classe thloria.

A mon père

Remerciements

Il m'est très agréable de remercier amicalement Michaël RUSINOWITCH pour sa permanente disponibilité, sa gentillesse et ses encouragements. Le travail d'équipe que nous avons entrepris ensemble a été pour moi une expérience très enrichissante et qui continuera de l'être. Qu'il trouve ici le témoignange de ma gratitude.

Je voudrais en second lieu remercier vivement les personnes qui m'ont fait l'honneur d'être membre de mon jury :

- Pascal MOLLI, Maître de Conférences à l'Université Henri Poincaré de Nancy, a toujours porté un grand intérêt à mes travaux. J'ai collaboré activement avec lui dans le cadre d'une coopération fructueuse entre les équipes CASSIS et ECOO. Grâce à son expérience, son pragmatisme et ses qualités humaines, j'ai pu traiter d'une problématique très en vogue, à savoir l'édition collaborative. Qu'il trouve ici ma gratitude pour sa collaboration et son soutien pour mener à bien cette thèse.
- Jacques JULLIAND, Professeur à l'Université de Franche-Comté, a accepté avec gentillesse de présider ce jury ; je l'en remrecie. Ses commentaires fort pertinents m'ont été très utiles dans la préparation de ce document.
- Jean FERRIE, Professeur à l'Université Montpellier II, m'a fait l'honneur de s'intéresser à ce travail. Je lui suis très reconnaissant d'être rapporteur de ma thèse. Ses compétences dans le domaine de l'édition collaborative, et notamment l'approche des transformées opérationnelles, ont apporté un point de vue très utile sur ce travail.
- Jean-François MONIN, Professeur à l'Université Joseph Fourier de Grenoble, a accepté d'être rapporteur de ma thèse. Je le remercie sincèrement pour le grand intérêt qu'il a montré pour ce travail. Ses remarques judicieuses m'ont été d'une grande utilité pour affiner mon travail sur le plan de la formalisation.
- Dominique MERY, Professeur à l'Université Henri Poincaré de Nancy, a accepté d'être examinateur de ma thèse. Je l'en remercie très sincèrement.

Mes remerciements s'adressent également à tous les membres de l'équipe CASSIS de Nancy pour les moments agréables ainsi que les discussions fructueuses que nous avons eus ensemble.

Sans oublier de remercier vivement les membres de l'équipe ECOO, plus particulièrement, Gérald Oster, Hala Skaf-Molli et Pascal Urso pour les débats captivants que nous avons eus au sujet de mon travail.

Comme une thèse est le résultat d'un travail collectif, elle est aussi le fruit d'une famille. Je voudrais ici témoigner ma plus haute gratitude à ma femme et mon fils, qui ont supporté ma longue absence malgré toutes les difficultés. Grâce surtout à la compréhension et le soutien permanent de ma femme, cette thèse a pu être menée à terme.

Enfin, merci du fond du cœur à ma mère, mes frères et soeurs, et mes amis pour leur encouragement et leur soutien amical.

Table des matières

Introduction Générale

1 Contexte

Le travail collaboratif assisté par ordinateur CSCW (en anglais "Computer Supported Cooperative Work") connaît un intérêt grandissant suite à une disponibilité de plus en plus importante de matériel informatique performant. Il est utilisé pour former un groupe d'utilisateurs réparti dans le temps, l'espace et à travers les organisations. Ces personnes collaborent le temps d'un projet pour répondre rapidement à un besoin donné. Par exemple, on peut appliquer le CSCW : (i) pour l'édition simultanée d'un même article scientifique par plusieurs chercheurs ; (ii) pour le développement d'un logiciel par une équipe dont les membres sont disséminés géographiquement ; ou, (iii) la réalisation d'un plan de bâtiment nécessitant l'expertise de plusieurs personnes.

L'outil de collaboration est un système logiciel, appelé *système collaboratif*, fournissant un ensemble d'objets partagés (documents textuels ou graphiques) pouvant à tout moment être visibles et accessibles par les utilisateurs. La communication entre ces derniers se fait de plus en plus sur un réseau informatique, comme Internet. Il y a deux catégories (synchrones/asynchrones) de systèmes collaboratifs selon le mode d'interactivité qu'ils proposent :

1. Le mode de travail *synchrone* permet aux collaborateurs de manipuler en même temps les mêmes données. Une modification sur un objet partagé est réalisée immédiatement et elle est visible en temps réel par les autres collaborateurs. Les *éditeurs collaboratifs* sont un exemple typique de ce mode d'interactivité [26; 73; 81; 86]. En effet, chaque utilisateur modifie localement son document et il envoie cette modification aux autres utilisateurs pour qu'elle soit immédiatement visible.
2. Le mode de travail *asynchrone* permet de travailler sur les mêmes données en même temps ou à des moments différents. Aussi, les mises-à-jour sont observées de manière différée. Les gestionnaires de versions comme CVS [12] et les synchroniseurs de fichiers comme Unison [67] supportent un tel mode de travail. A titre d'exemple un synchroniseur de fichiers permet aux utilisateurs de modifier le système de fichiers à des moments différents et de fusionner plus tard leurs modifications pour obtenir la même vue.

La conception d'un système collaboratif doit rendre son utilisation aussi conviviale qu'un système mono-utilisateur. Aussi, il est en particulier primordial d'assurer un temps de réponse court. Dans un contexte distribué, cela signifie que l'accès aux objets partagés ne nécessite pas de solliciter un site (un utilisateur ou un collaborateur) distant. Pour avoir donc une disponibilité permanente des données, les objets partagés sont répliqués sur les différents sites. De cette manière, l'accès à un objet peut être réalisé en accédant à la copie locale. Cependant, les modifications simultanées sur les différentes copies peuvent entraîner une *divergence* (ou une différence) entre les états de ces copies. Par conséquent, il est crucial de mettre en place une procédure de contrôle de la concurrence et ce pour assurer la *convergence* des copies vers un même état. Cette procédure doit concilier la contrainte "temps réel", la gestion des accès simultanés aux objets, le maintien de la convergence des copies et l'évolution dans le temps du groupe d'utilisateurs.

2 Vers l'édition collaborative sur des réseaux pair-à-pair

Les réseaux Pair-à-Pair (P2P) se caractérisent par une topologie variable du réseau ainsi que l'absence de la notion client/serveur [2]. En effet, un site peut être à la fois client et serveur et joindre ou quitter le réseau à tout moment. Les réseaux P2P représentent aujourd'hui le trafic dominant sur Internet [35]. Ce trafic correspond à une diffusion massive des données. La propagation des données est réalisée en recopiant les données d'un site du réseau à un autre. Ainsi, les données les plus demandées sont hébergées par un nombre important de sites. Malheureusement, si une copie est modifiée alors cette modification n'est pas diffusée aux autres copies. Ainsi la cohérence des données répliquées n'est pas gérée dans de tels systèmes.

Etablir une édition collaborative sur un réseau P2P permettrait de réutiliser le formidable potentiel de ces réseaux non seulement pour diffuser du contenu mais également pour créer et éditer ce contenu. Nous pouvons imaginer des applications nouvelles exploitant les possibilités des P2P. Par exemple, un réseau mondial de médecins pourrait se fédérer pour mener des études épidémiologiques. Les médecins doivent pouvoir répliquer les données partagées et faire des requêtes du type "l'âge et le dernier poids de tous les patients masculins avec la maladie X". Si un médecin met à jour une copie d'un dossier alors cette modification est propagée aux autres copies. Aussi, la cohérence des données partagées joue un rôle crucial dans cette application de par son impact direct sur la qualité de l'étude épidémiologique que peut mener tel ou tel médecin.

Le problème de fond est donc de disposer d'un système d'édition collaborative sur un réseau P2P qui *passe à l'échelle* (il peut gérer un nombre variable de sites) et garantir en conséquence la *convergence des copies*.

3 Etat de l'art en réplication

La réplication basée sur des techniques dites *pessimistes* donne l'illusion à l'utilisateur qu'il n'existe qu'une seule copie [6]. Les protocoles les plus connus sont basés sur le principe ROWA (Read One Write All). Les lectures peuvent être faites sur n'importe quelle copie tandis qu'une écriture doit être appliquée de manière atomique sur toutes les copies (à un instant donné un seul utilisateur peut écrire sur une copie). Ces protocoles font appel à l'utilisation des verrous dont la distribution est régie par un site central. Aussi, appliquer une modification sur un nombre très important de copies et cela de manière atomique sur un réseau P2P n'est pas réaliste.

La réplication basée sur des techniques *optimistes* [76] est une approche *a priori* plus adaptée au réseau P2P. Dans ce cas, les copies peuvent diverger. A un instant donné, un utilisateur peut donc observer des copies d'un même objet avec des valeurs différentes. Mais, quand le système sera au repos, i.e. quand toutes les opérations auront été propagées à toutes les copies, alors toutes les copies devront être identiques. La réplication optimiste laisse donc les copies diverger, à condition qu'elles finissent par converger. Comparée à la réplication

pessimiste, la réplication optimiste ne nécessite pas une mise à jour atomique de toutes les copies, et donc elle est potentiellement plus performante. Cependant, il faut être capable de faire converger des copies ayant divergé.

La règle de Thomas utilisée dans Usenet [77] gère la cohérence des copies de manière optimiste et passe à l'échelle. Malheureusement, le mécanisme de résolution de conflit de cette règle n'est pas adapté à l'édition collaborative. En effet, si deux copies sont modifiées en parallèle, alors une des deux modifications sera perdue. Les gestionnaires de configuration comme CVS [12] utilisent le paradigme du copier/modifier/fusionner pour réconcilier les données. Si la réconciliation permet de ne pas perdre des mises à jour, elle nécessite néanmoins un serveur central. Ce qui est inapproprié dans le cadre d'un réseau P2P, car ce serveur peut soit disparaître du réseau, soit être surchargé.

La réplication optimiste dans les bases données [16] ne nécessite aucun serveur central. La propagation des mises à jour se fait de manière épidémique en réconciliant les différents serveurs deux à deux. Néanmoins, la convergence n'est pas garantie dans tous les cas.

Avec IceCube [46; 71; 78], les problèmes de réconciliation sont vus comme un problème d'optimisation de contraintes. Il assure la convergence puisqu'il impose un ordre total sur l'exécution des modifications. Mais cette technique de restauration vers un état unique risque de perdre des mises-à-jour. En plus, IceCube désigne un site comme responsable de la réconciliation des autres sites. Cependant dans le cadre d'un réseau P2P, ce site peut disparaître du réseau durant le calcul sans pouvoir terminer le processus de réconciliation. Faire converger des copies peut être vu comme un problème de consensus [57]. Les différents sites doivent décider la valeur finale. Cependant, les hypothèses des algorithmes de consensus sont basées sur un nombre fixe de sites.

Une approche intéressante, appelée *transformées opérationnelles*, a été utilisée dans le domaine des éditeurs collaboratifs synchrones [26; 27]. Elle se présente sous forme d'algorithmes qui consistent à transformer les opérations reçues pour tenir compte de celles qui ont pu entre temps modifier l'état de l'objet. L'avantage substantiel des transformées opérationnelles est d'assurer la convergence des copies dans un contexte où les collaborateurs peuvent échanger leurs modification sans la moindre contrainte. En effet, elle est indépendante de la taille du groupe (le nombre de sites à considérer) ainsi que de sa topologie (la manière dont les sites sont interconnectés). Néanmoins, cet avantage n'a jamais été pleinement exploité sur un réseau P2P.

Premier objectif :
Nous nous intéressons dans cette thèse à concevoir un éditeur collaboratif fondé sur l'approche des transformées opérationnelles qui peut être facilement déployé sur un réseau P2P.

4 Approche des transformées opérationnelles

L'approche des transformées opérationnelles a été utilisée pour pallier le problème de divergence des copies dans les éditeurs collaboratifs synchrones [26; 73; 81; 85]. Elle utilise les propriétés sémantiques des opérations pour garan-

tir la convergence des copies. Plus précisément, elle exploite la transformation des opérations pour construire une histoire de chaque copie de l'objet. Les histoires des différentes copies ne sont pas identiques mais elles sont équivalentes (elles aboutissent au même état final) puisque l'ordre d'exécution des opérations concurrentes peut être différent d'une histoire à une autre.

Comparée à d'autres techniques optimistes, elle possède les avantages suivants : (i) Elle supporte un travail collaboratif sans contraintes. En effet, elle n'impose pas un ordre total sur les opérations concurrentes. Aussi, les sites peuvent échanger leurs modifications dans n'importe quel ordre. (ii) Elle permet la transformation des opérations pour les exécuter dans un ordre arbitraire même si à l'origine ces opérations ne commutent pas. (iii) Elle produit un état de convergence sans perte de mises-à-jour.

Chaque objet partagé doit avoir son propre algorithme de transformation dont l'écriture incombe aux développeurs de l'éditeur collaboratif. Ainsi, pour chaque paire d'opérations, le développeur doit préciser comment transformer ces opérations l'une par rapport à l'autre. Pour garantir la convergence des copies, un algorithme de transformation doit satisfaire des conditions très contraignantes [26; 68; 73]. Aussi, l'écriture d'un algorithme de transformation devient une tâche fastidieuse puisqu'elle nécessite des vérifications qui sont très difficiles (voire même impossibles) à effectuer à la main. D'autre part, cette difficulté varie selon la sémantique de l'objet partagé. Ainsi, la transformation pour des objets ayant une structure linéaire (comme la liste ou le texte) demeure toujours un problème ouvert.

Sans une approche formelle [62], la conception des algorithmes de transformation restera une activité ardue et sujette à erreurs, étant donné le nombre important de cas à considérer pour garantir des propriétés aussi critiques comme la convergence des copies.

Deuxième objectif :
Proposer un modèle formel pour l'approche des transformées opérationnelles qui nous permettra de concevoir des algorithmes corrects.

5 Contributions

La première contribution de cette thèse est la proposition d'une méthodologie formelle pour spécifier et vérifier des objets synchronisés par une une transformation opérationnelle [40; 43]. En utilisant une sémantique observationnelle, nous considérons un objet comme une boîte noire. Nous spécifions seulement les interactions (les séquences d'opérations) entre l'utilisateur et l'objet. Nous avons conçu un outil qui permet de définir les opérations et l'algorithme de transformation pour un objet donné (comme l'objet texte)[42]. A partir de cette description, l'outil engendre une spécification algébrique [90]. Pour vérifier la convergence, nous utilisons un prouveur automatique de théorèmes [11]. L'exploitation de notre méthodologie a été couronnée de succès puisqu'elle nous a permis de détecter des situations de divergence dans des systèmes collaboratifs bien connus dans la littérature [39; 44].

La deuxième contribution est la composition d'objets simples pour former des objets complexes tout en préservant des critères de convergence sur les algorithmes de transformation [38]. Deux types de composition sont étudiés : une *composition statique* où le nombre d'objets composants est connu ; ainsi on peut concevoir un agenda partagé comme étant un objet complexe composé d'un nombre fixe d'objets texte. Une *composition dynamique* est caractérisée par la création et la suppression dynamiques d'objets composants. Ainsi un document XML peut être conçu comme un arbre dont les nœuds sont des objets texte, qui peuvent être supprimés ou crées.

La troisième contribution concerne la synchronisation des objets linéaires (tels que la liste, le texte, l'arbre ordonné XML, etc). L'écriture d'algorithmes de transformation corrects pour de tels objets demeure un problème ouvert. A ce titre, nous avons constaté que les conditions de convergence connues dans la littérature sont très difficiles à satisfaire pour des objets linéaires. Aussi, nous avons proposé un nouvel algorithme de transformation basé sur une forme affaiblie de ces conditions [41]. Néanmoins, nous avons remarqué que l'intégration de cet algorithme dans les environnements collaboratifs connus dans la littérature est impossible, étant donné que ces environnements utilisent des conditions de convergence "très fortes" et difficiles à satisfaire pour les objets linéaires.

Aussi, comme quatrième et dernière contribution, nous avons conçu un nouvel environnement pour l'édition automatique basé sur les idées développés dans [41]. Cet environnement supporte un travail collaboratif sans contraintes (absence d'un site central) et un accès simultané aux données partagées avec une convergence des copies automatique grâce à la transformation des opérations. Il peut également passer à l'échelle grâce à la mise en œuvre d'une relation causale minimale entre les opérations. Par conséquent, cet environnement peut être déployé sur un réseau P2P.

6 Plan de la thèse

Ce document est organisé en trois parties.

La première partie est consacrée à une vue générale des éditeurs collaboratifs ainsi que l'approche des transformées opérationnelles :

- Le chapitre 1 passe en revue les éditeurs collaboratifs existants tout en mettant l'accent sur la convergence des données et le passage à l'échelle.
- Le chapitre 2 donne une présentation générale du modèle des transformées opérationnelles.

La deuxième partie présente notre méthodologie formelle pour concevoir des objets synchronisés par l'approche des transformées opérationnelles :

- Le chapitre 1 présente les notions fondamentales que nous utilisons tout au long de la deuxième partie : algèbres universelles, spécifications algébriques et spécifications observationnelles.
- Le chapitre 2 donne les ingrédients de la spécification formelle des objets partagés ainsi que la vérification automatique de la convergence.
- Le chapitre 3 présente une méthode compositionnelle pour spécifier et vérifier des objets complexes. Ce chapitre est indépendant du reste de la

thèse.

La troisième partie traite de la convergence des objets partagés ayant une structure linéaire :

- Le chapitre 1 passe en revue les cas de divergence détectés dans les algorithmes de transformation existants, et donne les causes réelles de cette divergence.
- Le chapitre 2 présente un nouvel algorithme de transformation pour des objets linéaires basé sur une forme relaxée des conditions de convergence connues dans le modèle des transformées opérationnelles.
- Le chapitre 3 donne une présentation détaillé d'un nouvel environnement pour l'édition collaborative et montre que ce dernier peut être utilisé sur un réseau P2P.

Nous concluons en résumant le travail effectué au cours de cette thèse et en indiquant quelques perspectives.

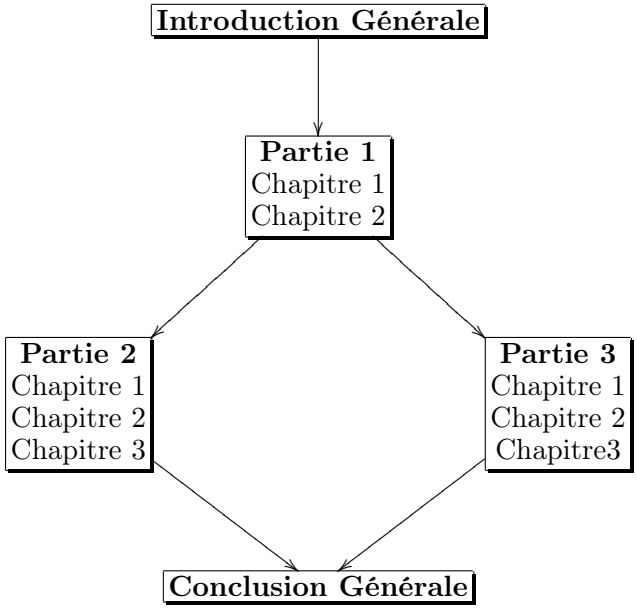


FIG. 1 – Ordres de lecture possibles.

Première partie

Contexte : Approche des Transformées Opérationnelles

Chapitre 1

Editeurs Collaboratifs

Sommaire

1.1 Introduction

Les systèmes collaboratifs sont utilisés pour former un groupe de personnes réparti dans le temps, l'espace et à travers des organisations [27]. Ces personnes collaborent ensemble le temps d'un projet pour répondre à un besoin donné. Un éditeur collaboratif est un exemple typique de tels systèmes. On peut imaginer par exemple l'édition simultanée d'un même article scientifique par plusieurs chercheurs qui sont éloignés géographiquement.

Pour avoir une disponibilité permanente des données, les objets partagés sont répliqués sur chaque site de l'utilisateur. Cependant, les modifications simultanées sur les différentes copies peuvent entraîner une *divergence* (ou une différence) entre les états de ces copies. Par conséquent, il est crucial de mettre en place une procédure de contrôle de la concurrence et ce pour assurer la *convergence* des copies vers un même état. Cette procédure doit concilier à la fois la contrainte "temps réel", la gestion des accès simultanés aux objets, le maintien de la convergence des copies et l'évolution dans le temps du groupe d'utilisateurs.

Dans ce chapitre, nous allons présenter quelques systèmes collaboratifs connus dans la littérature tout en mettant l'accent sur deux critères, à savoir la convergence des données et le passage à l'échelle. Ces deux critères seront définis dans la section 1.2. La section 1.3 présentera deux catégories de systèmes colla-

boratifs selon le contrôle de concurrence : centralisé et réparti. Nous terminons le chapitre par une conclusion.

1.2 Problématique

Un éditeur collaboratif est un système qui permet à plusieurs sites utilisateurs (un utilisateur par site) de modifier simultanément un objet (qui peut être du texte et/ou du graphique). Un tel système est un moyen pour établir des collaborations dans le but de réaliser une tâche commune. On peut imaginer un groupe de chercheurs disséminés à travers le monde qui préparent conjointement un article.

Pour mener à bien une collaboration, chaque site dispose d'une copie de l'objet partagé qu'il peut modifier à volonté. Les changements sont ensuite propagés pour être exécutés sur les autres copies. Afin d'assurer aux utilisateurs un temps de réponse minimum, toute modification générée sur un site doit être exécutée immédiatement.

Nous nous intéressons à des éditeurs collaboratifs qui possèdent les caractéristiques suivantes :

- *Convergence* : Si deux utilisateurs modifient en parallèle leurs copies respectives, celles-ci divergent (c-à-d elles seront différentes). Le système doit alors assurer que les copies vont finir par converger de telle sorte que *toutes les modifications soient prises en considération.*
- *Passage à l'échelle* : Les utilisateurs peuvent quitter ou joindre le groupe sans pour autant affecter le fonctionnement du système. En d'autres termes, un éditeur collaboratif ne doit pas être tributaire du nombre d'utilisateurs à participer durant les sessions de collaboration.

Pour mieux comprendre les éditeurs collaboratifs, nous nous servirons d'un éditeur de texte collaboratif comme exemple. Le texte est considéré comme un objet représenté par une séquence de caractères. Nous supposons que les caractères crées sont identifiés de manière unique durant toute la collaboration. Soit $Char$ l'ensemble des caractères et $Char^*$ l'ensemble des chaînes de caractères où chaque caractère apparaît au plus une fois dans la chaîne.

Les opérations qui peuvent modifier l'état de cet objet sont :

- $Ins(p, c)$: l'effet de cette opération est d'insérer le caractère c à la position p ;
- $Del(p, c)$: cette opération a pour effet d'effacer le caractère c à la position p.

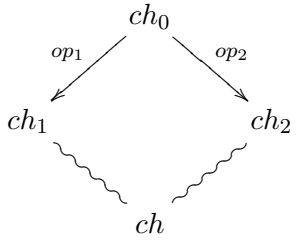
Soit $\mathcal{O}$ l'ensemble de toutes les opérations. Nous définissons la fonction $Do : \mathcal{O} \times Char^* \rightarrow Char^*$ pour appliquer une opération à une chaîne de caractère. Par exemple, $Do(Ins(2, x), \text{“}abc'') = \text{“}axbc''$ et $Do(Del(1, z), \text{“}abc'') = \text{“}abc''$ en supposant que les chaînes de caractères sont indexées à partir de 1. Nous définissons également une fonction booléenne qui vérifie l'existence d'un caractère dans une chaîne de caractères :

$$Exist : Char \times Char^* \rightarrow \{vrai, faux\}.$$

La fonction $Exist$ va nous permettre d'exprimer formellement les effets des opérations :

– Si $op_1 = Ins(p_1, c_1)$ et $ch_1 = Do(op_1, ch_0)$ alors l'effet de op_1 est $Exist(c_1, ch_1) = vrai$ (c-à-d le caractère c_1 existe dans la chaîne ch_1).
– Si $op_2 = Del(p_2, c_2)$ et $ch_2 = Do(op_2, ch_0)$ alors l'effet de op_2 est $Exist(c_2, ch_2) = faux$ (c-à-d le caractère c_2 n'existe pas dans ch_2).

Supposons que $op_1 = Ins(p_1, c_1)$ et $op_2 = Del(p_2, c_2)$ sont exécutées en parallèle sur deux copies ayant la même valeur ch_0. Comme le montre la figure suivante, les deux copies vont diverger puisque ch_1 et ch_2 sont différentes :



L'objectif d'un éditeur collaboratif est de faire converger ch_1 et ch_2 vers un état unique ch qui tient compte des effets de op_1 et op_2. Ce qui est formellement décrit par les égalités suivantes :

$$Exist(c_1, ch_1) = Exist(c_1, ch) \text{ et}$$
$$Exist(c_2, ch_2) = Exist(c_2, ch)$$

1.2.1 Convergence

Il est clair que si chaque opération est exécutée localement et ensuite propagée pour être exécutée sur les autres sites, alors les opérations seront exécutées dans différents ordres sur les différentes copies de l'objet partagé (voir la figure 1.1).

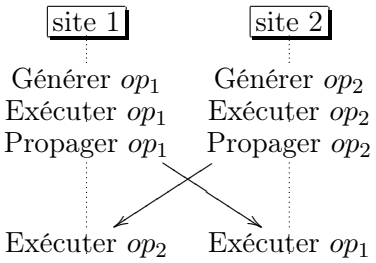


FIG. 1.1 – Exécution concurrente de deux opérations.

Pour restaurer les copies vers un état de convergence, des techniques de contrôle de concurrence sont nécessaires. Il y a deux classes de techniques : *pessimistes* et *optimistes*.

Techniques pessimistes. Elles évitent l'apparition de la divergence en utilisant des *verrous* [5]. Une copie n'est modifiée que si l'on acquiert exclusivement un verrou. Ainsi, dans l'exemple de la figure 1.1, les opérations op_1 et op_2 peuvent être exécutées sur toutes les copies selon l'ordre dans lequel elles acquièrent les verrous [70]. La figure 1.2 montre comment éviter la divergence illustrée dans la

figure 1.1. Les techniques pessimistes génèrent des attentes à cause de l'obtention et la libération des verrous et donc elles sont moins attractives pour les éditeurs collaboratifs nécessitant des temps de réponse courts.

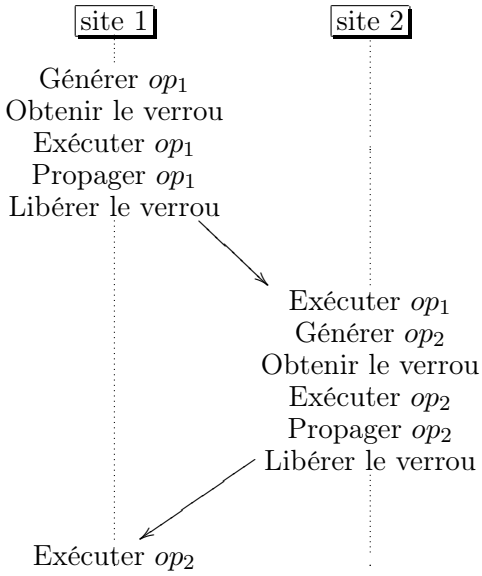


FIG. 1.2 – Exécution pessimiste avec verrouillage.

Techniques optimistes. Elles n'empêchent pas l'occurrence de la divergence mais une fois qu'elle apparaît des procédures sont utilisées pour restaurer le système à un état convergent [76]. Elles permettent à chaque utilisateur de travailler de façon indépendante sur sa copie et de la synchroniser avec les autres de temps en temps. Toutes les opérations sont estampillées d'un ordre total de telle façon qu'elles seront exécutées dans le même ordre sur toutes les copies. Considérons l'exemple de la figure 1.1. Si l'opération op_1 a une plus faible estampille par rapport à celle de op_2, alors op_1 doit s'exécuter avant op_2 sur toutes les copies. Pour ce faire, op_1 et op_2 doivent être réordonnées sur le site 2 par l'annulation de op_2, l'exécution de op_1 et la ré-exécution de op_2 (voir la figure 1.3).

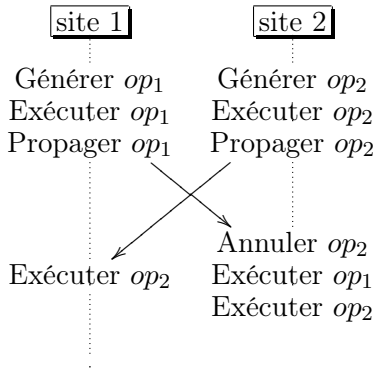


FIG. 1.3 – Exécution concurrente optimiste.

Cette vision optimiste pour converger permet de réduire les temps de réponse et donc elle est appropriée pour les éditeurs collaboratifs. Mais elle nécessite de

fournir pour chaque opération son opération inverse.

Tout au long de cette thèse nous allons nous intéresser à des techniques optimistes.

Néanmoins, nous allons voir à travers deux exemples que la méthode de restauration utilisée dans les techniques optimistes ne tient pas compte tout le temps des effets de toutes les opérations.

Exemple 1.1 *Deux sites* 1 *et* 2 *partagent une chaîne de caractères dont la valeur initiale est* $''abc''$*. Supposons que les opérations sont estampillées par les numéros des sites. Les copies respectives des deux sites sont modifiées en parallèle (voir la figure 1.4) :*

1. *Le site* 1 *génère et exécute* $op_1 = Ins(1, x)$ *dont l'effet consiste à insérer le caractère* x *; il produit la chaîne* $''xabc''$*.*
2. *En parallèle, le site* 2 *génère et exécute* $op_2 = Del(3, c)$ *dont l'effet est de supprimer le caractère* c *qui se trouve à la position* 3*. La chaîne produite est* $''ab''$*.*
3. *Lorsque* op_2 *arrive sur le site* 1*, elle sera directement exécutée puisque son estampille est supérieure à celle de* op_1*. Mais l'exécution de* op_2 *sur* $''xabc''$ *est sans effet car le caractère* c *n'existe pas à la position* 3*.*
4. *Comme l'estampille de* op_1 *est plus faible, elle ne sera donc pas exécutée directement. D'abord,* op_2 *est annulée en restaurant le caractère effacé. Ensuite,* op_1 *est exécutée sur l'état initial* $''abc''$ *pour produire l'état* $''xabc''$*. Enfin, la ré-exécution de* op_2 *est sans effet puisque encore une fois le caractère* c *ne se trouve pas à la bonne position.*

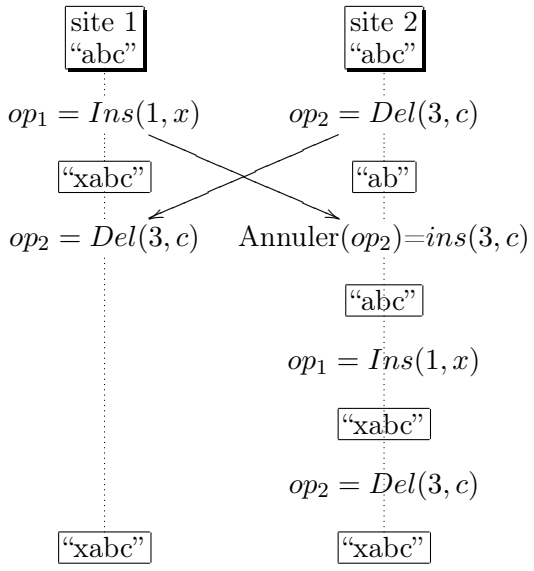


FIG. 1.4 – Une convergence avec une perte de suppression.

Les deux sites convergent vers la même chaîne $''xabc''$ *mais cette dernière ne comporte pas tous les effets, car :*

$$Exist(c,'' ab'') \neq Exist(c,'' xabc'')$$

Cela signifie que le caractère effacé par op_2 existe toujours. L'effet de op_2 a été ignoré pour préserver l'ordre total entre les opérations. Par conséquent, on a perdu une opération durant le processus de restauration vers l'état de convergence, ce qui est une situation indésirable dans un travail collaboratif. ■

Exemple 1.2 *Deux opérations $op_1 = Del(1, a)$ et $op_2 = Ins(4, z)$ modifient en parallèle deux copies ayant la valeur initiale $''abc''$ (voir la figure 1.5). Ces opérations sont également estampillées par les numéros des sites. Le déroulement de ce scénario est comme suit :*

1. *L'opération op_1 a pour effet de supprimer le caractère a qui se trouve à la position 1. Son exécution produit l'état $''bc''$.*
2. *L'opération op_2 a pour effet d'insérer à la fin de la chaîne le caractère z. Son exécution produit l'état $''abcz''$.*
3. *Lorsque op_2 arrive sur le site 1, elle est directement exécutée puisque son estampille est supérieure à celle de op_1. Par contre, elle sera sans effet puisque la position 4 n'existe pas sur la chaîne $''bc''$.*
4. *L'exécution de op_1 sur le site 2 va nécessiter l'annulation de op_2 pour respecter l'ordre total (c-à-d op_1 avant op_2 sur tous les sites). Le caractère inséré par op_1 est détruit pour restaurer l'état initial $''abc''$. Ensuite, op_1 est exécutée pour produire l'état $''bc''$. Enfin, op_2 est ré-exécutée mais en vain puisque la position d'insertion n'existe pas sur $''bc''$.*

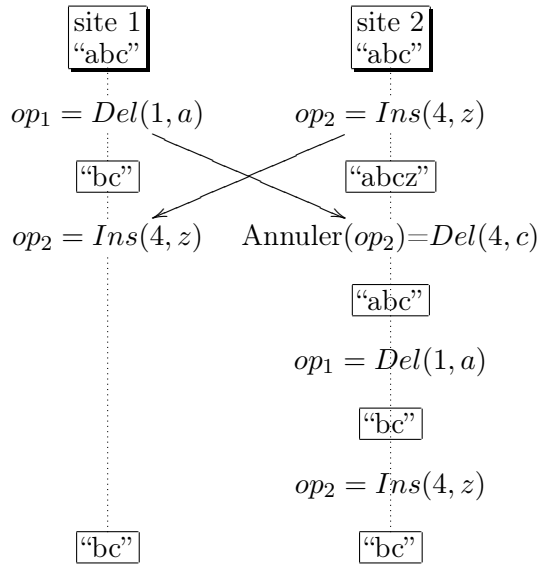


FIG. 1.5 – Une convergence avec une perte d'insertion.

Nous remarquons que les deux sites ont convergé vers le même état $''bc''$ dans lequel l'effet de op_2 a été ignoré :

$$Exist(z,'' abcz'') \neq Exist(z,'' bc'')$$

Le caractère ajouté par op_2 a été perdu au cours de la restauration vers l'état de convergence. L'exemple est simple, mais dans des cas pratiques, le caractère

perdu peut être assimilé à un paragraphe, une page, ... Par conséquent, la perte d'informations est une situation indésirable dans le travail collaboratif. ■

A travers les deux exemples ci-dessus, il est clair que la méthode de restauration utilisée dans les techniques optimistes est insuffisante pour produire un état convergent qui contient les effets de toutes les opérations. Au contraire, elle engendre dans certaines situations des pertes d'information.

Pour résoudre le problème, les éditeurs de texte collaboratifs utilisent l'approche des *transformées opérationnelles* [26]. Cette approche consiste à transformer les opérations (provenant d'un autre site) par rapport aux opérations locales avant d'être exécutées. Cette transformation nécessite l'écriture d'un algorithme, notée IT, qui prend en paramètres deux opérations et retourne en résultat une autre opération. De manière intuitive, on peut écrire la transformation IT pour les opérations Ins et Del :

```
IT(Ins(p1,c1), Del(p2,c2)) =
    si (p1 <= p2) alors retourner Ins(p1,c1)
    sinon retourner Ins(p1-1,c1)
    finsi;

IT(Del(p2,c2), Ins(p1,c1)) =
    si (p2 >= p1) alors retourner Del(p2+1,c2)
    sinon retourner Del(p2,c2)
    finsi;
```

Dans la figure 1.6, nous donnons une solution au problème soulevé à la figure 1.4, à savoir la perte d'une suppression. Sur le site 1, op_2 est transformée par rapport à op_1 pour tenir compte des modifications de cette dernière. Ainsi, la position d'effacement de op_2 est incrémentée pour qu'elle puisse effacer correctement le caractère c. On a donc un état de convergence sur les deux sites qui contient les effets des opérations op_1 et op_2 :

$$Exist(x,''xabc'') = Exist(x,''xab'') = vrai$$
$$Exist(c,''ab'') = Exist(c,''xab'') = faux$$

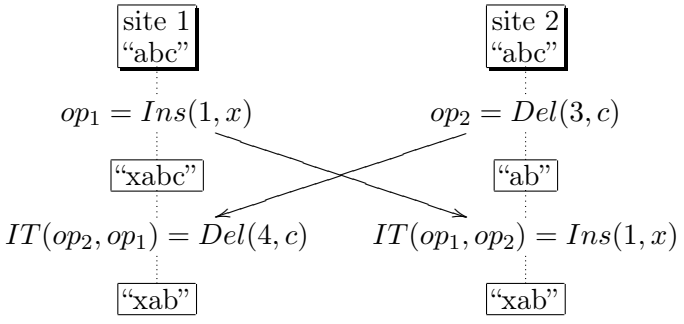


FIG. 1.6 – Une convergence sans perte de suppression.

le problème illustré à la figure 1.5 concernait la perte d'une insertion après convergence. La figure 1.7 montre comment résoudre ce problème en utilisant la

transformation de op_2. En effet, quand celle-ci arrive sur le site 1, elle est d'abord transformée par rapport à op_1 pour mettre à jour sa position d'insertion. Ainsi, la forme transformée de op_2 pourra insérer correctement le caractère z. L'état de convergence des deux sites $''bcz''$ contient donc les effets des opérations op_1 et op_2 :

$$Exist(a,''bc'') = Exist(a,''bcz'') = faux$$
$$Exist(z,''abcz'') = Exist(z,''bcz'') = vrai$$

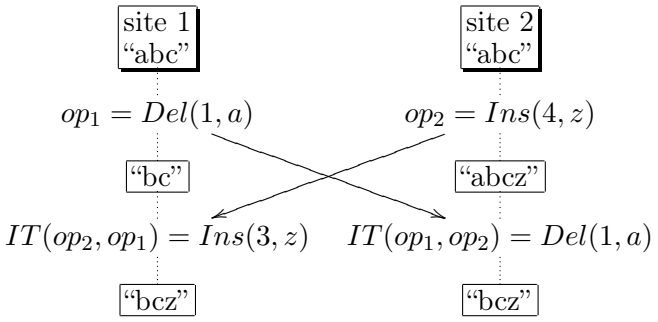


FIG. 1.7 – Une convergence sans perte d'insertion.

1.2.2 Passage à l'échelle

Dans le domaine des éditeurs collaboratifs, le passage à l'échelle est défini comme la capacité pour l'application de pouvoir rester opérationnelle malgré une montée du nombre de sites. En effet, ce nombre peut être variable et la topologie d'interconnexion entre les sites peut changer durant les sessions de collaboration. Les utilisateurs peuvent joindre ou quitter à tout moment une session de collaboration.

Comme composante essentielle des éditeurs collaboratifs, les algorithmes de diffusion des modifications sont sensibles au problème du passage à l'échelle. Il est bien connu que les algorithmes de diffusion basé sur un ordre total passe difficilement à l'échelle [18]. Seuls les algorithmes de diffusion à grande échelle, tels que les algorithmes à propagation épidémique, passent facilement à l'échelle [19; 28]. Dans une approche épidémique, les modifications se propagent de proche en proche aux différents sites (à chaque fois que ces derniers se connectent sur le réseau) à la façon dont une épidémie se propage.

A titre d'exemple, nous pouvons citer USENET qui est une architecture sur laquelle reposent des groupes de discussions (news group) [77]. Elle est très intéressante car elle est un service répliqué reliant de milliers de serveurs, opérationnel depuis longtemps et utilisé à très grande échelle. Le modèle de réplication utilisé a facilité la multiplication des groupes de discussion. Il se base sur l'envoi des messages contenant des opérations sur les données. Malheureusement, ces opérations ne concernent que la création des données. La destruction n'est pas prise en considération.

1.3 Analyse des Systèmes Existants

Pour l'étude des éditeurs collaboratifs, nous nous concentrons sur des systèmes utilisant des techniques optimistes. Une analyse sera faite sur ces systèmes sur la base des critères de convergence et du passage à l'échelle.

1.3.1 Contrôle de concurrence centralisé

Editeur Wiki

Un wiki est un système centralisé de gestion de contenu de site Web qui rend les pages Web librement et également modifiables par tous les visiteurs autorisés [15]. Les wikis sont utilisés pour faciliter l'écriture collaborative de documents avec un minimum de contrainte. Le wiki a été inventé par Ward Cunningham en 1995, pour une section d'un site sur la programmation informatique qu'il a appelée WikiWikiWeb. Le mot « wiki » vient du terme hawaiien wiki wiki, qui signifie « rapide » ou « informel ». Au milieu des années 2000, les wikis ont atteint un bon niveau de maturité et sont associés au Web 2.0. Créée en 2001, l'encyclopédie Wikipédia est devenue le wiki le plus visité au monde.

Une page Wiki peut être changée en parallèle par deux utilisateurs. Les modifications de ces derniers sont estampillées selon un ordre total. Ainsi une modification concurrente d'une page 0 peut produire deux copies à savoir les pages 1 et 2 (voir la figure 1.8). Les deux pages seront ensuite sérialisées l'une après l'autre selon l'ordre de modification. En effet, seule la dernière modification sera prise en considération pour l'affichage.

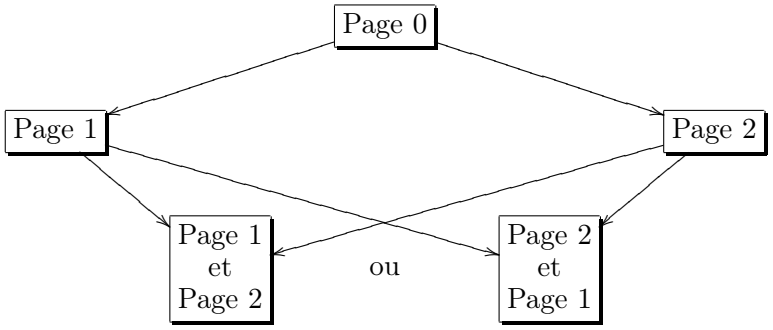


Fig. 1.8 – Modification concurrente d'une page Wiki.

D'un point de vue convergence et passage à l'échelle, l'éditeur Wiki se présente comme suit :

- **Convergence.** Les modifications concurrentes sont ordonnées selon un ordre total dictées par les estampilles. Le système converge donc vers un état qui tient compte seulement de la dernière modification. Les autres modifications sont ignorées.
- **Passage à l'échelle.** L'éditeur Wiki est hébergé sur un serveur central qui est un point de congestion. Donc si ce dernier tombe en panne, alors le système sera hors usage.

Synchroniseur de fichiers

Un synchroniseur de fichiers permet de stocker plusieurs copies d'un système de fichiers sur des machines différentes (station de travail, ordinateur portable, ...) et de les synchroniser à partir d'une machine centrale en propageant les changements de chaque copie sur les autres copies. A titre d'exemple, nous pouvons citer Unison [67], HotSync [36], ActiveSync [1] et iSync [45].

Nous avons choisi de nous intéresser à un synchroniseur *SO*6[1], qui est développé par l'équipe ECOO de l'INRIA Lorraine. Il est basé sur l'approche des transformées opérationnelles [60; 64]. Dans *SO*6, les opérations sont estampillées selon un ordre total. La propagation des opérations est basée sur un mécanisme de diffusion différée. En effet, l'envoi d'une opération nécessite la réception de toutes les opérations estampillées pour transformer les opérations locales en conséquence. Il est ensuite possible d'estampiller ces opérations locales et de les envoyer aux autres sites. Le fonctionnement de *SO*6 se prête bien au paradigme "copier-modifier-fusionner" utilisé dans le gestionnaire de versions CVS [12], où un utilisateur ne peut publier ses modifications que s'il a pris en compte toutes les autres modifications déjà publiées.

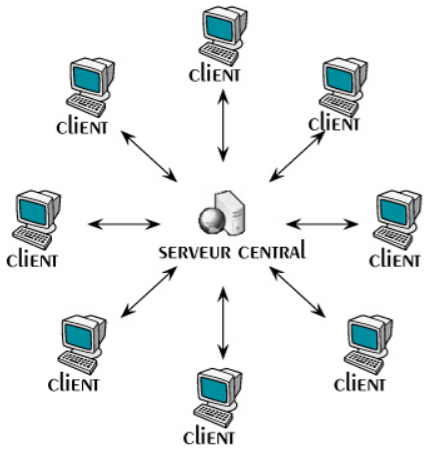


FIG. 1.9 – Architecture générale de *SO*6.

Comme le montre la figure 1.9, l'architecture de *SO*6 s'articule autour d'un serveur central. Ce dernier héberge un estampilleur qui délivre une estampille à chaque opération. Une application cliente s'exécute sur chaque machine des utilisateurs. Les applications clientes n'échangent pas directement les opérations ; elles doivent les faire transiter par l'estampilleur.

D'un point de vue convergence et passage à l'échelle, le synchroniseur *SO*6 se présente comme suit :

– **Convergence.** Pour restaurer les copies divergentes vers une copie identique, *SO*6 utilise l'approche des transformées opérationnelles qui est

[1] `http://libresource.inria.fr/`

connue par la qualité meilleure des états de convergence qu'elle permet d'obtenir.
- **Passage à l'échelle.** *SO*6 ne peut pas passer à l'échelle car il se base sur un serveur central (qui héberge l'estampilleur).

1.3.2 Contrôle de concurrence réparti

Bayou

Bayou est un système de gestion de données pour des applications collaboratives dans un environnement mobile [66; 87]. Une copie de l'objet partagé se trouve sur chaque serveur. Les applications interagissent avec cette copie à travers une interface de programmation Bayou API (Application Interface Programming), qui permet à l'utilisateur d exécuter deux types d opérations sur la copie locale : il s'agit des opérations de lecture et d'écriture. Les opérations de lecture consistent à obtenir certaines valeurs contenues dans la copie, tandis que les opérations d écriture permettent d'insérer, de modifier, et de supprimer certaines valeurs de la copie de données.

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.

Une opération d'écriture comprend la mise à jour à effectuer (`update`), la précondition 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 précondition est fausse, et dans ce cas, la procédure de fusion est exécutée.

Considérons l'exemple de la figure 1.4. Quand op_2 arrive sure le site 1, sa précondition (le caractère c se trouvant à la position 3), est fausse sur l'état $''xabc''$. Dans ce cas, Bayou peut déclencher une procédure de réconciliation qui consiste à chercher la position correcte du caractère c et ensuite le détruire.

D'un point de vue convergence et passage à l'échelle, Bayou *SO*6 se présente comme suit :
- **Convergence.** Si l'on ne considère que les opérations dans le préfixe p sont ordonnées selon 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. Les opérations sont munies de précondition et d'une procédure de fusion qui sont utiles pour exécuter correctement l'opération sur d'autres copies. Aussi l'état de convergence pourra tenir compte de toutes les opérations concurrentes.

– **Passage à l'échelle.** Comme les opérations sont ordonnées dans le préfixe p selon un ordre total dicté par un serveur central (désigné au lancement du système), alors le passage à l'échelle sera limité. En effet, une panne du serveur central engendrerait une perte de l'ordre total sur les opérations.

IceCube

IceCube [46; 71; 78] 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 modifiés 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.

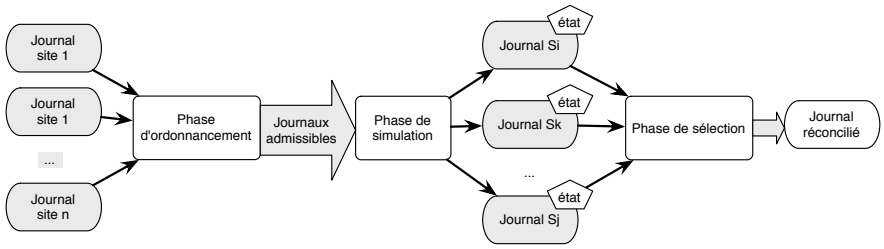


FIG. 1.10 – Processus de réconciliation dans IceCube.

La réconciliation se déroule en trois phases illustrées par la figure 1.10 :

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.

Considérons les exemples des figures 1.4 et 1.5. Dans de telles situations, IceCube doit indiquer explicitement un ordre entre les deux opérations pour pouvoir converger sur les deux sites. Deux cas sont possible :

- Ins avant Del : dans ce cas on aura un état de convergence semblable à celui de la figure 1.4 dans lequel une suppression a été ignorée.
- Del avant Ins : dans ce cas on aura la même état de convergence que celui de la figure 1.5 dans lequel une insertion a été ignorée.

D'un point de vue convergence et passage à l'échelle, IceCube $SO6$ se présente comme suit :

- **Convergence.** Certes IceCube assure la convergence puisqu'il impose un ordre total sur l'exécution des opérations. Mais cette technique de restauration n'est pas sans risque de perdre des opérations.
- **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 réconciliation vers ce site. L'algorithme principal d'IceCube peut alors calculer un journal réconcilié. 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.

1.4 Conclusion

Dans ce chapitre, nous avons passé en revue certains systèmes collaboratifs tout en mettant l'accent sur les critères de la convergence des données et le passage à l'échelle. A ce sujet, nous avons remarqué que ces systèmes présentent des lacunes majeures. Par conséquent, il est difficile de réutiliser ces systèmes dans un contexte pair-à-pair sans modifier leurs algorithmes.

Dans le chapitre suivant, nous allons présenter l'approche des transformées opérationnelles qui a priori répond surtout aux exigences de la convergence des copies.

Chapitre 2

Modèle des Transformées Opérationnelles

Sommaire

2.1 Introduction

L'approche des *transformées opérationnelles* a été utilisée dans le domaine des éditeurs collaboratifs synchrones [26] pour sérialiser les opérations concurrentes. Elle se présente sous forme d'algorithmes qui consistent à transformer les opérations reçues pour tenir compte de celles qui ont pu entre temps modifier l'état de l'objet. Elle utilise les propriétés sémantiques des opérations pour garantir la convergence des copies. Plus précisément, elle exploite la transformation des opérations pour construire une histoire de chaque copie de l'objet. Les histoires des différentes copies ne sont pas identiques mais elles sont équivalentes (elles aboutissent au même état final) puisque l'ordre d'exécution des opérations concurrentes peut être différent d'une histoire à une autre.

Comparée à d'autres techniques optimistes, elle possède les avantages suivants : (i) Elle supporte un travail collaboratif sans contraintes. En effet, elle n'impose pas un ordre total sur les opérations concurrentes. Aussi, les sites peuvent échanger leurs modifications dans n'importe quel ordre. (ii) Elle permet la transformation des opérations pour les exécuter dans un ordre arbitraire même si à l'origine ces opérations ne commutent pas. (iii) Elle produit un état de convergence sans perte de mises-à-jour.

Chaque objet partagé doit avoir son propre algorithme de transformation dont l'écriture incombe aux développeurs de l'éditeur collaboratif. Ainsi, pour chaque paire d'opérations, le développeur doit préciser comment transformer ces

opérations l'une par rapport à l'autre. Pour garantir la convergence des copies, un algorithme de transformation doit satisfaire deux conditions [68; 73].

Dans ce chapitre, nous donnons une présentation générale de l'approche des transformées opérationnelles. Dans le section 2.2, nous donnons les éléments de base du modèle des transformées opérationnelles. La section 2.3 présente les conditions qui doivent être satisfaites pour garantir la convergence des copies. Dans la section 2.4 nous donnons les étapes nécessaires pour intégrer localement les opérations provenant d'autres sites. Enfin, nous terminons le chapitre par une conclusion.

2.2 Modèle

Un éditeur collaboratif est considéré comme un groupe de sites (ou utilisateurs) qui manipulent un objet partagé, tel que un document texte ou graphique.

Définition 2.1 (Objet collaboratif) *Un objet est dit* collaboratif *s'il est une ressource de données partagée en lecture/écriture par plusieurs sites.* ■

L'objet collaboratif est répliqué sur plusieurs sites de telle façon que chaque site pourra lire et/ou écrire sur sa propre copie. Chaque objet collaboratif possède :

1. Un type (*i.e.* un texte, un arbre XML, un système de fichiers, etc.) qui définit un ensemble d'états possibles, noté par $\mathcal{S}$;
2. Un ensemble d'opérations primitives, noté par $\mathcal{O}$, où chaque opération possède une précondition de telle façon que l'opération ne peut s'exécuter sur un état de l'objet que si sa précondition est vérifiée.
3. Une fonction de transition $Do : \mathcal{S} \times \mathcal{O} \to \mathcal{S}$.

Définition 2.2 (Opération locale/distante) *Une opération est dite* locale *si elle est générée localement sur site. Une opération est dite* distante *si elle provient d'un autre site.* ■

Chaque site génère séquentiellement des opérations et les stocke dans une liste appelée une *histoire* (ou *journal des opérations*).

Définition 2.3 (Histoire) *Une séquence d'opérations est appelée une* histoire. *Nous définissons les histoires comme les éléments de l'ensemble $\mathcal{H}$ qui sont exprimées par la syntaxe suivante :*

$$h ::= \Lambda \mid [op] \mid [h; h]$$

où $op \in \mathcal{O}$. Le symbole Λ dénote l'histoire vide (que nous noterons également $[]$). Nous utilisons la notation $|h|$ pour représenter la longueur de h. ■

L'expression $(st)h$ représente l'état obtenu en exécutant l'histoire h sur l'état st. Elle est récursivement définie comme suit :

$$(st)\Lambda = st \text{ et}$$
$$(st)[op_1; op_2; \ldots; op_n] = Do(Do(Do(Do(st, op_1), op_2), \ldots), op_n).$$

Définition 2.4 (Légalité) *Une histoire h est dite* légale *sur un état st si la précondition de chaque opération de h est satisfaite.* ■

Définition 2.5 (Equivalence des histoires) *Deux histoires h_1 et h_2 sont dites* équivalentes *pour tout état st si les conditions suivantes sont satisfaites :*

1. *h_1 et h_2 sont légales sur st ;*
2. *$|h_1| = |h_2|$;*
3. *$(st)h_1 = (st)h_2$;*

Cette relation d'équivalence est notée $\equiv_{st}$. ■

Lemme 2.1 *La relation $\equiv_{st}$ est une congruence.* ■

Preuve. La relation $\equiv_{st}$ est une congruence si pour tous h_1, h_2, $h_3 \in \mathcal{H}$, on a $h_1 \equiv_{st} h_2$ implique $[h_3; h_1] \equiv_{st} [h_3; h_2]$ et $[h_1; h_3] \equiv_{st} [h_2; h_3]$. La preuve de ce lemme peut se faire par induction sur la longueur des histoires h_1 et h_2. ■

Durant l'exécution des opérations, il existe une relation de précédance qui doit être préservée par tous les sites. Une opération op_1 précède causalement une autre opération op_2 si et seulement si op_2 a été générée sur un site après que op_1 ait été exécutée sur le même site.

Définition 2.6 (Relation de causalité) *Soient deux opérations op_1 et op_2 générées respectivement sur les sites i et j. On dit que que op_1* précède causalement *op_2 (ou op_2* dépend causalement *de op_1), noté $op_1 \rightarrow op_2$, ssi : (i) $i = j$ et op_1 était générée avant op_2 ; ou, (ii) $i \neq j$ et l'exécution de op_1 sur le site j est arrivée avant la génération de op_2.* ■

Généralement, cette dépendance est maintenue en utilisant des vecteurs d'horloges [59; 30]. Dans un système à n sites, un tel vecteur V possède n composantes. Pour un site j, chaque composante $V[i]$ compte le nombre d'opérations générées du site i qui ont été déjà exécutées sur le site j. 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.

Une importante propriété que doit posséder les éditeurs collaboratifs est la préservation de la précédance causale.

Définition 2.7 (Préservation de la précédence causale) *Si $op_1 \rightarrow op_2$ alors op_2 doit s'exécuter toujours après op_1 sur tous les sites.* ■

Définition 2.8 (Relation de concurrence) *Deux opérations op_1 et op_2 sont dites* concurrentes*, notée $op_1 \parallel op_2$, ssi ni $op_1 \rightarrow op_2$, ni $op_2 \rightarrow op_1$.* ■

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 $V_{op_2}[S_{op_1}] \leq V_{op_1}[S_{op_1}]$ et $V_{op_2}[S_{op_2}] \geq V_{op_1}[S_{op_2}]$.

Deux opérations concurrentes op_1 et op_2 peuvent être exécutées sur les différents sites dans un ordre quelconque. Comme vu dans le chapitre précédent, si l'on exécute op_1 avant 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. Dans le modèle des transformées opérationnelles, les opérations distantes doivent être transformées par rapport aux opérations concurrentes locales.

La transformation est réalisée par l'intermédiaire d'un algorithme connu sous différents noms dans la littérature [26; 73; 81; 85]. Ainsi, il est appelé *transposition en avant* dans [81] et *transformation inclusive* dans [85].

Définition 2.9 (Algorithme de transformation) *Un algorithme de transformation est une fonction $IT : \mathcal{O} \times \mathcal{O} \rightarrow \mathcal{O}$ définie pour tout (op_1,op_2)$\in \mathcal{O} \times \mathcal{O}$ tel que :*

1. *op_1 et op_2 sont concurrentes et définies sur le même état, et ;*
2. *la précondition de $IT(op_1, op_2)$ est satisfaite sur l'état résultant de l'exécution de op_2.* ■

L'algorithme de transformation est utilisée comme suit : soient op_i et op_j deux opérations concurrentes définies sur le même état. Supposons que op_i et op_j sont générées respectivement sur les sites i et j (avec $i \neq j$). Soient $op'_i = IT(op_i, op_j)$ et $op'_j = IT(op_j, op_i)$. Ainsi, le site i exécute l'histoire $[op_i; op'_j]$ et le site j exécute l'histoire $[op_j; op'_i]$. D'une manière générale, lorsqu'une opération distante arrive sur un site donné, elle est transformée pour inclure les effets d'autres opérations (qu'elle n'a pas vu sur son site originel) pour être correctement intégrée dans l'histoire locale.

Exemple 2.1 *Deux sites (ou utilisateurs) modifient en parallèle un document partagé, dont les copies contiennent initialement la chaîne de caractères $"efet"$ (voir la figure 2.1). Le site 1 exécute l'opération $op_1 = Ins(2, f)$ pour insérer le caractère f à la position 2. Au même moment, le site 2 ajoute à la fin de la chaîne le caractère s en exécutant l'opération $op_2 = Ins(5, s)$. Les opérations op_1 et op_2 sont donc concurrentes puisqu'aucune n'a vu l'autre au moment de la génération. Lorsque op_2 arrives sur le site 1, elle doit tenir compte que op_1 a été exécutée avant son arrivée. Ainsi, op_2 est transformée par rapport à op_1 pour devenir $op'_2 = IT(op_2, op_1) = Ins(6, s)$ (la position d'insertion a été incrémentée car op_1) a inséré un caractère avant celui de op_2). Par contre, lorsque op_2 arrive sur le site 2, elle reste inchangée car sa position d'insertion se trouve avant celle de op_2. Les deux sites convergent donc vers la même chaîne de caractères $"effets"$ qui contient les effets de op_1 et op_2. De manière intuitive, on peut écrire la transformation IT pour les opérations Ins comme suit :*

```
IT(Ins(p1,c1), Ins(p2,c2)) =
    si (p1 <= p2) alors retourner Ins(p1,c1)
    sinon retourner Ins(p1+1,c1)
    finsi;

IT(Ins(p1,c1), Del(p2,c2)) =
    si (p1 <= p2) alors retourner Ins(p1,c1)
    sinon retourner Ins(p1-1,c1)
    finsi;

IT(Del(p2,c2), Ins(p1,c1)) =
    si (p2 >= p1) alors retourner Del(p2+1,c2)
    sinon retourner Del(p2,c2)
    finsi;
```

■

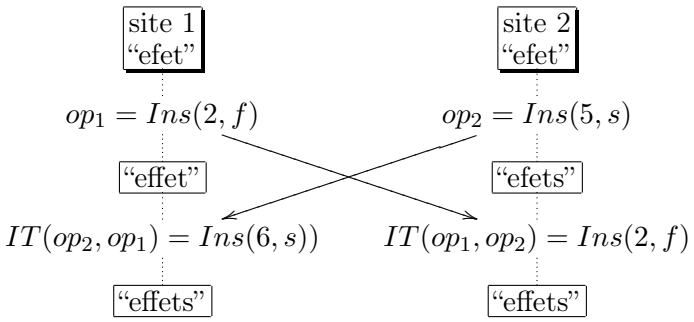


FIG. 2.1 – Transformation des opérations concurrentes.

Il y a des cas où il faut transformer des opérations concurrentes qui ne sont pas générées sur le même état. De tels cas correspondent à une situation de concurrence partielle [14; 80].

Définition 2.10 (Concurrence partielle) *Soient deux opérations concurrentes op_1 et op_2 définies respectivement sur les états st_1 et st_2. Elles sont dites en* concurrence partielle *ssi $st_1 \neq st_2$.* ■

Dans une édition collaborative, la concurrence partielle peut occasionner des situations de divergence. L'exemple suivant montre les résultats incorrects que l'on peut avoir si l'on applique directement la transformation entre deux opérations partiellement concurrentes.

Exemple 2.2 *Considérons deux sites qui tentent de corriger la chaîne partagée "têts", comme le montre la figure 2.2. Le site* 1 *génère deux opérations consécutives $op_1 = Ins(4, u)$ et $op_2 = Ins(5, e)$. En parallèle, le site* 2 *génère l'opération $op_3 = Del(4, s)$. Nous avons donc $op_1 \rightarrow op_2$, $op_1 \parallel op_3$ et $op_3 \parallel op_1$. Les opérations op_2 et op_3 ont été générées respectivement sur les états "têtus" et "têts". Ces deux états sont différents. Ainsi op_2 et op_3 sont partiellement concurrents.*

Lorsque op_3 arrive sur le site 1*, elle est transformée consécutivement par rapport à op_1 et op_2. On obtient donc $op'_3 = IT(IT(op_3, op_1), op_2) = Del(6, s)$. En exécutant op'_3, on obtient la chaîne voulue "têtue". Au niveau du site* 2*, la transformation de op_1 par rapport op_3 produit $op'_1 = IT(op_1, op_3) = op_1$. Quant à la transformation de op_2 par rapport à op_3, elle donne $op'_2 = IT(op_2, op_3) = Ins(4, s)$ dont l'exécution produit la chaîne "têteu". Il est clair que les deux sites* 1 *et* 2 *divergent puisque leurs états sont différents. Cette divergence est causée par une mauvaise application de la fonction IT. En effet, $IT(op_2, op_3)$ est incorrecte selon la définition 2.9, car op_2 et op_3 sont définies sur deux états différents.* ■

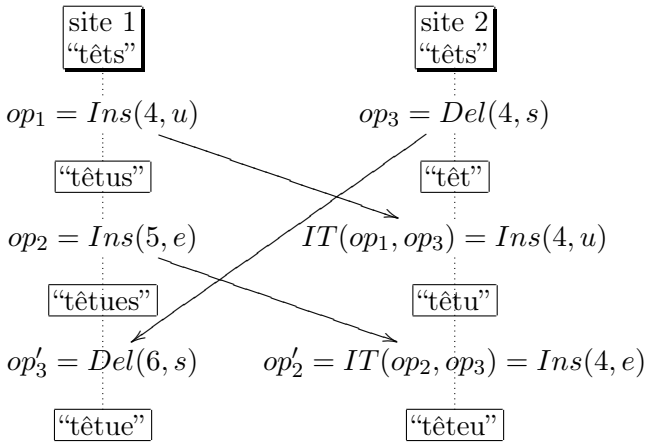


FIG. 2.2 – Transformation incorrecte dans une situation de concurrence partielle.

Pour résoudre le problème de la concurrence partielle, op_2 ne doit pas être transformée directement par rapport à op_3 car op_2 dépends causalement de op_1. Aussi, pour intégrer correctement op_2 sur le site 2, il faut tenir compte des effets des opérations op_3 et $op'_1 = IT(op_1, op_3)$. En d'autres termes, il faut d'abord transformer op_3 par rapport à op_1 pour obtenir une opération op'_3 définie sur le même état que celui de op_2. Ensuite, il faut transformer op_2 par rapport à op'_3. L'intégration correcte de op_2 est illustrée dans la figure 2.3.

Lorsque tous les sites cessent de propager des opérations, le système arrivera à un état où il n'y a plus de modifications sur l'objet collaboratif.

Définition 2.11 (Système au repos) *Un éditeur collaboratif est dit* au repos *si toutes les opérations générées ont été exécutées sur tous les sites.* ■

S'il n'y a aucune modification qui transite sur le réseau alors toutes les copies de l'objet collaboratif doivent être identiques.

Définition 2.12 (Propriété de convergence) *La propriété de* convergence *signifie que toutes les copies de l'objet collaboratif sont identiques lorsque l'éditeur collaboratif est au repos.* ■

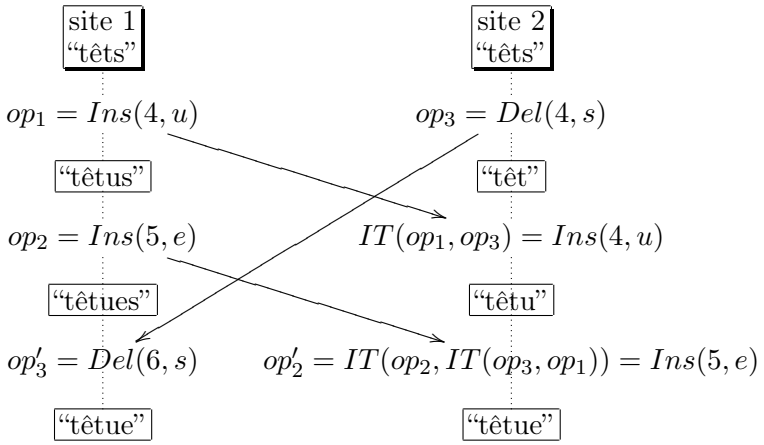


FIG. 2.3 – Transformation correcte dans une situation de concurrence partielle.

2.3 Conditions de Convergence

La préservation des relations de causalité entre les opérations est nécessaire mais pas suffisante pour obtenir dans toutes les situations des exécutions qui garantissent la convergence des copies sur tous les sites. L'exemple suivant montre un cas de divergence entre deux opérations concurrentes.

Exemple 2.3 *Considérons deux sites partageant une chaîne de caractères dont la valeur initiale est* $''mis''$ *voir la figure 2.4. Le site* 1 *insère le caractère* $'a'$ *à la position* 2 *en exécutant l'opération* $op_1 = Ins(2, a)$*. Au même moment, le site* 2 *exécute l'opération* $op_2 = Ins(2, o)$ *dont l'effet consiste à ajouter le caractère* $'o'$ *à la position* 2*. Pour intégrer les opérations concurrentes nous utilisons la fonction de transformation donnée à l'exemple 2.1. Sur le site* 1*,* op_2 *est transformée par rapport à* op_1 *pour produire* $op'_2 = IT(op_2, op_1) = Ins(2, o)$ *dont l'exécution donne la chaîne* $''moais''$ *qui contient les effets des deux opérations. Quant à* op_1*, elle est transformée sur le site* 2 *par rapport à* op_2 *pour donner* $op'_1 = IT(op_1, op_2) = Ins(2, a)$*. Le résultat de l'exécution de* op'_1 *est la chaîne* $''maois''$ *qui contient aussi les effets des deux opérations. Nous sommes en présence d'une situation de divergence puisque les deux sites* 1 *et* 2 *ont des états différents.* ■

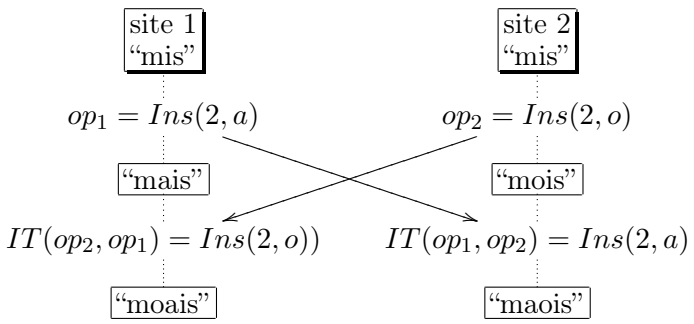


FIG. 2.4 – Divergence de copies due à la violation de la condition $TP1$.

Afin de garantir la convergence des copies après l'application de la fonction de transformation IT, cette fonction doit satisfaire la condition suivante [26; 68; 73; 81; 85] :

Définition 2.13 (Condition TP1) *La fonction IT satisfait la condition $TP1$ ssi pour toutes les opérations concurrentes définies sur le même état op_1, $op_2 \in \mathcal{O}$:*

$$[op_1; op'_2] \equiv_{st} [op_2; op'_1]$$

où $op'_1 = IT(op_1, op_2)$ et $op'_2 = IT(op_2, op_1)$. ■

La condition $TP1$ définit une *identité d'états*. Elle stipule que l'état produit en exécutant op_1 avant op_2 est le même que celui résultant de l'exécution de op_2 avant op_1.

La fonction IT défini à l'exemple 2.1 satisfait $TP1$ sauf dans le cas de deux insertions concurrentes à la même position, comme l'a montré l'exemple 2.3. Dans ce cas, on est en présence d'une situation de *conflit* entre deux insertions concurrentes. Il faut faire un choix pour déterminer quel caractère ajouter avant l'autre. Cependant, il faut assurer que le même choix sera fait sur tous les sites. Plusieurs solutions ont été proposées dans la littérature, comme l'utilisation d'un *ordre de priorité* sur les opérations [26], l'utilisation d'un ordre total sur les *identifiants des sites* [73], et l'utilisation de l'ordre *alphabétique* sur les caractères [81]. Ainsi si l'on utilise l'identifiant du site où l'opération a été générée, on doit ajouter cette information comme un nouveau paramètre dans les opérations. La définition de IT pour les opérations Ins devient donc :

```
IT(Ins(p1,c1,id1), Ins(p2,c2,id2)) =
si ((p1<p2) ou (p1 = p2 et id1<id2)) alors retourner Ins(p1,c1)
sinon retourner Ins(p1+1,c1)
finsi;
```

D'après cette définition, dans le cas où les positions d'insertion sont égales le caractère qui a le plus grand identifiant du site sera inséré après l'autre. La figure 2.5 illustre l'utilisation de cette nouvelle définition pour éviter la divergence de l'exemple 2.3.

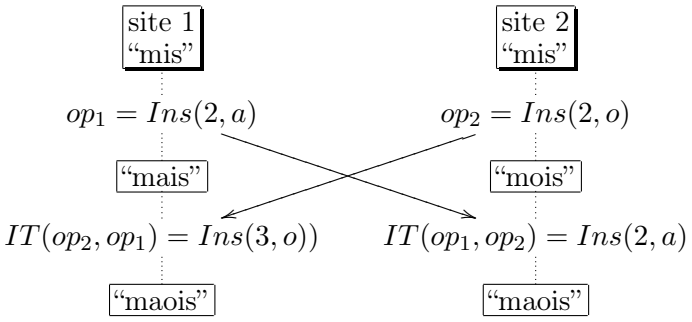


FIG. 2.5 – Convergence de copies avec la satisfaction de la condition $TP1$.

La condition $TP1$ est nécessaire mais pas suffisante pour assurer la convergence des copies lorsque l'on a plus de deux opérations concurrentes. L'exemple suivant montre une telle situation :

Exemple 2.4 *Considérons trois sites* 1, 2 *et* 3 *qui éditent simultanément le même texte (voir la figure 2.6). Initialement, les utilisateurs démarrent à partir du même état "ciré". Ils génèrent concurremment les opérations* $op_1 = Ins(3, `r')$, $op_2 = Del(2)$ *et* $op_3 = Ins(2, `a')$ *pour changer leurs états respectivement en "cirré", "cré" et "cairé". Pour intégrer les trois opérations, nous utilisons la nouvelle définition de IT qui vérifie* $TP1$. *Au site* 1, *quand* op_2 *arrive, elle est d'abords transformée par rapport à* op_1, *i.e.* $op'_2 = IT(op_2, op_1) = Del(2)$ *et ensuite* op'_2 *est exécuté donnant lieu à l'état "crré". Sur le même site, l'intégration de* op_3 *passe par la transformation par rapport à la séquence* $[op_1; op'_2]$ *qui résulte en* $op''_3 = Ins(2, `a')$ *dont l'exécution mène à l'état "carré".*

Au niveau du site 2, op_1 *est en premier transformée par rapport à* op_2, *i.e.* $op'_1 = IT(op_1, op_2) = Ins(2, `r')$, *et ensuite le résultat* $op'_1$ *est exécutée produisant l'état "crré". Quand* $op_3$ *arrive, elle est transformée par rapport à* $[op_2; op'_1]$ *dont le résultat* $op'_3 = Ins(3, `a')$ *est ensuite exécuté produisant l'état "craré". On constate que la transformation de* op_3 *par rapport aux deux séquences équivalentes* $[op_1; op'_2]$ *et* $[op_2; op'_1]$ *donne deux opérations différentes, i.e* $op'_3 = Ins(3, `a')$ *et* $op''_3 = Ins(2, `a')$, *ce qui induit à une divergence de données puisque les états des sites* 2 *et* 3 *sont différents.* ■

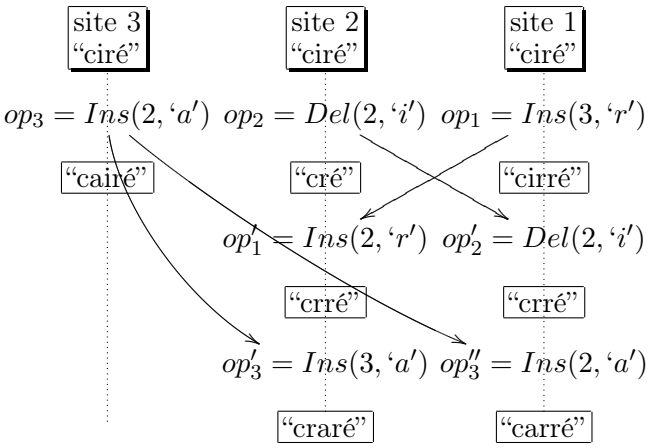


FIG. 2.6 – Divergence de copies pour trois opérations concurrentes.

En plus de $TP1$, la fonction IT doit satisfaire la condition suivante [26; 68; 73; 81; 85] :

Définition 2.14 (Condition TP2) *Soient trois opérations concurrentes définies sur le même état* op, op_1 *et* op_2. *La fonction IT satisfait la condition* $TP2$ *ssi :*

$$IT(IT(op, op_1), op'_2) = IT(IT(op, op_2), op'_1)$$

où $op'_1 = IT(op_1, op_2)$ *et* $op'_2 = IT(op_2, op_1)$. ■

La condition $TP2$ définit une *identité d'opérations*. Elle stipule que 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 cette séquence ont été transformées. Dans [73; 81; 56], il a été montré que $TP1$ et

$TP2$ sont suffisantes pour satisfaire la propriété de convergence (voir la définition 2.12) pour un *nombre arbitraire d'opérations concurrentes* qui peuvent s'exécuter dans *ordre arbitraire*.

Nous verrons dans la partie 3 de cette thèse que la condition $TP2$ est difficile à satisfaire pour les opérations Ins et Del.

Dans ce qui suit, nous allons étendre la fonction IT pour qu'elle soit appliquée sur les histoires d'opérations.

Définition 2.15 (Extension de IT) *Nous définissons* $IT^* : \mathcal{H} \times \mathcal{H} \rightarrow \mathcal{H}$ *comme suit :*

$$
\begin{aligned}
IT^*(h, \Lambda) &= h && (2.1)\\
IT^*(\Lambda, h) &= \Lambda && (2.2)\\
IT^*(h_1, [h_2; h_3]) &= IT^*(IT^*(h_1, h_2), h_3) && (2.3)\\
IT^*([h_1; h_2], h_3) &= [IT^*(h_1, h_3)\,;\, IT^*(h_2, IT^*(h_3, h_1))] && (2.4)\\
IT^*([op_1], [op_2]) &= [IT(op_1, op_2)] && (2.5)
\end{aligned}
$$

pour toutes histoires légales h, h_1, h_2 *et* h_3, *et pour toutes opérations* op_1 *et* op_2. ■

Soient h_1 et h_2 deux histoires concurrentes et légales sur le même état. Si $h'_1 = IT^*(h_1, h_2)$, alors IT^* est utilisée pour ajouter une histoire légale à h_2. Les deux premières équations de la définition 2.15 sont triviales. L'équation (2.3) signifie que la transformation de h_1 par rapport à $[h_2; h_3]$ procède en deux étapes : transformer d'abord h_1 par rapport à h_2 pour produire $IT^*(h_1, h_2)$; ensuite transformer $IT^*(h_1, h_2)$ par rapport à h_3. Quant à l'équation (2.4), la transformation de l'histoire $[h_1; h_2]$ par rapport à h_3 se fait comme suit : on transforme d'abord h_1 par rapport h_2. Ensuite, h_2 est transformée par rapport $IT^*(h_3, h_1)$ car h_1 précède h_2 et les opérations de h_1 ne sont pas dans h_3. Enfin, l'équation (2.5) donne le lien entre IT^* et IT.

En orientant les équations de la définition 2.15 de la gauche vers la droite (comme des règles de réécriture [21]), nous devons montrer deux propriétés importantes à savoir : la *terminaison* et la *confluence*. Ces propriétés vont nous assurer la terminaison du calcul de transformation ainsi que l'unicité du résultat retourné par IT^*.

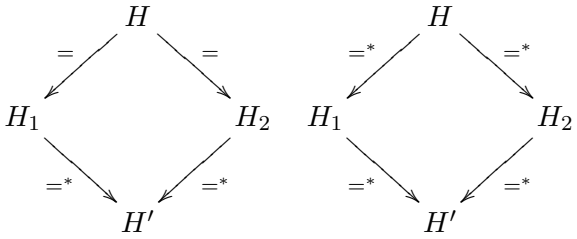
Il faut noter que la preuve de la terminaison n'est pas triviale. En effet, il n'est pas possible d'utiliser un ordre syntaxique [21] (tel que RPO ou LPO) pour ordonner les équations de la gauche vers la droite. C'est pourquoi nous avons utilisé l'outil AProVE [2] qui est basé sur des techniques plus subtiles pour montrer la terminaison [31]. La trace retournée par cet outil est donnée à l'Annexe (voir la page 177).

La figure 2.7 illustre les différentes formes de confluence que l'on peut avoir pour transformer des histoires. Soient H et H' deux histoires :

1. la transformation $H \stackrel{=}{\rightarrow} H'$ signifie qu'une seule équation a été utilisée pour passer de H vers H' ;

[2] `http://aprove.informatik.rwth-aachen.de/`

2. la transformation $H \stackrel{=^*}{\rightarrow} H'$ signifie qu'un ensemble d'équations (qui peut être vide) a été utilisé pour passer de H vers H'.



(a) Confluence Locale (b) Confluence

FIG. 2.7 – Différentes formes de confulence.

Le lemme suivant montre que le système d'équations de la définition 2.15 est localement confluent :

Lemme 2.2 *Le calcul de transformation est localement confluent.* ■

Preuve. Soient h_1, h_2, h_3 et h_4 quatre histoires. Pour ce faire, il est suffisant de montrer que l'utilisation des équations (2.3) ou (2.4) pour calculer $IT^*([h_1; h_2], [h_3; h_4])$ mène au même résultat.

(i) Appliquer d'abord (2.3)
$IT^*([h_1; h_2], [h_3; h_4]) = IT^*(IT^*([h_1; h_2], h_3), h_4])$
$= IT^*([IT^*(h_1, h_3); IT^*(h_2, IT^*(h_3, h_1))], h_4)$
$= [IT^*(IT^*(h_1, h_3), h_4); IT^*(IT^*(h_2, IT^*(h_3, h_1)), IT^*(h_4, IT^*(h_1, h_3)))]$

(ii) Appliquer d'abord (2.4)
$IT^*([h_1; h_2], [h_3; h_4]) = [IT^*(h_1, [h_3; h_4]); IT^*(h_2, IT^*([h_3; h_4], h_1))]$
$= [IT^*(IT^*(h_1, h_3), h_4); IT^*(h_2, IT^*([h_3; h_4], h_1))]$
$= [IT^*(IT^*(h_1, h_3), h_4); IT^*(h_2, [IT^*(h_3, h_1); IT^*(h_4, IT^*(h_1, h_3))])]$
$= [IT^*(IT^*(h_1, h_3), h_4); IT^*(IT^*(h_2, IT^*(h_3, h_1)), IT^*(h_4, IT^*(h_1, h_3)))]$

■

En utilisant la proposition de Newman [63], nous pouvons montrer l'unicité des résultats retournés par notre calcul de transformation.

Théorème 2.1 ([63]) *Si le calcul de transformation se termine et il est localement confluent alors il est confluent.*

Dans ce qui, nous supposons que la fonction IT satisfait $TP1$ et $TP2$. Nous allons montrer que ces conditions peuvent être étendues aux histoires.

Théorème 2.2 *Soient h_1 et h_2 deux histoires concurrentes et légales. Alors nous avons :*

$$[h_1; IT^*(h_2, h_1)] \equiv [h_2; IT^*(h_1, h_2)]$$

■

Preuve. Supposons que $[h_1; IT^*(h_2, h_1)]$ et $[h_2; IT^*(h_1, h_2)]$ sont légales et $|h_1| = |IT^*(h_1, h_2)|$ (resp. $|h_2| = |IT^*(h_2, h_1)|$). Soient n et m les longueurs respectives de h_1 et h_2. Nous procédons par double induction sur n et m.
Base d'induction : Si $n = 0$ ou $m = 0$ le résultat est trivial.
Hypothèse d'induction : pour $n \geq 0$ et $m \geq 0$, $[h_1; IT^*(h_2, h_1)] \equiv [h_2; IT^*(h_1, h_2)]$.
Etape d'induction : Soient $n+1$ et $m+1$ les longueurs respectives deh'_1 et h'_2, où $h'_1 = [op_1; h_1]$ et $h'_2 = [op_2; h_2]$ pour certaines opérations op_1 et op_2. Supposons que h'_1 et h'_2 sont légales. Soient $H_1 = [h'_1; IT^*(h'_2, h'_1)]$ et $H_2 = [h'_2; IT^*(h'_1, h'_2)]$:

$$
\begin{array}{rcl}
H_1 & = & [op_1; h_1; IT^*(op_2; h_2, op_1; h_1)] \\
 & & \text{[par réécriture de } h'_1 \text{ et } h'_2\text{]} \\
 & = & [op_1; h_1; IT^*(IT(op_2, op_1), h_1); \\
 & & \quad IT^*(IT^*(h_2, IT(op_1, op_2)), IT^*(h_1, IT(op_2, op_1)))] \\
 & & \text{[Définition de } IT^*\text{]} \\
 & \equiv & [op_1; IT(op_2, op_1); IT^*(h_1, IT(op_2, op_1)); \\
 & & \quad IT^*(IT^*(h_2, IT(op_1, op_2)), IT^*(h_1, IT(op_2, op_1)))] \\
 & & \text{[Hypothèse d'induction et } TP1\text{]} \\
H_2 & = & [op_2; h_2; IT^*(op_1; h_1, op_2; h_2)] \\
 & & \text{[par réécriture de } h'_1 \text{ et } h'_2\text{]} \\
 & = & [op_2; h_2; IT^*(IT(op_1, op_2), h_2); \\
 & & \quad IT^*(IT^*(h_1, IT(op_2, op_1)), IT^*(h_2, IT(op_1, op_2)))] \\
 & & \text{[Définition de } IT^*\text{]} \\
 & \equiv & [op_2; IT(op_1, op_2); IT^*(h_2, IT(op_1, op_2)); \\
 & & \quad IT^*(IT^*(h_1, IT(op_2, op_1)), IT^*(h_2, IT(op_1, op_2)))] \\
 & & \text{[Hypothèse d'induction et } TP1\text{]}
\end{array}
$$

Nous pouvons conclure que $H_1 \equiv H_2$ en utilisant la condition $TP1$ et l'hypothèse d'induction. ■

Théorème 2.3 *Soient h_1, h_2 et h_3 trois histoires concurrentes et légales. Alors nous avons :*

$$IT^*(h_3, [h_1; IT^*(h_2, h_1)]) = IT^*(h_3, [h_2; IT^*(h_1, h_2)])$$

■

Preuve. Soient n, m et p les longueurs respectives de h_1, h_2 et h_3. Nous procédons par triple induction sur n, m et p.
Base d'induction : Si $n = 0$, $m = 0$ ou $p = 0$ le résultat est trivial.
Hypothèse d'induction : pour $n \geq 0$, $m \geq 0$ et $p \geq 0$ $IT^*(h_3, [h_1; IT^*(h_2, h_1)]) = IT^*(h_3, [h_2; IT^*(h_1, h_2)])$.
Etape d'induction : Soient $n + 1$, $m + 1$ et $p + 1$ les longueurs respectives de h'_1, h'_2 et h'_3, où $h'_1 = [op_1; h_1]$, $h'_2 = [op_2; h_2]$ et $h'_3 = [op_3; h_3]$ pour toutes opérations op_1, op_2 et op_3. Soient $H_1 = IT^*(h'_3, h'_1; IT^*(h'_2, h'_1))$ et $H_2 = IT^*(h'_3, h'_2; IT^*(h'_1, h'_2))$.

En utilisant la définition de IT^* et le théorème 2.2, $H_1 = H'_1; H''_1$ où :

$$
\begin{aligned}
H'_1 &= IT^*(IT(IT(op_3, op_1), IT(op_2, op_1)), [IT^*(h_1, IT(op_2, op_1)); \\
&\qquad IT^*(IT^*(h_2, IT(op_1, op_2)), IT^*(h_1, IT(op_2, op_1)))]) \\
H''_1 &= IT^*(IT^*(h_3, IT(op_1, op_3)); IT(IT(op_2, op_3), IT(op_1, op_3))), \\
&\qquad IT^*(IT^*(h_1, IT(op_2, op_1)), IT(op_3, op_1; IT(op_2, op_1))); \\
&\qquad IT^*(IT^*(IT^*(h_2, IT(op_1, op_2)), IT(op_3, op_1; IT(op_2, op_1))), \\
&\qquad\quad IT^*(IT^*(h_1, IT(op_2, op_1)), IT(op_3, op_1; IT(op_2, op_1)))))
\end{aligned}
$$

De la même manière, en utilisant la définition de IT^* et le théorème 2.2, $H_2 = H'_2; H''_2$ où :

$$
\begin{aligned}
H'_2 &= IT^*(IT(IT(op_3, op_2), IT(op_1, op_2)), IT^*(h_2, IT(op_1, op_2)); \\
&\qquad IT^*(IT^*(h_1, IT(op_2, op_1)), IT^*(h_2, IT(op_1, op_2)))) \\
H''_2 &= IT^*(IT^*(h_3, IT(op_2, op_3)); IT(IT(op_1, op_3), IT(op_2, op_3))), \\
&\qquad IT^*(IT^*(h_2, IT(op_1, op_2)), IT(op_3, op_2; IT(op_1, op_2))); \\
&\qquad IT^*(IT^*(IT^*(h_1, IT(op_2, op_1)), IT(op_3, op_2; IT(op_1, op_2))), \\
&\qquad\quad IT^*(IT^*(h_2, IT(op_1, op_2)), IT(op_3, op_2; IT(op_1, op_2)))))
\end{aligned}
$$

En utilisant la condition $TP2$ et l'hypothèse d'induction, nous pouvons conclure que $H'_1 = H'_2$ et $H''_1 = H''_2$. ■

En utilisant la fonction IT^* et les théorèmes 2.2 et 2.3, nous fournissons une procédure intéressante pour construire des scénarios plus complexes dans des éditeurs collaboratifs basés sur l'approche des transformées opérationnelles.

2.4 Algorithmes d'Intégration

Dans l'approche des transformées opérationnelles, chaque site est équipé de deux composants principaux : *le composant d'intégration* et *le composant de transformation*. Le premier composant se présente comme un algorithme qui est responsable de recevoir, diffuser et exécuter les opérations. Il est *indépendant* de la sémantique des objets collaboratifs. Plusieurs algorithmes d'intégration ont été proposés dans le domaine des éditeurs collaboratifs, comme dOPT [26], adOPTed [73] et SOCT2,4 [81; 89]. Le composant de transformation est un ensemble d'algorithmes de transformation qui est responsable de sérialiser deux opérations concurrentes définies sur le même état. Chaque algorithme de transformation est *spécifique* à la sémantique d'un objet collaboratif.

Chaque site génère séquentiellement les opérations et stocke ces opérations dans un journal d'opérations (ou histoire). Lorsqu'un site reçoit une opération distante op, le composant d'intégration procède par les étapes suivantes :

1. Déterminer dans l'histoire locale h les opérations qui sont concurrentes à op.
2. Utiliser le composant de transformation pour produire op' qui est la forme transformée de op par rapport aux opérations de h qui sont concurrentes à op.
3. Exécuter op' sur l'état courant du site.
4. Ajouter op' à l'histoire locale.

Le composant d'intégration permet donc de construire l'histoire des opérations exécutées sur le site, tout en préservant la relation causale entre les opérations. Lorsque le système est au repos, les histoires des différents sites

ne sont pas forcément identiques mais elles sont équivalentes (elles aboutissent au même état final) malgré que l'ordre d'exécution des opérations concurrentes peut être différent d'une histoire à une autre. Cette équivalence ne peut être assurée que si l'algorithme de transformation utilisé satisfait les conditions $TP1$ et $TP2$.

2.5 Conclusion

Dans ce chapitre, nous avons donné une vue générale du modèle des transformées opérationnelles, qui est utilisé pour sérialiser par transformation des opérations concurrentes. Chaque objet collaboratif doit avoir son propre algorithme de transformation. Pour garantir la convergence des copies, un algorithme de transformation doit satisfaire deux conditions $TP1$ et $TP2$. Aussi, l'écriture d'un algorithme de transformation devient une tâche fastidieuse puisqu'elle nécessite des vérifications qui sont très difficiles (voire même impossibles) à effectuer à la main. D'autre part, cette difficulté varie selon la sémantique de l'objet collaboratif. Ainsi, la transformation pour des objets ayant une structure linéaire (comme la liste ou le texte) demeure toujours un problème ouvert.

Sans une approche formelle [62], la conception des algorithmes de transformation restera une activité ardue et sujette à erreurs, étant donné le nombre important de cas à considérer pour garantir des propriétés aussi critiques que la convergence des copies.

Dans la deuxième partie de cette thèse, nous allons proposer un modèle formel pour l'approche des transformées opérationnelles qui nous permettra de concevoir des algorithmes corrects.

Deuxième partie

Conception Formelle des Transformées Opérationnelles

Chapitre 1

Notions de Base

Sommaire

1.1 Introduction

Nous présentons dans ce chapitre les notions de base et les notations qui vont nous servir pour décrire formellement la première partie de cette thèse. Plus précisément, nous donnons un bref aperçu sur les concepts de l'algèbre universelle, des spécifications conditionnelle et observationnelle, des systèmes de réécriture conditionnelle et du prouveur automatique des théorèmes `SPIKE`. Cependant, pour plus de détails, nous renvoyons le lecteur intéressé aux références suivantes [25], [65], [90], [9], [5], et [74].

1.2 Algèbres

Nous introduisons les principaux éléments des algèbres universelles, qui constituent la base de la réécriture de termes.

1.2.1 Algèbre universelle

Le formalisme des types abstraits algébriques permet de décrire rigoureusement l'univers d'un problème ou le comportement attendu d'un programme. Les algèbres universelles fournissent un cadre formel pour l'étude de ces types

abstraits de données. Ce sont des structures typées munies d'opérations agissant sur ces structures.

Définition 1.1 (S-ensemble) *Un* S-ensemble *est un ensemble* A *muni d'une partition indexée par* S *:* $A = \uplus_{s \in S} A_s$*, où* $\uplus$ *est l'union disjointe d'ensembles.* ■

Une algèbre est constituée d'ensembles et de fonctions sur ces ensembles. L'ensemble des symboles associés à une algèbre est appelé sa signature. La signature d'une algèbre est la *syntaxe* de l'algèbre

Définition 1.2 (Signature) *Une* signature $\Sigma = (S, F)$ *est la donnée d'un ensemble* S *de sortes et d'un ensemble* F *de noms de fonctions. Chaque symbole de fonction* f *est muni d'un profil* $s_1 \times \ldots \times s_n \rightarrow s_{n+1} \in S$*. Le* domaine *de* f *est* $s_1 \times \ldots \times s_n$*. Le* codomaine *de* f *est* s_{n+1}*. Une fonction qui n'a pas de domaine est une* constante*.* ■

L'algèbre donne une sémantique en assignant aux symboles des interprétations.

Définition 1.3 (Σ-algèbres) *Soit* Σ *une signature. Une* Σ-algèbre $\mathcal{A}$ *est un* S*-ensemble* A *appelé* support *de* $\mathcal{A}$*, muni pour chaque symbole de fonction* f *de profil* $s_1 \times \ldots \times s_n \rightarrow s_{n+1}$*, d'une application* $f_A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_{s_{n+1}}$*, appelée* interprétation *de* f*. Dans le cas où* $n = 0$*,* f_A *est simplement un élément de* A*. La classe de toutes les* Σ*-algèbres sera dénotée* $Alg(\Sigma)$*.* ■

Par abus de notation, on notera une Σ-algèbre par A.

Un homomorphisme entre deux algèbres est une application entre les ensembles support qui préservent les fonctions.

Définition 1.4 (Homomorphisme) *Soient* Σ *une signature et* A*,* B *deux* Σ*-algèbres. Un* homomorphisme $h : A \rightarrow B$ *est une application entre les supports de* A *et* B *telle que : pour tout* n*-uplet* $(a_1, \ldots, a_n) \in A_{s_1} \times \ldots \times A_{s_n}$*,* $h(f_A(a_1, \ldots, a_n)) = f_B(h(a_1), \ldots, h(a_n))$*.* ■

Un *isomorphisme* est un homomorphisme bijectif.

Un morphisme de signatures définit une application des sortes et des noms de fonction d'une signature vers ceux d'une autre signature.

Définition 1.5 (Morphisme) *Soient* $\Sigma = (S, F)$ *et* $\Sigma' = (S', F')$ *deux signatures. Un* morphisme de signatures $\Phi : \Sigma \rightarrow \Sigma'$ *est une paire* $((\sigma_{sorte}, \sigma_{fonc})$*, tels que* $\sigma_{sorte} : S \rightarrow S'$ *et* $\sigma_{fonc} = (F_{\omega,s} \rightarrow F'_{\sigma^*_{sorte}(\omega),\sigma(s)})_{\omega \in S^*, s \in S}$*, où pour* $\omega = s_1 \ldots s_n \in S^*$ $\sigma^*_{sorte}(\omega) = \sigma_{sorte}(s_1) \ldots \sigma_{sorte}(s_n)$*. Nous utilisons* $\Phi(s)$ *pour* $\sigma_{sorte}(s)$ *et* $\Phi(f)$ *pour* $\sigma_{fonc}(f)$ *tel que* $f \in F_{s_1 \ldots s_n, s}$*.* ■

Définition 1.6 (Restriction d'une algèbre) *Etant donnés un morphisme de signature* $\Phi : \Sigma \rightarrow \Sigma'$ *et une* Σ'*-algèbre* A'*. La* restriction *de* A' *à* Σ*, notée* $\Phi(A')$*, est définie par les supports* $A'_{\Phi(s)}$ *pour* $s \in S$ *et les fonctions* $f_{\Phi(s)}$ *pour* $f \in F_{s_1 \ldots s_n, s}$*.* ■

On distingue deux Σ-algèbres : l'algèbre initiale et l'algèbre finale.

Définition 1.7 (Algèbre initiale) *Une Σ-algèbre $A \in Alg(\Sigma)$ est dite* initiale *si pour toute Σ-algèbre $B \in Alg(\Sigma)$ il existe un unique homomorphisme $h : A \rightarrow B$.* ■

Définition 1.8 (Algèbre finale) *Une Σ-algèbre $A \in Alg(\Sigma)$ est dite* finale *si pour toute Σ-algèbre $B \in Alg(\Sigma)$ il existe un unique homomorphisme $h : B \rightarrow A$.* ■

L'algèbre triviale qui est une algèbre dont le support est constitué d'un élément est une algèbre finale. Dans cette algèbre, on considère que deux éléments sont égaux à moins qu'on puisse prouver qu'ils sont différents. Les algèbres initiales et finales n'existent pas toujours, mais si elles existent, alors elles sont uniques à un isomorphisme unique près.

1.2.2 Termes

Soit une signature Σ. Une ensemble de variables est un S-ensemble X ne contenant pas de noms de fonctions de F. Les éléments de X sont appelées *variables*. Chacune des variables de X possède une et une seule sorte. On note par $\equiv$ l'égalité syntaxique de deux termes.

Définition 1.9 (Ensemble des termes avec variables) *Soit Σ une signature. L'ensemble des termes avec variables noté $T(F, X)$ est défini comme le plus petit ensemble tel que :*

(i) si x est une variable de sorte s alors $x \in T(F, X)_s$;

(ii) si f est une constante de sorte s alors $f \in T(F, X)_s$;

(iii) pour tout $f \in F_{s_1 \ldots s_n, s}$, pour tout $(t_1, \ldots, t_n) \in (T(F, X)_{s_1} \times \ldots T(F, X)_{s_n})$, $f(t_1, \ldots, t_n) \in T(F, X)_s$.

$T(F, X)$ est appelée l'algèbre des termes. *Si $X = \emptyset$ alors $T(F)$ est appelée* l'algèbre des termes clos. ■

Définition 1.10 (Substitution) *Une* substitution *est une application d'un ensemble de variables X dans $T(F, X)$. Une substitution de X dans $T(F)$ est appelée* substitution close. *L'application d'une substitution σ à un terme est notée par $t\sigma$. Le domaine de σ est $\{x \mid x\sigma \neq x\}$.* ■

Une substitution σ se prolonge de façon unique en un homomorphisme de $T(F, X)$ vers lui-même, noté également σ. Il vérifie donc pour tout f de F et pour tout t_1, …, t_n de $T(F, X)$:

$$(f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$$

L'algèbre des termes clos possède la propriété suivante :

Proposition 1.1 ([5]) *L'algèbre $T(F)$ est une Σ-algèbre initiale.* ■

1.3 Spécification conditionnelles

L'avantage des spécifications conditionnelles est qu'elles sont assez expressives. En effet, elles permettent de définir des types de données de manière plus souple et plus naturelle que dans le cadre équationnel. Les équations conditionnelles sont des équations ayant en plus une *précondition* qui détermine la condition d'égalité de deux termes.

Définition 1.11 (Equation, Equation conditionnelle) *Soit Σ une signature. Une* Σ-équation *est une expression de la forme $t_1 = t_2$ où t_1, t_2 sont deux termes de même sorte. Une Σ-équation* conditionnelle *est de la forme $\bigwedge_{i=1}^{n} a_i = b_i \implies l = r$, où a_i, $b_i \in T(F,X)_{s_i}$. Dans ce cas, $l = r$ est appelée la* conclusion *et $\bigwedge_{i=1}^{n} a_i = b_i$ est appelée la* précondition. ■

On dira équation à la place de Σ-équation quand il n'y a pas d'ambiguïté.

Définition 1.12 (Spécification conditionnelle) *Une* spécification conditionnelle *SP est un couple (Σ, E), où Σ est une signature et E est un ensemble d'axiomes exprimés sous forme d'équations conditionnelles. Elle est communément appelée* spécification algébrique. ■

Définition 1.13 (Assignation) *Soit A une algèbre. Une* assignation *est un homomorphisme θ de X dans A. L'application de l'assignation θ à un terme t donne le terme $t\theta$, défini comme suit :*

- *si t est une constante c, $t\theta = c_A$;*
- *si t est une variable x, $t\theta$ est l'image de t par θ ;*
- *si $t = f(t_1, \ldots, t_n)$ alors $t\theta = f_A(t_1\theta, \ldots, t_n\theta)$.*

■

Toute assignation peut être étendue à l'ensemble des termes :

Proposition 1.2 ([74]) *Etant donné une Σ-algèbre A, alors toute assignation $\theta : X \to A$ peut être uniquement étendue à un homomorphisme $\theta^* : T(F,X) \to A$.* ■

Définition 1.14 (Satisfaction) *Soit A une Σ-algèbre et $C \equiv \bigwedge_{i=1}^{n} a_i = b_i \implies l = r$. On dit que A* satisfait *C, ou C est* valide *dans A, ssi : pour toute assignation θ, si (pour tout $i \in [1..n]$, $a_i\theta = b_i\theta$ alors $l\theta = r\theta$. Cette satisfaction est notée $A \models^{\Sigma} C$ (ou s'il n'y a pas d'ambiguïté $A \models C$).* ■

Etant donnés un morphisme de signature $\Phi : \Sigma \to \Sigma'$ et une Σ-équation e de la forme $l = r$. La translation de e par Φ, notée $\Phi(e)$, est une Σ'-équation $\overline{\Phi}(l) = \overline{\Phi}(r)$, où $\overline{\Phi} : T(F,X) \to T(F',X')$ et $X' = \Phi(X)$. Une importante propriété de ces translations sur les algèbres et les équations est appelée *condition de satisfaction*, qui exprime l'invariance de la satisfaction par les changements de notation

Théorème 1.1 (Condition de satisfaction) *Etant donnés un morphisme* $\Phi : \Sigma \rightarrow \Sigma'$*, une* Σ'*-algèbre* A' *et une* Σ*-équation conditionnelle* C*. Alors, nous avons :*

$$\Phi(A') \models^{\Sigma} C \text{ ssi } A' \models^{\Sigma'} \Phi(C)$$

■

La preuve de ce théorème est donnée dans [74].

Définition 1.15 (Modèle) *Une algèbre* A *est un* modèle *d'une spécification algébrique* $SP = (\Sigma, E)$ *ssi* A *est une* Σ*-algèbre qui satisfait toutes les équations conditionnelles de* E*.* ■

La *théorie équationnelle* de la spécification $SP = (\Sigma, E)$, notée par $Mod(SP)$, est constituée par toutes les Σ-algèbres qui satisfont tous les axiomes de E. Soit C est une équation conditionnelle. On écrit $Mod(SP) \models C$ ou bien $E \models C$, si C est valide dans toutes les algèbres de $Mod(SP)$. Si $Mod(SP)$ est non vide, on dit alors que E est *cohérente*, sinon, E est *incohérente.*

1.3.1 Sémantique initiale

Pour prouver la validité des propriétés, nous nous plaçons dans l'algèbre initiale. Cette algèbre apparaît comme "le plus petit" modèle d'une spécification dans le sens qu'elle satisfait seulement les équations valides dans toutes les modèles de cette spécification. Dans ce qui suit, nous allons introduire les notions nécessaires pour définir la sémantique initiale.

Définition 1.16 (Homomorphisme d'évaluation) *Soit* A *une* Σ*-algèbre. L'homomorphisme* $eval_A : T(F) \rightarrow A$ *est défini comme suit :*

- $eval_A(f) = f_A$ *si* f *est une constante ;*
- $eval_A(f(t_1, \ldots, t_n)) = f_A(eval_A(t_1), \ldots, eval_A(t_n))$ *si* f *n'est pas une constante.*

■

Si t est un terme de $T(F)$, alors on note par t_A son image dans A par $eval_A$. Une algèbre *finiment engendrée* est une algèbre où chaque valeur d'élément est le résultat d'évaluation d'un terme clos. Ainsi, $A_s = t_A | t \in T(F)_s$ pour $s \in S$.

Définition 1.17 (Algèbre finiment engendrée) *Une* Σ*-algèbre* A *est une algèbre* finiment engendrée *si* $eval_A$ *est surjectif.* ■

Définition 1.18 (Congruence) *Une* congruence *sur une* Σ*-algèbre* A *est une relation d'équivalence* $\sim$ *sur* A *vérifiant pour tout* f *de profil* $s_1 \times \ldots \times s_n \rightarrow s$*, pour tout* a_1*,* b_1*, ...,* a_n*,* b_n *dans* A *:*

$$a_1 \sim b_1 \wedge \ldots \wedge a_n \sim b_n \implies f_A(a_1, \ldots, a_n) = f_A(b_1, \ldots, b_n)$$

■

Définition 1.19 (Algèbre quotient des termes) *Si $\sim$ est une congruence sur A, l'ensemble* quotient *noté $A/\sim$, est un ensemble muni d'une structure d'algèbre tel que pour tout f de profil $s_1 \times \ldots \times s_n \rightarrow s$ dans F, pour tout a_1, …, $a_n \in A$,*

$$f_{A/\sim}([a_1]_\sim, \ldots, [a_n]_\sim) = [f_A(a_1, \ldots, a_n)]_\sim$$

où pour tout $i \in [1..n]$, $[a_i]_\sim$ est la classe d'équivalence de a_i dans A. ■

Soit $SP = (\Sigma, E)$ une spécification algébrique. Si E est cohérente, alors l'algèbre quotient de l'algèbre des termes clos par la congruence $\sim_E$, notée $T(F)/\sim_E$ est un modèle dans $Mod(SP)$. De plus, $T(F)/\sim_E$ comme le montre le théorème suivant de [25].

Théorème 1.2 *Si E est cohérente alors $T(F)/\sim_E$ est initiale dans $Mod(SP)$.* ■

Nous noterons $T(F)/\sim_E$ par $Ini(SP)$.

Définition 1.20 (Conséquence inductive) *On dit qu'une équation conditionnelle C est une* propriété *ou* conséquence inductive*, si elle est valide dans $Ini(SP)$. Dans ce cas, on la note $Ini(SP) \models C$ ou bien $E \models_{ind} C$.* ■

Le théorème suivant de [65], donne une caractérisation opérationnelle des propriétés inductives.

Théorème 1.3 *Soit $C \equiv \bigwedge_{i=1}^{n} a_i = b_i \implies l = r$. Alors $Ini(SP) \models C$ ssi pour toute substitution close σ :*

$$(\textit{pour tout } i \in [1..n] : E \models a_i\sigma = b_i\sigma \textit{ alors } E \models l\sigma = r\sigma.$$

■

1.3.2 Spécification observationnelle

Une spécification observationnelle est une spécification conditionnelle, dans laquelle l'ensemble des sortes est scindé en sortes visibles et sortes cachées (ou non observables). Elle est définie comme suit :

Définition 1.21 (spécification observationnelle) *Une* spécification observationnelle *SP_{obs} est un couple (SP, S_{obs}) tel que $SP = (\Sigma, E)$ est une spécification conditionnelle et S_{obs} est l'ensemble des* sortes observables. ■

Le passage d'une spécification conditionnelle à une spécification observationnelle se fait en précisant les sortes à observer. Les spécifications observationnelles s'avèrent appropriées pour modéliser le comportement d'un système et notamment celui qui est articulé autour du modèle objet. En effet, la sorte cachée représente l'espace des états de l'objet et les sortes observables sont utilisées pour représenter les attributs de l'objet. La figure 1.1 donne un exemple de spécification observationnelle d'une pile, la sorte à observer est celle des entiers naturels.

```
Spécification: Pile

sortes: nat pile

sorte observable: nat

fonction définies:
0 :-> nat
Nil :-> pile
push: nat  pile ->  pile
top: pile ->  nat
pop: pile -> pile

axiomes:
top(push(i,p))=i;
top(Nil)=0;
pop(Nil)=Nil;
pop(push(i,p))=p;
```

FIG. 1.1 – Spécification observationnelle d'une pile.

Définition 1.22 (contexte) *Soit $T(F, X)$ une algèbre de termes, et Σ sa signature.*

- un contexte *sur F est un terme non clos $c \in T(F, X)$ où une variable est distinguée et appelée variable contextuelle de c. Pour indiquer la variable contextuelle z_s apparaissant dans c, on note souvent $c[z_s]$ au lieu de c, où s est la sorte de z_s.*
- *un contexte réduit à une variable z_s de sorte s est appelé* contexte vide *de sorte s.*
- *l'application d'un contexte $c[z_s]$ à un terme $t \in T(F, X)$ de sorte s, noté par $c[t]$, est défini par la substitution de z_s par t dans $c[z_s]$. Le contexte c est dit* applicable *à t.* ■

Définition 1.23 (contexte observable, terme, équation, précondition) *Un terme observable est un terme dont la sorte appartient à S_{obs}. L'ensemble des contextes observables est noté par C_{obs}. Une équation $a = b$ est observable si a et b sont observables. La précondition d'une règle est observable si toutes ses équations sont observables.* ■

Exemple 1.1 *Prenons par exemple la spécification de la pile donnée par la figure 1.1, elle admet une infinité de contexte observable :*

$top(z_{pile}), top(pop(z_{pile})), \ldots, top(pop(\ldots(pop(z_{pile}))\ldots)), \ldots$

$top(push(i, z_{pile})), top(push(i, pop(z_{pile}))), \ldots$

La notion de validité observationnelle repose sur l'idée de base que deux objets d'une algèbre donnée sont *observationnellement égaux* s'ils ne peuvent

être distingués par des calculs à résultats observables. Ces calculs sont formalisés par les contextes observables.

Définition 1.24 (égalité observable dans une algèbre) *Soit A une algèbre, et a, b deux éléments de A. On dit que a et b sont observationnellement égaux dans A, et on le note par $A \models_{obs} a = b$, ssi pour tout $c_{obs}[z_s] \in C_{obs}$, pour toutes assignations θ_a, θ_b telles que $z_s\theta_a = a$, $z_s\theta_b = b$, et pour tout $x \in X \setminus \{z_s\}$, $x\theta_a = x\theta_b$, on a : $A \models c_{obs}\theta_a = c_{obs}\theta_b$.* ■

Définition 1.25 (satisfaction observable) *Soit A une algèbre. Soit C une clause. On dit que C est observationnellement valide dans A et on le note par $A \models_{obs} C$, ssi : Pour toute assignation θ, si (pour tout $i \in [1..n], A \models_{obs} a_i\theta = b_i\theta$) alors (il existe $j \in [1..m]$, $A \models_{obs} a'_j\theta = b'_j\theta$)* ■

Définition 1.26 ($=_{obs}$) *Soit a et b deux termes. On dit que a et b sont observationnellement égaux, et on le note par $E \models_{obs} a = b$ ou par $a =_{obs} b$ ssi pour tout $c \in \mathcal{C}_{obs}, E \models_{ind} c[a] = c[b]$.* ■

Exemple 1.2 *Prenons la spécification de la pile donnée par la figure 1.1, la propriété $push(top(s), pop(s)) = s$ n'est pas satisfaite dans le sens classique car l'instance close $push(top(Nil), pop(Nil)) = Nil$ n'est pas valide. Mais, intuitivement, elle est observationnellement satisfaite si on observe seulement les éléments des séquences $push(top(s), pop(s))$ et s. Ceci est formellement démontré en considérant tous les contextes observables.* ■

1.4 Réécriture

Un ensemble E d'équations conditionnelles définit une congruence sur les termes. Ainsi deux termes sont congrus si on peut passer de l'un à l'autre par une succession de remplacements utilisant les axiomes de E. Cependant, cette congruence engendre des calculs indéterministes car il existe deux possibilités d'utiliser une équation conditionnelle. Afin d'orienter les équations conditionnelles, et pouvoir les utiliser dans un seul sens, on a recours à des ordres bien fondés sur les termes [47; 20; 21].

Une règle de réécriture est une équation conditionnelle dont la conclusion est orientée de la gauche vers la droite. Soit $\mathcal{R}$ un ensemble de règles conditionnelles. On associe à $\mathcal{R}$ une relation binaire sur les termes notée $\rightarrow_R$ ($\rightarrow_R^*$ est la fermeture transitive de $\rightarrow_R$).

Appliquer une règle de réécriture sur un terme t revient à remplacer l'instance du membre gauche dans t, par l'instance correspondante du membre droit, à condition que la précondition soit vérifiée. Cependant, l'évaluation de la précondition peut mener à des dérivation infinies, comme le montre l'exemple suivant :

Exemple 1.3 *Soit l'équation conditionnelle suivante :*

$$p(x) > 0 = true \implies x > 0 = true$$

où p désigne la fonction prédécesseur *sur les entiers. L'évaluation de l'équation* $0 > 0$ *donne une infinité de dérivations :*

$$0 > 0 \rightarrow_R true \ si \ -1 > 0 \rightarrow_R^* true$$
$$-1 > 0 \rightarrow_R true \ si \ -2 > 0 \rightarrow_R^* true$$
$$\ldots$$

■

Pour éviter ce type de dérivation cyclique, on impose des restrictions sur les règles.

Définition 1.27 (Règle de réécriture conditionnelle) *Soit* $\succ$ *un ordre bien fondé sur les termes. Une équation conditionnelle* $\bigwedge_{i=1}^{n} a_i = b_i \implies l = r$ *est dite règle de réécriture conditionnelle, notée par* $\bigwedge_{i=1}^{n} a_i = b_i \implies l \rightarrow r$, *si toute variable de* $\bigwedge_{i=1}^{n} a_i = b_i$ *et de* r *est une variable de* l, *et pour toute substitution* σ, *on a* $\{l\sigma\} \gg \{l\sigma, a_1\sigma, b_1\sigma, \ldots, a_n\sigma, b_n\sigma\}$ *où* $\gg$ *est l'extension multi-ensemble de* $\succ$. *Le terme* l *est appelé membre gauche de la règle.* ■

Les objets d'un type abstrait algébrique sont souvent définis à partir de fonctions de base, appelées *constructeurs*. L'ensemble F est donc partitionné en un ensemble C de constructeurs, et un ensemble D de *fonctions définies* à partir de ces constructeurs.

Définition 1.28 (Système de réécriture structuré) *Un système de réécriture* $\mathcal{R}$ *est* structuré *ssi pour toute règle de réécriture* $\bigwedge_{i=1}^{n} a_i = b_i \implies l = r$ *dans* $\mathcal{R}$, *si* $l \in T(C, X)$ *alors* $r \in T(C, X)$. ■

Définition 1.29 (Réécriture des termes) *Soit* $\mathcal{R}$ *un système de réécriture conditionnelle. La relation de réécriture conditionnelle est définie comme étant la plus petite relation réflexive et transitive* $\rightarrow_R$ *telle que pour tout terme* s *et* t, *tout sous-terme* u *de* s, *toute substitution* σ *et toute règle* $\bigwedge_{i=1}^{n} a_i = b_i \implies l = r \in \mathcal{R}$:
si $u = l\sigma$, $t = s[u \leftarrow r\sigma]$ *et il existe un terme* c *tels que* $a_i\sigma \rightarrow_R^* c$ $b_i\sigma \rightarrow_R^* c$ *alors* $s \rightarrow_R t$. ■

Définition 1.30 (Forme normale) *Un terme* t *est dit* irréductible *ou en* forme normale *s'il n'existe pas de terme* t' *tel que* $t \rightarrow_R t'$. ■

Définition 1.31 (Confluence, confluence sur les termes clos) *La relation* $\rightarrow_R$ *est* confluente *(resp. confluente sur les termes clos) ssi pour tous termes* t, t_1, t_2 *dans* $T(F, X)$ *(resp.* $T(F)$*), tels que* $t \rightarrow_R^* t_1$ *et* $t \rightarrow_R^* t_2$, *il existe* t' *dans* $T(F, X)$ *(resp.* $T(F)$*) tel que* $t_1 \rightarrow_R^* t'$ *et* $t_2 \rightarrow_R^* t'$. ■

Si la relation $\rightarrow_R$ est bien fondée et confluente (resp. confluente sur les termes clos), on dit que $\rightarrow_R$ est *convergente* (resp. convergente sur les termes clos). Pour vérifier la propriété de confluence sur les termes clos d'un système de réécriture conditionnelle, des techniques sont proposées dans [48]. Si $\rightarrow_R$ est convergente alors tout terme a une forme normale.

Définition 1.32 (Fonction complètement définie) *Une fonction* $f \in F_{s_1 \ldots s_n, s}$ *est complètement définie par rapport aux constructeurs si pour toute substitution close* σ*, il existe* $t' \in T(C)$ *tel que* $f(x_1, \ldots, x_n) \rightarrow_R^* t'$. ■

Définition 1.33 (Système de réécriture suffisamment complet) *Un système de réécriture est* suffisamment complet *si pour toutes ses fonctions de D sont complètement définies par rapport aux constructeurs.* ■

La complétude suffisante des spécifications conditionnelles est indécidable. Cependant, une technique de test est proposée dans [10].

1.5 SPIKE

SPIKE est prouveur automatique de théorèmes dont la méthode de preuve est basée sur l'induction par ensembles couvrants ainsi que sur d'autres techniques de raisonnement telles que la réécriture conditionnelle et l'analyse par cas [8; 3]. Les principales fonctionnalités de SPIKE sont :

- Test de complétude suffisante d'une spécification conditionnelle.
- Preuve par réfutation d'une conjecture.
- Preuve par induction : Ce processus commence d'abord par instancier les variables d'induction de la conjecture à prouver par les éléments de même sorte de l'ensemble test, ensuite réécrit et simplifie le résultat.
- Utilisation des procédures de décision pour éliminer automatiquement les tautologies arithmétiques [3].

SPIKE a été appliqué avec succès à des études de cas complexes telles que la preuve de conformité du protocole ABR (Available Bit Rate [75] ainsi que la preuve de la correction du bytecode JavaCard [4].

Exemple 1.4 (Commutativité de l'addition) *Considérons la spécification suivante :*

```
Constructeurs
0 :-> nat
s: nat -> nat

Fonctions définies
+ : nat nat  -> nat

Axiomes
0 + x = x;
s(x) + y = s(x+y);

Conjectures
x + y = y + x;
```

D'abord SPIKE *oriente les axiomes en règles de réécriture. La conjecture à prouver est mise dans l'ensemble* E_0*,* H_0 *étant vide. Le système procède donc comme suit :*

1. *Etat initial :* $\mathcal{E}_0 = \{x + y = y + x\},\ \mathcal{H}_0 = \emptyset$.
2. *Schéma d'induction*
   - *x est une variable d'induction.*
   - *L'ensemble test* $TS = \{0,\ s(x)\}$ *(définition récursive des entiers naturels).*
3. *Génération :*
   - *En remplaçant x par* 0 *on obtient :* $0 + y = y + 0$ (e_1).
   - *En remplaçant x par* $s(z)$ *on obtient :* $s(z) + y = y + s(z)$ (e_2).

   $\mathcal{E}_1 = \{0 + y = y + 0;\ s(z) + y = y + s(z)\},\ \mathcal{H}_1 = \{x + y = y + x\}$
4. *Simplification :*
   - *Simplification de* (e_1) *par le premier axiome :* $y = y + 0$ (e'_1).
   - *Simplification de* (e_2) *par le deuxième axiome :* $s(z+y) = y+s(z)$ (e'_2).
5. *SPIKE applique de nouveau la règle de génération pour prouver les équations* (e'_1) *et* (e'_2).

Le système manipule les théories dont les axiomes sont des équations conditionnelles du premier ordre. La signature est supposée typée et manipule des sortes. Elle est partitionnée en deux sous ensembles : les constructeurs et les fonctions définies. SPIKE dispose d'un langage de stratégie qui oriente la priorité des applications des règles, il fournit aussi la possibilité d'interagir avec l'utilisateur lors du déroulement de la preuve.

Chapitre 2

Spécification et Vérification des Objets Collaboratifs

Sommaire

2.1 Introduction

Un objet collaboratif est conçu pour être partagé par un groupe de personnes qui sont réparties dans le temps et l'espace. Ce partage est rendu à la fois convivial et performant grâce à la réplication. En effet, chaque utilisateur modifie une copie de l'objet collaboratif et transmet ses modifications aux autres membres du groupe. Ainsi, les modifications peuvent être réalisées dans différents ordres et sur les différentes copies ; ce qui mène potentiellement vers une divergence de copies (copies différentes).

A cet effet, l'approche des transformées a été proposée pour éviter la divergence des copies [26; 73; 85]. L'idée de base de cette approche est d'utiliser

un algorithme pour transformer une modification (provenant d'un autre utilisateur) par rapport aux modifications exécutées localement. Les transformées opérationnelles ont été utilisées dans plusieurs éditeurs collaboratifs [26; 73; 85; 81; 89; 86], et plus récemment dans un synchroniseur de fichiers [60]. Cependant, la convergence ne peut être assurée que si l'algorithme de transformation vérifie les deux conditions $TP1$ et $TP2$ [69; 73; 82; 84]. Ainsi, il est crucial de vérifier de tels algorithmes pour avoir plus de confiance dans l'approche.

Tout au long de ces dix sept dernières années, on a montré que les algorithmes de transformation publiés pour manipuler des chaînes de caractères sont faux car ils violent $TP1$ et/ou $TP2$ [39]. Ceci implique que toutes les propositions qui ont été faites pour d'autres structures de données linéaires sont fausses, comme c'est la cas de l'algorithme de transformation développé pour réconcilier un document XML modélisé sous forme d'un arbre ordonné [17].

La conception d'un algorithme de transformation n'est pas une tâche facile au regard de la satisfaction de $TP1$ et $TP2$. En effet, la preuve à la main de ces conditions est très difficile – voire impossible – vu le problème de l'explosion combinatoire des cas à considérer. Au fur et à mesure que le nombre d'opérations permettant de modifier l'état de l'objet collaboratif augmente, le nombre de triplets d'opérations concurrentes à considérer pour vérifier $TP2$ devient de plus en plus important. Et plus la structure de l'objet 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 $TP2$ requiert le traitement de huit triplets d'opérations. Pour chaque triplet, il faut prendre en compte les relations possibles entre les paramètres des opérations ; ce qui donne 13 cas pour notre exemple. Par conséquent, la preuve de $TP2$ pour les chaînes de caractères nécessite le traitement de 104 cas.

Pour pallier ce problème, il est nécessaire que l'approche des transformées opérationnelles soit basée sur l'utilisation des méthodes formelles. Le développeur d'un algorithme de transformation peut écrire une spécification formelle pour décrire le comportement de l'objet collaboratif et ensuite il utilise un prouveur automatique de théorèmes pour vérifier $TP1$ et $TP2$. Cependant, l'usage effectif d'un prouveur nécessite typiquement une expertise qui est rare chez les développeurs de logiciels. Aussi, dans ce chapitre, nous allons présenter un environnement pour la conception des algorithmes de transformation qui permet à la fois : (i) une écriture facile des spécifications formelles ; et, (ii) un haut degré d'automatisation pour le processus de preuve.

En utilisant une sémantique observationnelle, nous considérons un objet collaboratif comme une boîte noire [33]. Nous spécifions seulement les interactions entre l'utilisateur et l'objet. Nous avons implémenté un outil qui permet au développeur de définir dans un langage algorithmique les opérations ainsi que l'algorithme de transformation. A partir de cette description, notre outil génère une spécification algébrique exprimée en termes d'équations conditionnelles. Comme moyen de vérification, nous utilisons `SPIKE`, un prouveur automatique basé sur l'induction et la réécriture ; ce prouveur est approprié pour raisonner sur des théories conditionnelles.

La structure du chapitre se décline comme suit : les ingrédients de la spé-

cification formelle des objets collaboratifs est donnée à la section 2.2. Dans la section 2.3, nous allons présenter comment exprimer formellement les conditions $TP1$ et $TP2$. La section 2.4 décrit les techniques utilisées pour vérifier $TP1$ et $TP2$ et construire des scénarios en cas de violation. La section 2.5 donne une description de l'outil `VOTE` qui aide à la conception des algorithmes de transformation. Enfin, nous terminons le chapitre par une conclusion.

2.2 Spécification d'Objets Collaboratifs

L'objet collaboratif est la brique de base dans les éditeurs collaboratifs. Chaque objet possède un ensemble d'opérations primitives. Les *méthodes* sont les opérations qui modifient l'état de l'objet. Les *attributs* sont les opérations qui extraient des informations à partir de l'état de l'objet. En fait, nous observons l'état à travers les attributs.

Nous modélisons l'objet collaboratif comme une boîte noire, en ce sens où nous ne tenons pas compte de la représentation interne de son état. A ce titre, nous utilisons une technique observationnelle qui dissimule l'état interne en extrayant seulement les informations pertinentes à partir des méthodes exécutées.

Nous représentons l'espace d'états d'un objet collaboratif par la sorte `State`. Les états sont modifiés par les méthodes dont l'ensemble est décrit par la sorte `Meth`. Pour passer d'un état à un autre, nous définissons une fonction de transition *Do*. Pour chaque méthode, nous devons connaître les conditions d'application sur un état donné. Pour ce faire, nous utilisons une fonction booléenne *Poss* qui donne pour chaque méthode sa précondition. Quant à l'algorithme de transformation, nous le dénotons par la fonction IT qui prend en arguments deux méthodes pour produire une autre méthode.

Définition 2.1 (OC-signature) *Soient S l'ensemble de toutes les sortes, $S_b = \{\texttt{State}, \texttt{Meth}\}$ l'ensemble des* sortes de base *et $S_d = S \setminus S_b$ l'ensemble des* sortes données. *Une* OC-signature $\Sigma = (S, S_{obs}, F)$ *est une signature observationnelle où `State` est la seule sorte non observable. L'ensemble des noms d'opérations F est défini comme suit :*

- $F_{\texttt{Meth State, State}} = \{Do\}$, $F_{\texttt{Meth Meth, Meth}} = \{IT\}$, $F_{\texttt{Meth State, Bool}} = \{Poss\}$, *et $F_{\omega,s} = \emptyset$ pour tous les autres cas où $\omega \in S_b^*$ et $s \in S_b$;*
- *l'opération $f : s_1 \times s_2 \times \ldots \times s_n \rightarrow \texttt{Meth}$ est appelée une* méthode *ssi $s_1 \cdot s_2 \cdot \ldots \cdot s_n \in S_d^*$;*
- *l'opération $f : s_1 \times s_2 \times \ldots \times s_n \rightarrow s$ est appelée un* attribut *ssi $s_1 \cdot s_2 \cdot \ldots \cdot s_n$ contient une seule sorte `State` et $s \in S_d$.*

La restriction d'une OC-signature Σ à des sortes observables est notée Σ^d. Nous utilisons Σ, Σ', Σ_1, Σ_2, ..., pour désigner des OC-signatures. ■

Exemple 2.1 (Cellule de mémoire) *Considérons le cas d'une cellule de mémoire qui peut contenir une valeur d'une certaine sorte, que nous désignons par `Elt`. La sorte `State` dénotera les états possible de la mémoire. La OC-signature de cette mémoire se présente comme suit :*

```
spec CELL =
sort:
  Elt Meth State
opns:
  Do : Meth State -> State
  put : Elt -> Meth
  get : State -> Elt
  IT : Meth Meth -> Meth
```

La méthode `put(e,s)` *permet de mettre une donnée* `e`*, alors que l'attribut* `get(s)` *extrait de l'état* `s` *l'élément courant. Cette cellule de mémoire peut être considérée comme une "boîte noire" dans le sens où sa structure interne (ou l'implémentation) est cachée.* ■

Comme modèle sémantique, chaque OC-signature peut avoir une ou plusieurs algèbres.

Définition 2.2 (Implémentation) *Une* Σ-implémentation $\mathcal{I}$ *est une* Σ-*algèbre pour une OC-signature* Σ*.*

Exemple 2.2 *Considérons la spécification d'une cellule de mémoire donnée à l'exemple 2.1. Nous donnons les deux* Σ*-implémentations suivantes (soit* $\mathcal{N}$ *l'ensemble des entiers naturels) :*

1. *La cellule de mémoire possède une case où seule la dernière valeur stockée est mémorisée.*

$$\begin{array}{lcl}
\mathcal{I}_{\texttt{Elt}} & = & \mathcal{N} \\
\mathcal{I}_{\texttt{Meth}} & = & \{put(n) \mid n \in \mathcal{N}\} \\
\mathcal{I}_{\texttt{State}} & = & \{[n] \mid n \in \mathcal{N}\} \\
\mathcal{I}_{\texttt{Do}}(put(n), [n']) & = & [n] \\
\mathcal{I}_{\texttt{get}}([n]) & = & n \\
\mathcal{I}_{\texttt{IT}}(put(n), put(n')) & = & put(n).
\end{array}$$

2. *La cellule de mémoire garde l'histoire des opérations qui l'ont affecté. Nous représentons cette histoire sous forme d'une liste* $[head, tail]$ *où* $head$ *est la tête de liste, et* $tail$ *est la queue de liste. La notation* $[]$ *représente la liste vide.*

$$\begin{array}{lcl}
\mathcal{I}_{\texttt{Elt}} & = & \mathcal{N} \\
\mathcal{I}_{\texttt{Meth}} & = & \{put(n) \mid n \in \mathcal{N}\} \\
\mathcal{I}_{\texttt{State}} & = & \mathcal{I}^*_{\texttt{Meth}} \\
\mathcal{I}_{\texttt{Do}}(m, l) & = & [m, l] \\
\mathcal{I}_{\texttt{get}}([]) & = & 0 \\
\mathcal{I}_{\texttt{get}}([put(n), l]) & = & n \\
\mathcal{I}_{\texttt{IT}}(put(n), put(n')) & = & put(n).
\end{array}$$

■

Comme nous utilisons une sémantique observationnelle, la notion de *contexte* doit être définie puisqu'elle est fondamentale dans toutes les sémantiques basées

sur la validation observationnelle. Une propriété observationnelle est acquise en ne prenant en considération que des informations observables. Aussi, pour montrer qu'elle est valide, on doit montrer sa validité dans tous *contextes observables* [7].

Définition 2.3 (contextes) *Soient Σ une OC-signature et $T_\Sigma(F, X)$ une algèbre de termes.*

- *un* Σ-contexte *est un terme non clos $c \in T_\Sigma(F, X)$ contenant une variable linéaire $z_{\texttt{State}}$ de sorte* `State`*. Cette variable est appelée* variable d'état *de c. Pour indiquer la variable d'état $z_{\texttt{State}}$ dans c, on utilise la notation $c[z_{\texttt{State}}]$;*
- *l'application d'un Σ-contexte $c[z_{\texttt{State}}]$ à un terme $t \in T_\Sigma(F, X)$ de sorte* `State`*, notée $c[t]$, est définie par la substitution de $z_{\texttt{State}}$ par t dans $c[z_{\texttt{State}}]$. Ainsi, c est dit* applicable *à t ;*
- *un Σ-contexte c est, soit* observable *si la sorte de c est dans S_d, soit un* contexte d'état *si la sorte de c est* `State` *;*
- *$\mathcal{E}_\Sigma$ est l'ensemble des Σ-contextes observables.* ■

Exemple 2.3 *Considérons la signature de l'exemple 2.1. Il existe une infinité de contextes d'état $Do^n(put(e), s)$ ainsi qu'une infinité de contextes observables $get(Do^n(put(e), s))$ avec $n > 0$.* ■

La notion de validité observationnelle repose sur l'idée de base que deux objets dans une implémentation donnée sont *observationnellement égaux* s'ils ne peuvent être distingués par des calculs à résultats observables. De tels calculs sont représentés par les contextes observables [5; 11].

Définition 2.4 (Egalité observable dans une implémentation) *Soient $\mathcal{I}$ une Σ-implémentation et i, i' deux éléments de $\mathcal{I}$. On dit que i et i' sont* observationnellement égaux *dans $\mathcal{I}$, et on le note par $\mathcal{I} \models_{obs} i = i'$, ssi : pour tout $c[z_{\texttt{State}}] \in \mathcal{E}_\Sigma$, pour toutes assignations θ_i, $\theta_{i'}$ telles que $z_{\texttt{State}}\theta_i = i$, $z_{\texttt{State}}\theta_{i'} = i'$ et pour tout $x \in X \setminus \{z_{\texttt{State}}\}$ $x\theta_i = x\theta_{i'}$, on a $\mathcal{I} \models c\theta_i = c\theta_{i'}$.* ■

Exemple 2.4 *Si l'on considère l'exemple 2.2, l'égalité observationnelle est tout simplement l'égalité stricte (ou l'identité) dans le cas de la première implémentation de la cellule de mémoire. En effet, tester $\mathcal{I} \models_{obs} [n_1] = [n_2]$ en utilisant le contexte observable $c = \texttt{get}(z_{\texttt{State}})$ se réduit à tester $\mathcal{I} \models \mathcal{I}_{\texttt{get}}([n_1]) = \mathcal{I}_{\texttt{get}}([n_2])$, ce qui mène à $\mathcal{I} \models n_1 = n_2$. Quant à la deuxième interprétation, $\mathcal{I} \models_{obs} [put(n), l] = [put(n), l']$ est vraie pour tout entier naturel n et toutes les listes d'entiers naturels l, l', car quel que soit le contexte observable c utilisé, $\mathcal{I}_c$ évalue les deux états à n.* ■

Définition 2.5 (Satisfaction observable) *Etant données une Σ-implémentation $\mathcal{I}$, et une Σ-équation conditionnelle $e \equiv \bigwedge_{i=1}^{n} a_i = b_i \Rightarrow a = b$. On dit que e est* observationnellement valide *dans $\mathcal{I}$, et on le note par $\mathcal{I} \models_{obs} e$, ssi pour toute assignation θ, si (pour tout $i \in [1..n]$, $\mathcal{I} \models_{obs} a_i\theta = b_i\theta$) alors $\mathcal{I} \models_{obs} a\theta = b\theta$.* ■

Si l'on considère les implémentations données dans l'exemple 2.2, l'équation $Do(put(n), s) = Do(put(n), s')$ est strictement satisfaite par la première implémentation et observationnellement satisfaite par la deuxième implémentation.

Nous donnons maintenant les ingrédients de la spécification formelle d'un objet collaboratif.

Définition 2.6 (Spécification d'Objet Collaboratif) *Une* spécification d'objet collaboratif $\mathcal{C}$ *est un tuple* (Σ, M, A, T, E) *tels que :*

- Σ *est une OC-signature ;*
- M *est l'ensemble de méthodes, i.e.* $M = \{m(x_1, \ldots, x_n) \mid m$ *est un nom de méthode avec une arité* $n \geq 0$ *et* $x_1, \ldots, x_n$ *sont des variables de sorte* $S_d\}$ *;*
- A *est l'ensemble d'attributs, i.e.* $A = \{a(x_1, \ldots, x_n, z) \mid a$ *est un nom d'attribut avec* $x_1, \ldots, x_n$*, sont des variables de sorte* S_d *et* z *est une variable d'état* $\}$ *;*
- T *est l'ensemble d'axiomes définissant la fonction de transformation ;*
- E *est l'ensemble de tous les axiomes.*

Pour désigner de telles spécifications, nous utilisons les symboles $\mathcal{C}$*,* $\mathcal{C}'$*,* $\mathcal{C}_1$*,* $\mathcal{C}_2$*, ... Une* Σ*-implémentation* $\mathcal{I}$ satisfait observationnellement *(ou est une* implémentation *d'une) spécification* $\mathcal{C} = (\Sigma, M, A, T, E)$*, et on le note* $\mathcal{I} \models_{obs} \mathcal{C}$*, ssi* $\mathcal{I} \models_{obs} E$*.* ■

Il est clair que l'enrichissement d'une OC-signature Σ par des axiomes (ou des propriétés) permet de restreindre le nombre des Σ-implémentations [90].

Exemple 2.5 *La spécification d'une cellule de mémoire stockant des valeurs de type caractère est définie comme suit :*

```
spec CCHAR =
sort:
  Char Meth State
opns:
  Do : Meth State -> State
  putchar : Char -> Meth
  getchar : State -> Char
  IT : Meth Meth -> Meth
axioms:
  (1) getchar(Do(putchar(c),st)) = c;
  (2) IT(putchar(c1),putchar(c2)) = putchar(maxchar(c1,c2));
```

A partir de cette spécification, nous pouvons déduire que les ensembles des méthodes et des attributs sont respectivement $M = \{$`putchar`$\}$ *et* $A = \{$`getchar`$\}$*. L'axiome* `(1)`*, qui est une équation observable, indique que nous observons ce que nous avons stockons. Quant à l'axiome* `(2)` *il donne comment transformer deux* `putchar` *concurrentes pour les sérialiser l'une après l'autre. Pour atteindre la convergence des données, nous utilisons la fonction* `maxchar` *qui retourne la valeur maximale de deux caractères, comme un moyen pour résoudre le conflit généré par les deux méthodes concurrentes. Il faut noter que*

l'on peut recourir à d'autres moyens pour assurer la convergence, tels que le minimum, la moyenne, etc. ■

Pour une présentation concise de ce qui suit, et sans perdre de généralité, nous supprimons les arguments de sortes observables des méthodes et des attributs. Nous pourrions supposer qu'il existe une fonction pour chacun de ces arguments observables. A titre d'exemple, la méthode `putchar(c)` peut être remplacée par $\texttt{putchar}_c$ pour tout $c \in$`CHAR`.

Définition 2.7 (Complétude) *Soit une spécification* $\mathcal{C} = (\Sigma, M, A, T, E)$. *L'ensemble des axiomes* E *est dit* (M, A)-complet *ssi toutes les équations conditionnelles contenant des méthodes sont de la forme suivante :*

$$\Delta \implies a(Do(m, x)) = t$$

pour tout x *de sorte* `State`, $a \in A$ *et* $m \in M$ *tels que* $t \in T_{\Sigma \setminus M}(\{x\})$ *et* Δ *est un ensemble fini d'équations observables* $t_1 = t'_1$, $t_2 = t'_2$, ..., $t_n = t'_n$ *avec* t_1, $t'_1 \in T_\Sigma(X)_{s_1}$, t_2, $t'_2 \in T_\Sigma(X)_{s_2}$, ..., t_n, $t'_n \in T_\Sigma(X)_{s_n}$. ■

Dans l'exemple précédent, E est (M, A)-complet car le seul axiome contenant un terme de sorte `Meth`, à savoir `(1)`, possède la forme requise par la définition précédente.

Dans ce qui suit, nous restreignons notre attention à des spécifications dont l'ensemble des axiomes est (M, A)-complet.

Définition 2.8 (La relation $=_{\mathbf{obs}}$) *Soient une spécification* $\mathcal{C} = (\Sigma, M, A, T, E)$ *et deux termes* t_1, t_2. *On dit que* t_1 *et* t_2 *sont* observationnellement égaux, *et on le note* $E \models_{obs} t_1 = t_2$ *ou simplement* $t_1 =_{obs} t_2$, *ssi pour tout* $c[z_{\texttt{State}}] \in \mathcal{E}_\Sigma$ $E \models_{ind} c[t_1] = c[t_2]$. ■

Lemme 2.1 ([11]) *La relation* $=_{obs}$ *est une congruence sur* $T(F)$. ■

Preuve. La preuve est donnée dans [11]. ■

Dans un souci d'utiliser des preuves par induction dans le cadre observationnel, nous nous bornons à la classe suivante des Σ-implémentations [65; 58; 5] :

Définition 2.9 (La classe $\mathbf{Imp}(\mathcal{C}, \mathbf{\Sigma^d})$) *Soient* $\mathcal{C} = (\Sigma, M, A, T, E)$ *une spécification d'un objet collaboratif et* Σ^d *sa signature de données (qui constitue le support d'observation). La classe* $Imp(\mathcal{C}, \Sigma^d)$ *des* Σ*-implémentations est telle que pour tout* $\mathcal{I} \in Imp(\mathcal{C}, \Sigma^d)$ *:*

1. *il existe un homomorphisme surjectif de* $T(F)$ *dans* $\mathcal{I}$ *;*
2. $\mathcal{I} \models_{obs} \mathcal{C}$ *;*
3. *pour tout terme clos* $g \in T(F^d)$ *et pour tout terme* t, *si* $\mathcal{I} \models g = t$ *alors* $E \models_{ind} g = t$. ■

Les deux premières conditions indiquent qu'une Σ-implémentation dans la classe $Imp(\mathcal{C}, \Sigma^d)$ doit être un modèle finiment engendré de E. Quant à la troisième condition, elle stipule que la classe d'équivalence d'un terme clos construit à partir de Σ^d doit être explicitement déduite de E. A partir de $Imp(\mathcal{C}, \Sigma^d)$, nous énonçons les propriétés suivantes :

Lemme 2.2 *Si $Imp(\mathcal{C}, \Sigma^d)$ contient une implémentation finale $Fin(\mathcal{C})$, alors pour toute implémentation $\mathcal{I} \in Imp(\mathcal{C}, \Sigma^d)$, l'homomorphisme $\alpha : \mathcal{I} \rightarrow Fin(\mathcal{C})$ est surjectif.* ■

Preuve. Supposons que $Fin(\mathcal{C}) \in Imp(\mathcal{C}, \Sigma^d)$. En utilisant la définition 2.9, les homomorphismes $\beta : T(F) \rightarrow Fin(\mathcal{C})$ et $\rho : T(F) \rightarrow \mathcal{I}$ sont surjectifs. Puisque β et ρ sont uniques[3], alors la composition de ρ et α est aussi unique ; ce qui couvre tous les éléments de $Fin(\mathcal{C})$. Par conséquent, tout élément de $Fin(\mathcal{C})$ doit être accessible à partir de $\mathcal{I}$ par α. ■

Lemme 2.3 *Soient $\mathcal{I}$ et $\mathcal{I}'$ deux Σ-implémentations arbitraires. S'il existe un homomorphisme surjectif $\alpha : \mathcal{I}' \rightarrow \mathcal{I}$, alors tout homomorphisme $\beta : T(F, X) \rightarrow \mathcal{I}$ peut être exprimé comme la composition de α et un homomorphisme $\rho : T(F, X) \rightarrow \mathcal{I}'$.* ■

Preuve. Les homomorphismes β et ρ sont uniques [4] :

- Soit $x \in T(F, X)$ une variable. Comme α est surjectif, alors on peut choisir $\rho(x)$ tel que $\beta(x) = \alpha(\rho(x))$.
- Soit $f(t_1, \ldots, t_n) \in T(F, X)$ une fonction avec $t_1 \in T(F, X)$, ..., $t_n \in T(F, X)$ sont des termes. Comme $\rho(f(t_1, \ldots, t_n)) = \mathcal{I}'_f(\rho(t_1), \ldots, \rho(t_n))$, alors on a :

$$\begin{array}{lcl} \alpha(\rho(f(t_1, \ldots, t_n))) & = & \mathcal{I}_f(\alpha(\rho(t_1)), \ldots, \alpha(\rho(t_n))) \\ \beta(f(t_1, \ldots, t_n)) & = & \mathcal{I}_f(\beta(t_1), \ldots, \beta(t_n)) \end{array}$$

  Ainsi les deux homomorphismes doivent être identiques.

■

A partir des propriétés ci-dessus, nous déduisons que $Imp(\mathcal{C}, \Sigma^d)$ possède une algèbre finale.

Théorème 2.1 *L'implémentation finale $Fin(\mathcal{C})$ dans la classe $Imp(\mathcal{C}, \Sigma^d)$ est l'algèbre quotient de $T(F)$ par $=_{obs}$.* ■

Preuve. Soit $\mathcal{I}$ est une Σ-implémentation dans $Imp(\mathcal{C}, \Sigma^d)$. Il existe donc deux homomorphimes uniques $\rho : T(F) \rightarrow \mathcal{I}$ et $\beta : T(F) \rightarrow Fin(\mathcal{C})$. Par définition, ρ et β sont aussi surjectifs. Soit un homomorphisme α de $\mathcal{I}$ vers $Fin(\mathcal{C})$:

1. α est unique : supposons qu'il y a deux homomorphismes distincts α_1 et α_2 de $\mathcal{I}$ vers $Fin(\mathcal{C})$. Or, comme β est unique, on a $\beta = \rho \circ \alpha_1$ et $\beta = \rho \circ \alpha_2$. Donc, α_1 et α_2 doivent être égaux.

[3] Dans la littérature, il est montré que $T(F)$ est une algèbre initiale et donc il existe un homomorphisme unique de $T(F)$ vers toute Σ-algèbre (voir la proposition 1.1).

[4] La proposition suivante peut être facilement prouvée : étant donné une Σ-algèbre A, alors toute assignation $\theta : X \rightarrow A$ peut être uniquement étendue à un homomorphisme $\theta^* : T(F, X) \rightarrow A$ (voir la proposition 1.2).

2. α est surjectif (en utilisant le lemme 2.2).

Maintenant montrons que si une équation $u = v$ est observationnellement valide dans $\mathcal{I}$, elle l'est aussi dans $Fin(\mathcal{C})$. Supposons que $Fin(\mathcal{C})$ n'est pas une implémentation de $u = v$. D'après le lemme 2.3, tout homomorphisme ϕ de $T(F, X)$ vers $Fin(\mathcal{C})$ peut être exprimé comme la composition de γ et α où γ est un homomorphisme de $T(F, X)$ vers $\mathcal{I}$. Comme $\mathcal{I}$ est une implémentation de $u = v$ alors $\gamma(u) = \gamma(v)$. Toute composition de γ et α satisfera donc $\alpha(\gamma(u)) = \alpha(\gamma(v))$. Or ceci signifie que $u = v$ est observationnellement valide dans $Fin(\mathcal{C})$ (Contradiction). ■

2.3 Propriétés de Convergence

Avant de définir les propriétés que doit satisfaire une spécification $\mathcal{C} = (\Sigma, M, A, T, E)$ pour assurer la convergence, nous donnons les notations suivantes.

Notations. Soient m_1, m_2, ..., m_n des variables de sorte `Meth` et s une variable d'état.

1. L'application d'une séquence de méthodes sur un état donnée est notée :

$$(s)[m_1; m_2; \ldots; m_n] \equiv Do(m_n, \ldots, Do(m_2, Do(m_1, s)) \ldots)$$

   l'ordre d'application des méthodes se lit de la gauche vers la droite.

2. Une séquence de méthodes est possible sur un état donné si toutes ses méthodes sont possibles sur les états successifs :

$$\begin{aligned} Legal([m_1; m_2; \ldots; m_n], s) \quad \equiv \quad & Poss(m_1, s) \wedge Poss(m_2, (s)m_1) \wedge \ldots \\ & \wedge Poss(m_n, (s)[m_1; m_2; \ldots; m_{n-1}]) \end{aligned}$$

3. Par abus de notation, nous utilisons IT^* comme une fonction qui prend en arguments une méthode et une séquence de méthodes pour en produire une autre méthode. Elle est définie comme suit :

$$\begin{aligned} IT^*(m_1, []) &= m_1 \\ IT^*(m_1, [m_2; m_3; \ldots; m_n]) &= IT^*(IT(m_1, m_2), [m_3; \ldots; m_n]) \end{aligned}$$

■

2.3.1 Condition TP1

Comme vu précédemment, **TP1** exprime une *identité d'états* entre deux séquences de méthodes. Pour ce faire, nous utilisons une approche observationnelle pour comparer deux états. Aussi, nous définissons **TP1** par la propriété d'état suivante :

$$\begin{aligned} CP1 \quad \equiv \quad & (Legal([m_1; IT(m_2, m_1)], s) = true \wedge \\ & Legal([m_2; IT(m_1, m_2)], s) = true) \\ & \Longrightarrow (s)[m_1; IT(m_2, m_1)] = (s)[m_2; IT(m_1, m_2)] \end{aligned}$$

tels que m_1, $m_2 \in M$.

Soit $M' \subseteq M$ un ensemble de méthodes. Nous notons $CP1|_{M'}$ comme la restriction de $CP1$ aux méthodes de M'. Soient $M_1, M_2 \subseteq M$ deux ensembles disjoints de méthodes. Nous utilisons la notation $CP1|_{M_1,M_2}$ comme suit :

$$\begin{aligned} CP1|_{M_1,M_2} \quad \equiv \quad & (Legal([m_i; IT(m_j, m_i)], s) = true \wedge \\ & Legal([m_j; IT(m_i, m_j), s) = true) \\ & \implies (s)[m_i; IT(m_j, m_i)] = (s)[m_j; IT(m_i, m_j)] \end{aligned}$$

tels que $m_i \in M_i$ et $m_j \in M_j$ pour tout $i \neq j \in \{1, 2\}$.

2.3.2 Condition TP2

TP2 stipule une *identité de méthodes* entre deux séquences équivalentes de méthodes. En effet, étant donnés trois méthodes m_1, m_2 et m_3, la transformation de m_3 par rapport à deux séquences de méthodes $[m_1; IT(m_2, m_1)]$ et $[m_2; IT(m_1, m_2)]$ doit produire la même méthode. Nous définissons **TP2** comme suit :

$$CP2 \equiv IT^*(m_3, [m_1; IT(m_2, m_1)]) = IT^*(m_3, [m_2; IT(m_1, m_2)])$$

tels que m_1, m_2, $m_3 \in M$.

De la même manière, $CP2|_{M'}$ est une restriction aux méthodes de M'. Etant données $M_1, M_2 \subseteq M$ deux ensembles disjoints de méthodes. La notation est défini de la manière suivante :

$$CP2|_{M_1,M_2} \equiv IT^*(m, [m'; IT(m'', m')]) = IT^*(m, [m''; IT(m', m'')])$$

tels que $m' \in M_i$, $m'' \in M_j$ et $m \in M_k$ pour tout $i, j, k \in \{1, 2\}$ avec $k \neq i$ ou $k \neq j$.

2.3.3 Consistance

Comme les objets collaboratifs sont modifiés de manière concurrente, alors nous nous intéressons à des objets ayant la propriété suivante :

Définition 2.10 (Consistance) *Une spécification $\mathcal{C} = (\Sigma, M, A, T, E)$ est dite* consistante *ssi $\mathcal{C} \models_{obs} CP1$ et $\mathcal{C} \models_{ind} CP2$. Elle est dite* semi-consistante *si elle ne satisfait que $CP1$.* ■

Exemple 2.6 *Considérons la spécification donnée à l'exemple 2.5. Il est facile de vérifier que* `CCHAR` *est consistante :*

***Condition* CP1**. *Elle est définie par l'équation suivante :*

```
Do(IT(putchar(c2),putchar(c1)), Do(putchar(c1),s)) =
Do(IT(putchar(c1),putchar(c2)), Do(putchar(c2),s))
```

En appliquant l'axiome `(2)` *nous obtenons :*

```
Do(putchar(maxchar(c1,c2)), Do(putchar(c1),s)) =
Do(putchar(maxchar(c2,c1)), Do(putchar(c2),s))
```

Puisque l'équation `maxchar(c1,c2)=maxchar(c2,c1)` *est toujours vraie, alors* $CP1$ *est une propriété d'état pour tous les contextes observables* `getchar(s)`, `getchar(Do(putchar(c),s))`, `getchar(Do(putchar(c2),Do(putchar(c1),s)))`, *etc.*

***Condition* CP2**. *Elle est décrite par l'équation suivante :*

```
IT(IT(putchar(c3),putchar(c1)), IT(putchar(c2),putchar(c1))) =
IT(IT(putchar(c3),putchar(c2)), IT(putchar(c1),putchar(c2)))
```

En utilisant l'axiome `(2)`, *l'équation ci-dessus est réécrite en*

```
putchar(maxchar(maxchar(c3,c1), maxchar(c2,c1))) =
putchar(maxchar(maxchar(c3,c2), maxchar(c1,c2)))
```

Quelque soient les combinaisons possibles des valeurs de `c3`, `c2` *et* `c1`, *l'équation ci-dessus est une conséquence de* `CCHAR`. ■

2.4 Techniques de Vérification

2.4.1 Systèmes de réécriture

Notre but est d'utiliser un prouveur de théorèmes pour vérifier automatiquement qu'un objet collaboratif est consistant. Il faut rappeler que les axiomes de nos spécifications sont exprimés par des équations conditionnelles. Néanmoins, l'obstacle majeur à une vérification automatique est la nature déclarative des équations. A cet égard, pour rendre les spécifications des objets collaboratifs exécutables, nous avons procédé comme suit.

Utilisation des symboles constructeurs. Les objets d'un type abstrait algébrique sont souvent définis à partir de symboles de fonctions particuliers appelés constructeurs de type [9]. Aussi, nous allons scinder l'ensemble des symboles de fonction F en deux sous-ensembles disjoints : (i) C contient les symboles constructeurs ; (ii) D contient les symboles d'opérateurs définis.

Etant donnée une OC-signature $\Sigma = (S, S_{obs}, F)$. Nous définissons donc F comme suit :

1. $C_{\texttt{Meth State,State}} = \{Do\}$ et $C_{\omega,s} = \emptyset$ pour tous les autres cas où soit $\omega \in S_b^*$, soit s est une sorte d'état ;
2. $D_{\texttt{Meth Meth,Meth}} = \{IT\}$, $D_{\texttt{Meth State,Bool}} = \{Poss\}$ et $D_{\omega,s} = \emptyset$ pour tous les autres cas où $\omega \in S_b^*$ et $s \in S_b$.

Pour toute sorte, l'utilisation des symboles constructeurs a l'avantage de donner une caractérisation finie pour un ensemble infini de termes. A titre d'exemple, considérons la spécification donnée à l'exemple 2.5. l'espace d'états pour `CCHAR` peut être décrit par le terme `Do(putchar(c),s)`.

Utilisation des systèmes de réécritures. Le raisonnement équationnel permet une liberté sans restriction quant au remplacement d'un terme par un terme

égal. Ceci conduit à une explosion combinatoire des possibilités de remplacement qui est très difficile (voire impossible) à gérer par un prouveur automatique. Une équation $t_1 = t_2$ peut être orientée en règle de réécriture $t_1 \rightarrow t_2$. Cette règle définit toujours l'égalité $t_1 = t_2$. Elle permet le remplacement d'une instance de t_1 par une instance de t_2, mais elle interdit le remplacement dans la direction opposée. Cette orientation d'équations transforme une spécification algébrique en système de réécriture. Elle a l'avantage de rendre les spécifications plus opérationnelles qui se prêtent bien à la vérification automatique.

Soit $\mathcal{R}_\mathcal{C}$ le système de réécriture correspondant à une spécification $\mathcal{C}$. Pour vérifier les conditions $CP1$ et $CP2$, nous supposons que $\mathcal{R}_\mathcal{C}$ possède les propriétés suivantes : convergence sur les termes clos (terminaison et confluence) et complétude suffisante.

2.4.2 Exemple illustratif

Dans cette sous-section nous allons présenter l'éditeur collaboratif de textes `GROVE` qui a été conçu par Ellis et Gibb [26]. Ces derniers sont les pionniers de l'approche des transformées opérationnelles. `GROVE` ne requiert aucune contrainte quant à l'édition de texte. En effet, il confère aux utilisateurs une liberté totale pour éditer n'importe quelle partie du texte et à n'importe quel moment. En d'autres termes, il permet une édition simultanée du même texte par plusieurs personnes, tout en gérant le problème des conflits occasionnés par les opérations d'édition concurrentes.

Algorithme de Transformation

Deux opérations d'édition sont utilisées, à savoir :

- $Ins(p, c, pr)$ qui insère un caractère c à la position p ;
- $Del(p, pr)$ qui détruit le caractère à la position p.

Le paramètre pr, appelé *priorité*, contient tout simplement l'identifiant du site où l'opération a été générée. Cette priorité est utilisée comme un moyen pour résoudre le conflit apparaissant au moment où deux insertions concurrentes tentaient d'insérer des caractères différents mais à la même position. Par ailleurs, il faut souligner le fait que deux opérations concurrentes ont toujours des priorités différentes puisqu'elles ont été générées sur deux sites différents.

Dans l'algorithme 1, Ellis et Gibb proposent pour chaque couple d'opérations leur transformée en tenant compte des valeurs des différents paramètres. Cette transformation est bien sûr établie pour assurer la convergence des textes dans `GROVE`. On remarquera que lorsque les deux opérations sont identiques (excepté leurs priorités), IT retourne l'opération nulle $Nop()$, qui est sans effet sur l'état d'un texte. Ce cas se présente quand deux utilisateurs insèrent simultanément le même caractère à la même position, ou lorsqu'ils effacent simultanément le même caractère. Il faut alors exécuter une opération et ignorer l'autre car une seule opération suffit pour mener à la convergence. Nous complétons l'algorithme 1

1: $IT(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) =$
2: **si** $(p_1 < p_2)$ **alors**
3: **retourner** $Ins(p_1, c_1, pr_1)$
4: **sinon si** $(p_1 > p_2)$ **alors**
5: **retourner** $Ins(p_1 + 1, c_1, pr_1)$
6: **sinon si** $(c_1 = c_2)$ **alors**
7: **retourner** $Nop()$
8: **sinon si** $(pr_1 > pr_2)$ **alors**
9: **retourner** $Ins(p_1 + 1, c_1, pr_1)$
10: **sinon** $\{pr_1 < pr_2\}$
11: **retourner** $Ins(p_1, c_1, pr_1)$
12: **fin si**

13: $IT(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) =$
14: **si** $(p_1 < p_2)$ **alors**
15: **retourner** $Ins(p_1, c_1, pr_1)$
16: **sinon**
17: **retourner** $Ins(p_1 - 1, c_1, pr_1)$
18: **fin si**

19: $IT(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)) =$
20: **si** $(p_1 < p_2)$ **alors**
21: **retourner** $Del(p_1, pr_1)$
22: **sinon**
23: **retourner** $Del(p_1 + 1, pr_1)$
24: **fin si**

25: $IT(Del(p_1, pr_1), Del(p_2, pr_2)) =$
26: **si** $(p_1 < p_2)$ **alors**
27: **retourner** $Del(p_1, pr_1)$
28: **sinon si** $(p_1 > p_2)$ **alors**
29: **retourner** $Del(p_1 - 1, pr_1)$
30: **sinon**
31: **retourner** $Nop()$
32: **fin si**

Algorithme 1: Algorithme de transformation utilisé dans `GROVE`.

par les définitions suivantes :

$$IT(Nop(), Ins(p, c, pr)) = IT(Nop(), Del(p, pr)) = Nop()$$
$$IT(Ins(p, c, pr), Nop()) = Ins(p, c, pr)$$
$$IT(Del(p, pr), Nop()) = Del(p, pr)$$

pour dire que $Nop()$ n'a pas d'effet sur la transformation des opérations.

Par ailleurs, lorsque deux utilisateurs insèrent au même moment deux caractères différents mais à la même position alors on est dans une situation de *conflit*. Pour résoudre un tel conflit, Ellis et Gibb utilisent l'ordre sur les priorités pour sérialiser les deux opérations. Ainsi, la position d'insertion sera incrémentée (ou

sera avancée vers la droite) pour toute opération qui a la plus haute priorité. Le reste des cas pour IT sont assez simples.

Spécification Algébrique

L'objet collaboratif dans `GROVE` est un texte dont la spécification est donnée à la figure 2.1. On suppose que la position du premier caractère est 0. La sorte `Meth` représente l'ensemble des méthodes que possède l'objet collaboratif. Chaque méthode est définie comme un constructeur de la sorte `Meth`. Aussi, l'objet texte possède trois méthodes :

- `Ins(p,c,pr)` pour insérer un caractère `c` à la position `p` ;
- `Del(p,pr)` pour effacer un caractère à la position `p` ;
- `Nop` est une méthode sans effet sur l'état de l'objet texte.

Comme vu précédemment, l'objet texte est considéré comme une boîte noire en ce sens où la description de l'état est cachée. En effet, l'état est assimilé à la séquence de méthodes exécutée à partir d'un état initial donné. La sorte `State` est non observable et elle représente l'espace d'états. Le seul constructeur de `State` est `Do` qui définit une fonction de transition entre les différents états de l'objet texte. Nous définissons une fonction booléenne `Poss` pour donner les conditions nécessaires à une méthode pour qu'elle soit exécutée sur un état donné. Par exemple, la condition "*il est possible d'insérer un caractère seulement à une position avant la fin du texte*" est exprimée par l'axiome `(1)` de la figure 2.1.

Par ailleurs, l'objet texte dispose de deux attributs :

- `Length` qui donne la longueur du texte ;
- `Car` qui extrait un caractère pour une position donnée et un état du texte donné.

Le changement d'états est liée aux changements de valeurs des attributs. Comment de tels attributs sont affectés par les méthodes est exprimé par les axiomes `(4)-(12)`. La définition équationnelle de la fonction de transformation IT est donnée par les axiomes `(13)-(26)`.

2.4.3 Vérification de $CP1$

A partir de la spécification d'un objet collaboratif nous allons vérifier si son algorithme de transformation satisfait la propriété $CP1$; en d'autres termes, nous allons vérifier si $CP1$ est une conséquence observationnelle. Notre objectif est double :

- utiliser une machinerie de preuve, Proof , pour vérifier ou réfuter cette propriété ;
- en cas d'échec fournir toutes les situations qui mènent potentiellement vers la violation de cette propriété.

Définition 2.11 ($CP1$-scénario) *Un $CP1$-scénario est un triplet $(m, m', \mathcal{E})$ tels que m, m' sont des méthodes et $\mathcal{E}$ est l'ensemble non vide de conjectures retourné par* Proof *($\{CP1[m_1 \leftarrow m, m_2 \leftarrow m']\}$) avec $CP1[m_1 \leftarrow m, m_2 \leftarrow m']$ représentant la substitution dans $CP1$ des variables m_1 et m_2 par les méthodes m et m'.*

```
Sorts: State, Meth, Bool, Nat, Char
Observable sorts: Meth, Bool, Nat, Char

Constructors:
Do : Meth State -> State
Ins : Nat Char Nat -> Meth
Del : Nat Nat -> Meth
Nop : -> Meth
C0 : -> Char;
C1 : -> Char;

Defined Operations:
Poss : Meth State -> Bool
Length : State -> Nat
Car : Nat State -> Char
IT : Meth Meth -> Meth

Axioms:
(1)  Poss(Ins(p1,c1,pr1),st) = (p1 <= Length(st));
(2)  Poss(Del(p1,pr1),st) = (p1 < Length(st));
(3)  Poss(Nop, st) = true;

(4)  Length(Do(Ins(p1,c1,pr1),st)) = Length(st)+1;
(5)  Length(Do(Del(p1,pr1),st)) = Length(st)-1;
(6)  x=p1 => Car(x,Do(Ins(p1,c1,pr1),st)) = c1;
(7)  (x>p1)=true => Car(x,Do(Ins(p1,c1,pr1),st)) = Car(x-1,st);
(8)  (x<p1)=true => Car(x,Do(Ins(p1,c1,pr1),st)) = Car(x,st);
(9)  (x >= p1)=true => Car(x,Do(Del(p1),st)) = Car(x+1,st);
(10) (x<p1))=true => Car(x,Do(Del(p1),st)) = Car(x,st);
(11) Length(Do(Nop,st)) = Length(st);
(12) Car(x, Do(Nop, st)) = Car(x, st);

(13) IT(Nop,m)=Nop;
(14) IT(m,Nop)=m;
(15) (p1<p2)=true => IT(Ins(p1,c1,pr1),Ins(p2,c2,pr2)) = Ins(p1,c1,pr1);
(16) (p1>p2)=true => IT(Ins(p1,c1,pr1),Ins(p2,c2,pr2)) = Ins(p1+1,c1,pr1);
(17) p1=p2, c1=c2 => IT(Ins(p1,c1,pr1),Ins(p2,c2,pr2)) = Nop;
(18) p1=p2, c1<>c2, (pr1>pr2)=true =>
              IT(Ins(p1,c1,pr1),Ins(p2,c2,pr2)) = Ins(p1+1,c1,pr1);
(19) p1=p2, c1<>c2, (pr1<pr2)=true =>
              IT(Ins(p1,c1,pr1),Ins(p2,c2,pr2)) = Ins(p1,c1,pr1);
(20) (p1<p2)=true => IT(Ins(p1,c1,pr1),Del(p2,pr2)) = Ins(p1,c1,pr1);
(21) (p1=>p2)=true => IT(Ins(p1,c1,pr1),Del(p2,pr2)) = Ins(p1-1,c1,pr1);
(22) (p1<p2)=true => IT(Del(p1,pr1),Del(p2,pr2)) = Del(p1,pr1);
(23) (p1>p2)=true => IT(Del(p1,pr1),Del(p2,pr2)) = Del(p1-1,pr1);
(24) p1=p2 => IT(Del(p1,pr1),Del(p2,pr2)) = Nop;
(25) (p1<p2)=true => IT(Del(p1,pr1),Ins(p2,c2,pr2)) = Del(p1,pr1);
(26) (p1=>p2)=true => IT(Del(p1,pr1),Ins(p2,c2,pr2)) = Del(p1+1,pr1);
```

FIG. 2.1 – Spécification de l'objet collaboratif Texte.

Entrée : Une spécification $\mathcal{C} = (\Sigma, M, A, T, E)$
Sortie : L'ensemble $\mathcal{S}$ des $CP1$-scénarios

1: **pour tout** m dans M **faire**
2: **pour tout** m' dans M **faire**
3: $\mathcal{E} \leftarrow \{CP1[m_1 \leftarrow m, m_2 \leftarrow m']\}$
4: $\mathcal{E} \leftarrow$ PROOF $(\mathcal{E})$
5: **si** $\mathcal{E} \neq \emptyset$ **alors**
6: **retourner** $\mathcal{S} \leftarrow \mathcal{S} \cup \{(m, m', \mathcal{E})\}$
7: **fin si**
8: **fin pour**
9: **fin pour**

Algorithme 2: Algorithme pour vérifier $CP1$.

Les $CP1$-scénarios sont utilisés pour représenter les situations qui mènent vers des divergences de données causés par la non satisfaction de $CP1$. L'algorithme 2 met en œuvre comment vérifier cette propriété. Il faut noter que si PROOF échoue à réduire l'ensemble des conjectures $\mathcal{E}$ alors on stockera le résultat de PROOF pour pouvoir éventuellement reconstruire le scénario de divergence.

Exemple 2.7 *En utilisant l'algorithme 2 avec comme machinerie de preuve* SPIKE*, nous avons constaté que l'algorithme de transformation décrit à la figure 1 n'est pas correct. En effet, l'algorithme proposé par Ellis et Gibb ne satisfait pas la propriété $CP1$. La figure 2.2 illustre le résultat donné par l'algorithme 2.*

```
Scenario 1:
-----------
op1 : Ins(u1,u2,u3)
op2 : Del(u4,u5)

S1 [op1;IT(op2,op1)]:
   [Ins(u1,u2,u3);Del(u1+1,u5)]

S2 [op2;IT(op1,op2)]:
   [Del(u1,u5);Ins(u1-1,u2,u3)]

Instance: Car(u1,S1) = Car(u1,S2)

Preconditions:
(u1 <= Length(u6))=true /\ (u4 < Length(u6))=true /\ u1 = u4;
```

FIG. 2.2 – Un $CP1$-scénario pour l'algorithme de Ellis et Gibb.

Nous allons voir maintenant comment SPIKE *arrive à déceler la réfutation de $CP1$. Considérons l'instance $m = Ins(p_1, c_1, pr_1)$ et $m' = Del(p_2, pr_2)$ (voir*

les lignes $1-2$ *de l'algorithme 2). Nous obtenons la conjecture suivante :*

$$\begin{aligned}(Legal([Ins(p_1,c_1,pr_1);IT(Del(p_2,pr_2),Ins(p_1,c_1,pr_1))],s) = true \wedge \\ Legal([Del(p_2,pr_2);IT(Ins(p_1,c_1,pr_1),Del(p_2,pr_2))],s) = true) \\ \Longrightarrow (s)[Ins(p_1,c_1,pr_1);IT(Del(p_2,pr_2),Ins(p_1,c_1,pr_1))] = \\ (s)[Del(p_2,pr_2);IT(Ins(p_1,c_1,pr_1),Del(p_2,pr_2))]\end{aligned} \tag{2.1}$$

Pour réduire la conjecture (2.1), une analyse par cas est appliquée en utilisant des règles conditionnelles correspondant aux fonctions IT *et* $Poss$*. Si nous considérons le cas où les positions* p_1 *et* p_2 *sont égales, alors la conjecture (2.1) sera normalisée comme suit :*

$$\begin{aligned}(p_1 < Length(s)) = true \implies (s)[Ins(p_1,c_1,pr_1);Del(p_1+1)] = \\ (s)[Del(p_1,pr_2);Ins(p_1-1,pr_1)]\end{aligned} \tag{2.2}$$

Pour vérifier que (2.2) est une propriété observationnelle, `SPIKE` *va utiliser l'ensemble de contextes test pour la sorte* `State`*. Pour notre exemple (voir la spécification de la figure 2.1), l'ensemble de contextes test est* $\{Length(z_{\texttt{State}}), Car(x, z_{\texttt{State}})\}$*. En appliquant une induction avec comme contexte test* $Car(x, z_{\texttt{State}})$ *tel que* $x = p_1$ *et en utilisant les règles conditionnelles propres à la fonction* Car*, alors nous obtenons :*

$$(p_1 < Length(s)) = true \implies c_1 = Car(p_1, s) \tag{2.3}$$

En utilisant l'ensemble test $\{C0, C1,\ Do(Ins(p,c,pr),s), Do(Del(p,pr),s)\}$*, une induction peut être appliquée à (2.3) :*

$$\begin{aligned}(p_1 < Length(Do(Ins(p,C0,pr),s))) = true \implies \\ C0 = Car(p_1, Do(Ins(p,C0,pr),s))\end{aligned} \tag{2.4}$$

$$\begin{aligned}(p_1 < Length(Do(Ins(p,C1,pr),s))) = true \implies \\ C0 = Car(p_1, Do(Ins(p,C1,pr),s))\end{aligned} \tag{2.5}$$

$$(p_1 < Length(Do(Del(p,pr),s))) = true \implies C0 = Car(p_1, Do(Del(p,pr),s)) \tag{2.6}$$

$$(p_1 < Length(Do(Del(p,pr),s))) = true \implies C1 = Car(p_1, Do(Del(p,pr),s)) \tag{2.7}$$

Considérons la conjecture (2.5). Dans le cas où les positions p_1 *et* p *sont égales, alors (2.5) est simplifiée comme suit :*

$$(p_1 < Length(s) + 1) = true \implies C0 = C1$$

L'équation $C0 = C1$ *est un témoin d'incohérence observable. Comme le système de réécriture correspondant à la spécification de la figure 2.1 est convergent sur les termes clos, alors* $CP1$ *n'est pas un théorème observationnel.* ■

Un $CP1$-scénario donne seulement le couple de méthodes ainsi que des conditions sur leurs arguments qui peut mener à une situation de divergence. A partir de telles informations, nous pouvons construire un scénario réel ; il suffit juste d'instancier les arguments des méthodes du $CP1$-scénario par des valeurs respectant les préconditions.

Exemple 2.8 *A partir du $CP1$-scénario donné à la figure 2.2, nous pouvons extraire les informations suivantes :*

1. *les méthodes $Ins(u1, u2, u3)$ et $Del(u4, u5)$ peuvent causer une divergence ;*
2. *l'observation qui distingue les états résultant des deux séquences de méthodes $S1$ et $S2$;*
3. *les conditions sur les arguments des méthodes (i.e. Preconditions) qui peuvent potentiellement mener vers à une divergence.*

Connaissant de telles informations, nous pouvons reconstruire un scénario réel qui est illustré dans la figure 2.3. Pour des raisons de lisibilité nous avons omis le paramètre de priorité. Nous avons deux utilisateurs qui démarrent une session d'édition collaborative à partir du même texte "abc". Le premier utilisateur insère "x" à la position 2 (op_1), alors que le deuxième utilisateur efface simultanément le caractère à la même position (op_2). L'intégration de op_2 au site 1 nécessite la transformation de op_2 par rapport à op_1, i.e. $IT(op_2, op_1) = Del(3)$, et l'exécution de $Del(3)$ donne l'état "axc". De la même manière, op_1 est transformée par rapport à op_2 au site 2, i.e. $IT(op_1, op_2) = Ins(1, 'x')$, et l'exécution de $Ins(1, 'x')$ donne l'état "xac". Aussi, nous constatons que les états finaux des deux sites sont différents. Par conséquent, nous sommes en présence d'une situation de divergence. En analysant l'algorithme 2 nous remarquons que l'erreur provient de la définition de $IT(Ins(p_1, c_1, u_1), Del(p_2, u_2))$ (voir la ligne 13). En effet, la condition $p_1 < p_2$ doit être réécrite en $p_1 \leq p_2$ pour que l'algorithme de transformation satisfasse $CP1$. ■

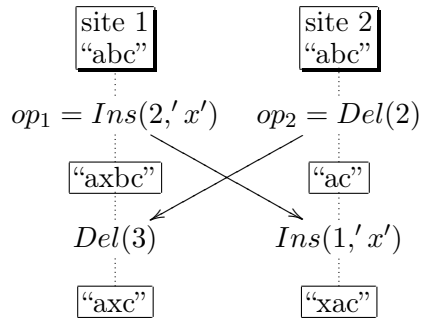


FIG. 2.3 – Un scénario réel violant la propriété $CP1$.

2.4.4 Vérification de $CP2$

Comme vu auparavant, la propriété $CP1$ est nécessaire mais pas suffisante pour assurer la convergence des données. En effet, il faut la satisfaction d'une autre propriété à savoir $CP2$. Dans cette sous-section nous allons présenter une technique pour vérifier ou réfuter une telle propriété. En cas d'échec nous allons fournir toute les situations qui conduisent potentiellement vers la non satisfaction de $CP2$.

Définition 2.12 ($CP2$-scénario) *Un* $CP2$-scénario *est un quadruplet $(m, m', m'', \mathcal{E})$ tels que m, m', m sont des méthodes et $\mathcal{E}$ est l'ensemble non vide de conjectures retourné par* PROOF *($\{CP2[m_1 \leftarrow m, m_2 \leftarrow m', m_3 \leftarrow m'']\}$).*

Entrée : Une spécification $\mathcal{C} = (\Sigma, M, A, T, E)$
Sortie : L'ensemble $\mathcal{S}$ des $CP2$-scénarios

1: **pour tout** m dans M **faire**
2: **pour tout** m' dans M **faire**
3: **pour tout** m'' dans M **faire**
4: $\mathcal{E} \leftarrow \{CP1[m_1 \leftarrow m, m_2 \leftarrow m', m_3 \leftarrow m'']\}$
5: $\mathcal{E} \leftarrow$ PROOF $(\mathcal{E})$
6: **si** $\mathcal{E} \neq \emptyset$ **alors**
7: **retourner** $\mathcal{S} \leftarrow \mathcal{S} \cup \{(m, m', m'', \mathcal{E})\}$
8: **fin si**
9: **fin pour**
10: **fin pour**
11: **fin pour**

Algorithme 3: Algorithme pour vérifier $CP2$.

L'algorithme 3 met en œuvre comment produire des $CP2$-scénarios quand PROOF échoue dans la preuve de $CP2$. Les $CP2$-scénarios contiennent simplement les méthodes, ainsi que des conditions sur leurs arguments, qui conduisent potentiellement à la non satisfaction de $CP2$.

Exemple 2.9 *Même après avoir corrigé l'algorithme 2 pour satisfaire $CP1$, il demeure toujours incorrect. En effet, nous avons constaté que la version corrigée de l'algorithme de Ellis et Gibb ne vérifie pas $CP2$ quand nous avons utilisé* `SPIKE`*. Deux $CP2$-scénarios sont donnés respectivement aux figures 2.4 et 2.5.*

Nous allons voir commet `SPIKE` *arrive à trouver un contre-exemple à la propriété $CP2$. Considérons l'instance $m = Ins(p_1, c_1, pr_1)$, $m' = Del(p_2, pr_2)$ et $m'' = Ins(p_3, c_3, pr_3)$ (voir les lignes $1-3$ de l'algorithme 3). Nous obtenons la conjecture suivante :*

$$IT^*(Ins(p_3, c_3, pr_3), [Ins(p_1, c_1, pr_1); IT(Del(p_2, pr_2), Ins(p_1, c_1, pr_1))]) = \\ IT^*(Ins(p_3, c_3, pr_3), [Del(p_2, pr_2); IT(Ins(p_1, c_1, pr_1), Del(p_2, pr_2))]) \tag{2.8}$$

Pour simplifier la conjecture (2.8), `SPIKE` *applique une analyse par cas en utilisant des règles conditionnelles propres à IT. Soit le cas où $p_1 = p_2 + 1$ et $p_2 = p_3$. Aussi, (2.8) sera simplifiée comme suit :*

$$IT(Ins(p_3, c_3, pr_3), Ins(p_1 - 1, c_1, pr_1)) = Ins(p_3, c_3, pr_3) \tag{2.9}$$

Deux cas sont possibles pour simplifier (2.9) :
- *si $c_3 = c_1$ alors (2.9) devient : $Nop = Ins(p_3, c_3, pr_3)$;*
- *si $c_3 \neq c_1$ et $pr_3 > pr_1$ alors (2.9) devient : $Ins(p_3 + 1, c_3, pr_3) = Ins(p_3, c_3, pr_3)$.*

Quel que soit le cas considéré, les deux simplifications mènent vers des témoins d'incohérence observables. Aussi $CP2$ n'est pas un théorème observationnel. ■

```
Scenario 1:
-----------
op1 : Ins(u1,u2,u3)
op2 : Del(u4,u5)
op3 : Ins(u6,u7,u8)

S1 [op2;IT(op3,op2)]:
  [Del(u4,u5);Ins(u6-1,u7,u8)]

S2 [op3;IT(op2,op3)]:
  [Ins(u6,u7,u8);Del(u4,u5)]

Transforming op1/S1: Ins(u1+1,u2,u3)

Transforming op1/S2: Ins(u1,u2,u3)

Preconditions:
u1 = u4 /\ (u4 < u6)=true /\ u1 = u6-1 /\ u2 = u7;
```

FIG. 2.4 – Un premier $CP2$-scénario pour l'algorithme de Ellis et Gibb.

```
Scenario 1:
-----------
op1 : Ins(u1,u2,u3)
op2 : Del(u4,u5)
op3 : Ins(u6,u7,u8)

S1 [op2;IT(op3,op2)]:
  [Del(u4,u5);Ins(u6-1,u7,u8)]

S2 [op3;IT(op2,op3)]:
  [Ins(u6,u7,u8);Del(u4,u5)]

Transforming op1/S1: Ins(u1+1,u2,u3)

Transforming op1/S2: Ins(u1,u2,u3)

Preconditions:
u1 = u4 /\ (u4 < u6)=true /\ u1 = u6-1 /\ u2 <> u7 /\ u8 > u3;
```

FIG. 2.5 – Un deuxième $CP2$-scénario pour l'algorithme de Ellis et Gibb.

En analysant les $CP2$-scénarios donnés par les figures 2.4 et 2.5, nous remarquons que la non satisfaction de $CP2$ est toujours causée par les mêmes méthodes, à savoir le triplet (Ins,Del,Ins). A partir de telles informations,

nous allons proposer un scénario réel.

Exemple 2.10 *Considérons trois sites* 1, 2 *et* 3 *qui éditent simultanément le même texte (voir la figure 2.6). Initialement, les utilisateurs démarrent à partir du même état "ciré". Ils génèrent concurremment les opérations* $op_1 = Ins(3, `r')$, $op_2 = Del(2)$ *et* $op_3 = Ins(2, `a')$ *pour changer leurs états respectivement en "cirré", "cré" et "cairé". Au site* 1, *quand* op_2 *arrive, elle est d'abords transformée par rapport à* op_1, *i.e.* $op'_2 = IT(op_2, op_1) = Del(2)$ *et ensuite* op'_2 *est exécuté donnant lieu à l'état "crré". Sur le même site, l'intégration de* op_3 *passe par la transformation par rapport à la séquence* $[op_1; op'_2]$ *qui résulte en* $op''_3 = Ins(2, `a')$ *dont l'exécution mène à l'état "carré".*

Au niveau du site 2, op_1 *est en premier transformée par rapport à* op_2, *i.e.* $op'_1 = IT(op_1, op_2) = Ins(2, `r')$, *et ensuite le résultat* $op'_1$ *est exécutée produisant l'état "crré". Quand* $op_3$ *arrive, elle est transformée par rapport à* $[op_2; op'_1]$ *dont le résultat* $op'_3 = Ins(3, `a')$ *est ensuite exécuté produisant l'état "craré". On constate que la transformation de* op_3 *par rapport aux deux séquences équivalentes* $[op_1; op'_2]$ *et* $[op_2; op'_1]$ *donne deux opérations différentes, i.e* $op'_3 = Ins(3, `a')$ *et* $op''_3 = Ins(2, `a')$, *ce qui induit à une violation de* $CP2$. *Cette violation se traduit directement par une divergence de données puisque les états des sites* 2 *et* 3 *sont différents.* ■

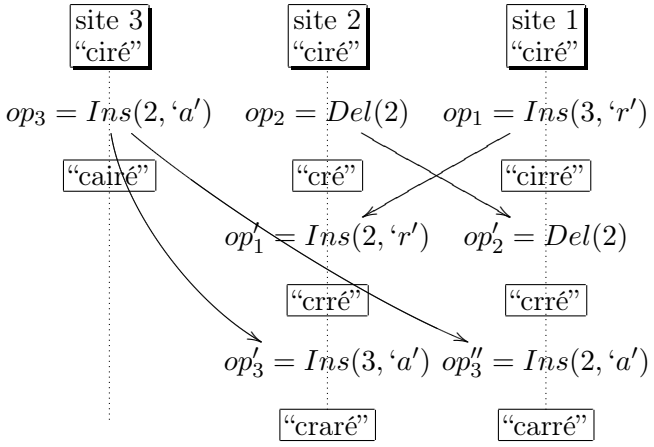


FIG. 2.6 – Un scénario réel violant la propriété $CP2$.

2.5 Outil VOTE

Dans cette section, nous allons présenter l'outil VOTE (Validation of Operational Transformation Environment), qui a été réalisé pour concevoir formellement des algorithmes de transformation utilisés par des objets collaboratifs [42].

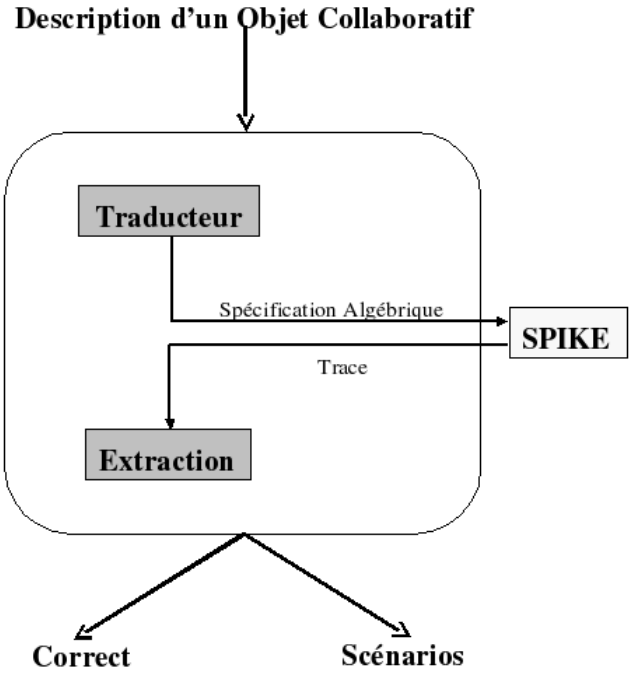


FIG. 2.7 – Architecture de VOTE.

2.5.1 Architecture

Entrée de l'outil

L'entrée de l'outil est une spécification rédigée dans un langage plus proche de l'algorithmique ; ce qui permet au développeur de décrire de manière concise les effets des opérations ainsi que l'algorithme de transformation. La figure 2.8 illustre la description de l'objet Texte avec l'algorithme de transformation proposé par Ellis et Gibb. La description de tout objet collaboratif procède donc par les étapes suivantes :

1. La déclaration des types de données.
2. La déclaration des attributs munis de leur profil.
3. La définition de chaque méthode est précédée par une expression booléenne indiquant les conditions de son application. A titre d'exemple (voir la figure 2.8), la méthode d'effacement de caractère $Del(p, pr)$ ne peut s'exécuter sur un état que si la position de suppression est inférieure à la longueur du texte.
4. Toutes les règles régissant un algorithme de transformation doivent être introduites de manière exhaustive. Ainsi pour le cas des méthodes, Ins et Del, il y a quatre règles. Il faut souligner que dans le cas de l'utilisation de la méthode nulle Nop, il n'est pas nécessaire de définir des règles de transformation pour cette méthode car elles seront générées automatiquement par VOTE, *i.e.* $IT(Nop, m) = Nop$ et $IT(m, Nop) = m$ pour toute méthode m.

5. Enfin, toutes les règles d'observation sont données pour définir comment les attributs sont affectés par l'exécution des méthodes. Le symbole $'$ (ou "prime") signifie que l'observation est liée à l'état produit par l'exécution d'une méthode.

Traduction

En concevant VOTE nous nous sommes fixé comme objectif de mettre à la disposition des utilisateurs un outil simple qui ne nécessite pas la connaissance d'une logique donnée. Aussi, la description introduite en entrée sera traduite automatiquement en une spécification algébrique. Toutes les règles de transformation ainsi que d'observation seront traduites en axiomes conditionnelles. A cet égard, VOTE permet d'assurer une définition complète des fonctions utilisées. A titre d'exemple, une expression telle que :

```
IT(Del(p1, pr1),Ins(p2, c2, pr2)) =
if (p1 < p2) then return Del(p1, pr1)
else return Del(p1+1, pr1)
endif;
```

est traduite en deux équations conditionnelles comme suit :

```
(p1 < p2) = true => IT(Del(p1, pr1),Ins(p2, c2, pr2)) = Del(p1, pr1)
(p1 < p2) = false => IT(Del(p1, pr1),Ins(p2, c2, pr2)) = Del(p1 + 1, pr1)
```

La spécification algébrique correspondant à la figure 2.8 est donnée à la figure 2.1.

Processus de vérification

Moyennant la spécification algébrique générée par l'étape de traduction, nous pouvons maintenant vérifier les propriétés de convergence $CP1$ et $CP2$. Pour ce faire, nous avons choisi d'utiliser le prouveur de théorèmes SPIKE. Le choix de ce prouveur est motivé par les raisons suivantes :

- SPIKE possède un haut degré d'automatisation de preuve ;
- sa capacité d'analyse par cas permet de traiter un nombre assez important de règles de transformation ;
- grâce à la propriété de complétude réfutationnelle, il est possible de trouver des contre-exemples pour les propriétés de convergence ;
- l'utilisation des procédures de décision, telle que l'arithmétique, permet d'accélérer le processus de preuve en éliminant automatiquement des tautologies appartenant à des théories décidables [5].

SPIKE commence d'abord par orienter les axiomes de la spécification algébrique en des règles de réécriture conditionnelle. Ensuite, la vérification des propriétés de convergence passe par deux possibilités :

[5] Par exemple, élimination des tautolgies comme $x+z > y = false \wedge z+x < y = false \implies x + z = y$.

– Soit la preuve réussit et dans ce cas on peut conclure que l'objet collaboratif en question est consistant, *i.e.* son algorithme de transformation satisfait les propriétés $CP1$ et $CP2$.
– Soit la preuve échoue et, en conséquence, la trace retournée par `SPIKE` est analysée pour extraire les cas problématiques qui ont conduit à l'échec de la preuve. Cette analyse peut révéler deux scénarios possibles. Le premier scénario est moins significatif car les propriétés à vérifier sont valides mais le prouveur échoue à établir cette validité. Pour pallier ce problème, il suffit juste d'introduire de nouveaux lemmes. Quant au deuxième scénario, il provient réellement de cas qui violent les propriétés de convergence. Aussi, `VOTE` donne toutes les informations nécessaires (les méthodes ainsi que des conditions sur leurs arguments) pour comprendre l'origine de la divergence de données.

2.5.2 Expérimentations

La méthode, que nous avons proposée pour spécifier et vérifier des objets collaboratifs, se révèle très efficace puisque elle nous a permis de déceler des cas de divergence de données dans plusieurs systèmes collaboratifs bien connus dans la littérature. Ces systèmes sont tous basés sur l'approche des transformées opérationnelles. La table 2.1 récapitule l'ensemble des systèmes pour lesquels nous avons vérifié les propriétés de convergence. Dans cette table nous avons :
– GROVE est un éditeur collaboratif de textes conçu par Ellis et Gibb [26].
– Joint EMACS est une version collaborative réalisée autour de l'éditeur Emacs [73].
– REDUCE[6] est système collaboratif pour éditer en temps réel des documents [85].
– CoWord[7] est une version collaborative réalisée autour du logiciel de traitement de textes Word de MicroSoft [86].
– SDT est un algorithme de transformation pour l'objet collaboratif Texte basé sur la compression des histoires d'opération [50].
– SAMS[8] est un éditeur collaboratif multi-synchrone basé sur des documents XML [61].
– *So6*[9] est un synchroniseur de fichiers [60].

2.6 Conclusion

Nous avons présenté une approche formelle pour détecter automatiquement une divergence de copies dans les systèmes collaboratifs. Pour garantir la convergence, l'algorithme de transformation doit être vérifié par rapport aux conditions $TP1$ et $TP2$. La preuve manuelle est difficile – voire impossible – à faire de par le nombre énorme des cas à considérer. Pour pallier ce problème, nous avons proposé un environnement formel pour assister à la conception des algorithmes

[6] `http://www.cit.gu.edu.au/~scz/projects/reduce`
[7] `http://www.cit.gu.edu.au/~scz/projects/coword/`
[8] `http://woinville.loria.fr/sams`
[9] `http://libresource.inria.fr/`

```
type nat, char;

attributes
  char car(nat);
  nat length();

methods
  p <= length() :  Ins(nat p, char c, nat pr);
  p < length()  : Del(nat p, nat pr);

transform
  IT(Ins(p1,c1,pr1), Ins(p2,c2,pr2)) =   IT(Ins(p1,c1,pr1), Del(p2,pr2)) =
    if (p1 < p2) then                         if (p1 < p2) then
      return Ins(p1,c1,pr1)                     return Ins(p1,c1,pr1)
    elseif (p1 > p2) then                     else
      return Ins(p1+1,c1,pr1)                   return Ins(p1-1,c1,pr1)
    elseif (c1 == c2) then                    endif;
      return Nop()
    elseif (pr1 > pr2) then             IT(Del(p1,pr1),Ins(p2,c2,pr2)) =
      return Ins(p1+1,c1,pr1)               if (p1 < p2) then
    else                                          return Del(p1,pr1)
      return Ins(p1,c1,pr1)                 else
    endif;                                        return Del(p1+1,pr1)
                                            endif;

  IT(Del(p1,pr1), Del(p2,pr2)) =
   if (p1 < p2) then
      return Del(p1,pr1)
   elseif (p1 > p2) then
      return Del(p1-1,pr1)
   else
      return Nop()
   endif;

observations
  car'(n)/Ins(p,c,pr) =                   car'(n)/Del(p,pr) =
   if (n == p) then                          if (n >= p) then
      return c                                  return car(n+1)
   elseif (n > p) then                       else
      return car(n-1)                           return car(n)
   else                                       end;
      return car(n)
   endif;

  length'()/Ins(p,c,pr) =                 length'()/Del(n,pr) =
   return length()+1;                        return length()-1;
```

FIG. 2.8 – Description de l'objet Texte dans VOTE.

Systèmes Collaboratifs	**CP1**	**CP2**
GROVE	non satisfaite	non satisfaite
Joint EMACS	non satisfaite	non satisfaite
REDUCE	satisfaite	non satisfaite
CoWord	satisfaite	non satisfaite
SDT	satisfaite	non satisfaite
SAMS	satisfaite	non satisfaite
So6	satisfaite	non satisfaite

TAB. 2.1 – Etudes de cas.

de transformation. Nous pensons que notre approche possède une valeur ajoutée car :

1. elle nous a permis de détecter des cas de divergence dans des systèmes collaboratifs très connus dans la littérature [39] ;
2. l'utilisation d'un prouveur automatique permet de balayer tous les cas possibles et produire automatiquement des contre-exemples ;
3. l'expertise pour l'utilisation d'un prouveur n'est pas tellement requise.

Il est vrai que l'utilisation d'un prouveur automatique a déplacé le problème de l'explosion combinatoire plus loin. Mais ce problème persiste toujours lorsque nous traitons des objets dont la structure est plus complexe (comme un système de fichiers ou un arbre XML ordonné). Plus la structure de l'objet est complexe, plus le nombre des ses méthodes est important. Notons qu'un objet complexe est souvent composé d'objets plus simples. Aussi, nous pensons que l'utilisation d'une technique compositionnelle pour concevoir un algorithme de transformation pour un objet complexe peut apporter une solution au problème de l'explosion combinatoire. Cela sera le sujet du chapitre suivant.

Chapitre 3

Composition des Objets Collaboratifs

Sommaire

3.1 Introduction

La conception d'un algorithme de transformation pour un objet collaboratif n'est pas une tâche facile au regard de la vérification des conditions de convergence $TP1$ et $TP2$. Même avec l'utilisation de techniques avancées de déduction automatique, le processus de vérification peut être ardu lorsque la structure de l'objet est complexe et le nombre d'opérations est important. En effet, plus le nombre d'opérations permettant de modifier l'état de l'objet collaboratif augmente, plus le nombre de triplets d'opérations concurrentes à considérer pour vérifier $TP2$ devient de plus en plus important. Et plus la structure de l'objet est complexe, plus le nombre de cas à vérifier pour $TP1$ ainsi que chaque triplet d'opérations de $TP2$ est considérable.

A titre d'exemple, pour un objet collaboratif dont l'état peut être modifié par n opérations, la vérification de $TP2$ nécessite le traitement de n^3 triplets d'opérations. Sans oublier que pour chaque triplet, il faut également considérer les cas inhérents aux possibles relations entre les paramètres des opérations. Par ailleurs, il faut souligner que les objets complexes sont souvent une agrégation d'objets plus simples.

Aussi, nous pensons que l'utilisation d'une technique basée sur le slogan "Diviser pour Régner" peut apporter une solution au problème de l'explosion

des cas à considérer. A ce titre, nous proposons dans ce chapitre une méthode compositionnelle pour spécifier et vérifier des objets collaboratifs complexes. La plus importante caractéristique de notre méthode est que la conception d'un algorithme de transformation pour un objet composite peut se faire par la *réutilisation* des algorithmes de transformation des objets composants. Ainsi, nous pouvons commencer une conception à partir de petits objets qui sont corrects (l'algorithme de transformation satisfait $TP1$ et $TP2$) et relativement faciles à vérifier. Ensuite, nous combinons ces objets de manière incrémentale pour produire des objets corrects plus complexes.

Pour spécifier les objets collaboratifs, nous avons utilisé des spécifications algébriques basées sur la sémantique observationnelle. Un certain nombre de travaux a vu le jour pour traiter de la combinaison des spécifications algébriques et notamment sous l'optique du modèle orienté-objet [34; 32; 24; 23; 22]. Nous sommes inspirés de ces travaux pour proposer un modèle formel de composition [38]. Deux constructions sont étudiées : une *composition statique* où le nombre d'objets composants est connu ; ainsi on peut concevoir un agenda partagé comme étant un objet complexe composé d'un nombre fixe d'objets texte. Une *composition dynamique* est caractérisée par la création et la suppression dynamiques d'objets composants. Ainsi un document XML peut être conçu comme un arbre dont les noeuds sont des objets texte.

La structure du chapitre se compose comme suit : la section 3.2 est consacrée à la première construction, à savoir la composition statique. Dans cette section, nous allons donner une définition formelle de cette composition ainsi que les conditions pour préserver les conditions $TP1$ et $TP2$. La composition dynamique sera traitée à la section 3.3. Nous allons présenter les étapes nécessaires pour réaliser une composition dynamique et les conditions pour conserver $TP1$ et $TP2$. Dans la section 3.4, nous allons illustrer notre approche par un exemple pratique concernant la construction d'un document de traitement de textes. Enfin, nous terminons le chapitre par une conclusion.

3.2 Composition Statique

La composition *statique* permet de construire un objet collaboratif à partir d'un nombre *fixe* (deux ou plusieurs) d'objets collaboratifs existants, de telle façon que la structure du nouvel objet reste figée durant l'exécution. Deux types de composition statique seront présentées selon le degré de concurrence qu'ils confèrent entre les objets composants.

Avant d'introduire ces types de composition, nous présentons quelques notions de base :

3.2.1 Notions de Base

Chaque objet collaboratif est considéré comme un composant :

Définition 3.1 (Composant de spécification). *Nous appellerons un composant de spécification, ou* composant*, toute spécification d'objet collaboratif* $\mathcal{C} = (\Sigma, M, A, T, E)$. ■

Dans ce qui suit, nous supposons que les composants sont (M, A)-complets.

Exemple 3.1 *Les composants* `CNAT` *et* `CCOLOR` *modélisent une cellule de mémoire qui peut stocker respectivement des entiers et des valeurs de type couleur :*

```
spec CNAT =
sort:
  Nat Meth State
opns:
  Do : Meth State -> State
  putnat : Nat -> Meth
  getnat : State -> Nat
  IT : Meth Meth -> Meth
axioms:
  (1) getnat(Do(putchar(n),st)) = n;
  (2) IT(putnat(n1),putnat(n2)) = putnat(minnat(n1,n2));

spec CCOLOR =
sort:
  Color Meth State
opns:
  Do : Meth State -> State
  putcolor : Color -> Meth
  getcolor : State -> Color
  IT : Meth Meth -> Meth
axioms:
  (1) getcolor(Do(putcolor(cl),st)) = cl;
  (2) IT(putcolor(cl1),putcolor(cl1)) = putcolor(mincolor(cl1,cl2));
```

Il faut noter que pour atteindre la convergence des données nous utilisons la fonction `minnat` *(respec.* `mincolor`*) qui retourne la valeur minimale de deux entiers (respec. de deux couleurs).* ■

Nous pouvons relier deux signatures par l'intermédiaire de la notion de morphismes [34; 32; 74].

Définition 3.2 (Σ-Morphisme). *Soient Σ_1 et Σ_2 deux OC-signatures. Un* Σ-morphisme $\Phi : \Sigma_1 \to \Sigma_2$ *est un morphisme de signatures tels que :*

1. $\Phi(s) = s$ *pour tout sorte* $s \in S_d$ *;*
2. $\Phi(f) = f$ *pour tout nom de fonction* $f \in \Sigma_{\omega,s}$ *où* $\omega \in S_d^*$ *et* $s \in S_d$ *;*
3. $\Phi(S_b) = S'_b$, *i.e.* $S'_b = \{$`State'`,`Meth'`$\}$, $\Phi($`State`$)=$ `State'` *et* $\Phi($`Meth`$)=$ `Meth'` *;*
4. $\Phi(Do) = Do$, $\Phi(IT) = IT$ *et* $\Phi(Poss) = Poss$. ■

Les trois premières conditions ci-dessus indiquent que les Σ-morphismes préservent les sortes de base (les sortes `State` et `Meth`), les sortes observables ainsi que les noms de fonctions dont le domaine et le codomaines sont sortés par les éléments de S_d. Les noms de fonctions Do (fonction de transition d'états), IT (fonction de transformation) et $Poss$ (fonction booléenne exprimant les conditions d'exécution d'une méthode), sont aussi préservées selon la quatrième condition.

Définition 3.3 ***(Morphisme de composants).*** *Etant donnés deux composants* $\mathcal{C}_1 = (\Sigma_1, M_1, A_1, T_1, E_1)$ *et* $\mathcal{C}_2 = (\Sigma_2, M_2, A_2, T_2, E_2)$*. Un* morphisme de composants $\Phi : \mathcal{C}_1 \to \mathcal{C}_2$ *est* Σ*-morphisme* $\Phi : \Sigma_1 \to \Sigma_2$ *tels que :*

1. $\Phi(M_1) \subseteq M_2$ *;*
2. $\Phi(A_1) \subseteq A_2$ *;*
3. $E_2 \models_{obs}^{\Sigma_2} \Phi(e)$ *pour tout axiome* $e \in E_1$*.* ■

La définition 3.3 fournit un support pour réutiliser des composants en utilisant la notion de morphisme de composants. En plus, comme nous utilisons que des composants (M, A)-complets alors la condition 3 ne nécessitera qu'un nombre fini d'équations à satisfaire.

3.2.2 Composition sans Synchronisation

Comme première construction, nous proposons la composition de plusieurs composants en un seul composite mais sans interaction (ou synchronisation) entre eux. En d'autres termes, chaque composant pourra changer indépendemment l'état de l'objet composite.

Définition 3.4 (Composition statique). *Soient* n *composants* $\mathcal{C}_i = (\Sigma_i, M_i, A_i, T_i, E_i)$ *avec* $i \in \{1, \ldots, n\}$ *et* $n > 1$*. Un composant* $\mathcal{C} = (\Sigma, M, A, T, E)$ *est dit une* composition parallèle *de* $\mathcal{C}_i$ *ssi il existe un morphisme de composants* $\Phi_i : \mathcal{C}_i \to \mathcal{C}$ *tel que pour tout* $i \neq j \in \{1, \ldots, n\}$ *:*

(i). $\Sigma = \bigcup_i \Phi_i(\Sigma_i)$ *;*

(ii). $M = \bigcup_i \Phi_i(M_i)$ *;*

(iii). $A = \bigcup_i \Phi_i(A_i)$ *;*

(iv). $T = \bigcup_i \Phi_i(T_i) \cup \bigcup_{i,j} T_{ij}$ *où :*

$$T_{ij} = \{IT(\Phi_i(m_i), \Phi_j(m_j)) = \Phi_i(m_i) \mid m_i \in M_i \text{ et } m_j \in M_j\}$$

(v). $E = \bigcup_i \Phi_i(E_i) \cup IE$ *où* IE *est appelée* interaction *entre* $\mathcal{C}_i$*, et elle est définie comme* $IE = \bigcup_{i,j}(T_{ij} \cup A_{ij})$ *avec :*

$$A_{ij} = \{\Phi_i(a_i)(Do(\Phi_j(m_j), x)) = \Phi_i(a_i)(x) \mid a_i \in A_i \text{ et } m_j \in M_j\}$$

Nous notons la composition parallèle $\mathcal{C} = \bigoplus_i \mathcal{C}_i$*.* ■

Exemple 3.2 *Considérons les composants* `CCHAR` $= (\Sigma_1, M_1, A_1, T_1, E_1)$*,* `CNAT` $= (\Sigma_2, M_2, A_2, T_2, E_2)$ *et* `CCOLOR` $= (\Sigma_3, M_3, A_3, T_3, E_3)$ *(voir l'exemple 3.1) qui modélisent des cellules de mémoire qui peuvent stocker des valeurs de type respectivement caractère, entier et couleur. La composition parallèle de* `CCHAR` *et* `CNAT` *est le composite* `SIZEDCHAR` $= (\Sigma, M, A, T, E)$ *(see Figure 3.1(a)) :*

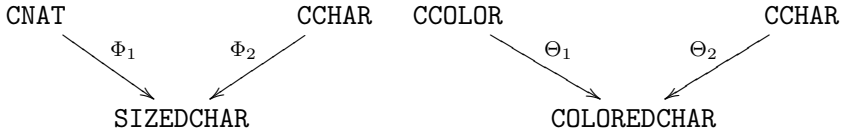
```
spec SIZEDCHAR =
sort:
 Char Nat Meth State
opns:
 Do : Meth State -> State
 putchar : Char -> Meth
 putnat  : Nat  -> Meth
 getchar : State -> Char
 getnat  : State -> Nat
 IT : Meth Meth -> Meth
axioms:
 (1) getchar(Do(putchar(c),st)) = c;
 (2) getnat(Do(putnat(n),st)) = n;
 (3) getchar(Do(putnat(n),st))=getchar(st);
 (4) getnat(Do(putchar(c),st)) = getnat(n);
 (5) IT(putchar(c1),putchar(c2)) = putchar(maxchar(c1,c2));
 (6) IT(putnat(n1),putnat(n2)) = putchar(minnat(n1,n2));
 (7) IT(putchar(c1),putnat(n1)) = putchar(c1);
 (8) IT(putnat(n1),putchar(c1)) = putnat(n1);
```

où T_{ij} contient les axiomes (7)-(8) *et A_{ij} est donné par les axiomes* (3)-(4) *pour tout i, $j \in \{1, 2\}$ et $i \neq j$. Cette composition parallèle peut être assimilée à la construction d'un objet collaboratif qui a deux attributs : une valeur de type caractère et une taille de police de caractère. Ces attributs sont modifiés indépendemment par des méthodes différentes. Par ailleurs, il faut noter que les morphismes de composant $\Phi_1 : \mathcal{C}_1 \to \mathcal{C}$ et $\Phi_2 : \mathcal{C}_2 \to \mathcal{C}$ sont tout simplement des morphismes d'inclusion.*

De la même manière, la composition parallèle de CCHAR *et* CCOLOR *produit le composite* COLOREDCHAR $= (\Sigma, M, A, T, E)$ *qui modélise un objet collaboratif ayant une valeur caractère et une couleur (see Figure 3.1(b)) :*

```
spec COLOREDCHAR =
sort:
 Char Color Meth State
opns:
 Do : Meth State -> State
 putchar : Char -> Meth
 putcolor  : Color  -> Meth
 getchar : State -> Char
 getcolor  : State -> Color
 IT : Meth Meth -> Meth
axioms:
 (1) getchar(Do(putchar(c),st)) = c;
 (2) getcolor(Do(putcolor(cl),st)) = cl;
 (3) getchar(Do(putcolor(cl),st))=getchar(st);
 (4) getcolor(Do(putchar(c),st)) = getcolor(st);
 (5) IT(putchar(c1),putchar(c2)) = putchar(maxchar(c1,c2));
 (6) IT(putcolor(cl1),putcolor(cl2)) = putcolor(mincolor(n1,n2));
 (7) IT(putchar(c1),putcolor(cl1)) = putchar(c1);
 (8) IT(putcolor(cl1),putchar(c1)) = putcolor(cl1);
```

■



(a) Première Composition (b) Deuxième Composition

FIG. 3.1 – Composition sans synchronisation.

Nous allons maintenant définir comment percevoir la *vraie concurrence* (ou la *commutativité* des méthodes) entre n composants combinés en un seul composite selon la définition 3.4.

Définition 3.5 (Composants indépendants). *Soit* $\mathcal{C} = \bigoplus_i \mathcal{C}_i$ *tels que* $i \in \{1, \ldots, n\}$ *et* $n > 1$. *Les composants* $\mathcal{C}_i$ *sont dits* indépendants *ssi :*

$$\mathcal{C} \models^{\Sigma}_{obs} (s)[\Phi_i(m_i); \Phi_j(m_j)] = (s)[\Phi_j(m_j); \Phi_i(m_i)]$$

où $m_i \in M_i$ *et* $m_j \in M_j$ *avec* $i \neq j \in \{1, \ldots, n\}$. ■

Dans l'exemple 3.2, il peut être facilement montré que les composants `CCHAR` et `CNAT` sont indépendants puisque l'équation suivante :

```
Do(putchar(c),Do(putnat(n),s))=Do(putnat(n),Do(putchar(c),s))
```

est une conséquence observable de `SIZEDCHAR`.

Dans ce qui suit, nous allons donner les principales caractéristiques de la composition parallèle sans interaction. Tout d'abords, nous allons montrer que ce type de composition permet de préserver les propriétés de convergence des composants (théorème 3.1).

Théorème 3.1 *Soient les composants* $\mathcal{C}_i$ *et* $\mathcal{C} = \bigoplus_i \mathcal{C}_i$ *(pour* $i \in \{1, \ldots, n\}$ *et* $n > 1$*), tels que* $\mathcal{C}_i$ *sont indépendants. Si* $\mathcal{C}_i$ *sont consistants alors* $\mathcal{C} \models^{\Sigma}_{obs} CP1|_{\Phi_i(M_i)} \wedge CP2|_{\Phi_i(M_i)}$. ■

Preuve. Nous considérons deux cas :

(1) *preuve de* $\mathcal{C} \models^{\Sigma}_{obs} CP1|_{\Phi_i(M_i)}$ *:* nous devons montrer que pour tout Σ_i-contexte observable c_i et pour tout Σ-contexte observable c nous avons :

$$\mathcal{C}_i \models^{\Sigma_i} c_i[CP1|_{M_i}] \text{ implique } \mathcal{C} \models^{\Sigma} c[CP1|_{\Phi_i(M_i)}]$$

avec $i \in \{1, \ldots, n\}$. Notons que c_i a la forme suivante :

$$a_i((st)[m_{i_1}; m_{i_2}; \ldots; m_{i_p}])$$

avec $p \geq 0$, m_{i_1}, m_{i_2}, ..., $m_{i_p} \in M_i$, et $a_i \in A_i$. De la même manière, c est :

$$a((st)[m_1; m_2; \ldots; m_q])$$

avec $q \geq 0$, m_1, m_2, ..., $m_q \in M$, et $a \in A$.

Deux cas sont à considérer :

(a) $c = \Phi_i(c_i)$: dans ce cas nous avons

$$\mathcal{C}_i \models^{\Sigma_i} c_i[CP1|_{M_i}] \text{ implique } \mathcal{C} \models^{\Sigma} \Phi_i(c_i)[CP1|_{\Phi_i(M_i)}]$$

qui résulte du théorème 1.1 (condition de satisfaction) et du fait que $\Phi_i(c_i[CP1|_{M_i}]) = \Phi_i(c_i)[CP1|_{\Phi_i(M_i)}]$.

(b) $c \neq \Phi_i(c_i)$: ceci signifie que a n'est pas une image de a_i par Φ_i. Sans perte de généralité, supposons que dans c les méthodes m_1, m_2, ..., m_l (pour $0 \leq l < q$) sont les images par Φ_i de certaines méthodes dans M_i. Ainsi $c[CP1|_{\Phi_i(M_i)}]$ est réécrite comme suit :

$$a((st)[\Phi_i(m); IT(\Phi_i(m'), \Phi_i(m)); m_1; \ldots; m_l; \ldots; m_q]) = a((st)[\Phi_i(m'); IT(\Phi_i(m), \Phi_i(m')); m_1; \ldots; m_l; \ldots; m_q])$$

où m, $m' \in M_i$. Comme les composants $\mathcal{C}_i$ ($i \in \{1, \ldots, n\}$) sont indépendants, alors les méthodes m_1, m_2, ..., m_l sont commutatives avec m_{l+1}, ..., m_q ; dans ce cas nous avons :

$$a((st)[m_{l+1}; \ldots; m_q; \Phi_i(m); IT(\Phi_i(m'), \Phi_i(m)); m_1; \ldots; m_l;]) = a((st)[m_{l+1}; \ldots; m_q; \Phi_i(m'); IT(\Phi_i(m), \Phi_i(m')); m_1; \ldots; m_l])$$

qui est vraie en utilisant la condition (v) de la définition 3.4.

(2) *preuve de* $\mathcal{C} \models_{obs} CP2|_{\Phi_i(M_i)}$: puisque $CP2$ ne contient pas de termes de sorte `State`, alors nous devons montrer :

$$\mathcal{C}_i \models^{\Sigma_i} CP2|_{M_i} \text{ implique } \mathcal{C} \models^{\Sigma} CP2|_{\Phi_i(M_i)}$$

qui résulte du théorème 1.1 en se basant sur le fait que $\Phi_i(CP2|_{M_i}) = CP2|_{\Phi_i(M_i)}$. ■

Le théorème 3.2 est très important dans le sens où il stipule que la propriété de consistance (voir la définition 2.10) peut être obtenue par construction. En effet, la composition parallèle de plusieurs composants consistants produit un composite consistant à condition que les composants sont indépendants.

Théorème 3.2 *Soient les composants* $\mathcal{C}_i$ *et* $\mathcal{C} = \bigoplus_i \mathcal{C}_i$ *tels que* $i \in \{1, \ldots, n\}$ *et* $n > 1$. *Si les composants* $\mathcal{C}_i$ *sont consistants et indépendants alors* $\mathcal{C}$ *est aussi consistant.* ■

Preuve. Soit $\mathcal{C}$ la composition parallèle de $\mathcal{C}_1$, $\mathcal{C}_2$, ..., $\mathcal{C}_n$ avec $n > 1$. Par définition, $\mathcal{C}$ est consistant ssi $\mathcal{C} \models^{\Sigma}_{obs} CP1 \wedge CP2$.

(1) *preuve de* $\mathcal{C} \models^{\Sigma}_{obs} CP1$ *:* la propriété de convergence $CP1$ peut être formulée comme suit :

$$CP1 \equiv \Phi_i(CP1|_{M_i}) \wedge CP1|_{\Phi_i(M_i), \Phi_j(M_j)}$$

avec $i \neq j \in \{1, \ldots, n\}$. Puisque les composants $\mathcal{C}_i$ sont consistants alors $\mathcal{C} \models^{\Sigma}_{obs} \Phi_i(CP1|_{M_i})$ est une conséquence du théorème 3.1 et du fait que $\Phi_i(CP1|_{M_i}) = CP1|_{\Phi_i(M_i)}$). En utilisant la définition 3.4, $\mathcal{C} \models^{\Sigma}_{obs} CP1|_{\Phi_i(M_i),\Phi_j(M_j)}$ est réécrite en :

$$\mathcal{C} \models_{obs} (s)[\Phi_i(m_i); \Phi_j(m_j)] = (s)[\Phi_j(m_j); \Phi_i(m_i)]$$

(où $m_i \in M_i$ et $m_j \in M_j$) qui est valide parce que les composants are indépendants.

(2) *preuve de* $\mathcal{C} \models^{\Sigma}_{obs} CP2$. La propriété de convergence $CP2$ peut être exprimée comme suit :

$$CP2 \equiv \Phi_i(CP2|_{M_i}) \wedge CP2|_{\Phi_i(M_i),\Phi_j(M_j)}$$

avec $i \neq j \in \{1, \ldots, n\}$. Comme les composants $\mathcal{C}_i$ sont consistants, alors $\mathcal{C} \models^{\Sigma}_{obs} \Phi_i(CP2|_{M_i})$ est une conséquence du théorème 3.1. Quant à $\mathcal{C} \models^{\Sigma}_{obs} CP2|_{\Phi_i(M_i),\Phi_j(M_j)}$, elle est réécrite en :

$$IT^*(\Phi_{k_1}(m), [\Phi_{i_1}(m'); IT(\Phi_{j_1}(m''), \Phi_{i_1}(m'))]) = IT^*(\Phi_{k_1}(m), [\Phi_{j_1}(m''); IT(\Phi_{i_1}(m'), \Phi_{j_1}(m''))])$$

où $m' \in M_{i_1}$, $m'' \in M_{j_1}$ et $m \in M_{k_1}$ pour tout $i_1, j_1, k_1 \in \{i, j\}$ tel que $k_1 \neq i_1$ or $k_1 \neq j_1$. Deux cas sont possibles :

(a) $i_1 = j_1$ et $k_1 \neq i_1$: comme les composants $\mathcal{C}_{k_1}$ et $\mathcal{C}_{i_1}$ sont indépendants

$$IT^*(\Phi_{k_1}(m), [\Phi_{i_1}(m'); \Phi_{i_1}(IT(m'', m'))]) = \Phi_{k_1}(m)$$

et

$$IT^*(\Phi_{k_1}(m), [\Phi_{i_1}(m''); \Phi_{i_1}(IT(m', m''))]) = \Phi_{k_1}(m)$$

(b) $i_1 \neq j_1$ et ($k_1 = i_1$ or $k_1 = j_1$) : considérons le cas où $k_1 = i_1$ (le cas $k_1 = j_1$ est similaire). Comme les composants $\mathcal{C}_{i_1}$ et $\mathcal{C}_{j_1}$ sont indépendants

$$IT^*(\Phi_{i_1}(m), [\Phi_{i_1}(m'); \Phi_{j_1}(m'')]) = IT(\Phi_{i_1}(IT(m, m')), \Phi_{j_1}(m'')) = \Phi_{i_1}(IT(m, m'))$$

et

$$IT^*(\Phi_{i_1}(m), [\Phi_{j_1}(m''); \Phi_{i_1}(m')]) = \Phi_{i_1}(IT(m, m'))$$

■

On peut facilement montrer que les composants `CCHAR`, `CNAT` et `CCOLR` sont consistants (voir l'exemple 3.2). Comme ces composants sont aussi indépendants, alors `SIZEDCHAR` et `COLOREDCHAR` sont par construction consistants.

Il faut noter que si les composants sont semi-consistants alors la composition est aussi semi-consistante.

Corollaire 3.1 *Soient les composants* $\mathcal{C}_i$ *et* $\mathcal{C} = \bigoplus_i \mathcal{C}_i$ *tels que* $i \in \{1, \ldots, n\}$ *et* $n > 1$. *Si les composants* $\mathcal{C}_i$ *sont semi-consistants et indépendants alors* $\mathcal{C}$ *est aussi semi-consistant.* ■

Par conséquent, si l'un des composants indépendants et semi-consistants $\mathcal{C}_i$ est aussi consistant alors la composition demeure toujours semi-consistante.

3.2.3 Composition avec Synchronisation

Comme deuxième construction, nous permettons maintenant à des composants à interagir entre eux. Il est possible d'avoir des situations où des méthodes et des attributs sont partagés par des composants. Ces derniers sont dits *synchronisés* par leur partie commune. Une telle partie partagée est juste un moyen pour permettre aux composants de communiquer.

Définition 3.6 (Composant partageable). *Soient $\mathcal{C}_i$ des composants pour $i \in \{0, \ldots, n\}$ et $n > 1$. Le composant $\mathcal{C}_0$ est appelé* composant partageable *de $\mathcal{C}_1$, …, $\mathcal{C}_n$ ssi il existe une famille de morphismes de composants $\Theta_i : \mathcal{C}_0 \to \mathcal{C}_i$ telle que pour toutes les méthodes $m \in M_0$ et $m' \in M_i \setminus \Theta_i(M_0)$ et pour tous les attributs $a \in A_0$ et $a' \in A_i \setminus \Theta_i(A_0)$ nous avons :*

(i). $\mathcal{C}_i \models^{\Sigma_i}_{obs} IT(\Theta_i(m), m') = \Theta_i(m)$;

(ii). $\mathcal{C}_i \models^{\Sigma_i}_{obs} IT(m', \Theta_i(m)) = m'$;

(iii). $\mathcal{C}_i \models^{\Sigma_i}_{obs} a'(Do(\Theta_i(m), x)) = a'(x)$;

(iv). $\mathcal{C}_i \models^{\Sigma_i}_{obs} \Theta_i(a)(Do(m', x)) = \Theta_i(a)(x)$;

(v). $\mathcal{C}_i \models^{\Sigma_i}_{obs} (s)[\Theta_i(m); m'] = (s)[m'; \Theta_i(m)]$. ■

Les conditions $(i) - (v)$ indiquent comment les composants $\mathcal{C}_i$ et $\mathcal{C}_0$ interagissent entre eux. En fait, il n'y a pas d'interférence entre eux et leurs méthodes sont commutatives d'après la condition (v).

Définition 3.7 (Composition statique avec synchronisation). *Soient les composants $\mathcal{C}_0$, $\mathcal{C}_1$, …, $\mathcal{C}_n$, tel que $\mathcal{C}_0$ est un composant partageable des composants $\mathcal{C}_i$ avec $i \in \{1, \ldots, n\}$ et $n > 1$. Un composant $\mathcal{C}$ est dit la* composition synchronisée *de $\mathcal{C}_i$ ssi il existe des morphismes de composants $\Theta_i : \mathcal{C}_0 \to \mathcal{C}_i$ et $\Phi_i : \mathcal{C}_i \to \mathcal{C}$ pour $i \in \{1, \ldots, n\}$ telles que les conditions suivantes sont satisfaites :*

(i). $\Theta_i \circ \Phi_i = \Theta_j \circ \Phi_j$ *avec* $i \neq j$ *et* $j \in \{1, \ldots, n\}$;

(ii). $\mathcal{C} = \bigoplus_i \mathcal{C}_i$.

Nous notons la composition synchronisée $\mathcal{C} = \bigoplus_i^{\mathcal{C}_0} \mathcal{C}_i$. ■

La signature de la composition synchronisée est construite à partir des signatures des composants. Il devrait être noté que la duplication des méthodes et des attributs du composant partageable n'est pas admise, comme indiquée par la condition (i) de la définition 3.7.

Exemple 3.3 *Considérons les composants donnés dans l'exemple 3.2. Il faut juste rappeler que* `SIZEDCHAR` *est un objet caractère avec un attribut pour la taille de police de caractère, et* `COLOREDCHAR` *est un autre objet caractère avec un attribut couleur. Ces objets ont une partie commune à savoir* `CCHAR` *(voir la Figure 3.2). Ainsi la composition de* `SIZEDCHAR` *et* `COLOREDCHAR` *impose donc une synchronisation par rapport à* `CCHAR`*. Nous obtenons la spécification suivante :*

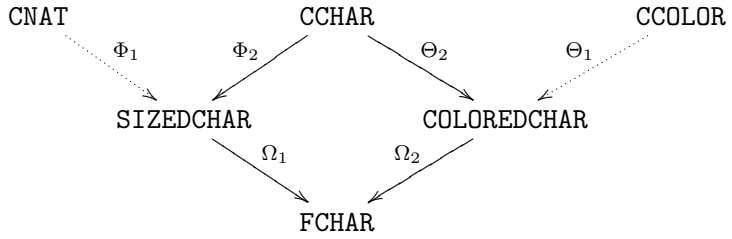


FIG. 3.2 – Une composition synchronisée.

```
spec FCHAR =
sort:
 Char Color Nat Meth State
opns:
 Do : Meth State -> State
 putchar : Char -> Meth
 putnat : Nat -> State
 putcolor  : Color  -> Meth
 getchar : State -> Char
 getnat : State -> Nat
 getcolor  : State -> Color
 IT : Meth Meth -> Meth
axioms:
 (1)  getchar(Do(putchar(c),st)) = c;
 (2)  getnat(Do(putnat(n),st)) = n;
 (3)  getcolor(Do(putcolor(cl),st)) = cl;
 (4)  getchar(Do(putcolor(cl),st))=getchar(st);
 (5)  getchar(Do(putnat(n),st)) = getchar(st);
 (6)  getnat(Do(putchar(c),st)) = getnat(st);
 (7)  getcolor(Do(putchar(c),st)) = getcolor(st);
 (8)  getnat(Do(putcolor(cl),st)) = getnat(st);
 (9)  getcolor(Do(putnat(n),st)) = getcolor(st);
 (10) IT(putchar(c1),putchar(c2)) = putchar(maxchar(c1,c2));
 (11) IT(putnat(n1),putnat(n2)) = putnat(maxnat(n1,n2));
 (12) IT(putcolor(cl1),putcolor(cl2)) = putcolor(maxcolor(cl1,cl2));
 (13) IT(putchar(c1),putcolor(cl1)) = putchar(c1);
 (14) IT(putcolor(cl1),putchar(c1)) = putcolor(cl1);
 (15) IT(putchar(c1),putnat(n1)) = putchar(c1);
 (16) IT(putnat(n1),putchar(c1)) = putnat(n1);
 (17) IT(putcolor(cl1),putnat(n1)) = putcolor(cl1);
 (18) IT(putnat(n1),putcolor(cl1)) = putnat(n1);
```

où la partie interaction IE contient les axiomes (4)-(9) *et* (13)-(18)*. Notons que Ω_1 et Ω_2 sont juste des morphismes d'inclusion telle que $\Phi_2 \circ \Omega_1 = \Theta_2 \circ \Omega_2$.* ■

Comme vu précédemment, les composants d'une composition synchronisée possèdent un composant partageable à partir duquel ils sont synchronisés. Aussi, la vraie concurrence est définie seulement sur les parties disjointes des composants.

Définition 3.8 (Composants indépendants). *Soit $\mathcal{C} = \bigoplus_i^{\mathcal{C}_0} \mathcal{C}_i$ pour $i \in$*

$\{1,\ldots,n\}$ *et* $n > 1$. *Les composants* $\mathcal{C}_i$ *sont dits* indépendants *ssi :*

$$\mathcal{C} \models^{\Sigma}_{obs} (s)[\Phi_i(m_i);\Phi_j(m_j)] = (s)[\Phi_j(m_j);\Phi_i(m_i)]$$

tels que $m_i \in M_i \setminus \Theta_i(M_0)$ *et* $m_j \in M_j \setminus \Theta_j(M_0)$ *avec* $i \neq j \in \{1,\ldots,n\}$. ■

Dans l'exemple 3.3, `SIZEDCHAR` et `COLOREDCHAR` ont un composant partageable `CCHAR` et ils sont indépendants selon la définition 3.8. En effet, l'équation suivante :

```
Do(putcolor(c),Do(putnat(n),s))=Do(putnat(n),Do(putcolor(c),s))
```

est une conséquence observable de `FCHAR`.

Dans ce qui suit, nous allons donner les caractéristiques de la composition synchronisée. Le théorème 3.3 stipule que les propriétés de convergence des composants sont préservées.

Théorème 3.3 *Etant donnés les composants* $\mathcal{C}_i$ *et* $\mathcal{C} = \bigoplus_i^{\mathcal{C}_0} \mathcal{C}_i$ *pour* $i \in \{1,\ldots,n\}$ *et* $n > 1$, *tels que* $\mathcal{C}_i$ *sont indépendants. Si les composants* $\mathcal{C}_i$ *sont consistants alors :*

$$\mathcal{C} \models^{\Sigma}_{obs} CP1|_{\Phi_i(M_i)} \wedge CP2|_{\Phi_i(M_i)}.$$

■

Preuve. Similaire à celle du théorème 3.1. ■

Des composants consistants ayant un composant partageable peuvent être combinés en un seul composant consistant.

Théorème 3.4 *La composition synchronisée de* n $(n > 1)$ *composants consistants et indépendants est aussi consistante.* ■

Preuve. Soit $\mathcal{C}$ la composition synchronisée de $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ $(n > 1)$ qui ont $\mathcal{C}_0$ comme un composant partageable, *i.e.* il existe des morphismes de composants $\Theta_i : \mathcal{C}_0 \to \mathcal{C}_i$ et $\Phi_i : \mathcal{C}_i \to \mathcal{C}$ pour $i \in \{1,\ldots,n\}$. Notons que $\Phi_i(M_i) \cap \Phi_j(M_j) = \Phi_i(\Theta_i(M_0)) = \Phi_j(\Theta_j(M_0))$ pour $i \neq j \in \{1,\ldots,n\}$. Soit la notation suivante $\overline{M_i} = M_i \setminus \Theta_i(M_0)$. Par définition, $\mathcal{C}$ est consistant ssi $\mathcal{C} \models^{\Sigma}_{obs} CP1 \wedge CP2$.

(1) *preuve de* $\mathcal{C} \models^{\Sigma}_{obs} CP1$ *:* $CP1$ est exprimée comme suit :

$$CP1 \equiv \Phi_i(CP1|_{M_i}) \wedge CP1|_{\Phi_i(\overline{M_i}),\Phi_j(\overline{M_j})}$$

avec $i \neq j \in \{1,\ldots,n\}$. Puisque les composants $\mathcal{C}_i$ sont consistants alors $\mathcal{C} \models^{\Sigma}_{obs} \Phi_i(CP1|_{M_i})$ résulte du théorème 3.3 et du fait que $\Phi_i(CP1|_{M_i}) = CP1|_{\Phi_i(M_i)})$. En utilisant la définition 3.4, $\mathcal{C} \models^{\Sigma}_{obs} CP1|_{\Phi_i(\overline{M_i}),\Phi_j(\overline{M_j})}$ est réécrite comme suit :

$$\mathcal{C} \models^{\Sigma}_{obs} (s)[\Phi_i(m_i);\Phi_j(m_j)] = (s)[\Phi_j(m_j);\Phi_i(m_i)]$$

(où $m_i \in \overline{M_i}$ et $m_j \in \overline{M_j}$) qui est valide car les composants $\mathcal{C}_i$ sont indépendants pour $i \in \{1,\ldots,n\}$.

(2) *preuve de* $\mathcal{C} \models_{obs} CP2$. $CP2$ peut être formulée comme suit :

$$CP2 \equiv \Phi_i(CP2|_{M_i}) \wedge CP2|_{\Phi_i(\overline{M_i}),\Phi_j(\overline{M_j})}$$

avec $i \neq j \in \{1, \ldots, n\}$. Remarquons que $\mathcal{C} \models_{obs} \Phi_i(CP2|_{M_i})$ est une conséquence du théorème 3.3 et du fait que les composants $\mathcal{C}_i$ sont consistants. Quant à $\mathcal{C} \models_{obs} CP2|_{\Phi_i(\overline{M_i}),\Phi_j(\overline{M_j})}$, elle est réécrite en :

$$IT^*(\Phi_{k_1}(m), [\Phi_{i_1}(m'); IT(\Phi_{j_1}(m''), \Phi_{i_1}(m'))]) =$$
$$IT^*(\Phi_{k_1}(m), [\Phi_{j_1}(m''); IT(\Phi_{i_1}(m'), \Phi_{j_1}(m''))])$$

où $m' \in \overline{M_{i_1}}$, $m'' \in \overline{M_{j_1}}$ et $m \in \overline{M_{k_1}}$ pour tout $i_1, j_1, k_1 \in \{i, j\}$ tel que $k_1 \neq i_1$ ou $k_1 \neq j_1$. Deux cas sont possibles :

(a) $i_1 = j_1$ et $k_1 \neq i_1$: comme les composants $\mathcal{C}_{k_1}$ et $\mathcal{C}_{i_1}$ sont indépendants alors

$$IT^*(\Phi_{k_1}(m), [\Phi_{i_1}(m'); \Phi_{i_1}(IT(m'', m'))]) = \Phi_{k_1}(m)$$
and
$$IT^*(\Phi_{k_1}(m), [\Phi_{i_1}(m''); \Phi_{i_1}(IT(m', m''))]) = \Phi_{k_1}(m)$$

(b) $i_1 \neq j_1$ et ($k_1 = i_1$ ou $k_1 = j_1$) : considérons le cas où $k_1 = i_1$ (le cas $k_1 = j_1$ est similaire). Puisque les composants $\mathcal{C}_{i_1}$ et $\mathcal{C}_{j_1}$ sont indépendants alors

$$IT^*(\Phi_{i_1}(m), [\Phi_{i_1}(m'); \Phi_{j_1}(m'')]) = IT(\Phi_{i_1}(IT(m, m')), \Phi_{j_1}(m'')) =$$
$$\Phi_{i_1}(IT(m, m')) \text{ and}$$
$$IT^*(\Phi_{i_1}(m), [\Phi_{j_1}(m''); \Phi_{i_1}(m')]) = \Phi_{i_1}(IT(m, m'))$$

■

Dans l'exemple 3.3, `CCHAR`, `CNAT` et `CCOLOR` sont consistants et indépendants. Par composition parallèle, `SIZEDCHAR` = `CNAT`⊕`CCHAR` et `COLOREDCHAR` = `CCOLOR`⊕`CCHAR` sont consistants. Finalement, par composition synchronisée, `FCHAR` = `SIZEDCHAR`$\oplus^{CCHAR}$`COLOREDCHAR` est aussi consistant.

Il faut souligner que si les composants sont semi-consistants alors la composition est aussi semi-consistante.

Corollaire 3.2 *La composition synchronisée de n (n > 1) composants semi-consistants et indépendants est aussi semi-consistante.*

Par conséquent, si l'un des composants indépendants et semi-consistants $\mathcal{C}_i$ est aussi consistant alors la composition demeure toujours semi-consistante.

3.3 Composition Dynamique

La construction proposée dans cette section permet de composer un nombre arbitraire d'un même objet collaboratif selon une structure donnée (que nous appellerons par la suite un *patron de composition*). En d'autres termes, l'objet en question est crée et/ou supprimé de manière dynamique. De ce fait, l'objet obtenu a une structure qui n'est pas figée durant l'exécution.

Définition 3.9 (Patron de Composition). *Un* patron de composition *est une spécification paramétrée* $\overline{\mathcal{C}} = (PA, \mathcal{C})$ *où :*

- $PA = (\Sigma_{PA}, E_{PA})$*, appelée* paramètre formel*, est une spécification algébrique ;*
- $\mathcal{C} = (\Sigma, M, A, T, E)$*, appelée* corps*, est une spécification d'un objet collaboratif (ou un composant) ;*

telles que les conditions suivantes sont satisfaites :

1. $S_{PA} = \{Elem, Bool\}$ *;*
2. $\Sigma_{PA} \subset \Sigma$ *;*
3. $E_{PA} \subset E$ *;*
4. *il existe une méthode* $m \in M$ *dont l'un de ses arguments est de sorte* $Elem$ *;* m *est appelée* méthode paramétrique *;*
5. *il existe un attribut* $a \in A$ *dont soit son résultat est de sorte* $Elem$ *ou soit l'un de ses arguments est de sorte* $Elem$ *;* a *est appelé* attribut paramétrique*.*

Pour désigner de tels patrons de composition, nous utilisons les symboles $\overline{\mathcal{C}}$, $\overline{\mathcal{C}}'$, $\overline{\mathcal{C}}_1$, $\overline{\mathcal{C}}_2$, ... ■

Exemple 3.4 *Le patron de composition* PSET $= (PA, \mathcal{C})$ *décrit les propriétés des ensembles finis qui sont paramétrés par leur contenu :*

Paramètre formel PA *:*

```
spec PA =
sorts:
 Elem Bool
opns:
 eq : Elem Elem -> Bool
axioms:
 (1) eq(x,y)=eq(y,x);
 (2) eq(x,y)=true, eq(y,z)=true => eq(x,z)=true;
```

Corps $\mathcal{C}$ *:*

```
spec C =
sorts:
 Set
opns:
 empty : -> Set
 Do : Meth Set -> Set
 nop : -> Meth
 add : Elem -> Meth
 remove : Elem -> Meth
 Poss : Meth Set -> Bool
 iselem : Elem Set -> Bool
 IT : Meth Meth -> Meth
axioms:
 (1)  Poss(nop,st)=true;
 (2)  Poss(add(x),st)=true;
 (3)  iselem(x,st)=true  => Poss(remove(x),st)=true;
```

```
(4)  iselem(x,st)=false => Poss(remove(x),st)=false;
(5)  eq(x,y)=true  => iselem(x,Do(add(y),st))=true;
(6)  eq(x,y)=false => iselem(x,Do(add(y),st))=iselem(x,st);
(7)  eq(x,y)=true  => iselem(x,Do(remove(y),st))=false;
(8)  eq(x,y)=false => iselem(x,Do(remove(y),st))=iselem(x,st);
(9)  eq(x,y)=true  => IT(add(x),add(y))=nop;
(10) eq(x,y)=false => IT(add(x),add(y))=add(x);
(11) IT(add(x),remove(y))=add(x);
(12) eq(x,y)=true  => IT(remove(x),remove(y))=nop;
(13) eq(x,y)=false => IT(remove(x),remove(y))=remove(x);
(14) IT(remove(x),add(y))=remove(x);
```
■

Dans la définition qui suit, nous donnons les conditions pour qu'un composant puisse potentiellement substituer un paramètre formel d'un patron de composition.

Définition 3.10 (Admissibilité). *Soient $\overline{\mathcal{C}} = (PA, \mathcal{C})$ un patron de composition et $\mathcal{C}_1 = (\Sigma_1, M_1, A_1, T_1, E_1)$ un composant tel que $(\Sigma \setminus \Sigma_{PA}) \cap \Sigma_1 = \emptyset$ (i.e. pas de noms similaires). Le composant $\mathcal{C}_1$ est dit* admissible *pour $\overline{\mathcal{C}}$ si pour tous les axiomes $e \in E_{PA}$, $E_1 \models_{obs} \Phi(e)$, où $\Phi : \Sigma_{PA} \to \Sigma_1$ est un morphisme de signatures tels que $\Phi(Elem) = \mathtt{State}_{\mathcal{C}_1}$ et $\Phi(Bool) = Bool$.* ■

Si nous considérons le composant caractère $\mathtt{CCHAR} = (\Sigma_1, M_1, A_1, T_1, E_1)$ de l'exemple 2.5, alors CCHAR est admissible pour le patron PSET (voir l'exemple 3.4) quand nous utilisons le morphisme suivant : $\Phi(Elem) = \mathtt{State}_{\mathtt{CCHAR}}$ et $\Phi(\mathtt{eq}) = (=_{obs}^{\mathtt{CCHAR}})$.

La substitution d'un paramètre formel par un admissible composant produit un nouveau composant.

Définition 3.11 (Instanciation de paramètre). *Soient $\overline{\mathcal{C}_1} = (PA, \mathcal{C}_1)$ un patron de composition et $\mathcal{C}_2 = (\Sigma_2, M_2, A_2, T_2, E_2)$ un admissible composant pour $\overline{\mathcal{C}_1}$ via un morphisme de signature $\Phi : \Sigma_{PA} \to \Sigma_2$. L'*instanciation *de $\overline{\mathcal{C}_1}$ par $\mathcal{C}_2$, que nous notons $\overline{\mathcal{C}_1}[PA \leftarrow \mathcal{C}_2]_\Phi$, est la spécification (Σ, M, A, T, E) tels que :*

(i) $\Sigma = \Sigma_2 \cup \Phi(\Sigma_1)$;

(ii) $M = \Phi(M_1)$;

(iii) $A = \Phi(A_1)$;

(iv) $T = \Phi(T_1)$;

(v) $E = E_2 \cup \Phi(E_1)$. ■

Nous donnons une définition formelle de la composition dynamique :

Définition 3.12 (Composition Dynamique). *Etant donnés un patron de composition $\overline{\mathcal{C}_1} = (PA, \mathcal{C}_1)$ et un composant $\mathcal{C}_2 = (\Sigma_2, M_2, A_2, T_2, E_2)$. Soient $\Phi : \Sigma_{PA} \to \Sigma_2$ un morphisme de signature et Update : $s_1 \dots s_n\ \mathtt{State}_{\mathcal{C}_2}\ \mathtt{State}_{\mathcal{C}_2} \to \mathtt{Meth}$ un symbole de méthode. Une spécification $\mathcal{C} = (\Sigma, M, A, T, E)$ est dite une a* composition dynamique *de $\mathcal{C}_2$ par rapport à $\overline{\mathcal{C}_1}$ (notée $\overline{\mathcal{C}_1}[\mathcal{C}_2]$) ssi $\mathcal{C}_2$ est admissible pour $\overline{\mathcal{C}_1}$ via Φ, et $\mathcal{C} = \overline{\mathcal{C}_1}[PA \leftarrow \mathcal{C}_2]_\Phi \cup (\Sigma', M', A', T', E')$ tels que :*

(i). $\Sigma' = (S', F')$ *avec* $S' = S_2 \cup \Phi(S_1)$ *et* $F' = \{Update\}$. *La méthode* $Update(U, x, y)$ *signifie le remplacement de l'ancienne valeur* x *par la nouvelle valeur* y *qui est le résultat de l'application d'une méthode de* $\mathcal{C}_2$ *sur* x *(*U *est une variable qui remplace la séquence de variables* $x_1, \ldots, x_n$*).*

(ii). $M' = \{Update(U, x, y) \mid x, y$ *sont des variables de sorte* $\mathtt{State}_{\mathcal{C}_2}$ *et* U *est une variable de sorte* $S^*_d\}$ *;*

(iii). $A' = \emptyset$ *;*

(iv). *Soient* $u_1 = Update(U, x, Do_{\mathcal{C}_2}(m_1, x))$ *et* $u_2 = Update(U', x', Do_{\mathcal{C}_2}(m_2, x'))$ *deux méthodes où* $m_1, m_2 \in \mathcal{C}_2$. *Pour toute méthode* $m \in \Phi(M_1)$, *nous avons :*

$$T' = \boldsymbol{Ax}(IT(u_1, m)) \cup \boldsymbol{Ax}(IT(m, u_1)) \cup \boldsymbol{Ax}(IT(u_1, u_2)))$$

tel que $\boldsymbol{Ax}(IT(u_1, u_2))$ *contient les axiomes suivants :*

$$\begin{aligned} U = U' \wedge x = x' &\implies IT(u_1, u_2) = u'_1 \\ x \neq x' &\implies IT(u_1, u_2) = u_1 \\ U \neq U' &\implies IT(u_1, u_2) = u_1 \end{aligned}$$

où $u'_1 = Update(U, Do_{\mathcal{C}_2}(m_2, x'), Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m_1, m_2),\ Do_{\mathcal{C}_2}(m_2, x')))$.

(v). *Pour tout symbole d'attribut* $a : s'_1 \ldots s'_m \to s'$, *nous avons*

$$E' = \boldsymbol{Ax}(Poss(Update(U, x, y), st)) \cup \boldsymbol{Ax}(a(Z, Do(Update(U, x, y), st)))$$

où $\boldsymbol{Ax}(a(Z, Do(Update(V, x, y), st)))$ *est définie comme suit :*

(a) *a est l'instance d'un attribut paramétrique dont l'un des arguments est de sorte* $\Phi(Elem)$ *:*

$$\begin{aligned} C[Z, x', U, x, y, st] &\implies a(Z, x', Do(Update(U, x, y), s)) = cst \\ \overline{C[Z, x', U, x, y, st]} &\implies a(Z, x', Do(Update(U, x, y), st)) = a(Z, x', st) \end{aligned}$$

avec cst est une constante de sorte s' *et* $C[Z, x', V, x, y, st]$ *(*$\overline{C[Z, x', V, x, y, st]}$ *est sa négation) est une formule contenant des variables libres construite de conjonction d'équations observables de telle façon que* $C[Z, x', U, x, y, st] \wedge C[Z, x', U', x, y, st]$ *est fausse à chaque fois que* $U \neq U'$.

(b) *a est l'instance d'un attribut paramétrique avec* $s' = \Phi(Elem)$ *:*

$$\begin{aligned} C'[Z, U, st] &\implies a(Z, Do(Update(U, x, y), st)) = y \\ \overline{C'[Z, U, st]} &\implies a(Z, Do(Update(U, x, y), st)) = a(Z, st) \end{aligned}$$

où $C'[Z, U, st]$ *(et sa négation) est une formule contenant des variables libres construite de conjonction d'équations observables de telle façon que* $C'[Z, U, st] \wedge C'[Z, U', st]$ *est fausse à chaque fois que* $U \neq U'$.

(c) *a n'est pas l'instance d'un attribut paramétrique :* $a(Z, Do(Update(U, x, y), st)) = a(U, st)$.

La notation $\boldsymbol{Ax}(f)$ *signifie l'ensemble des axiomes correspondant à la définition de la fonction* f. ■

Bien que la définition ci-dessus semble assez complexe à comprendre, elle est juste une formulation mathématique de quelques idées simples :

- Le paramètre formel du patron $\overline{\mathcal{C}_1}$ est remplacé par un admissible composant $\mathcal{C}_2$ pour produire un nouveau composant $\mathcal{C}$.
- Ce nouveau composant $\mathcal{C}$ est enrichi par une nouvelle méthode *Update* dont le rôle est de relier l'espace d'état de $\overline{\mathcal{C}_1}$ à celui de $\mathcal{C}_2$. En d'autres termes, l'utilisation de *Update* signifie que le changement d'états dans $\mathcal{C}_2$ implique le changement d'états dans $\mathcal{C}$.
- Les axiomes mentionnées dans (iv) montrent comment transformer des méthodes *Update*. D'une part, on doit ajouter des axiomes pour définir comment transformer *Update* par rapport aux autres méthodes de $\overline{\mathcal{C}_1}$. D'autre part, quand on modifie le même objet de $\mathcal{C}_2$ alors la transformation utilisée est celle de $\mathcal{C}_2$, *i.e.* $IT_{\mathcal{C}_2}$. Par contre, la modification de deux objets différents de $\mathcal{C}_2$ peut se faire dans n'importe quel ordre.
- Les axiomes données dans (v) expriment comment les attributs sont affectés par la méthode *Update*.

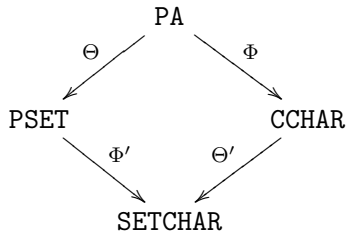


FIG. 3.3 – Composition Dynamique.

Exemple 3.5 *La composition dynamique de* CCHAR *(voir l'exemple 2.5) par rapport à* SET *(voir l'exemple 3.4) en utilisant le morphisme suivant* $\Phi(Elem) = \mathit{State}_{\mathit{CCHAR}}$ *et* $\Phi(\mathit{eq}) = (=_{obs}^{\mathit{CCHAR}})$ *procède par les étapes suivantes :*

1. *l'instanciation de* SET *par* Φ, *i.e.* SETCHAR $= \Phi(\mathit{SET})$;
2. *ajouter à* SETCHAR *une nouvelle méthode Update* : $\mathit{State}_{\mathit{CHAR}}\ \mathit{State}_{\mathit{CHAR}} \rightarrow \mathit{Meth}$ *avec les axiomes suivants :*
   (a) *transformation des méthodes Update (voir la définition 3.12.(iv)) :*

   ```
   (16) c1 = c2  =>
           IT(Update(c1,c2),Update(c3,c4)) = Update(c4,c')
   (17) c1 <> c2 =>
           IT(Update(c1,c2),Update(c3,c4)) = Update(c1,c2)
   ```

   où `c'=Do_CCHAR(IT_CCHAR(m1,m2),c4)`, `m1` *et* `m2` *sont des méthodes de* CCHAR *tel que* `c2 = Do_CCHAR(m1,c1)` *and* `c4 = Do_CCHAR(m2,c3)`.

   (b) *axiomes pour la fonction Poss :*

```
(18) iselem(c,st)=true  => Poss(Update(c,c'),st) = true
(19) iselem(c,st)=false => Poss(Update(c,c'),st) = false
```

(c) axiomes pour tous les attributs observant les effets de Update (voir la définition 3.12.(v)) :

```
(20) c = c2  => iselem(c,Do(Update(c1,c2),st)) = true
(21) c1 <> c2 => iselem(c,Do(Update(c1,c2),st)) =
                                                 iselem(c,st)
```

■

Dans ce qui suit, nous allons énoncer quelques propriétés propres à la composition dynamique.

L'application de *Update* à deux objets différents peut se faire dans n'importe quel ordre.

Lemme 3.1 *Soit* $a : s_1 \dots s_n\texttt{State} \rightarrow s$ *un symbole d'attribut tel que* a *est une instance d'un attribut paramétrique. Etant données deux méthodes* $u_1 = Update(U, x, x')$ *et* $u_2 = Update(V, y, y')$. *Si* $U \neq V$ *ou* $x \neq y$ *alors :*

$$a(Z, (st)[u_1; u_2]) = a(Z, (st)[u_2; u_1]).$$

pour tout état st. ■

Preuve. Deux cas sont à considérer :
Premier cas : il y a un seul $s_i = \Phi(Elem) = \mathit{State}_{\mathcal{C}_2}$ avec $i \in \{1, \dots, n\}$ tel que : $a : s_1 \dots s_{n-1}\mathit{State}_{\mathcal{C}_2}\ \mathit{State} \rightarrow s$. Selon la définition 3.12 nous avons :

$$a(Z, z, (st)[Update(U, x, x'); Update(V, y, y')]) = a(Z, z, (st)[Update(V, y, y'); Update(U, x, x')])$$

1. $U = V$ et $x \neq y$:
   (a) si $C[Z, z, U, x, x', st] \wedge C[Z, z, V, y, y', st]$ est vraie alors $cst = cst$;
   (b) si $C[Z, z, U, x, x', st] \wedge \overline{C[Z, z, V, y, y', st]}$ est vraie alors $cst = cst$;
   (c) si $\overline{C[Z, z, U, x, x', st]} \wedge C[Z, z, V, y, y', st]$ est vraie alors $cst = cst$;
   (d) si $\overline{C[Z, z, U, x, x', st]} \wedge \overline{C[Z, z, V, y, y', st]}$ est vraie alors $a(Z, z, st) = a(Z, z, st)$;
2. $U \neq V$: Selon la définition 3.12 nous avons $C[Z, z, U, x, y, st] \wedge C[Z, z, U', x, y, st]$ est fausse chaque fois que $U \neq U'$. Trois cas sont possibles :
   (a) si $C[Z, z, U, x, x', st] \wedge \overline{C[Z, z, V, y, y', st]}$ est vraie alors $cst = cst$;
   (b) si $\overline{C[Z, z, U, x, x', st]} \wedge C[Z, z, V, y, y', st]$ est vraie alors $cst = cst$;
   (c) si $\overline{C[Z, z, U, x, x', st]} \wedge \overline{C[Z, z, V, y, y', st]}$ est vraie alors $a(Z, z, st) = a(Z, z, st)$;

Deuxième cas : $s = \Phi(Elem) = \mathit{State}_{\mathcal{C}_2}$ tel que : $a : s_1 \dots s_{n-1}\ \mathit{State} \rightarrow \mathit{State}_{\mathcal{C}_2}$. Selon la définition 3.12 nous obtenons :

$$a(Z, (st)[Update(U, x, x'); Update(V, y, y')]) = a(Z, (st)[Update(V, y, y'); Update(U, x, x')])$$

1. $U = V$ et $x \neq y$: comme u_1 et u_2 sont appliquées sur l'état st alors $a(Z, st) = x$ et $a(Z, st) = y$. Ainsi nous avons $x = y$ qui contredit l'hypothèse de ce cas.
2. $U \neq V$: Selon la définition 3.12 nous avons $C'[Z, U, st] \wedge C'[Z, U', st]$ est fausse chaque fois que $U \neq U'$. Aussi, nous avons les cas suivants :
   (a) si $C'[Z, U, st] \wedge \overline{C'[Z, V, st]}$ est vraie alors $x' = x'$;
   (b) si $\overline{C'[Z, U, st]} \wedge C'[Z, V, st]$ est vraie alors $x' = x'$;
   (c) si $\overline{C'[Z, U, st]} \wedge \overline{C'[Z, V, st]}$ est vraie alors $a(Z, st) = a(Z, st)$; ■

Si deux *Update* u_1 et u_2 modifient respectivement deux objets différents alors les séquences $[u_1; u_2]$ et $[u_2; u_1]$ ont le même effet.

Lemme 3.2 *Soient $u_1 = Update(U, x, x')$ et $u_2 = Update(V, y, y')$ deux méthodes. Pour tout état st, si $U \neq V$ ou $x \neq y$ alors $(st)[u_1; u_2] =_{obs} (st)[u_2; u_1]$.* ■

Preuve. Nous considérons un contexte arbitraire $C[st] = a \cdot m_1 \cdot \ldots \cdot m_n$ pour $n > 0$ avec $a \in A$ et $m_i \in M$ tel que $i \in \{1, \ldots, n\}$. Ensuite nous avons :

$$C[(st)[u_1; u_2]] = C[(st)[u_2; u_1]]$$

Il est suffisant de montrer par induction sur n que :

$$a(Z, (st)[u_1; u_2; m_1(X_1); \ldots; m_n(X_n)]) = a(Z, (st)[u_2; u_1; m_1(X_1); \ldots; m_n(X_n)])$$

Base d'induction : Pour $n = 0$ et $C[st] = a$ nous avons :

$$a(Z, (st)[u_1; u_2]) = a(Z, (st)[u_2; u_1]). \tag{3.1}$$

Pour prouver l'équation (3.1) nous devons considérer deux cas :

(i) a est une instance d'un attribut paramétrique : l'équation (3.1) est donc vraie en utilisant le lemme 3.1.

(ii) a n'est pas une instance d'un attribut paramétrique : selon la définition 3.12 nous avons $a(Z, (st)[u_1; u_2]) = a(Z, st)$ et $a(Z, (st)[u_2; u_1]) = a(Z, st)$.

Hypothèse d'induction : Pour $n > 0$

$$a(Z, (st)[u_1; u_2; m_1(X_1); \ldots; m_n(X_n)]) = a(Z, (st)[u_2; u_1; m_1(X_1); \ldots; m_n(X_n)])$$

Pas d'induction : Montrons que lorsque $C'[st] = a \cdot m_1 \cdot \ldots \cdot m_n \cdot m_{n+1}$ alors $C[(st)[u_1; u_2]] = C[(st)[u_2; u_1]]$. Soient $st_1 = (st)[u_1; u_2; m_1(X_1); \ldots; m_n(X_n)]$ et $st_2 = (st)[u_2; u_1; m_1(X_1); \ldots; m_n(X_n)]$. Par l'hypothèse d'induction nous déduisons que $st_1 =_{obs} st_2$. Comme $=_{obs}$ est une congruence alors $a(Z, (st_1)[m_{n+1}]) = a(Z, (st_2)[m_{n+1}])$. ■

La composition dynamique d'un composant consistant par rapport à un patron consistant produit un composant qui satisfait $CP1$ pour les méthodes *Update*.

Théorème 3.5 *Etant donnés un patron de composition* $\overline{\mathcal{C}_1} = (PA, \mathcal{C}_1)$ *et un composant* $\mathcal{C}_2 = (\Sigma_2, M_2, A_2, T_2, E_2)$. *Soit* $\mathcal{C} = (\Sigma, M, A, T, E)$ *la composition dynamique de* $\mathcal{C}_2$ *par rapport à* $\overline{\mathcal{C}_1}$. *Si* $\overline{\mathcal{C}_1}$ *et* $\mathcal{C}_2$ *sont consistants alors* $E \models_{obs} CP1 \mid_{M'}$ *avec* M' *contenant des méthodes Update.* ■

Preuve. $CP1 \mid_{M'}$ est défini comme suit :

$$(st)[Update(X, u, v); IT(Update(Y, u', v'), Update(X, u, v))] = (st)[Update(Y, u', v'); IT(Update(X, u, v), Update(Y, u', v'))]$$

où $v = Do_{\mathcal{C}_2}(m_1(V), u)$ et $v' = Do_{\mathcal{C}_2}(m_2(W), u')$ avec m_1 et m_2 sont des méthodes de $\mathcal{C}_2$. Selon la définition 3.12 nous considérons deux cas :
Premier cas : $X = Y$ et $u = u'$

$CP1 \mid_{M'}$ est réécrite comme suit :

$$(st)[Update(X, u, v); Update(Y, v, Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m_2(W), m_1(V)), v))] = (st)[Update(Y, u, v'); Update(X, v', Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m_1(V), m_2(W)), v'))]$$

Comme $v = Do_{\mathcal{C}_2}(m_1(V), u)$, $v' = Do_{\mathcal{C}_2}(m_2(W), u)$ et $\mathcal{C}_2$ est consistant alors

$$Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m_2(W), m_1(V)), v) = Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m_1(V), m_2(W)), v') = u''$$

Ainsi, nous obtenons :

$$(st)[Update(X, u, v); Update(Y, v, u'')] = (st)[Update(Y, u, v'); Update(X, v', u'')]$$

qui est vraie.
Deuxième cas : $X \neq Y$ ou $u \neq u'$

$CP1 \mid_{M'}$ est réécrite comme suit :

$$(st)[Update(X, u, v); Update(Y, u', v')] = (st)[Update(Y, u', v'); Update(X, u, v)]$$

Cette équation est toujours vraie selon le lemme 3.2. ■

La composition dynamique d'un composant consistant par rapport à un patron consistant produit un composant qui satisfait $CP2$ pour les méthodes *Update*.

Théorème 3.6 *Etant donnés un patron de composition* $\overline{\mathcal{C}_1} = (PA, \mathcal{C}_1)$ *et un composant* $\mathcal{C}_2 = (\Sigma_2, M_2, A_2, T_2, E_2)$. *Soit* $\mathcal{C} = (\Sigma, M, A, T, E)$ *la composition dynamique de* $\mathcal{C}_2$ *par rapport à* $\overline{\mathcal{C}_1}$. *Si* $\overline{\mathcal{C}_1}$ *et* $\mathcal{C}_2$ *sont consistants alors* $E \models_{obs} CP2 \mid_{M'}$ *avec* M' *contenant des méthodes Update.* ■

Preuve. Soient $up \equiv Update(R, v, w)$, $up_1 \equiv Update(P, x, y)$ et $up_2 \equiv Update(Q, z, t)$ trois méthodes, où $w = Do_{\mathcal{C}_2}(m(Z), v)$, $y = Do_{\mathcal{C}_2}(m_1(V), x)$ et $t = Do_{\mathcal{C}_2}(m_2(W), z)$ avec m, m_1 et m_2 sont des méthodes de $\mathcal{C}_2$. Alors $CP2 \mid_{M'}$ est défini comme suit :

$$IT^*(up, [up_1; IT(up_2, up_1)]) = IT^*(up, [up_2; IT(up_1, up_2)])$$

Selon la définition 3.12 nous considérons deux cas :
Premier cas : $P = Q$ et $x = z$

$CP2\ |_{M'}$ est réécrite comme $IT^*(up, [up_1; up'_2]) = IT^*(up, [up_2; up'_1])$ où :

$$up'_1 \equiv Update(P, Do_{\mathcal{C}_2}(m_2(W), z), Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m_1(V), m_2(W)), Do_{\mathcal{C}_2}(m_2(W), z)))$$

et

$$up'_2 \equiv Update(Q, Do_{\mathcal{C}_2}(m_1(V), x), Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m_2(W), m_1(V)), Do_{\mathcal{C}_2}(m_1(V), x)))$$

Deux cas sont possibles :

1. $R = P$ et $v = x$
   Dans ce cas nous obtenons :

$$Update(R, u_1, Do_{\mathcal{C}_2}(m', u_1)) = Update(R, u_2, Do_{\mathcal{C}_2}(m'', u_2)) \text{ où}$$
$$u_1 \equiv Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m_2(W), m_1(V)), Do(m_1(V), x))$$
$$m' \equiv IT^*_{\mathcal{C}_2}(m(Z), [m_1(V); IT_{\mathcal{C}_2}(m_2(W), m_1(V))])$$
$$u_2 \equiv Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m_1(V), m_2(W)), Do(m_2(W), z))$$
$$m'' \equiv IT^*_{\mathcal{C}_2}(m(Z), [m_2(W); IT_{\mathcal{C}_2}(m_1(V), m_2(W))])$$

   Comme $\mathcal{C}_2$ est consistant, alors $u_1 = u_2$ et $m' = m''$. Par conséquent, l'équation ci-dessus est vraie.

2. $R \neq P$ ou $v \neq x$
   Nous avons $IT^*(up, [up_1; up'_2]) = up$ et $IT^*(up, [up_2; up'_1]) = up$.

Deuxième cas : $P \neq Q$ ou $x \neq z$

$CP2\ |_{M'}$ est réécrite comme suit :

$$IT^*(Update(R, v, w), [Update(P, x, y); Update(Q, z, t)]) =$$
$$IT^*(Update(R, v, w), [Update(Q, z, t); Update(P, x, y)])$$

Trois cas sont à considérer :

1. $R = P$ et $v = x$
   Nous obtenons :

$$Update(R, Do_{\mathcal{C}_2}(m_1(V), x), Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m(Z), m_1(V)), Do(m_1(V), x))) =$$
$$Update(R, Do_{\mathcal{C}_2}(m_1(V), x), Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m(Z), m_1(V)), Do(m_1(V), x)))$$

2. $R = Q$ et $v = z$
   Nous obtenons :

$$Update(R, Do_{\mathcal{C}_2}(m_2(W), z), Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m(Z), m_2(W)), Do(m_2(W), z))) =$$
$$Update(R, Do_{\mathcal{C}_2}(m_2(W), z), Do_{\mathcal{C}_2}(IT_{\mathcal{C}_2}(m(Z), m_2(W)), Do(m_2(W), z)))$$

3. $R \neq P$, $R \neq Q$, $v \neq x$ ou $v \neq z$
   Nous obtenons $Update(R, v, w) = Update(R, v, w)$.

■

Le théorème suivant est d'une importance majeure puisqu'il stipule que la propriété de consistance peut être préservée par une composition dynamique.

Théorème 3.7 *Etant donnés un patron consistant* $\overline{\mathcal{C}_1} = (PA, \mathcal{C}_1)$ *et un composant consistant* $\mathcal{C}_2 = (\Sigma_2, M_2, A_2, T_2, E_2)$*. Soit* $\mathcal{C} = (\Sigma, M, A, T, E)$ *la composition dynamique de* $\mathcal{C}_2$ *par rapport à* $\overline{\mathcal{C}_1}$ *en utilisant le morphisme* Φ*. Si* $E \models_{obs} CP1 \mid_{M', \Phi(M_1)}$ *et* $E \models_{obs} CP2 \mid_{M', \Phi(M_1)}$ *alors* $\mathcal{C}$ *est consistant avec* M' *contenant des méthodes Update.*

Preuve. Supposons que $E \models_{obs} CP1 \mid_{M', \Phi(M_1)}$ et $E \models_{obs} CP2 \mid_{M', \Phi(M_1)}$. Par définition, $\mathcal{C}$ est consistant ssi $E \models_{obs} CP1 \wedge CP2$.

1. **Preuve de** $E \models_{obs} CP1$**.** $CP1$ peut être formulée comme suit :

$$CP1 \equiv CP1 \mid_{M'} \wedge \Phi(CP1 \mid_{M_1}) \wedge CP1 \mid_{M', \Phi_2(M_1)}$$

   Comme $\overline{\mathcal{C}_1}$ est consistant et selon le théorème 3.5 $CP1$ est donc satisfaite.

2. **Preuve de** $E \models_{obs} CP2$**.** $CP2$ peut être exprimée comme suit :

$$CP2 \equiv CP2 \mid_{M'} \wedge \Phi(CP2 \mid_{M_1}) \wedge CP2 \mid_{M', \Phi_2(M_1)}$$

   Puisque $\overline{\mathcal{C}_1}$ est consistant et selon le théorème 3.6 $CP2$ est vraie.

■

Il faut souligner que la composition dynamique $\mathcal{C}$ est semi-consistante ssi :

1. $E \models_{obs} CP1 \mid_{M', \Phi(M_1)}$;
2. $E \models_{obs} CP2 \mid_{M', \Phi(M_1)}$;
3. le patron $\overline{\mathcal{C}_1}$ est semi-consistant ou/et le composant $\mathcal{C}_2$ est semi-consistant.

3.4 Exemple Illustratif

Un document d'un traitement de textes a une structure hiérarchique. En effet, il ne contient pas que du texte mais aussi des objets de mise en forme (la police, la couleur, la taille, ...) comme les documents de MicroSoft Word. Un document est divisé en pages, paragraphes, phrases, mots et caractères [37]. Un objet de mise en forme peut appartenir à chacun de ces niveaux ; cela dépend de sa position dans le document. Par exemple, si l'objet de mise en forme est à l'intérieur du paragraphe, alors il est considéré comme faisant partie de la structure du paragraphe. Par contre, s'il est entre deux paragraphes alors il est considéré comme un élément du niveau correspondant au paragraphe (ou faisant partie de la structure de la page). Plusieurs éditeurs collaboratifs se basent sur cette structure de document. A titre d'exemple, nous pouvons citer CoWord [86], qui est une version collaborative réalisée autour du logiciel de traitement de textes MicroSoft Word.

Maintenant, nous allons voir comment modéliser ce document en utilisant les constructions de composition que nous avons présentées dans les sections précédentes. A l'exception du caractère, chaque niveau a une structure linéaire. Aussi, nous utilisons le patron de composition `STRING` qui représente une séquence d'éléments d'un type donné :

```
spec = STRING
sorts:
 State Meth Elem Bool Nat
cons:
 Do : Meth State -> State
 Ins : Nat Elem Nat -> Meth
 Del : Nat Nat -> Meth
 Nop : -> Meth
opns:
 Poss : Meth State -> Bool
 Length : State -> Nat
 GetElem : Nat State -> Elem
 IT : Meth Meth -> Meth
axioms:
(1)  p <= Length(st) = true => Poss(Ins(p,e,n),st) = true
(2)  p <= Length(st) = false => Poss(Ins(p,e,n),st) = false
(3)  p < Length(st) = true => Poss(Del(p,n),st) = true
(4)  p < Length(st) = false => Poss(Del(p,n),st) = false
(5)  Length(Init) = 0
(6)  Length(Do(Ins(p,e,n),st)) = Length(st)+1
(7)  Length(Do(Del(p,n),st)) = Length(st)-1
(8)  p=p1 => GetElem(p,Do(Ins(p1,e1,n1),st)) = e1
(9)  p > p1 = true  => GetElem(p,Do(Ins(p1,e1,n1),st)) =
                              GetElem(p-1,st)
(10) p < p1 = true => GetElem(p,Do(Ins(p1,e1,n1),st)) =
                              GetElem(p,st)
(11) p >= p1 = true => GetElem(p,Do(Del(p1,n1),st)) =
                              GetElem(p+1,st)
(12) p < p1 = true => GetElem(p,Do(Del(p1,n1),st)) = GetElem(p,st)
(13) p1 < p2 = true => IT(Ins(p1,e1,n1),Ins(p2,e2,n2)) = Ins(p1,e1,n1)
(14) p1 > p2 = true => IT(Ins(p1,e1,n1),Ins(p2,e2,n2)) = Ins(p1+1,e1,n1)
(15) p1=p2, n1 < n2 = true => IT(Ins(p1,e1,n1),Ins(p2,e2,n2)) =
                                 Ins(p1,e1,n1)
(16) p1=p2, n1 > n2 = true => IT(Ins(p1,e1,n1),Ins(p2,e2,n2)) =
                                 Ins(p1+1,e1,n1)
(17) p1 < p2 =true => IT(Ins(p1,e1,n1),Del(p2,n2)) = Ins(p1,e1,n1)
(18) p1 >= p2 =true => IT(Ins(p1,e1,n1),Del(p2,n2)) = Ins(p1-1,e1,n1)
(19) p1 < p2 =true => IT(Del(p1,n1),Del(p2,n2)) = Del(p1,n1)
(20) p1 > p2 =true => IT(Del(p1,n1),Del(p2,n2)) = Del(p1-1,n1)
(21) p1=p2 => IT(Del(p1,n1),Del(p2,n2)) = Nop
(22) p1 < p2 =true => IT(Del(p1,n1),Ins(p2,e2,n2)) = Del(p1,n1)
(23) p1 >= p2 =true => T(Del(p1,n1),Ins(p2,e2,n2)) = Del(p1+1,n1)
(24) IT(Nop,op) = Nop
(25) IT(op,Nop) = op
```

La fonction de transformation *IT* est utilisée pour synchroniser un objet collaboratif dont la structure est linéaire. Il y a deux méthodes : `Ins(p,e,n)` pour ajouter un élément `e` à la position `p` ; `Del(p,n)` pour supprimer l'élément à la position `p`. Le paramètre `n` est un entier (l'identifiant du site générateur) qui est utilisé pour résoudre le conflit entre deux insertions.

D'après cette spécification, le paramètre formel de `STRING` n'a pas d'axiomes. Aussi, tous les composants possibles peuvent être admissibles pour `STRING`. Sup-

posons que nous voulons doter le document des mises en forme concernant la taille et la couleur. Aussi considérons les composantes suivants (voir respectivement les exemples 2.5 et 3.1) :

- `CCHAR` est une cellule de mémoire contenant un caractère ;
- `CNAT` est la cellule de mémoire contenant un entier pour représenter la taille ;
- `CCOLOR` est la cellule de mémoire contenant une couleur.

L'élément de base du document est le caractère formaté `FCHAR` qui est obtenu par une composition statique :

$$\texttt{FCHAR} = \texttt{CCHAR} \oplus \texttt{CNAT} \oplus \texttt{CCOLOR}$$

Un mot est la séquence de plusieurs caractères qui est construit par des compositions dynamique et statique :

$$\begin{aligned} \texttt{MOT} &= \texttt{STRING}[\texttt{FCHAR}] \\ \texttt{FMOT} &= \texttt{MOT} \oplus \texttt{CNAT} \oplus \texttt{CCOLOR} \end{aligned}$$

La phrase est une chaîne de mots ; elle est donc construite de la façon suivante :

$$\begin{aligned} \texttt{PHRASE} &= \texttt{STRING}[\texttt{FMOT}] \\ \texttt{FPHRASE} &= \texttt{PHRASE} \oplus \texttt{CNAT} \oplus \texttt{CCOLOR} \end{aligned}$$

Le reste des niveaux du document sont construit de la même manière :

$$\begin{aligned} \texttt{PARAGRAPHE} &= \texttt{STRING}[\texttt{FPHRASE}] \\ \texttt{FPARAGRAPHE} &= \texttt{PARAGRAPHE} \oplus \texttt{CNAT} \oplus \texttt{CCOLOR} \\ \texttt{PAGE} &= \texttt{STRING}[\texttt{FPARAGRAPHE}] \\ \texttt{FPAGE} &= \texttt{PAGE} \oplus \texttt{CNAT} \oplus \texttt{CCOLOR} \end{aligned}$$

Il est clair que la consistance est régie par le type de composition ainsi que les composants utilisés à chaque niveau.

3.5 Conclusion

Dans ce chapitre, nous avons présenté une méthode compositionnelle pour spécifier et vérifier des objets collaboratifs complexes. A ce titre, nous avons proposé deux constructions pour composer les objets :

1. La composition statique permet de construire un objet collaboratif à partir d'un nombre fixe d'objets existants, de telle façon que la structure du nouvel objet reste figée durant l'exécution. Deux types de composition statique ont été présentées selon le degré de concurrence qu'ils confèrent entre les objets composants. La composition sans synchronisation construit un objet composite sans interaction (donc plus de concurrence) entre les objets composants. Par contre, la composition avec synchronisation permet aux composants d'interagir entre eux.

2. La composition dynamique permet de combiner un nombre arbitraire d'un même objet collaboratif selon un structure donnée. En d'autres termes, l'objet en question est crée et/ou supprimé de manière dynamique. De ce fait, l'objet obtenu a une structure qui n'est pas figée durant l'exécution.

En perspective, nous envisagerons d'affiner notre approche en étudiant, d'un point de vue de déduction automatique, les propriétés sémantiques que présentent les différentes compositions. Enfin, nous projetterons également d'enrichir l'outil `VOTE` par ces techniques de composition.

Troisième partie

Convergence des Structures de Données Linéaires

Chapitre 1

Problématique

Sommaire

1.1 Introduction

L'approche des transformées opérationnelles considère l'existence de deux ou plusieurs sites (ou utilisateurs) qui collaborent pour réaliser une tâche commune [73; 81; 84]. Chaque site possède une copie de l'objet partagé ainsi qu'un journal des opérations exécutées (appelé également *histoire*) H. Un site modifie localement la copie par l'exécution d'une opération, ensuite il envoie cette opération aux autres sites. Pour intégrer une opération distante op, H est réorganisée en deux séquences consécutives H_h et H_c, de telle sorte que H_h contient les opérations qui précèdent causalement op et H_c contient les opérations concurrentes à op. En utilisant un algorithme de transformation spécifique à la sémantique de l'objet partagé, op est d'abord transformée par rapport à H_c, ensuite l'opération résultant de cette transformation est exécutée après H. L'avantage majeur de l'approche des transformées opérationnelles est le fait que les histoires des sites ne sont pas identiques mais elles sont équivalentes (elles produisent le même état). En d'autres termes, les opérations concurrentes peuvent s'exécuter dans n'importe quel ordre.

Théoriquement, l'approche des transformées opérationnelles requiert la satisfaction des conditions $TP1$ et $TP2$ de telle façon que l'algorithme de transformation puisse garantir la convergence sur n'importe quelle histoire H_c [68; 73; 81; 84]. L'acquisition de ces conditions simplifie énormément la conception

des éditeurs collaboratifs basés sur la transformation des opérations. En effet, de tels éditeurs peuvent assurer la convergence sur toute architecture de communication (avec ou sans site central). Malheureusement, ceci s'avère extrêmement difficile à réaliser, car la satisfaction de $TP2$ pose un défi significatif quant à la conception des algorithmes de transformation.

Depuis dix sept ans, plusieurs algorithmes de transformation spécifiques à des objets linéaires (tels que la liste, la chaîne de caractère, ...) ont été proposées [26; 73; 81; 39; 50]. Mais aucun de ces algorithmes n'arrive à vérifier $TP2$. Dans ce chapitre, nous allons montrer la difficulté du problème au travers les erreurs détectées dans de tels algorithmes. Nous avons spécifié et vérifié ces algorithmes selon la méthodologie que nous avons présentée dans la partie précédente (voir le chapitre 2). L'identification du problème inhérent à la violation de $TP2$ pour des objets linéaires va nous conduire à proposer une nouvelle solution qui sera illustrée dans le chapitre 2.

Le chapitre est composé comme suit. Dans la section 1.2, nous allons présenter quatre algorithmes de transformation spécifiques à des objets linéaires, ainsi que les situations de divergence que nous avons décelées dans de tels algorithmes. Quant à la section 1.3, elle va mettre en évidence les causes réelles inhérentes à la violation de $TP2$. Enfin, nous terminons le chapitre par une conclusion.

1.2 Etat de l'art : scénarios de divergence

Dans cette section, nous allons mettre l'accent sur la violation de la condition $TP2$ qui mène inévitablement vers une divergence de données. Pour ce faire, nous allons passer en revue les algorithmes de transformation existants dans la littérature ainsi que les problèmes de divergence qu'ils produisent quant à la violation de $TP2$. Pour présenter les scénarios de divergence, nous allons utiliser comme objet linéaire les chaînes de caractères.

1.2.1 Algorithme de Ressel

Description

Dans Ressel et *al.* [73], les auteurs ont proposé un algorithme de transformation pour les chaînes de caractères qui apporte deux modifications à celui proposé par Ellis et *al.* [26] (voir section 2.4.2). La première modification consiste à remplacer le paramètre *pr* par un autre paramètre u. Ce dernier représente tout simplement l'*identifiant* (ou le numéro) du site où l'opération a été générée. De la même manière que l'algorithme de Ellis et *al.*, u est utilisé comme un moyen pour résoudre le conflit occasionné au moment où deux insertions concurrentes tentaient d'insérer deux caractères à la même position.

Quant à la deuxième modification, elle concerne la transformation du couple d'opérations d'insertion. En effet, lorsque deux insertions concurrentes produisent à la même position deux caractères (identiques ou différents), la position d'insertion de l'opération ayant le plus grand identifiant sera incrémentée (ou sera avancée vers la droite). Alors que l'autre opération (celle ayant le plus petit

identifiant) gardera la même position d'insertion. En d'autres termes, les deux caractères seront insérés même s'ils sont identiques. Ce qui est à l'opposé de la solution adoptée par Ellis et *al.*, qui ne garde qu'un seul exemplaire en cas d'insertions concurrentes de deux caractères identiques.

En dehors de ces deux modifications, les autres cas de transformation restent similaires à ceux de Ellis et *al.*. A ce titre, rappelons que la transformation du couple Ins et Del comportait une erreur qui causait la violation de $TP1$ (voir section 2.4.2). Aussi, nous supposons ici que les auteurs font référence à une version corrigée.

L'algorithme 4 illustre le contenu détaillé de la solution de Ressel et *al.* [73].

1: $IT(Ins(p_1, c_1, u_1), Ins(p_2, c_2, u_2)) =$
2: **si** ($p_1 < p_2$ or ($p_1 = p_2$ and $u_1 < u_2$)) **alors**
3: **retourner** $Ins(p_1, c_1, u_1)$
4: **sinon**
5: **retourner** $Ins(p_1 + 1, c_1, u_1)$
6: **fin si**

7: $IT(Ins(p_1, c_1, u_1), Del(p_2, u_2)) =$
8: **si** ($p_1 \leq p_2$) **alors**
9: **retourner** $Ins(p_1, c_1, u_1)$
10: **sinon**
11: **retourner** $Ins(p_1 - 1, c_1, u_1)$
12: **fin si**

13: $IT(Del(p_1, u_1), Ins(p_2, c_2, u_2)) =$
14: **si** ($p_1 < p_2$) **alors**
15: **retourner** $Del(p_1, u_1)$
16: **sinon**
17: **retourner** $Del(p_1 + 1, u_1)$
18: **fin si**

19: $IT(Del(p_1, u_1), Del(p_2, u_2)) =$
20: **si** ($p_1 < p_2$) **alors**
21: **retourner** $Del(p_1, u_1)$
22: **sinon si** ($p_1 > p_2$) **alors**
23: **retourner** $Del(p_1 - 1, u_1)$
24: **sinon**
25: **retourner** $Nop()$
26: **fin si**

Algorithme 4: Algorithme de Ressel et *al.*

Contre-exemple

En vérifiant cet algorithme nous avons constaté qu'il viole la condition $TP2$. Le scénario, qui mène vers cette violation, est illustré à la figure 1.1. Ce scénario est connu communément dans la littérature sous le nom anglais "$TP2$ puzzle" [55]. Il consiste en trois utilisateurs qui exécutent concurremment trois

opérations : $op_1 = Ins(3, x, u_1)$, $op_2 = Del(2, u_2)$ et $op_3 = Ins(2, y, u_3)$ avec $u_1 < u_2$ et $u_2 < u_3$. Le déroulement du scénario procède comme suit :

1. D'abord, op_3 est intégrée sur le site u_2 sous une forme transformée $op'_3 = IT(Ins(2, y, u_3), Del(2, u_2)) = Ins(2, y, u_3)$.
2. L'opération op_2 est reçue sur le site u_3 pour être intégrée sous la forme $op'_2 = IT(Del(2, u_2), Ins(2, y, u_3)) = Del(3, u_2)$.
3. Quand op_1 arrive sur le site u_2, elle sera intégrée de la manière suivante : op_1 est en premier transformée par rapport à op_2 et ensuite le résultat de cette transformation est de nouveau transformé par rapport à op'_3. L'enchaînement de ces transformation est donné par l'expression suivante :

$$IT(IT(\overbrace{Ins(3, x, u_1)}^{op_1}, \overbrace{Del(2, u_2)}^{op_2}), \overbrace{Ins(2, y, u_3)}^{op'_3}) = \overbrace{Ins(2, x, u_1)}^{op'_1}$$

4. De la même façon, op_1 est intégrée sur le site u_3 sous la forme suivante :

$$IT(IT(\overbrace{Ins(3, x, u_1)}^{op_1}, \overbrace{Ins(2, y, u_3)}^{op_3}), \overbrace{Del(3, u_2)}^{op'_2}) = \overbrace{Ins(3, x, u_1)}^{op'_1}$$

L'intégration de op_1 sur les sites u_2 et u_3 a produit respectivement deux opérations op'_1 et op''_1 qui sont syntaxiquement différentes. Ce qui viole la condition $TP2$. Nous constatons également que cette violation engendre une divergence d'états dans les sites u_2 et u_3.

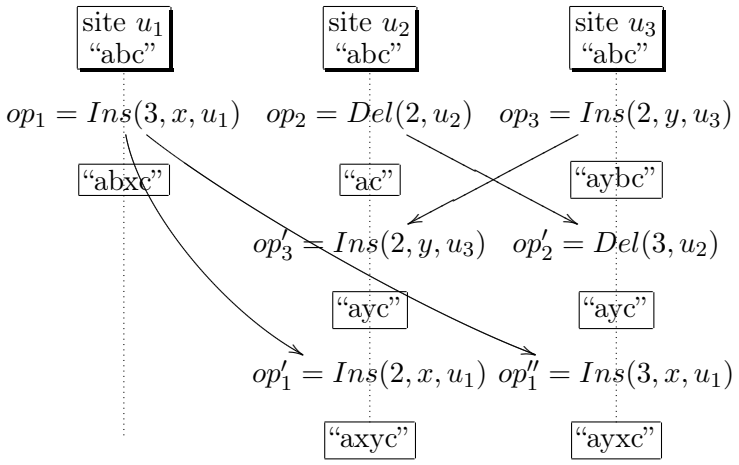


FIG. 1.1 – Le "$TP2$ puzzle".

Analyse

Notons qu'à leur état de génération, il y a une relation entre les positions d'insertion de op_1 et op_3 : la position de op_3 est avant (ou à gauche de) celle de op_1. Une fois que les deux opérations sont transformées par rapport à op_2 sur le site u_2, elles perdent cette relation. En effet, leurs positions d'insertion deviennent identiques. Par conséquent, transformer l'une par rapport à l'autre

sur le site u_2 relève de la résolution de conflit (utilisation des identifiants des sites pour décider laquelle des opérations insère avant l'autre) puisque les deux opérations sont censées insérer deux caractères à la même position. Cette résolution peut éventuellement ne pas respecter la relation existante à l'état de génération de op_1 et op_3, comme c'est le cas dans le contre-exemple illustré à la figure 1.1.

Dans ce qui suit, nous allons présenter des algorithmes de transformation conçus dans le but de résoudre le "$TP2$ puzzle".

1.2.2 Algorithme de Suleiman

Description

Le problème "$TP2$ puzzle" a déjà été soulevé par Suleiman et *al.* [81]. Pour le résoudre, ils ont proposé une nouvelle solution qui enrichit l'opération d'insertion par deux paramètres av et ap. Ces paramètres mémorisent l'ensemble des opérations de suppression concurrentes à l'opération d'insertion. L'ensemble av contient les opérations de suppression qui ont détruit un caractère devant la position d'insertion. Quant à l'ensemble ap, il contient les opérations qui ont effacé un caractère derrière la position d'insertion.

Trois cas sont donc possibles pour résoudre le conflit occasionné par 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 :

1. $(av_1 \cap ap_2) \neq \emptyset$: le caractère c_2 est inséré avant le caractère c_1,
2. $(ap_1 \cap av_2) \neq \emptyset$: le caractère c_2 est inséré après le caractère c_1,
3. $(av_1 \cap ap_2) = (ap_1 \cap av_2) = \emptyset$: dans ce cas une fonction $code(c)$ 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 avant. La fonction $code(c)$ remplace donc le système de priorité utilisé dans les algorithmes précédents.

Il faut noter que lorsque les deux opérations insèrent le même caractère (*i.e.* $code(c_1) = code(c_2)$) à la même position, l'une est exécutée et l'autre est ignorée en retournant l'opération nulle $Nop()$. En d'autres termes, un seul exemplaire des caractères ajoutés est gardé (similaire à la solution préconisée par Ellis et *al.* [26]). Par ailleurs, lors de la génération d'une opération d'insertion, les ensembles av et ap sont vides.

L'algorithme 5 donne le contenu détaillé de la solution de Suleiman et *al.*

Le problème soulevé dans le "$TP2$ puzzle" (voir la figure 1.1) est résolu par l'utilisation des deux paramètres av et ap comme le montre le scénario illustré à la figure 1.2. Le problème était situé sur le site u_2. Nous rejouons donc l'intégration des différentes opérations sur ce site comme suit :

- Lorsque op_3 est reçue sur le site u_2, elle est transformée par rapport à op_2. Le résultat de cette transformation est $op'_3 = Ins(2, x, \{\}, \{Del(2)\})$. L'opération op_2 a été mémorisée dans l'ensemble ap de op'_3 car le caractère qu'elle détruit est situé après le caractère que op_3 insère.
- Lorsque op_1 arrive sur le site u_2, elle doit être transformée par rapport aux opérations op_2 et op'_3. La première transformation donne l'opération

1: $IT(Ins(p_1, c_1, av_1, ap_1), Ins(p_2, c_2, av_2, ap_2)) =$
2: **si** $(p_1 < p_2)$ **alors**
3: **retourner** $Ins(p_1, c_1, av_1, ap_1)$
4: **sinon si** $(p_1 > p_2)$ **alors**
5: **retourner** $Ins(p_1 + 1, c_1, av_1, ap_1)$
6: **sinon si** $(av_1 \cap ap_2 \neq \emptyset)$ **alors**
7: **retourner** $Ins(p_1 + 1, c_1, av_1, ap_1)$
8: **sinon si** $(ap_1 \cap av_2 \neq \emptyset)$ **alors**
9: **retourner** $Ins(p_1, c_1, av_1, ap_1)$
10: **sinon si** $(code(c_1) > code(c_2))$ **alors**
11: **retourner** $Ins(p_1, c_1, av_1, ap_1)$
12: **sinon si** $(code(c_1) < code(c_2))$ **alors**
13: **retourner** $Ins(p_1 + 1, c_1, av_1, ap_1)$
14: **sinon**
15: **retourner** $Nop()$
16: **fin si**

17: $IT(Ins(p_1, c_1, av_1, ap_1), Del(p_2)) =$
18: **si** $(p_1 \leq p_2)$ **alors**
19: **retourner** $Ins(p_1, c_1, av_1, ap_1 \cup \{Del(p_2)\})$
20: **sinon**
21: **retourner** $Ins(p_1 - 1, c_1, av_1 \cup \{Del(p_2)\}, ap_1)$
22: **fin si**

23: $IT(Del(p_1), Ins(p_2, c_2, av_2, ap_2)) =$
24: **si** $(p_1 < p_2)$ **alors**
25: **retourner** $Del(p_1)$
26: **sinon**
27: **retourner** $Del(p_1 + 1)$
28: **fin si**

29: $IT(Del(p_1), Del(p_2)) =$
30: **si** $(p_1 < p_2)$ **alors**
31: **retourner** $Del(p_1)$
32: **sinon si** $(p_1 > p_2)$ **alors**
33: **retourner** $Del(p_1 - 1)$
34: **sinon**
35: **retourner** $Nop()$
36: **fin si**

Algorithme 5: Algorithme de Suleiman et *al.*

$op_1^2 = Ins(2, y, \{Del(2)\}, \{\})$. L'opération 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 de savoir laquelle des deux opérations à son effet devant l'autre. Dans la proposition de Ressel, l'identifiant du site est utilisé pour décider dans quel ordre l'insertion se fait. Avec la solution de Suleiman, les informations stockées dans av et ap sont susceptibles de prendre une

telle décision. Comme op_2 est dans l'ensemble av de op_1^2, on en déduit que 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. Ayant de telles informations, on déduit que la position de op_1^2 est derrière celle de op_2^2. Par conséquent, la position de op_1^2 est incrémentée. Le résultat de la transformation est l'opération $op_1' = Ins(3, x, \{Del(2)\}, \{\})$.

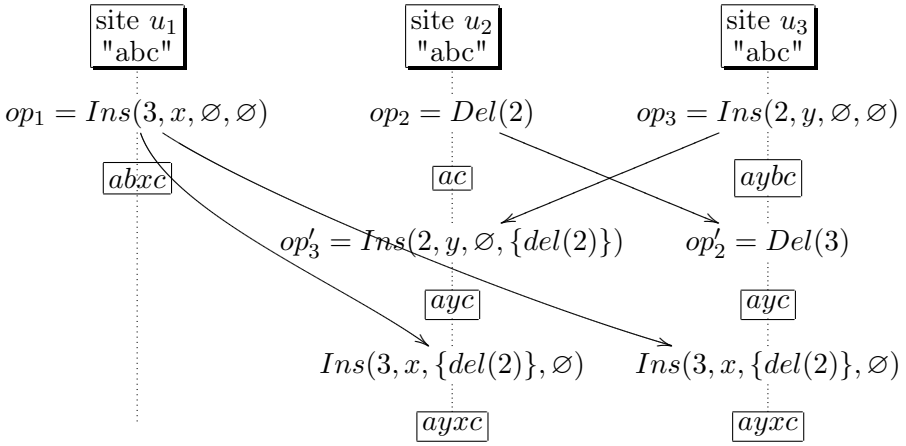


FIG. 1.2 – Solution de Suleiman pour le contre-exemple de Ressel.

Contre-exemple

Malheureusement, la solution de Suleiman ne marche pas dans tous les cas. En effet, une analyse automatique de cette solution a révélé une situation où la condition $TP2$ est violée. Le contre-exemple illustrant cette violation est donné à la figure 1.3.

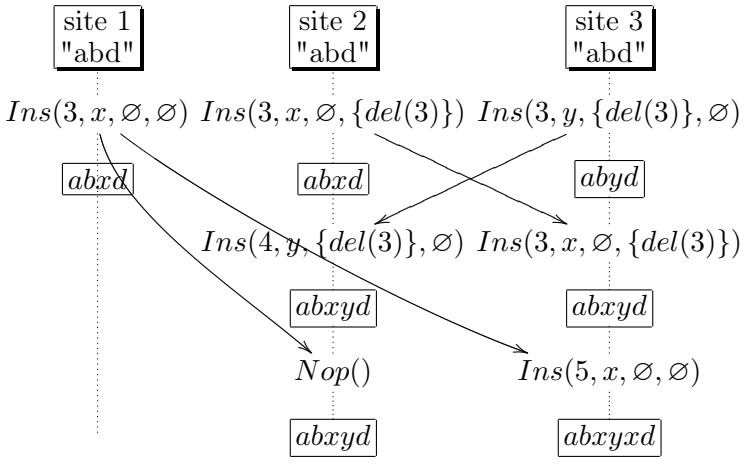


FIG. 1.3 – Contre-exemple pour la solution de Suleiman.

Le scénario de la figure 1.3 fait intervenir trois opérations concurrentes : $op_1 = Ins(3, x, \{\}, \{\})$, $op_2 = Ins(3, x, \{\}, Del(3))$ et $op_3 = Ins(3, y, \{Del(3)\}, \{\})$. Il se déroule comme suit :

1. Quand op_2 arrive sur le site 3, elle est intégrée sous la forme suivante :
$$IT(\overbrace{Ins(3,x,\{\},\{Del(3)\})}^{op_2},\overbrace{Ins(3,y,\{Del(3)\},\{\})}^{op_3}) = \overbrace{Ins(3,x,\{\},\{Del(3)\})}^{op'_2}$$
2. Quant à op_3, elle est intégrée sur site 2 intègre sous la forme suivante :
$$IT(\overbrace{Ins(3,y,\{Del(3)\},\{\})}^{op_3},\overbrace{Ins(3,x,\{\},\{Del(3)\})}^{op_2}) = \overbrace{Ins(4,y,\{Del(3)\},\{\})}^{op'_3}$$
3. Ensuite, le site 2 intègre op_1 :
$$IT(IT(\overbrace{Ins(3,x,\{\},\{\})}^{op_1},\overbrace{Ins(3,x,\{\},\{Del(3)\})}^{op_2}),\overbrace{Ins(4,y,\{Del(3)\},\{\})}^{op'_3}) =$$
$$\overbrace{Nop()}^{op'_1}$$
4. Finalement, le site 3 intègre op_1 :
$$IT(IT(\overbrace{Ins(3,x,\{\},\{\})}^{op_1},\overbrace{Ins(3,y,\{Del(3)\},\{\})}^{op_3}),\overbrace{Ins(3,x,\{\},\{Del(3)\})}^{op'_2}) =$$
$$\overbrace{Ins(5,x,\{\},\{\})}^{op''_1}$$

Nous constatons que l'intégration de op_1 sur les sites 2 et 3 ne produit pas la même opération (respectivement $Nop()$ et $Ins(5,x,\{\},\{\})$). La condition $TP2$ n'est donc pas satisfaite. Ceci mène vers une divergence de données sur les sites 2 et 3.

Le scénario de la figure 1.3 soulève une question : est-ce que ce scénario est réaliste ? En d'autres termes : est-ce qu'il existe une exécution réelle des opérations op_1, op_2 et op_3 avec les mêmes paramètres mentionnés dans le scénario ? Donc à partir du scénario de la figure 1.3 nous allons essayer de construire un scénario réel en se basant sur les interprétations suivantes :

- $op_1 = Ins(3,x,\{\},\{\})$: Les deux ensembles av et ap sont vides, ce qui signifie que op_1 n'a pas subi une transformation par rapport à une opération de suppression.
- $op_2 = Ins(3,x,\{\},\{Del(3)\})$: L'ensemble ap contient une opération $Del(3)$, ce qui nous amène à dire que op_2 est le résultat d'une transformation par rapport à une opération $Del(3)$. En utilisant l'algorithme 5, nous pouvons déduire que la position d'insertion de op_2 avant d'être transformée par rapport à $Del(3)$ était égale à 3. Aussi, $op_2 = IT(op''_2, Del(3))$ tel que $op''_2 = Ins(3,x,\{\},\{\})$.
- En effectuant le même raisonnement, nous déduisons que $op_3 = IT(op''_3, Del(3))$ tel que $op''_3 = Ins(4,y,\{\},\{\})$.

A partir des interprétations précédentes, nous pouvons construire un scénario réel (illustré à la figure 1.4) qui est une version détaillée de celui de la figure 1.3.

Analyse

Sur le site 2, op_1 et op_2 sont considérées comme équivalentes puisqu'elles insèrent à la même position deux caractères identiques. Aussi, comme la pro-

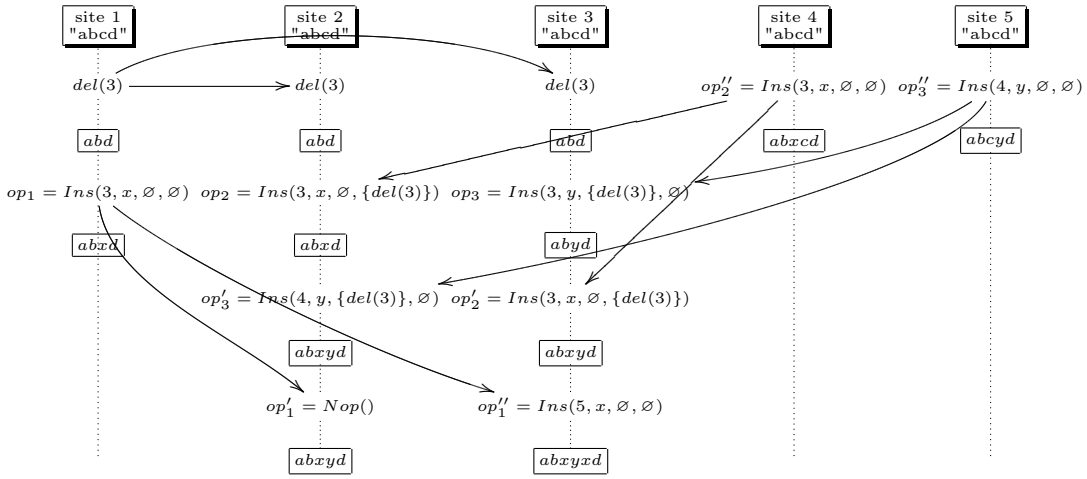


FIG. 1.4 – Contre-exemple détaillé pour la solution de Suleiman.

position de Suleiman garde un seul exemplaire, la transformation de op_1 par rapport à op_2 résulte en une opération nulle $Nop()$. Cependant, cette équivalence est perdue sur le site 3 lorsque op_1 et op_2 sont transformées par rapport à op_3. Cette perte d'information va causer donc la violation de $TP2$.

1.2.3 Algorithme d'Imine

Description

Dans [39], nous avons proposé un algorithme de transformation plus simple qui apporte une solution au problème du "$TP2$ puzzle". Pour ce faire, nous avons proposé un nouveau profil pour l'opération d'insertion à savoir $Ins(p, o, c)$, où p est la position d'insertion actuelle et o est la position originelle (ou initiale) d'insertion définie pendant la génération 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 des caractères effectivement à la même position. Si tel est le cas, nous comparons les "codes des caractères" comme dans Suleiman [81]. Sinon, nous décalons les positions en fonction de la relation entre les positions initiales.

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

L'algorithme 6 donne le contenu détaillé de notre solution.

L'utilisation de l'algorithme 6 apporte une solution au problème de Ressel. Le déroulement correct du scénario est donné à la figure 1.5. Intéressons-nous juste à l'intégration des différentes opérations sur le site u_2 :

– Quand op_3 arrive sur le site u_2, elle est transformée par rapport à op_2. Le résultat de cette transformation est l'opération $op'_3 = Ins(2, 2, y)$.

1: $IT(Ins(p_1, o_1, c_1), Ins(p_2, o_2, c_2)) =$
2: **si** $(p_1 < p_2)$ **alors**
3: **retourner** $Ins(p_1, o_1, c_1)$
4: **sinon si** $(p_1 > p_2)$ **alors**
5: **retourner** $Ins(p_1 + 1, o_1, c_1)$
6: **sinon si** $(o_1 < o_2)$ **alors**
7: **retourner** $Ins(p_1, o_1, c_1)$
8: **sinon si** $(o_1 > o_2)$ **alors**
9: **retourner** $Ins(p_1 + 1, o_1, c_1)$
10: **sinon si** $(code(c_1) < code(c_2))$ **alors**
11: **retourner** $Ins(p_1, o_1, c_1)$
12: **sinon si** $(code(c_1) > code(c_2))$ **alors**
13: **retourner** $Ins(p_1 + 1, o_1, c_1)$
14: **sinon**
15: **retourner** $Nop()$
16: **fin si**

17: $IT(Ins(p_1, o_1, c_1), Del(p_2)) =$
18: **si** $(p_1 \leq p_2)$ **alors**
19: **retourner** $Ins(p_1, o_1, c_1)$
20: **sinon**
21: **retourner** $Ins(p_1 - 1, o_1, c_1)$
22: **fin si**

23: $IT(Del(p_1), Ins(p_2, o_2, c_2)) =$
24: **si** $(p_1 < p_2)$ **alors**
25: **retourner** $Del(p_1)$
26: **sinon**
27: **retourner** $Del(p_1 + 1)$
28: **fin si**

29: $IT(Del(p_1), Del(p_2)) =$
30: **si** $(p_1 < p_2)$ **alors**
31: **retourner** $Del(p_1)$
32: **sinon si** $(p_1 > p_2)$ **alors**
33: **retourner** $Del(p_1 - 1)$
34: **sinon**
35: **retourner** $Nop()$
36: **fin si**

Algorithme 6: Algorithme de Imine et *al.*

- L'intégration de op_1 sur le site u_2 requiert une transformation par rapport à la séquence d'opérations $[op_2; op'_3]$. La première transformation résulte en $op_1^2 = IT(op_1, op_2) = Ins(2, 3, x)$. Nous remarquons que la position d'insertion est décrémentée mais la position initiale demeure la même.
- D'après leurs positions actuelles, les opérations $op_1^2 = Ins(2, 3, x)$ et $op'_3 = Ins(2, 2, y)$ sont en conflit. Aussi, les positions actuelles sont insuffisantes pour résoudre correctement ce conflit. Pour transformer op_1^2 par rapport

à op'_3, 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 $op'_1 = Ins(3, 3, x)$.

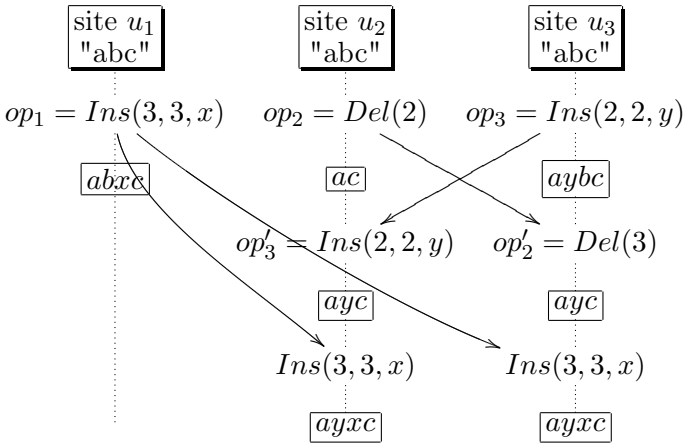


FIG. 1.5 – Solution de Imine pour le "$TP2$ puzzle".

Contre-exemple

Dans [39], nous avons affirmé que l'algorithme 6 satisfait la condition $TP2$ en se basant sur l'hypothèse suivante : les opérations d'insertion intervenant dans $TP2$ n'avaient pas subi de transformation auparavant. En d'autres termes, elles possèdent leurs positions actuelles égales à leurs position originales. Or cette hypothèse n'est pas réaliste puisqu'une opération d'insertion peut être le résultat d'une série de transformation et donc elle peut exister sous la forme où sa position actuelle est différente de sa position initiale.

En prenant en compte que les positions actuelle et initiale peuvent être quelconques, une nouvelle vérification de l'algorithme 6 a révélé un contre-exemple qui est présenté à la figure 1.6.

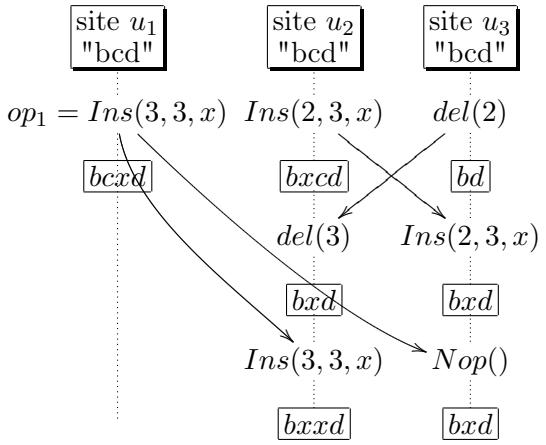


FIG. 1.6 – Contre-exemple pour la solution de Imine.

Le scénario de la figure 1.6 comporte trois opérations concurrentes : $op_1 = Ins(3, 3, x)$, $op_2 = Ins(2, 3, x)$ et $op_3 = Del(2)$. Nous remarquons que les positions actuelle et initiale de op_2 sont différentes. Ceci signifie que op_2 est le résultat d'une transformation par rapport à une opération d'effacement, à savoir $op_2 = IT(op_2'', Del(p))$ tels que $op_2'' = Ins(3, 3, x)$ et $p < 3$. Cette constatation va nous amener à construire un scénario plus détaillé et réel (voir la figure 1.7) pour le contre-exemple de la figure 1.6.

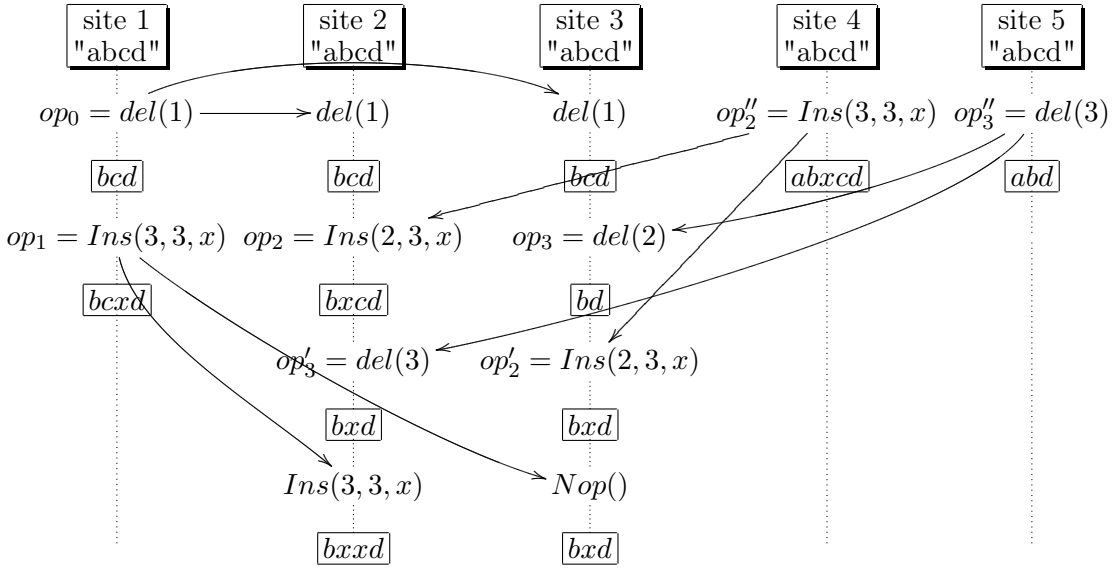


FIG. 1.7 – Contre-exemple détaillé pour la solution de Imine.

Analyse

L'utilisation des positions originales n'est pas suffisante pour résoudre le conflit occasionné par deux opérations transformées et qui n'ont pas été générées sur le même état initial. En effet, dans le contre-exemple de la figure 1.7, op_1 et op_2'' sont considérées comme équivalentes lorsqu'elles sont intégrées sur le site 3. Or, si nous comparons les effets de ces opérations à leur génération, elles ne sont pas équivalentes : l'effet de op_2'' est situé devant celui de op_1 puisque cette dernière insère le caractère x entre b, et c, tandis que op_2'' insère le caractère x entre les caractères a et b. Cette relation doit être préservée au cours des transformations sous peine de violation de $TP2$.

1.2.4 Algorithme de Du Li

Description

Toujours dans un souci de résoudre le problème du "$TP2$ puzzle", Du Li et *al.* [50] ont proposé une autre solution basée sur le concept de *transformation par différence d'états*, ou en anglais "State Difference Transformation (SDT)". Ils utilisent deux notions importantes, à savoir :

- La transformation inverse de IT, notée ET, et que l'on appelle en anglais "exclusion transformation" [85]. Ce type de transformation exclut l'effet d'une opération par rapport à une autre opération. Par exemple, si $op'_1 = IT(op_1, op_2)$ alors $ET(op'_1, op_2) = op_1$.
- Une fonction β qui calcule pour toute opération la position qu'elle aurait pu avoir sur un état précédent de convergence (ou de synchronisation). A titre d'exemple, considérons un état de synchronisation s_0 = "*abcd*" et une séquence d'opérations $SQ = [op_1; op_2]$ exécutée sur s_0, tels que $op_1 = Del(2)$ et $op_2 = Ins(3, y)$. L'exécution de op_1 sur s_0 donne l'état s_1 = "*acd*" et l'exécution de op_2 sur s_1 produit l'état "*acyd*". Le calcul de β par rapport à s_0 se fait comme suit : comme op_1 est définie directement sur s_0 alors $\beta(op_1) = 2$. Par contre pour calculer $\beta(op_2)$ nous devons exclure l'effet de op_1 de op_2, car op_2 n'est pas exécutée directement sur s_0 ; nous obtenons donc $\beta(op_2) = \beta(Ins(4, y)) = 4$ tel que $Ins(4, y) = ET(op_2, op_1)$.

Ainsi, lors de la transformation de deux opérations concurrentes d'insertion, Du Li et *al* recourent à l'utilisation de β pour voir si les deux opérations ont des positions égales ou différentes sur un état de synchronisation donné. Si sur cet état les positions sont égales, alors la transformation se fera de la même manière que la proposition de Ressel (voir l'algorithme 4).

L'algorithme 7 donne le contenu détaillé de la solution Du Li *al.*

L'algorithme de Du Li apporte une solution au problème du "$TP2$ puzzle" (voir figure 1.1). En effet, en considérant l'état "abc" comme étant le dernier état de synchronisation sur lequel β sera calculée, nous avons donc $\beta(op_1) = 3$, $\beta(op_2) = 2$ et $\beta(op_3) = 2$. Ces valeurs nous rappellent la notion de position initiale utilisée dans la proposition de Imine et *al.* Dans ce cas, le scénario utilisant les valeurs de β va se comporter de la même façon que celui utilisant des positions initiales (voir figure 1.5).

Contre-exemple

La vérification de l'algorithme 7 a décelé un contre-exemple potentiel qui est illustré sur la figure 1.8. Supposons qu'il existe un état antérieur de convergence (autre que s_2) sur lequel β est calculée, telles que $\beta(op_1) = \beta(op_2) = \beta(op_3)$. Le scénario se déroule comme suit :

1. Quand op_2 (resp. op_3) arrive sur le site 2 (resp. 1), elle sera transformée par rapport à op_3 (resp. op_2) pour produire $op'_2 = Ins(2, z)$ (resp. $op'_3 = Del(2)$).
2. L'intégration de op_1 sur le site 1 nécessite la transformation par rapport à op_2 et ensuite op'_3. Comme $\beta(op_1) = \beta(op_2)$ et les positions de op_1 et op_2 sont différentes (*i.e.* $2 < 3$) alors $IT(op_1, op_2) = op_1$; la forme finale à intégrer est $IT(op_1, op'_3) = Ins(2, x)$.
3. Une fois sur le site 2, nous avons $IT(op_1, op_3) = op_1$. Transformer op_1 par rapport op'_2 requiert l'utilisation des identifiants des sites puisque les deux opérations ont les mêmes valeurs pour β ainsi que les mêmes positions d'insertion. Ce qui résulte en $IT(op_1, op'_2) = Ins(3, x)$.

1: Soient $op_1 = Ins(p_1, c_1, u_1)$ et $op_2 = Ins(p_2, c_2, u_2)$
2: $IT(op_1, op_2) =$
3: **si** $\beta(op_1) < \beta(op_2)$ **alors**
4: **retourner** $Ins(p_1, c_1, u_1)$
5: **sinon si** $\beta(op_1) > \beta(op_2)$ **alors**
6: **retourner** $Ins(p_1 + 1, c_1, u_1)$
7: **sinon si** $p_1 < p_2$ **alors**
8: **retourner** $Ins(p_1, c_1, u_1)$
9: **sinon si** $p_1 > p_2$ **alors**
10: **retourner** $Ins(p_1 + 1, c_1, u_1)$
11: **sinon si** $u_1 < u_2$ **alors**
12: **retourner** $Ins(p_1, c_1, u_1)$
13: **sinon**
14: **retourner** $Ins(p_1 + 1, c_1, u_1)$
15: **fin si**

16: $IT(Ins(p_1, c_1, u_1), Del(p_2, u_2)) =$
17: **si** $(p_1 \leq p_2)$ **alors**
18: **retourner** $Ins(p_1, c_1, u_1)$
19: **sinon**
20: **retourner** $Ins(p_1 - 1, c_1, u_1)$
21: **fin si**

22: $IT(Del(p_1, u_1), Ins(p_2, c_2, u_2)) =$
23: **si** $(p_1 < p_2)$ **alors**
24: **retourner** $Del(p_1, u_1)$
25: **sinon**
26: **retourner** $Del(p_1 + 1, u_1)$
27: **fin si**

28: $IT(Del(p_1, u_1), Del(p_2, u_2)) =$
29: **si** $(p_1 < p_2)$ **alors**
30: **retourner** $Del(p_1, u_1)$
31: **sinon si** $(p_1 > p_2)$ **alors**
32: **retourner** $Del(p_1 - 1, u_1)$
33: **sinon**
34: **retourner** $Nop()$
35: **fin si**

Algorithme 7: Algorithme de Du Li et *al.*

Là aussi, nous nous posons la même question : le scénario de la figure 1.8 est-il réel ? Existe-t-il une exécution réelle pour les opérations op_1, op_2 et op_3 ? La réponse à de telles questions se trouve dans la figure 1.9 où nous donnons un scénario complet. Les valeurs de β sont calculées par rapport à l'état s_0 = "*abcd*". Nous avons donc les valeurs suivantes :

- Comme $op_1 = IT(op_1, op_0)$ alors $\beta(op_1) = 2$.
- Nous avons $\beta(op_2) = \beta(op_2'') = 2$ puisque $op_2 = IT(IT(op_2'', op_0), op_5)$.
- Comme il n'existe pas une opération op telle que $op_3 = IT(op, op_5)$, alors

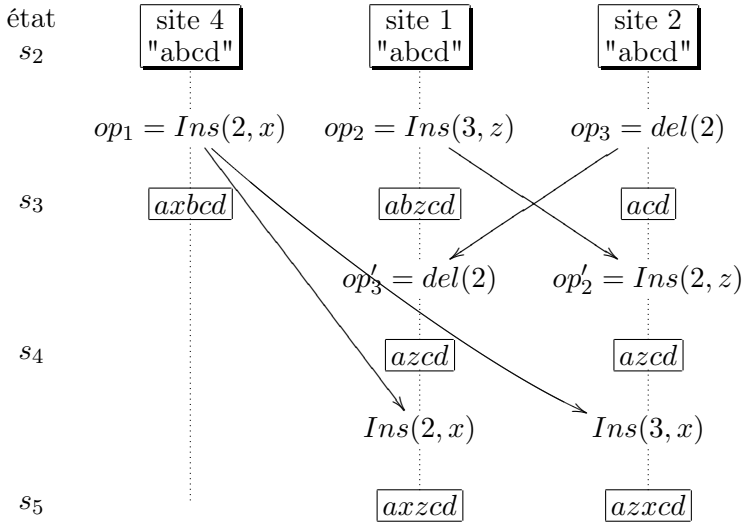


FIG. 1.8 – Contre-exemple pour la solution de Du Li.

$\beta(op_3) = \beta(op_5) = 2$.

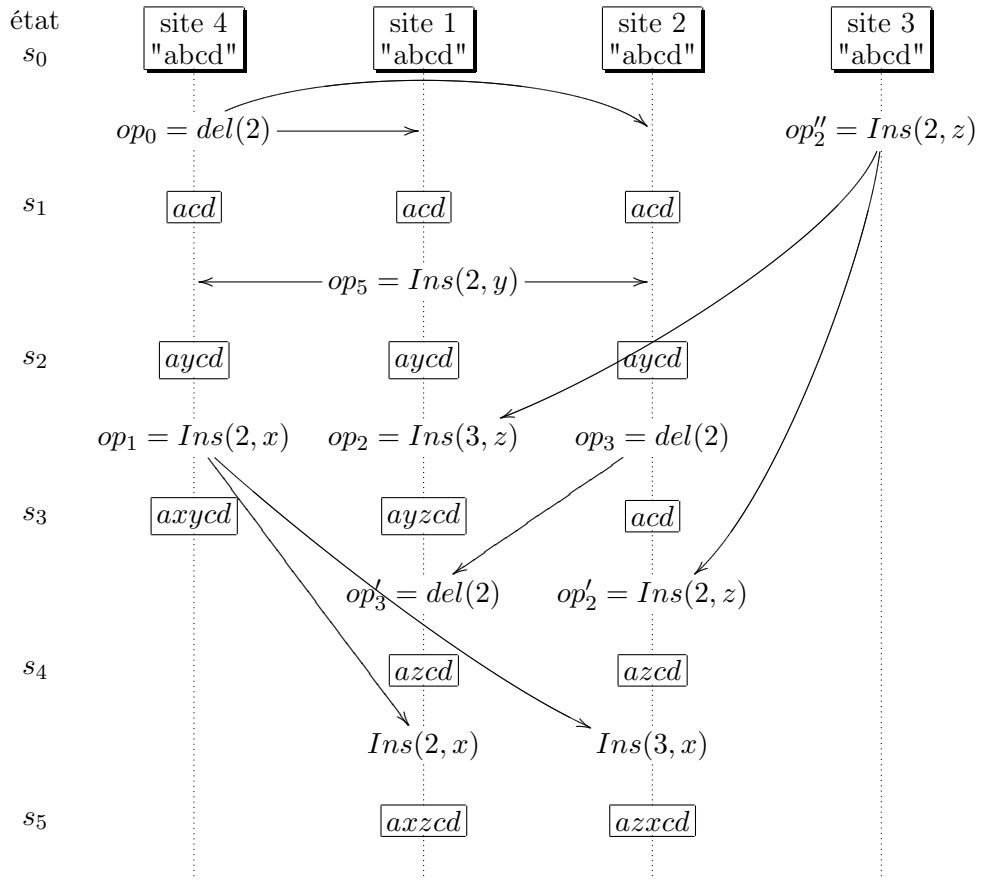


FIG. 1.9 – Contre exemple détaillé pour la solution de Du Li.

Analyse

D'après la contre-exemple de la figure 1.9, il est clair que l'utilisation de la fonction β s'avère inefficace pour résoudre le conflit causé par deux opérations d'insertion, en l'occurrence op_1 et op_2''. Le lecteur pourra facilement remarquer que l'effet de op_1 est devant celui de op_2''. Or cette relation est perdue une fois que les deux opérations sont intégrées sur le site 2. En effet, op_1 et op_2'' seront en conflit sur le site 2 (même valeur pour β et même positions). Dans ce cas, les identifiants des sites sont utilisés pour les départager ; ce qui impose un ordre d'insertion contraire à la relation mentionnée auparavant.

1.3 Source du Problème

Situations de conflit

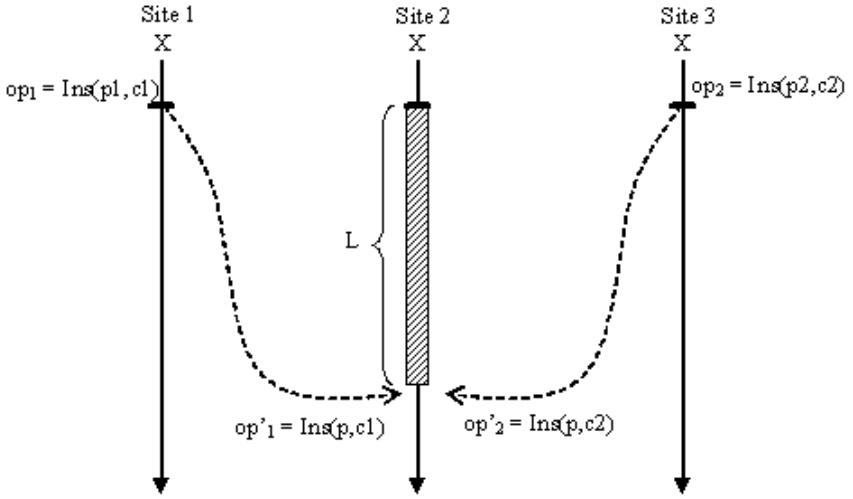
Comme vu précédemment, un conflit apparaît lorsque deux opérations concurrentes d'insertion tentent d'insérer des caractères à la même position. Dans la figure 1.10, nous présentons deux situations de conflit qui impliquent trois sites (ou utilisateurs) démarrant une session de collaboration à partir du même état initial X. Dans les deux situations, il y a deux opérations d'insertions op_1 et op_2 générées sur des sites différents ; op_1 et op_2 peuvent avoir ou non les mêmes positions d'insertion à leur génération.

Concernant la première situation (voir la figure 1.10(a)), op_1 et op_2 sont générées sur le même état X. Sur le site 2, une séquence d'opérations L a été exécutée, dont les opérations sont considérées comme étant concurrentes à op_1 et op_2. Il est clair que op_1 et op_2 peuvent arriver après L dans un ordre arbitraire. Leur intégration sur le site 2 nécessite donc une transformation par rapport aux opérations de L.

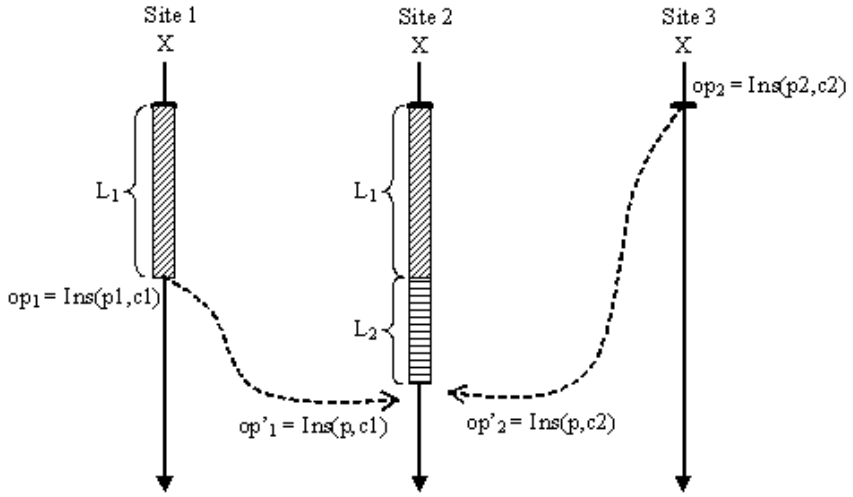
Quant à la deuxième situation, op_1 et op_2 ne sont pas générées sur le même état ; il y a un décalage d'états (voir la figure 1.10(b)). Sur le site 1, une séquence d'opérations L_1 se trouve sur le contexte de génération de op_1. La séquence L_1 est équivalente à une autre séquence L_1' exécutée sur le site 2. En considérant $L = [L_1'; L_2]$, op_1 et op_2 peuvent également arriver dans n'importe quel ordre après L. Pour être intégrées sur le site 2, op_1 et op_2 doivent être transformées respectivement par rapport à L_2 et L.

Quelle que soit la situation considérée, l'intégration de op_1 et op_2 sur le site 2 (après L) va produire deux opérations op_1' et op_2' qui ont la même position d'insertion p. Deux séquences d'opérations peuvent donc être exécutées après L : $[op_1'; IT(op_2', op_1')]$ et $[op_2'; IT(op_1', op_2')]$. Par ailleurs, il faut noter que op_1' et op_2' sont en conflit alors que peut être op_1 et op_2 ne le sont pas.

Le processus de transformation peut conduire deux opérations concurrentes (ayant des positions d'insertion différentes avant la transformation) à devenir en conflit. Malheureusement, la relation initiale entre les positions de ces opérations est perdue à cause de leurs transformations par rapport à d'autres opérations. Par conséquent, il est primordial de récupérer une telle relation pour éviter des problèmes de divergence.



(a) Etats de génération identiques



(b) Etats de génération différents

FIG. 1.10 – Situations de conflit pour des opérations d'insertion.

Utilisation d'informations supplémentaires

En plus des paramètres effectifs des opérations d'insertion (position et caractère à insérer), plusieurs algorithmes de transformation recourent à l'utilisation d'informations supplémentaires pour résoudre un conflit, telles que :

- les priorités sur les sites [26] ;
- les identifiants des sites [73] ;
- ensembles mémorisant les opérations de suppression [81] ;
- position à la génération de l'opération [39] ;
- position par rapport à un antécédent état de convergence [50].

Un ordre total est défini sur la plupart des informations définies au-dessus. Pour fixer et analyser le problème, nous allons considérer $\mathcal{I}_{op}$ comme étant une information supplémentaire quelconque utilisée par une opération d'insertion *op*,

et $\mathcal{R}$ est une relation d'ordre total sur l'information supplémentaire. De manière informelle, nous définissons la relation d'effets existante entre deux opérations $op_1 = Ins(p_1, c_1)$ et $op_2 = Ins(p_2, c_2)$ comme suit :

$$op_1 \sqsubset op_2 \text{ ssi } (p_1 < p_2) \text{ ou } (p_1 = p_2 \text{ et } \mathcal{R}(\mathcal{I}_{op_1}, \mathcal{I}_{op_2}))$$

$op_1 \sqsubset op_2$ signifie tout simplement que l'effet de op_1 est devant celui de op_2. Nous avons besoin de vérifier que cette relation est préservée (ou monotone) par tout algorithme de transformation IT. En d'autres termes, nous avons :

$$\text{si } op_1 \sqsubset op_2 \text{ alors } IT(op_1, op) \sqsubset IT(op_2, op) \tag{1.1}$$

pour toute opération op. La préservation de cette relation par transformation signifie que les opérations d'insertions seront transformées l'une par rapport à l'autre de la même manière sur tout site où elles seront intégrées. Malheureusement, la condition (1.1) n'est pas préservée par les algorithmes que nous avons passés en revue dans la section précédente.

Le problème du "$TP2$ puzzle" illustré à la figure 1.1 corresponds à la première situation de conflit que nous avons décrite auparavant (voir la figure 1.10(a)). En effet, nous avons deux opérations $op_1 = Ins(3, x)$ et $op_3 = Ins(2, y)$ générées sur le même état ($X =$ "abc'') et qui ne sont pas en conflit. L'algorithme de transformation IT de Ressel utilise comme information supplémentaire $\mathcal{I}_{op}$ l'identifiant des sites et comme relation d'ordre $\mathcal{R}$ la relation "inférieur strictement ($<$)". Nous avons donc $\mathcal{I}_{op_1} = u_1$ et $\mathcal{I}_{op_3} = u_3$ avec $u_1 < u_3$. Si nous considérons $op = Del(2)$, alors il est clair que $op_3 \sqsubset op_1$ mais $IT(op_3, op) \not\sqsubset IT(op_1, op)$. Cette perte de relation fait que op_1 et op_3 ne se transforment pas l'une par rapport à l'autre de la même façon sur les sites u_2 et u_3. Ce qui engendre un problème de divergence.

Quant aux contre-exemples donnés dans les figures 1.4, 1.7 et 1.9, ils correspondent à la deuxième situation de conflit (voir la figure 1.10(b)). Le problème provient de deux opérations concurrentes d'insertions qui ne sont pas générées sur le même état. A titre d'exemple, considérons le contre-exemple illustré à la figure 1.7 et qui concerne l'algorithme de Imine. Les opérations en question sont $op_1 = Ins(2, 3, x)$ et $op_2' = Ins(3, 3, x)$. L'information supplémentaire $\mathcal{I}_{op}$ utilisée ici est la position initiale. Nous avons donc $\mathcal{I}_{op_1} = 3$ et $\mathcal{I}_{op_2} = 3$. Il peut être facilement vérifié que la relation $op_1 \sqsubset op_2$ n'est pas préservée par transformation puisque $IT(op_1, op) \not\sqsubset IT(op_2, op)$ pour $op = Del(2)$. Par conséquent, les deux opérations op_1 et op_2 se transforment différemment l'une par rapport à l'autre sur les sites 2 et 3. Ce qui cause aussi un problème de divergence.

1.4 Conclusion

Dans ce chapitre, nous avons passé en revue quatre algorithmes de transformation spécifiques à des objets linéaires. Aucun de ces algorithmes ne parvient à satisfaire $TP2$. Ainsi, pour chaque algorithme, nous avons présenté un possible scénario de divergence qui est du à la violation de $TP2$.

Nous avons constaté que le problème provenait du conflit que peut engendrer deux opérations d'insertion. Le processus de transformation peut conduire deux

opération concurrentes d'insertion (ayant des positions d'insertion différentes avant la transformation) à devenir en conflit. Malheureusement, la relation originelle entre les positions de ces deux opérations est perdue à cause de leurs transformations par rapport à d'autres opérations. Par conséquent, il est crucial de récupérer une telle relation pour éviter des problèmes de divergence.

L'identification du problème de la violation de $TP2$ nous a permis de proposer un nouvel algorithme de transformation qui sera présentée dans le chapitre suivant.

Chapitre 2

Algorithme de Transformation pour des Objets Collaboratifs Linéaires

Sommaire

2.1 Introduction

Dans ce chapitre, nous allons proposer une nouvelle solution de transformation pour des objets collaboratifs qui ont une structure linéaire. Pour illustrer cette solution par des exemples, nous avons utilisé un objet "chaîne de caractère". Le principe de la solution est simple : nous stockons les différentes positions occupées par un caractère durant le processus de transformation. Au lieu d'utiliser une seule position nous allons maintenir une trace de positions. En raisonnant sur ces traces, il sera possible dans certaines situations de conserver une relation entre les positions de deux opérations concurrentes d'insertion.

Le présent chapitre se compose comme suit. Dans la section 2.2, nous allons définir le concept de "mot de positions" qui n'est d'autre qu'une trace de positions. Nous allons présenter le nouvel algorithme de transformation à la

section 2.3. Dans la section 2.4, nous allons traiter de la correction de cet algorithme en explicitant dans quelles situations la convergence peut être garantie. La section 2.5 va présenter un examen critique quand à l'utilisation pratique de notre solution dans les éditeurs collaboratifs. Enfin, nous terminons ce chapitre par une conclusion.

2.2 Mot de Positions

Nous allons présenter quelques notions de bases sur les alphabets et les mots. Pour plus de détails, le lecteur pourra consulter les références suivantes [91; 49].

Définition 2.1 (Alphabet et mots) *Pour tout ensemble de symboles $\mathcal{A}$ totalement ordonné, appelé* alphabet*, $\mathcal{A}^*$ représente l'ensemble des* mots *sur$\mathcal{A}$, tels que :*

(i) pour tout $\omega \in \mathcal{A}^*$, $|\omega|$ représente la longueur *de ω ;*

(ii) si $\omega = uv$ alors u est un préfixe de ω et v est un suffixe *de ω, pour u, $v \in \mathcal{A}^*$;*

(iii) pour tout $\omega \in \mathcal{A}^*$, $Base(\omega)$ et $Top(\omega)$ donnent respectivement le dernier et le premier élément de ω ;

(iv) $Tail(\epsilon) = \epsilon$ et $Tail(n\rho) = \rho$ pour $n \in \mathcal{A}$ et $\rho \in \mathcal{A}^*$;

(v) si ω_1, $\omega_2 \in \mathcal{A}^*$ alors $\omega_1 \preceq \omega_2$ est l'ordre lexicographique *défini comme suit :*

(a) ω_1 est un préfixe de ω_2, ou

(b) $\omega_1 = \rho u$ et $\omega_2 = \rho v$, où $\rho \in \mathcal{A}^*$ est le plus long préfixe de ω_1 et ω_2, et $Top(u)$ précède $Top(v)$ selon l'ordre alphabétique de $\mathcal{A}$. ■

Par exemple, si nous avons $\omega_1 = abcde$ et $\omega_2 = z$ alors $Top(abcde) = a$, $Base(abcde) = e$ et $\omega_1 \prec \omega_2$ selon l'ordre alphabétique.

Définition 2.2 (Mot des positions) *Nous considérons l'ensemble des nombres naturels $\mathbb{N}$ comme un alphabet. L'ensemble des* mots des positions*, ou* p-mots*, est défini comme suit :*

(i) $\epsilon \in \mathcal{P}$;

(ii) si $n \in \mathbb{N}$ alors $n \in \mathcal{P}$;

(iii) si ω est un p-mot non vide et $n \in \mathbb{N}$ alors $n\omega \in \mathcal{P}$ ssi $Top(\omega) \in \{n, n+1, n-1\}$. ■

Nous pouvons immédiatement remarquer que la concaténation de deux p-mots produit un autre p-mot si le dernier élément du premier p-mot diffère d'au plus 1 par rapport au premier élément du deuxième p-mot.

Proposition 2.1 *Soient ω_1 et ω_2 deux p-mots non vides. La concaténation de ω_1 et ω_2, notée $\omega_1 \cdot \omega_2$ ou simplement $\omega_1\omega_2$, est un p-mot ssi (i) $Base(\omega_1) = Top(\omega_2)$, ou (ii) $Base(\omega_1) = Top(\omega_2) \pm 1$.* ■

Preuve. Nous allons procéder par induction sur la longueur de ω_1 :

- *Base d'induction* : $|\omega_1| = 1$ et ω_1 est un entier naturel n. Alors $n\omega_2$ est aussi un p-mot en utilisant la définition 2.2.
- *Hypothèse d'induction* : Si $|\omega_1| \leq l$ alors $\omega_1\omega_2$ est un p-mot.
- *Pas d'induction* : Soit $|\omega_1| = l + 1$. Alors ω_1 peut s'écrire comme $m\rho$ tels que $m \in \mathbb{N}$ et $\rho \in \mathcal{P}$. Comme $|\rho| = l$, nous avons donc $\omega_1\omega_2 = (m\rho)\omega_2 = m(\rho\omega_2)$ puisque la concaténation est associative. En utilisant la définition 2.2 et l'hypothèse d'induction, nous pouvons alors conclure que $\omega_1\omega_2$ est un p-mot.

■

A titre d'exemple, si $\omega_1 = 00$ et $\omega_2 = 1232$ sont deux p-mots alors $\omega_1\omega_2$ est un p-mot ; par contre $\omega_2\omega_1$ n'est pas un p-mot.

Nous allons définir une relation d'équivalence sur l'ensemble des p-mots.

Définition 2.3 (Equivalence des p-mots) *Deux p-mots ω_1 et ω_2 sont dits* équivalents*, notée $\omega_1 \equiv_{\mathcal{P}} \omega_2$, ssi $Top(\omega_1) = Top(\omega_2)$ et $Base(\omega_1) = Base(\omega_2)$.* ■

La relation d'équivalence $\equiv_{\mathcal{P}}$ est une congruence à droite.

Proposition 2.2 (Congruence à droite) *Considérons les p-mots ω_1, ω_2 et ρ tels que $\omega_1\rho$ et $\omega_2\rho$ sont aussi des p-mots. Si $\omega_1 \equiv_{\mathcal{P}} \omega_2$ alors $\omega_1\rho \equiv_{\mathcal{P}} \omega_2\rho$.* ■

Preuve. En utilisant les définitions 2.1 et 2.3. ■

2.3 Algorithme de Transformation

Afin de préserver une relation entre les positions de deux opérations d'insertions, nous proposons de stocker les différentes positions occupées par un caractère durant le processus de transformation. En d'autres termes, au lieu d'utiliser un seule position nous allons maintenir un mot de positions, ou un p-mot. A chaque fois qu'une opération d'insertion est transformée nous gardons sa dernière position avant la transformation dans un p-mot. La taille de ce mot est proportionnelle au nombre des opérations concurrentes.

Dans notre nouvel algorithme de transformation, nous avons redéfini le profil d'une opération d'insertion comme $Ins(p, c, \omega, i)$, où :

- p est la position d'insertion ;
- c est le caractère à insérer ;
- ω est le p-mot qui garde toutes les positions occupées par c ;
- i est l'identifiant du site.

A la génération le p-mot d'une opération d'insertion est toujours vide, *i.e.* $Ins(p, c, \epsilon, i)$. Lorsqu'une opération d'insertion est transformée de telle façon que sa position est modifiée, la position originelle sera donc concaténée au p-mot. Par exemple :

$$IT(Ins(3, x, \epsilon, i), Del(1)) = Ins(2, x, [3], i)$$
$$IT(Ins(2, x, [3], i), Ins(1, y, \epsilon, i)) = Ins(3, x, [2.3], i)$$

Nous définissons une fonction PW qui est utilisée pour construire des p-mots à partir des opérations d'édition. Elle prend comme argument une opération et retourne un p-mot :

$$\begin{aligned} PW(Ins(p,c,\omega,pr)) &= \begin{cases} p & \text{si } \omega = \epsilon \\ p\omega & \text{si } \omega \neq \epsilon \text{ et } (p = Top(\omega) \text{ ou } p = Top(\omega) \pm 1) \\ \epsilon & \text{sinon} \end{cases} \\ PW(Del(p)) &= p \\ PW(Nop()) &= \epsilon \end{aligned}$$

Lorsque deux opérations d'insertions créent deux caractères à la même position (par définition elles sont en conflits), une décision doit être prise : quel caractère doit être inséré avant l'autre ? En général, la solution adoptée consiste à munir chaque opération d'une priorité, tel que l'identifiant du site où l'opération a été générée [73]. Ce faisant, le caractère ayant la plus haute priorité sera inséré après l'autre. Si les p-mots de deux opérations d'insertion sont identiques alors ces deux opérations sont équivalentes. Sinon, leurs p-mots permettent d'inférer la relation qui existe entre les positions des deux opérations en question.

L'algorithme 8 illustre le contenu détaillé de notre solution.

Nous définissons la partie stricte d'une relation d'ordre sur les opérations d'insertion.

Définition 2.4 (Relation d'effet) *Soient deux opérations d'insertion* $op_1 = Ins(p_1, c_1, \omega_1, i_1)$ *et* $op_1 = Ins(p_2, c_2, \omega_2, i_2)$ *:* $op_1 \sqsubset op_2$ *ssi l'une des conditions suivantes est satisfaite : (i)* $PW(o_1) \prec PW(o_2)$ *; (ii)* $PW(o_1) = PW(o_2)$ *et* $i_1 < i_2$*. Nous utilisons aussi la notation* $op_2 \sqsupset op_1$ *pour définir que* $op_1 \sqsubset op_2$*.* ■

La relation d'effet indique tout simplement comment les caractères insérés sont reliés entre eux. Ainsi, $op_1 \sqsubset op_2$ signifie que le caractère inséré par op_1 se trouve à gauche (ou avant) de celui de op_2.

Dans le chapitre précédent (voir section 1.3), nous avons présenté deux situations de conflit qui mettent en jeu deux opérations concurrentes d'insertion. Aussi, nous allons appliquer notre algorithme sur deux scénarios de divergence issus des deux situations de conflit et ce pour montrer comment nous pouvons atteindre la convergence.

La figure 2.1 montre comment la convergence est garantie pour le "$TP2$ puzzle" (contre-exemple pour la proposition de Ressel). Le "$TP2$ puzzle" est issu de la première situation de conflit, puisqu'il y a deux opérations d'insertion générées sur le même état. Nous pouvons remarquer que l'effet de op_1 est derrière l'effet de op_2, *i.e.* $op_2 \sqsubset op_1$. Cette relation doit être préservée durant le processus de transformation. Ainsi, nous avons les étapes suivantes :

- quand op_1 arrive sur le site u_2, elle est transformée par rapport à op_2 pour devenir l'opération $Ins(2, x, [3])$;
- puis l'opération $Ins(2, x, [3])$ est transformée par rapport à $op'_3 = Ins(2, y, [2])$; en comparant les p-mots nous avons la relation $[2.2] \prec [2.3]$ ce qui permet de produire après transformation $op'_1 = Ins(3, x, [2.3]$

1: Soient $\alpha_1 = PW(Ins(p_1, c_1, \omega_1, i_1)$ et $\alpha_2 = PW(Ins(p_2, c_2, \omega_2, i_2)$
2: $IT(Ins(p_1, c_1, \omega_1, i_1), Ins(p_2, c_2, \omega_2, i_2)) =$
3: **si** ($\alpha_1 \prec \alpha_2$ or ($\alpha_1 = \alpha_2$ and $i_1 < i_2$)) **alors**
4: **retourner** $Ins(p_1, c_1, \omega_1, i_1)$
5: **sinon**
6: **retourner** $Ins(p_1 + 1, c_1, p_1\omega_1, i_1)$
7: **fin si**

8: $IT(Ins(p_1, c_1, \omega_1, i_1), Del(p_2)) =$
9: **si** ($p_1 > p_2$) **alors**
10: **retourner** $Ins(p_1 - 1, c_1, p_1\omega_1, i_1)$
11: **sinon si** ($p_1 < p_2$) **alors**
12: **retourner** $Ins(p_1, c_1, \omega_1, i_1)$
13: **sinon**
14: **retourner** $Ins(p_1, c_1, p_1\omega_1, i_1)$
15: **fin si**

16: $IT(Del(p_1), Ins(p_2, c_2, \omega_2, i_2)) =$
17: **si** ($p_1 < p_2$) **alors**
18: **retourner** $Del(p_1)$
19: **sinon**
20: **retourner** $Del(p_1 + 1)$
21: **fin si**

22: $IT(Del(p_1), Del(p_2)) =$
23: **si** ($p_1 < p_2$) **alors**
24: **retourner** $Del(p_1)$
25: **sinon si** ($p_1 > p_2$) **alors**
26: **retourner** $Del(p_1 - 1)$
27: **sinon**
28: **retourner** $Nop()$
29: **fin si**

Algorithme 8: Algorithme de transformation utilisant des p-mots.

La figure 2.2 montre comment nous atteignons la convergence dans un contre-exemple compliqué (voir le contre-exemple de la proposition de Imine illustré à la figure 1.7), qui relève de la deuxième situation de conflit. En effet, nous avons deux opérations concurrentes d'insertion $op_1 = Ins(3, x, \epsilon)$ et $op_2'' = Ins(3, x, \epsilon)$, générées sur deux états différents (respectivement "bcd" et "abcd"). Malgré le décalage d'états, il est clair que l'effet de op_1 est derrière celui de op_2'', *i.e.* $op_2 \sqsubset op_1$. Nous allons voir comment cette relation est préservée en utilisant les p-mots. Intéressons-nous juste à l'intégration de op_1 sur le site 3 :

- la transformation de op_1 par rapport à op_3 donne l'opération $Ins(2, x, [3])$;
- pour transformer $Ins(2, x, [3])$ par rapport à $op_2' = Ins(2, x, [2.3])$ nous devons comparer leurs p-mots ; comme $[2.2.3] \prec [2.3]$ nous obtenons donc $Ins(3, x, [2.3])$.

Ainsi, la relation entre op_1 et op_2'' est préservée et les sites 2 et 3 convergent vers le même état.

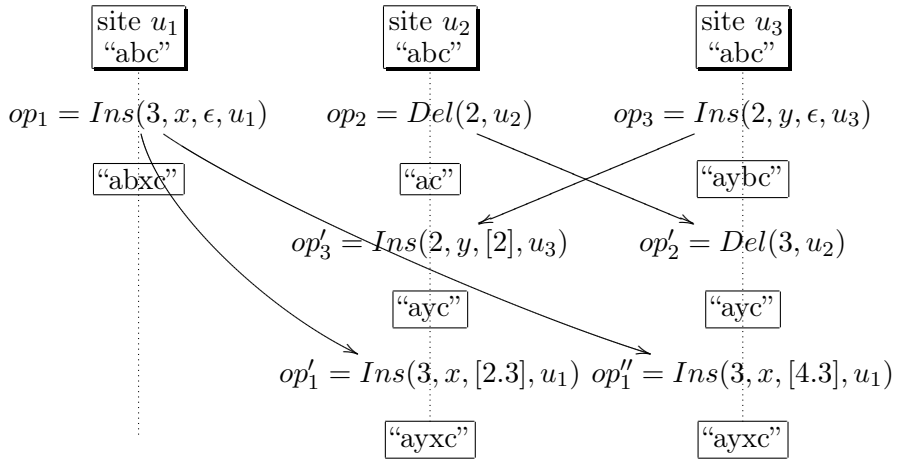


FIG. 2.1 – Exécution correcte du $TP2$-puzzle en utilisant les p-mots.

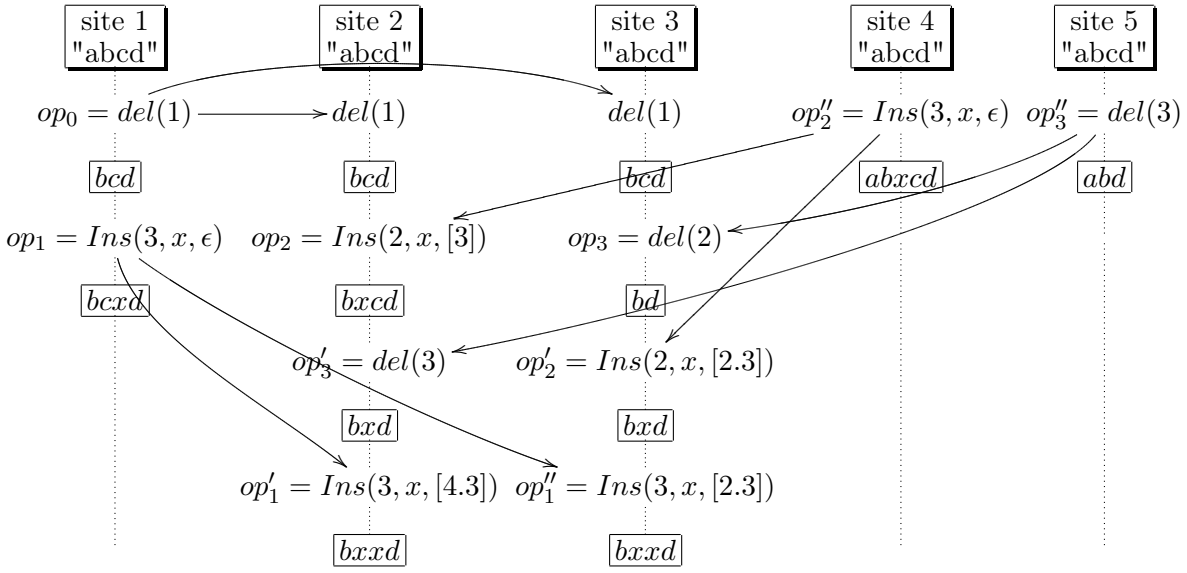


FIG. 2.2 – Correcte exécution du contre-exemple pour la proposition de Imine eu utilisant les p-mots.

Bien que l'utilisation des p-mots assure la convergence dans les scénarios précédents, la condition $TP2$ est toujours violée. Considérons la figure 2.1, il est clair que les opérations op'_1 et op''_1 (qui sont respectivement les formes intégrées de op_1 dans les sites u_2 et u_3) sont syntaxiquement différentes puisque les paramètres [2.3] et [4.3] ne sont pas identiques. Mais les deux opérations ont le même effet car leurs p-mots, $PW(op'_1) = [3.2.3]$ et $PW(op''_1) = [3.4.3]$, sont équivalents selon la définition 2.3 ; ce qui conduit à une convergence d'états.

A partir de l'équivalence des p-mots, nous définissons une relation d'équivalence entres les opérations :

Définition 2.5 (Equivalence des opérations) *Deux opérations sont dites* équivalentes *ssi l'une des conditions suivantes est satisfaite :*

(i) $op_1 = Ins(p_1, c_1, w_1, i_1)$, $op_2 = Ins(p_2, c_2, w_2, i_2)$, $c_1 = c_2$, $i_1 = i_2$, *et* $PW(op_1) \equiv_{\mathcal{P}} PW(op_2)$;

(ii) $op_1 = Del(p_1)$, $op_2 = Del(p_2)$ *et* $p_1 = p_2$.

Par abus de notation, nous écrivons $op_1 \equiv_{\mathcal{P}} op_2$. ■

En utilisant l'équivalence des opérations, nous proposons une forme faible de la condition $TP2$. Nous l'appellerons condition $TP2'$:

Définition 2.6 (Condition TP2′) *Si IT satisfait* $TP1$ *alors :*

$$IT^*(op, [op_1 ; IT(op_2, op_1)]) \equiv_{\mathcal{P}} IT^*(op, [op_2 ; IT(op_1, op_2)])$$

pour toutes opérations op, op_1 *et* op_2. ■

2.4 Correction

Dans cette section, nous allons présenter la correction de notre algorithme de transformation en abordant les points suivants :

1. La conservation par transformation des mots de positions ;
2. La préservation des relations d'effet entre les opérations d'insertion ;
3. la convergence des données.

2.4.1 Conservation des p-mots

Dans ce qui suit, nous montrons que notre algorithme de transformation ne perd aucune information sur les mots de positions :

Lemme 2.1 *Soit une opération d'insertion* $op_1 = Ins(p_1, c_1, \omega_1, i_1)$. *Pour toute opération op telle que* $op \parallel op_1$, $PW(op_1)$ *est un suffixe de* $PW(IT(op_1, op))$. ■

Preuve. Soient $op'_1 = IT(op_1, op)$ et $PW(op_1) = p_1\omega_1$. Alors, nous considérons deux cas :

1. $op = Ins(p, e, \omega, i)$: Comme $op \parallel op_1$ alors $i \neq i_1$. Soient $\alpha_1 = PW(op_1)$ et $\alpha_2 = PW(op)$.
   (a) Si $\alpha_1 \prec \alpha_2$ ou ($\alpha_1 = \alpha_2$ et $i_1 < i$) alors $op'_1 = op_1$.
   (b) Si $\alpha_1 \succ \alpha_2$ ou ($\alpha_1 = \alpha_2$ et $i_1 > i$) alors $op'_1 = Ins(p_1 + 1, c_1, p_1\omega_1, i_1)$ et $p_1\omega_1$ est un suffixe de $PW(op'_1)$.
2. $op = Del(p)$
   (a) Si $p_1 > p$ alors $op'_1 = Ins(p_1 - 1, c_1, p_1\omega_1, i_1)$; par conséquent $p_1\omega_1$ est un suffixe de $PW(op'_1)$.
   (b) Si $p_1 < p$ alors $op'_1 = op_1$.
   (c) Si $p_1 = p$ alors $op'_1 = Ins(p_1, c_1, p_1\omega_1, i_1)$ et $p_1\omega_1$ est un suffixe de op'_1.

■

Le théorème suivant énonce que l'extension de notre algorithme de transformation à des séquences d'opération préserve aussi les informations sur les mots de positions :

Théorème 2.1 *Etant donnée une opération d'insertion* $op_1 = Ins(p_1, c_1, \omega_1, i_1)$. *Pour toute séquence d'opérations seq,* $PW(op_1)$ *est un suffixe de* $PW(IT^*(op_1, seq))$. ■

Preuve. Soit n la longueur de seq. Nous allons procéder par induction sur la longueur de seq.

1. *Base d'induction* : $n = 0$. Alors seq est vide et nous avons $IT^*(op_1, []) = op_1$.
2. *Hypothèse d'induction* : pour $n \geq 0$, $PW(op_1)$ est un suffixe de $PW(T^*(op_1, seq))$.
3. *Etape d'induction* : Soit $seq = [seq'; op]$ où seq' est une séquence de longueur n et op est une opération quelconque. Nous avons $IT^*(op_1, [seq'; op]) = IT(IT^*(op_1, seq'), op)$. En utilisant le lemme 2.1, $PW(T^*(op_1, seq'))$ est un suffixe de :
   $PW(IT^*(op_1, [seq'; op])) = PW(IT(IT^*(op_1, seq'), op))$. En utilisant l'hypothèse d'induction et la transitivité de la relation "est suffixe", nous pouvons conclure que $PW(op_1)$ est un suffixe de $PW(T^*(op_1, seq))$ pour toute séquence d'opération seq.

■

2.4.2 Relation entre les opérations d'insertion

Nous pouvons utiliser des relation d'effet entre des positions d'insertion pour les considérer comme un *invariant* qui doit être préservé durant le processus de transformation. Considérons deux opérations d'insertion op_1 et op_2 ainsi que leurs formes transformées respectives op'_1 et op'_2. La propriété d'invariance s'énonce comme suit : si $op_1 \sqsubset op_2$ alors $op'_1 \sqsubset op'_2$.

Le lemme suivant montre qu'une relation d'effet est préservée par transformation par rapport à une autre opération.

Lemme 2.2 *Etant données deux opérations concurrentes d'insertion* op_1 *et* op_2. *Pour toute opération op telle que* $op \parallel op_1$ *et* $op \parallel op_2$ *:*

$$op_1 \sqsubset op_2 \text{ implique } IT(op_1, op) \sqsubset IT(op_2, op)$$

■

Preuve. Soient $op_1 = Ins(p_1, c_1, \omega_1, i_1)$, $op_2 = Ins(p_2, c_2, \omega_2, i_2)$, $op'_1 = IT(op_1, op)$ et $op'_2 = IT(op_2, op)$. Sans perdre de généralité, nous allons juste considérer $op = Del(p)$. Deux cas sont donc possibles :

1. $PW(op_1) \prec PW(op_2)$: Nous traitons tous les cas possibles des relations entre p, p_1 et p_2.

(a) $p_1 = p_2$:

i. $p < p_1 : op'_1 = Ins(p_1 - 1, c_1, p_1\omega_1, i_1)$ et $op'_2 = Ins(p_2 - 1, c_2, p_2\omega_2, i_2)$; comme $p_1\omega_1 \prec p_2\omega_2$ et $p_1 = p_2$ alors $PW(op'_1) \prec PW(op'_2)$.

ii. $p = p_1 : op'_1 = Ins(p_1, c_1, p_1\omega_1, i_1)$ et $op'_2 = Ins(p_2, c_2, p_2\omega_2, i_2)$; comme $p_1\omega_1 \prec p_2\omega_2$ alors $PW(op'_1) \prec PW(op'_2)$.

iii. $p > p_1 : op'_1 = op_1$ et $op'_2 = op_2$ alors $PW(op'_1) \prec PW(op'_2)$..

(b) $p_1 < p_2$:

i. $p < p_1 : op'_1 = Ins(p_1 - 1, c_1, p_1\omega_1, i_1)$ et $op'_2 = Ins(p_2 - 1, c_2, p_2\omega_2, i_2)$; $PW(op'_1) \prec PW(op'_2)$ car $p_1 - 1 < p_2 - 1$.

ii. $p = p_1 : op'_1 = Ins(p_1, c_1, p_1\omega_1, i_1)$ et $op'_2 = Ins(p_2 - 1, c_2, p_2\omega_2, i_2)$; $PW(op'_1) \prec PW(op'_2)$ pour $p_1 \leq p_2 - 1$.

iii. $p_1 < p < p_2 : op'_1 = op_1$ et $op'_2 = Ins(p_2 - 1, c_2, p_2\omega_2, i_2)$; $PW(op'_1) \prec PW(op'_2)$ car $p_1 < p_2 - 1$.

iv. $p = p_2 : op'_1 = op_1$ et $op'_2 = Ins(p_2, c_2, p_2\omega_2, i_2)$; $PW(op'_1) \prec PW(op'_2)$ car $p_1 < p_2$.

v. $p > p_2 : op'_1 = op_1$ et $op'_2 = op_2$ alors $PW(op'_1) \prec PW(op'_2)$.

2. $PW(op_1) = PW(op_2)$ et $i_1 < i_2$: nous déduisons que $p_1 = p_2$; ainsi la preuve de ce cas est la même que celle du cas $(1)(a)$.

■

Le théorème suivant montre que l'extension de notre algorithme à des séquences identiques préserve aussi les relations d'effet :

Théorème 2.2 *Considérons deux opérations concurrentes d'insertion op_1 et op_2. Pour toute séquence d'opérations seq dont les opérations sont concurrentes à op_1 et op_2, nous avons :*

Si $PW(op_1) \sqsubset PW(op_2)$ *alors* $PW(IT^*(op_1, seq)) \sqsubset PW(IT^*(op_2, seq))$.

■

Preuve. Considérons n comme étant la longueur de *seq*. Pour prouver ce théorème nous allons procéder par induction sur la longueur de *seq*.

1. *Base d'induction :* pour $n = 0$ nous avons $IT^*(op_1, []) = op_1$ et $IT^*(op_2, []) = op_2$.
2. *Hypothèse d'induction :* si $n \geq 0$ alors $PW(op_1) \prec PW(op_2)$ implique $PW(T^*(op_1, seq)) \prec PW(T^*(op_2, seq))$.
3. *Etape d'induction* : considérons $seq = [seq'; op]$ comme une séquence formée par seq' de longueur n et une opération quelconque op. Nous avons donc :

$$IT^*(op_1, [seq'; op]) = IT(IT^*(op_1, seq'), op) \text{ et}$$
$$IT^*(op_2, [seq'; op]) = IT(IT^*(op_2, seq'), op)$$

Après cette réécriture, nous pouvons utiliser l'hypothèse d'induction et le lemme 2.2 pour conclure que $\sqsubset$ est préservée par transformation pour toute séquence *seq*. ■

2.4.3 Convergence

Conditions $TP1$ et $TP2'$

Il faut rappeler que la condition $TP2'$ est une forme relaxée de $TP2$ (voir la définition 2.6). En effet, elle stipule que la transformation d'une opération par rapport à deux séquences équivalentes ne donne pas forcément le même résultat mais deux opérations équivalentes selon la définition 2.5.

Par ailleurs, les conditions $TP1$ et $TP2'$ ne font intervenir que des opérations concurrentes. A ce titre, notre algorithme de transformation basé sur les p-mots satisfait les deux conditions. Les théorèmes 2.3, 2.4 et 2.5 énoncés ci-dessous ont été automatiquement vérifié par l'intermédiaire du prouveur `SPIKE`.

Le théorème suivant montre que l'algorithme 8 satisfait $TP1$.

Théorème 2.3 *Pour toutes opérations op_1 et op_2 les séquences $[op_1; IT(op_2, op_1)]$ et $[op_2; IT(op_1, op_2)]$ sont équivalentes.* ■

Preuve. Soient $op'_1 = IT(op_1, op_2)$ et $op'_2 = IT(op_2, op_1)$. Sans perdre de généralité, nous allons considérer les couples suivants :

1. $op_1 = Ins(p_1, c_1, \omega_1, i_1)$ et $op_2 = Ins(p_2, c_2, \omega_2, i_2)$: selon les paramètres de op_1 et op_2 nous avons les cas suivants :
   (a) $PW(op_1) \prec PW(op_2)$: Selon cet ordre le caractère c_1 est inséré avant le caractère c_2. En effet, si op_1 a été exécutée alors lorsque op_2 arrive sa position d'insertion est incrémentée ($op'_2 = Ins(p_2{+}1, c_2, p_2\omega_2, i_2)$) de telle sorte que c_2 sera insérée à droite de c_1. Maintenant, si op_1 arrive après l'exécution de op_2 alors op_1 n'est pas transformée ($op'_1 = op_1$)et c_1 est inséré à gauche de c_2. Ainsi, les séquences $[op_1; op'_2]$ et $[op_2; op'_1]$ ont le même effet.
   (b) $PW(op_1) = PW(op_2)$ et $i_1 < i_2$: les deux opérations sont en conflit puisqu'elles insèrent deux caractères à la même position. Dans ce cas l'ordre d'insertion suivra l'ordre des identifiants des sites. Ainsi, $[op_1; op'_2]$ et $[op_2; op'_1]$ produisent le même effet.
2. $op_1 = Del(p_1)$ et $op_2 = Del(p_2)$: selon les paramètres de op_1 et op_2 trois cas sont possibles :
   (a) $p_1 = p_2$: les deux opérations ont détruit le même caractère. L'arrivée de l'une après l'exécution de l'autre produira l'opération nulle ($op'_1 = op'_2 = Nop()$) et donc $[op_1; op'_2]$ et $[op_2; op'_1]$ ont le même effet.
   (b) $p_1 < p_2$: le caractère détruit à la position p_1 se trouve à gauche du caractère effacé à la position p_2. Cet ordre d'effacement est respecté par les transformations : si op_2 arrive après l'exécution de op_1 alors la position de op_2 est décrémentée ($op'_2 = Del(p_2 - 1)$) ; par contre, l'arrivée de op_1 après l'exécution de op_2 ne changera pas la position p_1 ($op'_1 = Del(p_1)$). De ce fait, $[op_1; op'_2]$ et $[op_2; op'_1]$ mènent au même état.
   (c) $p_1 > p_2$: même raisonnement que le cas précédent. ■

Le théorème 2.4 montre que la transformation d'une opération d'effacement par rapport à deux séquences équivalentes produit deux opérations identiques.

Théorème 2.4 *Soit une opération d'effacement op. Si IT satisfait TP*1 *alors pour toutes opérations* op_1 *et* op_2 *nous avons :*

$$IT^*(op, [op_1; op'_2]) = IT^*(op, [op_2; op'_1])$$

avec $op'_1 = IT(op_1, op_2)$ *et* $op'_2 = IT(op_2, op_1)$. ■

Preuve. Soient $op = Del(p)$, $op' = IT^*(op, [op_2; op'_1])$ et $op" = IT^*(op, [op_1; op'_2])$. Sans perdre de généralité, nous allons considérons les opérations $op_1 = Del(p_1)$ et $op_2 = Del(p_2)$. Nous avons les cas suivants en tenant compte des paramètres de op_1, op_2 et op

1. $p_1 = p_2 : op'_1 = Nop()$ et $op'_2 = Nop()$; en fonction de op nous avons les cas suivants :
   (a) $p < p_1 : op' = Del(p)$ et $op" = Del(p)$.
   (b) $p = p_1 : op' = Nop()$ et $op" = Nop()$.
   (c) $p > p_1 : op' = Del(p-1)$ et $op" = Del(p-1)$.
2. $p_1 < p_2 : op'_1 = Del(p_1)$ et $op'_2 = Del(p_2 - 1)$; nous considérons les cas suivants en fonction de op :
   (a) $p < p_1 : op' = Del(p)$ et $op" = Del(p)$.
   (b) $(p = p_1)$ ou $(p_1 < p$ et $p = p_2) : op' = Nop()$ et $op" = Nop()$.
   (c) $p_1 < p < p_2 : op' = Del(p-1)$ et $op" = Del(p-1)$.
   (d) $p_2 < p : op' = Del(p-2)$ et $op" = Del(p-2)$.
3. $p_1 > p_2 : op'_1 = Del(p_1 - 1)$ et $op'_2 = Del(p_2)$; le même raisonnement que le cas (2).

■

Par contre, le théorème 2.5 énonce que transformer une opération d'insertion par rapport à deux séquences équivalentes peut résulter en deux opérations équivalentes selon la définition 2.5.

Théorème 2.5 *Etant donnée une opération d'insertion op. Si IT satisfait TP*1 *alors pour toutes opérations* op_1 *et* op_2 *nous avons :*

$$IT^*(op, [op_1; op'_2]) \equiv_{\mathcal{P}} IT^*(op, [op_2; op'_1])$$

avec $op'_1 = IT(op_1, op_2)$ *et* $op'_2 = IT(op_2, op_1)$. ■

Preuve. Soient $op = Ins(p, c, \omega, i)$, $op' = IT^*(op, [op_1; op'_2])$ et $op" = IT^*(op, [op_2; op'_1])$. Sans perdre de généralité, nous allons considérons les opérations $op_1 = Ins(p_1, c_1, \omega_1, i_1)$ et $op_2 = Del(p_2)$, tel que $op_1 \sqsubset op$. Nous avons les cas suivants en tenant compte des paramètres de op_1, op_2 et op

1. $p_1 < p_2 : op'_1 = Ins(p_1, c_1, \omega_1, i_1)$ et $op'_2 = Del(p_2 + 1)$. Nous considérons les cas suivants en fonction de op :
   (a) $p = p_1 : op' = Ins(p+1, c, p\omega, i)$ et $op" = Ins(p+1, c, p\omega, i)$.

(b) $p = p_2 : op' = Ins(p+1, c, (p+1)p\omega, i)$ et $op" = Ins(p+1, c_1, pp\omega, i)$. Nous remarquons que $(p+1)(p+1)p \equiv_{\mathcal{P}} (p+1)pp$. En utilisant le fait que $\equiv_{\mathcal{P}}$ est une congruence à droite (voir la proposition 2.2) nous pouvons affirmer que
$(p+1)(p+1)p\omega \equiv_{\mathcal{P}} (p+1)pp\omega$. Ainsi, $op' \equiv_{\mathcal{P}} op"$.

(c) $p > p_2 : op' = Ins(p, c, (p+1)p\omega, i)$ et $op" = Ins(p, c, (p-1)p\omega, i)$. Nous avons $p(p+1)p\omega \equiv_{\mathcal{P}} p(p-1)p\omega$ en utilisant la proposition 2.2 (page 127). Ainsi, $op' \equiv_{\mathcal{P}} op"$.

2. $p_1 = p_2 : op'_1 = Ins(p_1, c, p_1\omega_1, i_1)$ et $op'_2 = Del(p_2 + 1)$. En fonction de op, nous avons les cas suivants :

(a) $p = p_1 : op' = Ins(p+1, c, (p+1)p\omega, i)$ et $op" = Ins(p+1, c, pp\omega, i)$. Nous avons donc $op' \equiv_{\mathcal{P}} op"$ (voir le cas $(1)(b)$).

(b) $p > p_1 : op' = Ins(p, c, (p+1)p\omega, i)$ et $op" = Ins(p, c, (p-1)p\omega, i)$. Nous avons donc $op' \equiv_{\mathcal{P}} op"$ (voir le cas $(1)(c)$).

3. $p_1 > p_2 : op'_1 = Ins(p_1 - 1, c_1, p_1\omega_1, i_1)$ et $op'_2 = Del(p_2)$. Si $p \geq p_1$, alors $op' = Ins(p, c, (p+1)p\omega, i)$ et $op" = Ins(p, c, (p-1)p\omega, i)$. Nous avons donc $op' \equiv_{\mathcal{P}} op"$ (voir le cas $(1)(c)$).

■

$TP2'$ n'est pas suffisante

Si nous considérons seulement des opérations concurrentes, nous avons montré dans la section précédente que $TP2'$ (en plus de $TP1$) est nécessaire pour assurer la convergence. Dans ce qui suit, nous allons montrer que cette condition s'avère insuffisante dans certaines situations où nous prenons en compte, en plus de la concurrence, des relations de dépendance causale entre les opérations.

Pour mettre en évidence les situations où $TP2'$ est insuffisante, la conjecture suivante énonce la préservation par transformation d'une relation d'effet entre deux opérations d'insertion qui sont partiellement concurrentes [10].

Conjecture 1 (Relation d'effet avec concurrence partielle) *Soient deux opérations concurrentes d'insertion op_1 et op_2 ($op_1 \parallel op_3$). Pour toutes les opérations op_3 et op_4, tels que $op_1 \parallel op_3$, $op_1 \parallel op_4$, $op_2 \parallel op_4$ et $op_3 \rightarrow op_2$, nous avons :*

$$op_2 \sharp IT(op_1, op_3) \text{ implique } IT(op_2, IT(op_4, op_3)) \sharp IT^*(op_1, s)$$

avec $\sharp$ est un symbole $\{\sqsubset, \sqsupset\}$ et $s \in SEQ(\{op_3, op_4\})$[11]. ■

D'après la conjecture ci-dessus il est clair que op_1 et op_2 sont partiellement concurrentes puisque $op_3 \rightarrow op_2$ et $op_1 \parallel op_3$. Pour sa vérification, nous devons considérons deux cas :

[10] Deux opérations sont partiellement concurrentes si elles sont générées sur deux états différents (cette notion est définie plus formellement à la page 29).

[11] $SEQ(O)$ est l'ensemble des séquences d'opérations construites par transformation à partir de l'ensemble des opérations O. A titre d'exemple, si $O_1 = \{op_1, op_2\}$ avec $op_1 \parallel op_2$ et $O_2 = \{op_3, op_4\}$ avec $op_4 \rightarrow op_3$ alors $SEQ(O_1) = \{[op_1; IT(op_2, op_1)], [op_2; IT(op_1, op_2)]\}$ et $SEQ(O_2) = \{[op_4; op_3]\}$.

1. $s = [op_3; IT(op_4, op_3)]$: dans ce cas la conjecture est réécrite comme suit :

   $op_2 \sharp IT(op_1, op_3)$ implique $IT(op_2, IT(op_4, op_3)) \sharp IT(IT(op_1, op_3), IT(op_4, op_3))$

   En remplaçant $IT(op_1, op_3)$ et $IT(op_4, op_3)$ respectivement par op'_1 et op'_4 nous obtenons :

   $$op_2 \sharp op'_1 \text{ implique } IT(op_2, op'_4) \sharp IT(op'_1, op'_4)$$

   qui est toujours vraie selon le lemme 2.2.
2. $s = [op_4; IT(op_3, op_4)]$: dans ce cas il y a plusieurs contre-exemples qui montrent que la conjecture est fausse. A titre d'exemple, considérons le contre-exemple suivant : $op_1 = Ins(p, c_1, \omega, i_1)$, $op_2 = Ins(p, c_2, \omega, i_2)$, $op_3 = Del(p)$, $op_4 = Del(p+1)$, $i_1 < i_2$ et $\sharp = \sqsubset$. Nous obtenons donc les transformations suivantes :
   - $op'_3 = IT(op_3, op_4) = Del(p)$;
   - $op'_4 = IT(op_4, op_3) = Del(p)$;
   - $op'_1 = IT(op_1, op_3) = Ins(p, c_1, p\omega, i_1)$;
   - $op''_1 = IT^*(op_1, [op_4; op'_3]) = Ins(p, c_1, p\omega, i_1)$;
   - $op'_2 = IT(op_2, op'_4) = Ins(p, c_2, p\omega, i_2)$

   A partir de ces transformations, nous déduisons que $op_2 \sqsubset op'_1$; or cette relation n'est pas préservée par transformation puisque $op'_2 \sqsupset op''_1$.

Il devrait être noté que cette perte de relation entre deux opérations concurrentes d'insertion en présence d'une dépendance causale, peut conduire à une situation de divergence. A ce titre, la figure 2.3 un scénario réel illustrant une telle situation. Les opérations $op_1 = Ins(3, r, \epsilon, 1)$ et $op_2 = Ins(3, s, \epsilon, 2)$ sont générées respectivement sur les sites 1 et 2. L'opération op_2 dépends causalement de $op_3 = Del(3)$. Nous pouvons donc dire que op_1 et op_2 sont partiellement concurrentes. Le déroulement du scénario procède par les étapes suivantes :

1. L'intégration de op_1 sur le site 2 nécessite qu'elle soit transformée par rapport à la séquence $[op_3; op_2; op'_4]$. Comme $IT(op_1, op_3) = Ins(3, r, [3], 1)$, alors nous déduisons que $op_2 \sqsubset IT(op_1, op_3)$. Ceci signifie que sur le site 2 le caractère r sera insérée à droite du caractère s de op_2.
2. L'opération op_2 est partiellement concurrente à op_4. Aussi son intégration sur le site 3 requiert une correcte transformation, à savoir $op'_2 = IT(op_2, IT(op_4, op_3)) = Ins(3, s, [3], 2)$.
3. Lorsque op_1 arrive sur le site 3, elle doit être transformée par rapport à la séquence $[op_4; op'_3; op'_2]$. La transformation de op_1 par rapport à $[op_4; op'_3]$ donne l'opération $Ins(3, r, [3], 1)$. Dans ce cas, il est clair que $op'_2 \sqsupset Ins(3, r, [3], 1)$ ($i_1 < i_2$ avec $i_1 = 1$ et $i_2 = 2$). Par conséquent, le caractère r sera inséré à gauche du caractère s ; ce qui est contraire à la relation des deux caractères sur le site 2.

Cette perte de relation entre deux opération d'insertion, partiellement concurrentes, conduit inévitablement à une situation comme le montre la figure 2.3 (les états finaux des sites 2 et 3 sont différents).

Par ailleurs, il faut souligner que la vérification de la conjecture 1 a été confiée au prouveur `SPIKE`. A ce titre, nous énumérons à la table 2.1 tous les contre-exemples donnés par `SPIKE`.

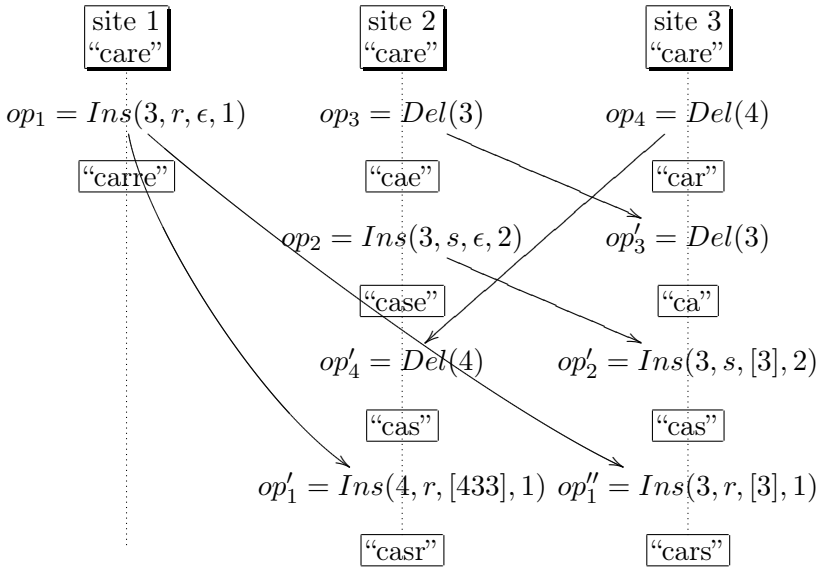


FIG. 2.3 – Divergence en présence de la concurrence partielle.

		op_3			
		$Del(p_3)$	$Del(p_3)$	$Del(p_3)$	$Ins(p_3, e_3, w_3, i_3)$
op_4	$Del(p_4)$	$p_1 = p_2$, $w_1 = w_2$, $p_2 = p_3$ $p_4 = p_3 + 1$, $\sharp = \sqsubset$			
	$Del(p_4)$		$p_1 = p_2$, $w_1 = w_2$, $i_1 < i_2$, $p_4 = p_2$, $p_3 = p_4 + 1$, $\sharp = \sqsubset$		
	$Del(p_4)$			$p_1 = p_2$, $w_1 = w_2$, $p_1 = p_4$, $p_4 = p_3 + 1$, $\sharp = \sqsupset$	
	$Del(p_4)$				$p_1 = p_2$, $p_2 = p_3$, $p_3 = p_4$, $w_1 = w_2$, $w_2 = w_3$, $i_1 < i_3$, $i_1 < i_2$, $\sharp = \sqsupset$

TAB. 2.1 – Contre-exemples pour la conjecture 1.

2.5 Synthèse

2.5.1 Etat de l'art

Un algorithme d'intégration doit être capable de calculer pour une opération op générée à l'état s, sa forme transformée op' à exécuter à l'état s' (dans la plupart des cas s et s' sont différents). Pour ce faire, les algorithmes d'intégration, comme adOPTed [73], SOCT2 [81], GOTO [84], procèdent en deux étapes :

(i) choisir ou construire un chemin de transformation C permettant de passer de s à s' ;

(ii) transformer op par rapport à C.

Un chemin de transformation est tout simplement une séquence d'opérations concurrentes à op et dont l'exécution à s produit s'. Il peut exister plusieurs

chemins de transformation de s à s'. Pour construire de tels chemin, l'histoire locale H du site receveur (celui qui va intégrer l'opération) est réorganisée en deux séquences H_h et H_c telle que $H = [H_h; H_c]$. La séquence H_h contient les opérations exécutées avant la génération de *op*. En d'autres termes, *op* dépend causalement des opérations de H_h. Quant à H_c, elle contient des opérations concurrentes à *op*. Donc H_c est le chemin de transformation sur lequel *op* sera transformée et dont le résultat pourra être correctement exécuté sur s'.

Cependant, les opérations de H_h et H_c peuvent être arbitrairement ordonnées d'un site à un autre. Ce qui rend la conception d'un algorithme de transformation pour un objet collaboratif linéaire (avec comme opérations Ins et Del) très difficile – voire même impossible – puisque cet algorithme doit satisfaire la condition $TP2$ [68; 73]. Théoriquement, cette condition assure que la transformation de toute opération par rapport à un chemin arbitraire produit le même résultat. Or, à notre connaissance, il n'existe aucun algorithme de transformation qui satisfait $TP2$ [39].

Devant cette difficulté à satisfaire $TP2$, d'autres approches [85; 89; 79] proposent des algorithmes de transformation qui satisfont seulement $TP1$. La convergence est assurée en construisant le même chemin de transformation sur tous les sites. Ceci nécessite un ordre total sur les opérations. Cette solution réduit la concurrence réelle des opérations puisqu'elle requiert un site central pour gérer l'ordre total.

2.5.2 Application de la transformation utilisant les p-mots

Notre algorithme de transformation basé sur les mots de positions satisfait seulement une forme relaxée de $TP2$, à savoir $TP2'$. L'utilisation du prouveur `SPIKE` nous a permis de couvrir un nombre important de situations pour analyser cet algorithme par rapport aux conditions de convergence $TP1$ et $TP2'$. Comme résultat de cette analyse, nous aboutissons aux conclusions suivantes :

- La condition $TP2'$ est suffisante pour assurer la convergence dans le cas où les chemins de transformation sont identiques sur tous les sites.
- En présence de certaines situation de concurrence partielle, $TP2'$ s'avère insuffisante pour atteindre la convergence (voir la table 2.1).

En conséquence, nous ne pouvons pas embarquer notre algorithme de transformation sur les environnements d'intégration basés sur la "forte" condition $TP2$, comme adOPTed [73]. Par contre, notre algorithme se prête bien pour être utilisé sur les autres environnements d'intégration basé seulement sur $TP1$. En effet, la condition $TP2'$ apporte à de tels environnements une nette amélioration quant à l'état de convergence.

A titre d'exemple, considérons le système SOCT4 [89] qui utilise un estampilleur (hébergé sur un site central) pour ordonner totalement les opérations. Comme algorithme de transformation des opérations d'édition Ins et Del, nous utilisons celui de Ressel (voir algorithme 4). Le scénario de la figure 2.4 montre une violation de relation d'effet dans SOCT4. Nous avons trois opérations concurrentes générées sur le même état "abc" : $op_1 = Ins(3, x, 1)$, $op_2 = Del(2)$ et $op_3 = Ins(2, y)$. Il est clair que sur l'état "abc" op_1 insère son caractère après celui de op_3. Pourtant, si nous supposons que l'estampilleur

ordonne les opérations dans l'ordre op_2, op_3 et op_1, alors l'intégration de ces opérations conduit à une violation de la relation d'effet entre op_1 et op_3. Intéressons-nous à l'intégration des opérations sur le site 2 :

1. lorsque op_3 arrive, elle est transformée par rapport à op_2, *i.e.* $op'_3 = IT(op_3, op_2) = Ins(2, y, 2)$;
2. l'opération op_1 doit ensuite être transformée par rapport à la séquence $[op_2; op_3]$ comme suit :

$$op'_1 = IT^*(op_1, [op_2; op_3]) = IT(IT(op_1, op_2), op'_3) = \\ IT(Ins(2, x, 1), Ins(2, y, 2)) = Ins(2, x, 1)$$

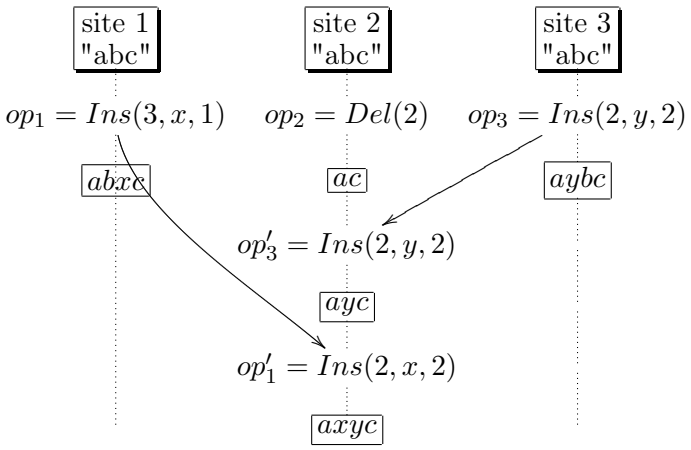


FIG. 2.4 – Violation de relation d'effet dans SOCT4.

Au final, l'état de convergence obtenu ne respecte pas la relation d'effet initiale entre op_1 et op_3. En effet, nous aurions dû avoir l'état "ayxc" au lieu de "axyc". Ceci montre que $TP1$ est insuffisante pour préserver les relations d'effet entre les opérations concurrentes d'insertion. Cependant, si nous utilisons l'algorithme de transformation basé sur les p-mots, la relation d'effet entre op_1 et op_3 sera forcément respectée sur tous les sites comme le montre la figure 2.5.

La plupart des éditeurs collaboratifs basés sur l'approche transformationnelle utilisent la technique des vecteurs d'état pour détecter la concurrence entre les opérations. Il est bien connu dans la littérature que les vecteurs d'état imposent des dépendances causales particulières qui ne sont même pas requises par la sémantique de l'objet collaboratif [13]. Notons que la majorité des contre-exemples donnés dans la table 2.1 ont la dépendance causale $op_3 \rightarrow op_2$ entre une opération d'insertion op_2 et une opération d'effacement op_3. Or cette dépendance s'avère fausse lorsque l'on étudie la sémantique des opérations d'insertion et d'effacement. En effet, toutes les opérations Ins et Del dépendent sémantiquement de Ins.

Pour rendre $TP2'$ suffisante pour atteindre la convergence, il est indispensable d'éviter les situations illustrées dans la table 2.1. Aussi, cette constatation nous a incité à réfléchir à la conception d'un nouvel algorithme d'intégration qui sera décrit en détail dans le chapitre suivant. Parmi les nouvelles caractéristiques de cet algorithme :

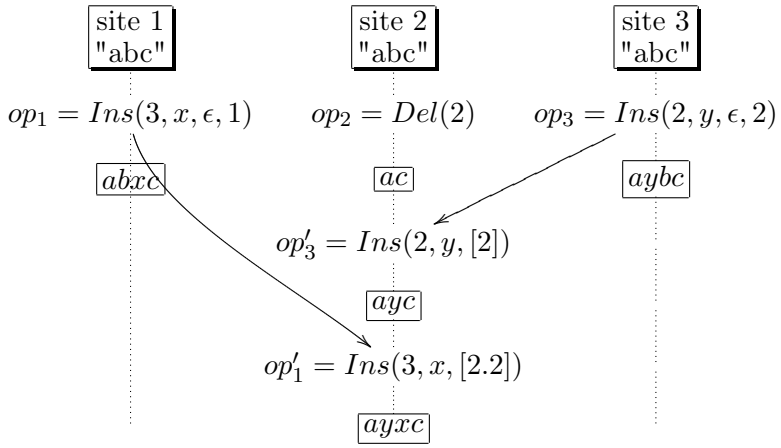


FIG. 2.5 – Respect de relation d'effet dans SOCT4 en utilisant les p-mots.

- La détection de la concurrence est basée sur une relation de dépendance sémantique entre Ins et Del. Ce qui procure plus de concurrence dans la collaboration.
- Il n'est pas tributaire d'un site central.
- Le nombre de sites peut être arbitraire.
- Il nécessite la satisfaction des conditions $TP1$ et $TP2'$ tout en permettant des chemins de transformation canoniques qui contiennent toujours les insertions avant les effacements.
- Il est adapté au réseau pair-à-pair.

2.6 Conclusion

Nous avons proposé une forme relaxée de $TP2$, à savoir $TP2'$, pour garantir la convergence pour des objets linéaires en utilisant un nouvel algorithme de transformation. Cet algorithme est basé sur la trace des positions stockées durant le processus de transformation. L'utilisation de `SPIKE` nous a permis de couvrir un nombre important de situations pour vérifier la convergence pour notre algorithme. Nous sommes arrivés aux conclusions suivantes :

- La condition $TP2'$ est nécessaire lorsque nous sérialisons des opérations concurrentes définies sur le même état.
- Dans certaines situations de concurrence partielle entre des opérations d'insertion, $TP2'$ s'avère insuffisante pour garantir la convergence.

Par conséquent, nous ne pouvons pas embarquer notre solution de transformation dans des algorithmes d'intégration tels que adOPTed [73] et SOCT2 [81] puisque ces derniers nécessitent la satisfaction de $TP2$ qui est "plus forte" que $TP2'$.

Par ailleurs, nous avons remarqué que $TP2'$ peut être suffisante si les transformations se font sur deux catégories d'histoires : (i) des histoires identiques ; et, (ii) des histoires équivalentes ayant les insertions avant les suppressions. La première catégorie concerne les histoires construites à partir d'un ordre global sur les opérations concurrentes. Ainsi, nous pouvons exploiter notre solution sur

un environnement comme SOCT4 [89]. La deuxième catégorie est plus intéressante car les histoires ne sont pas forcément identiques ; il y a juste un ordre entre l'insertion et l'effacement. Aussi, pour construire de telles histoires, il faut concevoir un nouvel algorithme d'intégration ; c'est l'objet du chapitre suivant.

Chapitre 3

Nouvel Environnement pour l'Edition Collaborative sur un Réseau Pair-à-Pair

Sommaire

3.1 Introduction

Pour garantir la convergence des données, les éditeurs collaboratifs basés sur la transformation des opérations, nécessitent la satisfaction des conditions $TP1$ et $TP2$ [73; 81; 84]. Cependant, la conception d'un algorithme de transformation pour des objets linéaires reste toujours un problème ouvert quant à la

satisfaction de $TP2$. Théoriquement, l'acquisition d'une telle condition va assurer la convergence dans un contexte où les collaborateurs peuvent échanger et intégrer les opérations sans la moindre contrainte. Or, à notre connaissance, il n'existe aucun algorithme de transformation qui vérifie $TP2$. Ce qui rend pour le moment impraticable des algorithmes d'intégration comme adOPTed [73] et SOCT2 [81], puisque avec ces algorithmes on ne peut pas assurer une collaboration sans divergence même pour un objet comme les chaînes de caractères.

Devant cette difficulté à satisfaire $TP2$, nous avons proposé dans le chapitre précédent un algorithme de transformation basé sur une condition plus faible $TP2'$ qui ne peut pas assurer la convergence dans tous les cas de figure. En effet, $TP'2$ est suffisante si les transformations se font sur des histoires identiques ou des histoires équivalentes ayant les insertions avant les effacements. La deuxième catégorie d'histoires revêt plus d'intérêt à nos yeux puisqu'elle ne nécessite pas un ordre total sur les opérations concurrentes ; elle requiert juste un ordre entre les insertions et les effacements. Néanmoins, il faut une procédure pour construire de telles histoires sur tous les sites.

A ce titre, nous présentons dans ce chapitre un nouvel environnement pour l'édition collaboratif basé sur la réplication et la transformation des opérations. Nous considérons un groupe de N sites (où N peut être arbitraire) qui démarrent une session de collaboration à partir du même état. Chaque site modifie localement sa copie et envoie cette modification aux autres sites. Nous verrons plus loin comment les histoires des opérations sur les sites sont toujours organisées de telle sorte que les insertions soient avant les effacements.

Notre environnement possède un certain nombre d'avantages. Il permet l'accès simultané aux données avec une restauration automatique (en utilisant la transformation) de la cohérence des données. Cette dernière est garantie grâce aux conditions $TP1$ et $TP2'$. Contrairement à l'utilisation des vecteurs d'état, notre environnement met en oeuvre une causalité minimale entre les opérations grâce à un mécanisme de détection des dépendances qui se base essentiellement sur la sémantique de l'objet collaboratif.

A notre connaissance, il n'y a aucun éditeur collaboratif basé sur la transformation qui peut passer à l'échelle. Tous les systèmes existants supposent un nombre fixe de sites [73; 81; 84; 89; 51; 53]. Comme notre environnement n'utilise pas les vecteurs d'état (dont la taille est fixe et reflète le nombre de sites), il n'est pas tributaire du nombre de collaborateurs. Ce nombre peut être variable pendant les sessions de collaboration. Par conséquent, notre environnement est approprié pour être déployé sur un réseau pair-à-pair.

Ce chapitre est composé comme suit. Dans la section 3.2, nous allons définir formellement un modèle de cohérence de données. La section 3.3 présentera les différents algorithmes de transformation utilisés pour construire les histoires. Dans la section 3.4, nous allons donner l'algorithme d'intégration ainsi que la preuve de sa correction. La section 3.5 sera consacrée à une étude comparative avec d'autres travaux. Enfin, nous terminons le chapitre avec une conclusion.

3.2 Modèle de Cohérence

Dans cette section, nous définissons formellement les ingrédients de notre approche :

3.2.1 Opérations

Il faut rappeler que nous traitons d'un objet collaboratif qui admet une structure linéaire. Cet objet peut être assimilé à une *liste* finie d'éléments d'un type de données $\mathcal{E}$. Ce type est juste un paramètre qui peut être remplacé par d'autres types de données. Par exemple, un élément de la liste peut être un caractère, un bloc de caractères, une page, etc. Nous notons l'ensemble des états de l'objet $\mathcal{L}$.

Nous supposons que l'état de l'objet est modifié seulement par les opérations primitives suivantes :

(i) $Ins(p, e, \omega)$ qui ajoute un élément e à la position p (ω est un p-mot) ;

(ii) $Del(p)$ détruit l'élément qui se situe à la position p.

Aussi, l'ensemble des opérations est donné comme suit :

$$\mathcal{O} = \{Ins(e, p, \omega) | e \in \mathcal{E} \text{ et } p \in \mathbb{N}\} \cup \{Del(p) | p \in \mathbb{N}\} \cup \{Nop()\}$$

où $Nop()$ est l'opération qui laisse l'état de l'objet inchangé. Nous utilisons la fonction $Do : \mathcal{O} \times \mathcal{L} \rightarrow \mathcal{L}$ pour calculer l'état l' obtenu après l'exécution d'une opération op sur un état l, *i.e.* $Do(op, l) = l'$.

Définition 3.1 (Opérations similaires) *Deux opérations op_1 et op_2 sont dites* similaires *(noté $op_1 \approx op_2$) ssi l'une des conditions est satisfaite :*

1. $op_1 = Ins(p_1, e_1, \omega_1)$, $op_2 = Ins(p_2, e_2, \omega_1)$, $p_1 = p_2$ *et* $e_1 = e_2$;
2. $op_1 = Del(p_1)$, $op_2 = Del(p_2)$ *et* $p_1 = p_2$;
3. $op_1 = Nop()$ *et* $op_2 = Nop()$.

■

D'après la définition 3.1 deux opérations similaires d'insertion n'ont pas forcément les mêmes p-mots. Il est clair que si $op_1 \equiv_{\mathcal{P}} op_2$ (voir la définition 2.5) alors $op_1 \approx op_2$. Le contraire n'est pas tout le temps vrai.

3.2.2 Requêtes

Nous définissons une *requête* comme un quadruplet $r = (u, k, a, o)$ où : (i) u est l'identifiant du site (ou l'utilisateur) qui a généré la requête [12] ; (ii) k est le compteur du nombre de requêtes générées par le site u ; (iii) a est l'identifiant de la requête qui précède r [13] (voir la sous-section 3.2.4) ; (iv) o est l'opération à exécuter sur l'état de l'objet. Nous utilisons r, r', r_1, r_2, ..., pour désigner les requêtes. Soit $\mathcal{R}$ l'ensemble de toutes les requêtes.

[12] Nous supposons que chaque site possède un identifiant unique.

[13] L'*identifiant* de r est le mot formé par $r.u$ et $r.k$. Cet identifiant est noté $Id(r)$.

Les notations $r.u$, $r.k$, $r.a$ et $r.o$ seront utilisées pour mentionner les composants d'une requête r. Si $r.o$ est une opération d'insertion (resp. d'effacement) alors nous remplaçons r par i (resp. d).

Durant une session de collaboration, les composants $r.u$, $r.k$ et $r.a$ restent inchangés. Seul le composant $r.o$ sera potentiellement modifié lors d'un processus de transformation.

Par abus de notation, nous utilisons les concepts suivants :

(i) $r \equiv_{\mathcal{P}} r'$ ssi $r.o \equiv_{\mathcal{P}} r'.o$ (r et r' sont dites *équivalents*) ;

(ii) $r \approx r'$ ssi $r.o \approx r'.o$ (r et r' sont dites *similaires*).

Nous définissons la partie stricte d'une relation d'ordre sur les requêtes d'insertion :

Définition 3.2 (Relation d'effet sur les requêtes d'insertion) *Soient deux requêtes d'insertion i_1 et i_2) : $i_1 \sqsubset i_2$ ssi l'une des conditions suivantes est satisfaite : (i) $PW(i_1.o) \prec PW(i_2.o)$; (ii) $PW(i_1.o) = PW(i_2.o)$ et $i_1.u < i_2.u$.* ■

La relation d'effet indique tout simplement comment les éléments insérés sont reliés entre eux. Ainsi, $i_1 \sqsubset i_2$ signifie que l'élément ajouté par i_1 se trouve à gauche (ou avant) de celui ajouté par i_2.

3.2.3 Liste des requêtes

Toutes les requêtes exécutées sur un site donné sont mémorisées dans une liste que nous définissons comme suit :

Définition 3.3 (Histoire) *Une* histoire *est un tuple $(H, \ll_H)$ où H est un ensemble fini de requêtes muni d'une relation d'ordre total $\ll_H$ reflétant l'ordre d'exécution. Nous désignons une histoire $(H, \ll_H)$ par la séquence $H = [u_1; u_2; \ldots; u_n]$ ssi $H = \{u_1, u_2, \ldots, u_n\}$ et $u_i \ll_H u_j$ pour $i < j \in \{1, \ldots, n\}$. Nous désignons l'ensemble des histoires par $\mathcal{H}$.* ■

Ainsi, chaque site dispose de sa propre liste de requêtes. Une histoire de requêtes peut être scindée en plusieurs sous-histoires.

Définition 3.4 *Une histoire $(H', \ll_{H'})$ est dite* sous-histoire *de $(H, \ll_H)$ ssi $H' \subseteq H$ et pour toutes requêtes r, $r' \in H'$ $(r, r') \in \ll_H$ ssi $(r, r') \in \ll_{H'}$.* ■

Etant donnée une histoire H, $H[i]$ représente la i-ème requête dans H et $H[i, j]$ dénote la sous-histoire de H allant de la i-ème jusqu'à la j-ème requête avec $0 \leq i \leq j \leq n-1$ où n est le nombre de requêtes dans H (ou $n = |H|$). Par abus de notation, nous utilisons $Do(H, l)$ pour dire que l'histoire H est exécutée sur l'état l.

3.2.4 Dépendance Sémantique

Dans une histoire, une requête peut dépendre des requêtes précédemment exécutées. En d'autres termes, l'effet de cette requête peut être influencé par les précédentes requêtes. En identifiant touts les prédécesseurs "appropriés" d'une requête donnée, nous sommes en mesure d'identifier les requêtes qui doivent s'exécuter sur tous les sites selon le même ordre.

L'étude sémantique d'un objet linéaire dont l'état est altéré par des opérations d'insertion et d'effacement, nous a permis de proposer la relation suivante :

Définition 3.5 (Dépendance sémantique) *Nous définissons la relation* $\xrightarrow{s}$ *sur l'ensemble* $\mathcal{R}$ *comme suit. Nous disons que* $r_1 \xrightarrow{s} r_2$ *ssi une des conditions suivantes est satisfaite :*

1. $o.r_1 = Ins(p, e)$, $o.r_2 = Ins(p, e')$ *et* $u.r_1 \leq u.r_2$;
2. $o.r_1 = Ins(p, e)$, $o.r_2 = Ins(p + 1, e')$ *et* $u.r_2 \leq u.r_1$;
3. $o.r_1 = Ins(p, e)$ *et* $o.r_2 = Del(p)$;
4. *il existe* r_3 *tel que* $r_1 \xrightarrow{s} r_3$ *et* $r_3 \xrightarrow{s} r_2$. ■

Les trois premières conditions constituent trois formes de dépendances que nous notons respectivement fd_1, fd_2 *et* fd_3.

La définition suivante donne comment détecter que deux requêtes sont indépendantes :

Définition 3.6 (Concurrence) *Deux requêtes* r_1 *et* r_2 *sont dites* concurrentes *(noté* $r_1 \parallel r_2$*) ssi nous n'avons ni* $r_1 \xrightarrow{s} r_2$ *ni* $r_2 \xrightarrow{s} r_1$. ■

Maintenant nous généralisons la relation de dépendance sémantique aux histoires.

Définition 3.7 (Sous-histoire fermée) *Une sous-histoire* $(H', \ll_{H'})$ *de* $(H, \ll_H)$ *est dite* fermée *ssi pour toutes requêtes* $r \in H$ *et* $r' \in H'$ *où* $(r, r') \in \ll_H$ *si* $r \xrightarrow{s} r'$ *alors* $r \in H'$. ■

Une sous-histoire fermée contient pour chaque requête sa précédente requête dont elle dépend. L'échange des requêtes entre les sites doit donc impérativement préserver les sous-histoires fermées.

Définition 3.8 (Sous-histoires concurrentes) *Deux sous-histoires* $(H', \ll_{H'})$, $(H'', \ll_{H''})$ *de* $(H, \ll_H)$ *sont dites* concurrentes *ssi* $H' \cap H'' = \emptyset$ *et pour toutes requêtes* $r' \in H'$ *et* $r'' \in H''$ *nous avons* $r' \parallel r''$. ■

Deux sous-histoires sont concurrentes si leurs ensembles de requêtes sont disjoints et il n'y a aucune dépendance entre les requêtes de ces deux ensembles.

3.2.5 Critères de Cohérence

Notre mécanisme de réplication doit garantir le modèle de cohérence que nous définissons par les critères suivants :

Définition 3.9 (Modèle de cohérence) *Un éditeur collaboratif est* cohérent *s'il est en mesure de satisfaire les propriétés suivantes :*

(1) Préservation de la dépendance : pour tout couple de requêtes r_1 et r_2, si $r_1 \xrightarrow{s} r_2'$ alors r_1 doit s'exécuter avant r_2 sur tous les sites.

(2) Convergence : lorsque tous les sites ont exécuté le même ensemble de requêtes, toutes les copies de l'objet collaboratif sont identiques. ■

Pour mettre en œuvre une précédance causale entre les requêtes, nous allons utiliser la relation de dépendance que nous avons donnée à la définition 3.5. Cette relation est minimale en ce sens où chaque requête ne connaît que l'identifiant de la requête dont elle dépend directement.

Pour atteindre la convergence des données, nous allons recourir à la transformation des requêtes. A ce titre, nous allons utiliser l'algorithme de transformation basé sur les p-mots que nous avons décrit au chapitre 2. En construisant des chemins de transformation particuliers (voir la sous-section 3.3.4), les conditions $TP1$ et $TP2'$ s'avèrent suffisantes pour assurer la convergence.

3.3 Transformation bidirectionnelle

Dans cette section, nous définissons trois formes de transformation qui sont basées sur les p-mots. De manière informelle, la transformation inclusive (IT) permet de sérialiser deux requêtes concurrentes. Si l'on considéré une séquence de requêtes $[r_1; r_2]$, alors la transformation exclusive (ET) tente de "casser" l'ordre entre r_1 et r_2 pour les rendre concurrentes. Quant à la permutation (PERM), elle permet tout simplement de réordonner la séquence sous la forme $[r_2'; r_1']$ où r_1' et r_2' sont respectivement les formes transformées de r_1 et r_2.

3.3.1 Transformation Inclusive

Nous réappliquons l'algorithme de transformation utilisant les p-mots (voir algorithme 8) aux requêtes. Etant données deux requêtes concurrentes et définies sur le même état r_1 et r_2. La transformation (inclusive) de r_1 par rapport à r_2, $IT(r_1, r_2) = r_1'$, consiste à inclure l'effet de r_2 dans r_1, de telle façon que r_1' peut être exécutée après r_2.

Soient les requêtes i_j et d_j tels que $i_j.o = Ins(p_j, e_j, \omega_j)$ et $d_j.o = Del(p_j)$ pour $j \in \{1, 2\}$. Les différents cas de transformation pour les requêtes sont donnés dans l'algorithme 9.

Comme vu dans le chapitre précédent, notre algorithme de transformation satisfait la condition $TP1$ mais il ne vérifie pas la condition $TP2$. En contrepartie, nous avons proposé une forme relaxée, $TP2'$, qui est satisfaite par notre algorithme. Néanmoins, moyennant $TP2'$, la convergence est assurée que dans certaines situations.

1: $IT(r_1, r_2) = r'_1$
2: $r'_1 \leftarrow r_1$
3: **Choix** de r_1 et r_2
4: **Cas :** $r_1 = i_1$ et $r_2 = i_2$
5: **si** $(PW(r_2.o) \prec PW(r_1.o)$ ou $(PW(r_1.o) = PW(r_2.o)$ et $r_2.u < r_1.u))$ **alors** $r'_1.o \leftarrow Ins(p_1 + 1, e_1, p_1\omega_1)$
6: **fin si**
7: **Cas :** $r_1 = i_1$ et $r_2 = d_2$
8: **si** $(p_2 < p_1)$ **alors** $r'_1.o \leftarrow Ins(p_1 - 1, e_1, p_1\omega_1)$
9: **sinon si** $(p_2 = p_1)$ **alors** $r'_1.o \leftarrow Ins(p_1, e_1, p_1\omega_1)$
10: **fin si**
11: **Cas :** $r_1 = d_1$ et $r_2 = i_2$
12: **si** $(p_2 \leq p_1)$ **alors** $r'_1.o \leftarrow Del(p_1 + 1)$
13: **fin si**
14: **Cas :** $r_1 = d_1$ et $r_2 = d_2$
15: **si** $(p_2 < p_1)$ **alors** $r'_1.o \leftarrow Del(p_1 - 1)$
16: **sinon si** $(p_2 = p_1)$ **alors** $r'_1.o \leftarrow Nop()$
17: **fin si**
18: **fin choix**
19: **retourner** r'_1

Algorithme 9: Transformation inclusive utilisant des p-mots.

Comme propriétés sous-jacentes à l'algorithme 9, nous énonçons les lemmes suivants :

La relation de similarité entre les requêtes est préservée par transformation par rapport à des requêtes d'effacement.

Lemme 3.1 *Etant données deux requêtes r_1 et r_2. Pour toute requête d'effacement d, si $r_1 \approx r_2$ alors $IT(r_1, d) \approx IT(r_2, d)$.* ■

Preuve. Soient $d.o = Del(p_1)$, $r'_1 = IT(r_1, d)$ et $r'_2 = IT(r_2, d)$. Deux cas sont à considérer :

1. $r_1.o = Ins(p, e, \omega_1)$ et $r_2.o = Ins(p, e, \omega_2)$: Il faut comparer p par rapport à p_1.
   (a) $p_1 < p$: $r'_1.o = Ins(p-1, e, p\omega_1)$, $r'_2.o = Ins(p-1, e, p\omega_2)$ et $r_1 \approx r'_2$.
   (b) $p_1 = p$: $r'_1.o = Ins(p, e, p\omega_1)$, $r'_2.o = Ins(p, e, p\omega_2)$ et $r_1 \approx r'_2$.
   (c) $p < p_1$: $r'_1 = r_1$ et $r'_2 = r_2$.
2. $r_1.o = Del(p)$ et $r_2.o = Del(p)$ (ou $r_1.o = Nop()$ et $r_2.o = Nop()$) : r'_1 et r'_2 sont toujours similaires.

■

La relation de similarité est également préservée par transformation par rapport à des histoires contenant seulement des requêtes d'effacement.

Théorème 3.1 *Etant données une histoire contenant seulement des requêtes d'effacement et deux requêtes quelconques r_1 et r_2. Si $r_1 \approx r_2$ alors $IT^*(r_1, H) \approx IT^*(r_2, H)$* ■

Preuve. Par induction sur $|H|$ et l'utilisation du lemme 3.1. ■

La transformation d'une requête d'insertion par rapport à deux séquences équivalentes d'insertion donne le même résultat.

Lemme 3.2 *Soient i_1 et i_2 deux requêtes d'insertion. Si $i'_1 = IT(i_1, i_2)$ et $i'_2 = IT(i_2, i_1)$ alors $IT^*(i, [i_1; i'_2]) = IT^*(i, [i_2; i'_1])$ pour toute requête d'insertion i.* ■

Preuve. Soient $i.o = Ins(p, e, \omega)$, $i_1.o = Ins(p_1, e_1, \omega_1)$ et $i_2.o = Ins(p_2, e_2, \omega_2)$. Sans perdre de généralité, prenons le cas où $PW(i_1.o) \prec PW(i.o)$ et $PW(i.o) \prec PW(i_2.o)$. Dans ce cas, nous avons $i'_1.o = Ins(p_1, e_1, \omega_1)$ et $i'_2.o = Ins(p_2 + 1, e_2, \omega_2)$. Si nous considérons $i_{11} = IT(i, i_1)$ et $i_{21} = IT(i_{11}, i'_2)$ alors nous avons $i_{11}.o = Ins(p + 1, e, \omega)$ et $i_{21}.o = Ins(p + 1, e, \omega)$ (comme $PW(i_{11}.o) \prec PW(i'_2.o)$). De la même manière, nous considérons $i_{12} = IT(i, i_2)$ et $i_{22} = IT(i_{12}, i'_1)$. Nous avons donc $i_{12}.o = Ins(p, e, \omega)$ et $i_{22}.o = Ins(p + 1, e, \omega)$. En conséquence, $i_{21} = i_{22}$. ■

La transformation d'une requête d'insertion par rapport à deux séquences équivalentes d'effacements produit deux requêtes similaires.

Lemme 3.3 *Soient i une requête d'insertion. Pour toutes requêtes d'effacement d_1 et d_2, si $d'_1 = IT(d_1, d_2)$ et $d'_2 = IT(d_2, d_1)$ alors $IT^*(i, [d_1; d'_2]) \approx IT^*(i, [d_2; d'_1])$.* ■

Preuve. En utilisant le théorème 2.5, nous avons $IT^*(i, [d_1; d'_2]) \equiv_{\mathcal{P}} IT^*(i, [d_2; d'_1])$. Par conséquent, $IT^*(i, [d_1; d'_2]) \approx IT^*(i, [d_2; d'_1])$. ■

3.3.2 Transformation Exclusive

Etant donnée une séquence de requêtes $[r_1; r_2]$. La transformation exclusive de r_2 par rapport à r_1, $ET(r_2, r_1) = r'_2$, permet d'exclure l'effet de r_1 de r_2. De ce fait, r'_2 et r_1 sont considérées comme concurrentes puisqu'elles sont désormais définies sur le même état. On peut dire que la transformation exclusive procède à l'inverse de la transformation inclusive.

Soient les requêtes i_j et d_j tels que $i_j.o = Ins(p_j, e_j, \omega_j)$ et $d_j.o = Del(p_j)$ pour $j \in \{1, 2\}$. Les différents cas de a transformation exclusive pour les requêtes sont donnés dans l'algorithme 10.

Supposons que r_1 et r_2 sont des requêtes d'insertion. Dans le cas où leurs positions d'insertion sont identiques, cela signifie que l'élément ajouté par r_1 précède celui ajouté par r_2. Si leurs p-mots ou les identifiants de leurs sites générateurs reflètent cette précédance alors $ET(r_1, r_2)$ retourne r_1 puisque r_2 n'a aucun effet sur la position d'insertion de r_1. Lorsque l'élément inséré par r_2 est avant celui de r_1 et les p-mots (ou les identifiants des sites) de r_1 et r_2 respectent cette précédance, alors nous décrémentons par un la position de r_1 pour exclure l'effet de r_2 (lignes $8 - 15$).

1: $ET(r_1, r_2) = r'_1$
2: $r'_1 \leftarrow r_1$
3: **Choix** de r_1 et r_2
4: **Cas :** $r_1 = i_1$ et $r_2 = i_2$
5: $\alpha_1 \leftarrow PW(r_1.o)$
6: $\alpha_2 \leftarrow PW(r_2.o)$
7: **si** $(p_1 = p_2)$ et $((\alpha_1 \succ \alpha_2)$ ou $(\alpha_1 = \alpha_2$ et $r_1.u \geq r_2.u))$
alors retourner "Indéfinie"
8: **sinon si** $(p_1 = p_2 + 1)$
9: **alors si** $(\omega_1 \neq \epsilon)$
10: **alors si** $(\omega_1 \succ \alpha_1)$ ou $(\omega_1 = \alpha_1$ et $r_1.u > r_2.u)$
alors $r'_1.o \leftarrow Ins(p_1 - 1, e_1, Tail(\omega_1))$
11: **sinon retourner "Indéfinie"**
12: **sinon si** $(r_1.u > r_2.u)$ **alors** $r'_1.o \leftarrow Ins(p_1 - 1, e_1, \omega_2)$
13: **sinon retourner "Indéfinie"**
14: **sinon si** $(p_1 > p_2 + 1)$ **alors** $r'_1.o \leftarrow Ins(p_1 - 1, e_1, Tail(\omega_1))$
15: **fin si**
16: **Cas :** $r_1 = i_1$ et $r_2 = d_2$
17: **si** $(\omega_1 \neq \epsilon)$
18: **alors si** $(p_1 = p_2$ et $p_1 = Top(\omega_1))$ **alors** $r'_1.o \leftarrow Ins(p_1, e_1, Tail(\omega_1))$
19: **sinon si** $(p_1 > p_2)$ **alors** $r'_1.o \leftarrow Ins(p_1 + 1, e_1, Tail(\omega_1))$
20: **sinon si** $(p_1 > p_2)$ **alors** $r'_1.o \leftarrow Ins(p_1 + 1, e_1, \omega_1)$
21: **fin si**
22: **Cas :** $r_1 = d_1$ et $r_2 = i_2$
23: **si** $(p_1 \geq p_2 + 1)$ **alors** $r'_1.o \leftarrow Del(p_1 - 1)$
24: **sinon si** $(p_1 = p_2)$ **alors retourner "Indéfinie"**
25: **fin si**
26: **Cas :** $r_1 = d_1$ et $r_2 = d_2$
27: **si** $(p_1 \geq p_2)$ **alors** $r'_1.o \leftarrow Del(p_1 + 1)$
28: **fin choix**
29: **retourner** r'_1

Algorithme 10: Transformation exclusive utilisant des p-mots.

Pour exclure l'effet d'une requête d'effacement r_2 d'une autre requête d'effacement r_1, nous incrémentons par un la position de r_1 (comme si r_2 n'avait pas été exécutée avant r_1) lorsqu'elle a effacé un élément après celui de r_2 (lignes $25 - 27$).

Les autres cas de transformation de ET sont assez simples. Néanmoins, il y a des situations où ET n'est pas définie, ce qui explique la valeur retournée **"Indéfinie"**. A titre d'exemple, considérons r_1 et r_2 comme étant respectivement des requêtes d'effacement et d'insertion. Si la position d'effacement de r_1 est la même que la position d'insertion de r_2 alors r_1 a détruit l'élément ajouté par r_2. Il est impossible d'exclure l'effet de r_2 de r_1 car cette dernière dépends sémantiquement de r_1. En fait, tous les cas où ET est indéfinie représentent

des situation de dépendance sémantique telle que nous l'avons exprimée à la définition 3.5. Aussi, nous pouvons énoncer la condition suivante :

$$\text{si } r_1 \xrightarrow{s} r_2 \text{ alors } ET(r_2, r_1) \text{ est indéfinie.}$$

Nous donnons une extension de ET pour des séquences de requêtes :

Définition 3.10 (Transformation exclusive pour les séquences) *Soit* $seq = [r_1; \ldots; r_{n-1}; r_n]$. *La transformation exclusive de toute requête* r *par rapport à seq, notée* $ET^*(r, seq)$, *est définie récursivement :*

$$ET^*(r, []) = r$$
$$ET^*(r, [r_1; \ldots; r_{n-1}; r_n]) = ET^*(ET(r, u_n), [r_1; \ldots; r_{n-1}]).$$

■

La dépendance sémantique entre deux requêtes d'une même histoire se définit comme suit :

Définition 3.11 *Etant donnée une histoire* $H = [r_1; r_2; \ldots; r_n]$. *Nous disons que* $r_i \xrightarrow{s} r_j$, *où* $i, j \in \{1, \ldots, n\}$ *et* $i < j$, *ssi il existe soit :*

(i) $r' = ET^*(r_j, H[i+1, j-1])$ *tel que* $r_i \xrightarrow{s} r'$ *; ou,*

(ii) soit une requête r_k *avec* $i < k < j$ *et* $r_i \xrightarrow{s} r_k \xrightarrow{s} r_j$.

■

Comme propriétés de l'algorithme 10, nous donnons les lemmes suivants :

Une requête d'insertion doit être réversible par rapport à toute requête en utilisant les deux formes de transformation.

Lemme 3.4 *Soit* i *une requête d'insertion. Pour toute requête* r, *si* $IT(i, r) = i'$ *alors* $ET(i', r) = i$. ■

Preuve. Soient $i.o = Ins(p, e, \omega)$ et $i' = IT(i, r)$. Nous devons considérer deux cas pour r :

1. $r.o = Ins(p_1, e_1, \omega_1)$: Si $PW(r.o) \prec PW(i.o)$ (ou ($PW(r.o) = PW(i.o)$ et $r.u < i.u$)) alors $i'.o = Ins(p+1, e, p\omega)$. Comme $PW(r.o) \prec p\omega$ (ou $PW(r.o) = p\omega$ et $r.u < i.u$), alors $ET(i', r) = i$.
2. $r.o = Del(p_1)$: Il faut comparer p à p_1.
   (a) $p < p_1$: $i'.o = Ins(p, e, \omega)$ et $ET(i', r) = i$.
   (b) $p = p_1$: $i'.o = Ins(p, e, p\omega)$ et $ET(i', r) = i$.
   (c) $p_1 < p$: $i'.o = Ins(p-1, e, p\omega)$ et $ET(i', r) = i$.

■

Lemme 3.5 *Etant données deux requêtes* r_1 *et* r_2 *tel que* $r_1 \not\xrightarrow{s} r_2$. *Si* $ET(r_1, r_2) = r_1'$ *alors* $IT(r_1', r_2) \approx r_1$. ■

Preuve. Soit $r''_1 = IT(r'_1, r_2)$. Sans perte de généralité, considérons r_1 comme étant une requête d'insertion telle que $r_1.o = Ins(p_1, e_1, \omega_1)$. Deux cas sont possibles pour le choix de r_2 :

1. $r_2.o = Ins(p_2, e_2, \omega_2)$: Trois cas sont à considérer.
   (a) $p_1 = p_2 + 1$, $\omega_1 = \epsilon$ et $r_1.u > r_2.u$: $r'_1.o = Ins(p_1 - 1, e_1, \omega_2)$. Nous avons $PW(r'_1.o) = PW(r_2.o)$ et $r'_1.u > r_2.u$. Ainsi, $r''_1.o = Ins(p_1, e_1, (p_1 - 1).\omega_2)$ et $r''_1 \approx r_1$.
   (b) $p_1 > p_2 + 1$: Nous avons donc $r'_1.o = Ins(p_1 - 1, e_1, Tail(\omega_1))$ et $r''_1.o = Ins(p_1, e_1, (p_1 - 1).\omega_1)$. Ainsi $r''_1 \approx r_1$.
2. $r_2.o = Del(p_2)$: Nous considérons les cas suivants.
   (a) $\omega_1 \neq \epsilon$, $p_1 = p_2$ et $p_1 = Top(\omega_1)$: Nous avons donc $r'_1.o = Ins(p_1, e_1, Tail(\omega_1))$ et $r''.o = Ins(p_1, e_1, p_1\omega_1)$. Ainsi $r''_1 \approx r_1$.
   (b) $\omega_1 \neq \epsilon$ et $p_1 > p_2$: Comme $r'_1.o = Ins(p_1 + 1, e_1, Tail(\omega_1))$ et $r''.o = Ins(p_1, e_1, (p_1 + 1).\omega_1)$ alors $r''_1 \approx r_1$.
   (c) $\omega_1 = \epsilon$ et $p_1 > p_2$: Nous avons $r'_1.o = Ins(p_1 + 1, e_1, \omega_1)$ et $r''.o = Ins(p_1, e_1, [p_1 + 1])$. Ainsi $r''_1 \approx r_1$.

■

La transformation exclusive d'une requête par rapport à deux séquences équivalentes (ayant le même type de requêtes) donne le même résultat.

Lemme 3.6 *Soient r_1 et r_2 deux requêtes de même type (insertion ou effacement), tels que $r'_1 = IT(r_1, r_2)$ et $r'_2 = IT(r_2, r_1)$. Pour toute requête r, nous avons :*

$$ET^*(r, [r_1; r'_2]) = ET^*(r, [r_2; r'_1]).$$

■

Preuve. Deux cas sont à considérer :

1. $r_1.o = Del(p_1)$ et $r_2.o = Del(p_2)$:
2. $r_1.o = Ins(p_1, e_1, \omega_1)$ et $r_2.o = Ins(p_2, e_2, \omega_2)$:

■

3.3.3 Permutation des requêtes

Il y a des situations où il est nécessaire de réordonner des requêtes dans une histoire, sans affecter l'état résultant de l'exécution de cette histoire [68; 82]. A ce titre, nous proposons une fonction qui permet, à partir d'une séquence $[r_1; r_2]$, de produire la séquence $[r'_2; r'_1]$ qui pourrait exister dans le cas où r_2 a été exécutée avant r_1.

Définition 3.12 *Soient r_1 et r_2 deux requêtes. Nous définition la fonction $PERM : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{H}$ tel que : si $r_1 \overset{s}{\not\rightarrow} r_2$ alors $PERM(r_2, r_1) = [r'_2; r'_1]$ où $r'_2 = ET(r_2, r_1)$ et $r'_1 = IT(r_1, r'_2)$.* ■

Comme exemple de permutation, considérons le scénario illustré à la figure 3.1. Sur le site i, l'état initial est la chaîne $s_0 =$ "$BULLE''$. L'utilisateur génère la requête r_1 dont l'opération est $r_1.o = Del(2, U)$, qui une fois exécutée sur s_0, elle produit l'état $s_1 =$ "$BLLE''$. Ensuite, l'utilisateur exécute la requête r_2 pour insérer 'A' à la position 2 ($r_2.o = Ins(2, A, \epsilon)$). Cette requête produit l'état $s_2 =$ "$BALLE''$. La fonction $PERM(r_2, r_1)$ permet de réordonner la séquence $[r_1; r_2]$ en une autre $[r'_2; r'_1]$. Pour ce faire, $PERM$ commence par exclure l'effet de r_1 par la transformation $ET(r_2, r_1) = r'_2$ telle que $r'_2.o = Ins(2, A, \epsilon)$. Cette transformation exclusive était possible parce que $r_2 \overset{s}{\not\rightarrow} r_1$. Ensuite, l'effet de la nouvelle requête r'_2 doit être inclue dans r_1 par $IT(r_1, r'_2) = r'_1$ avec $r'_1.o = Del(3)$. Finalement, les deux séquences $[r_1; r_2]$ et $[r'_2; r'_1]$ sont équivalentes puisqu'elles mènent vers le même état.

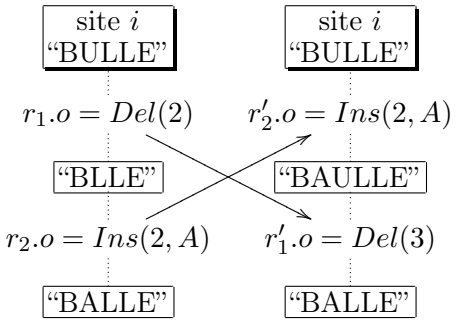


FIG. 3.1 – Permutation de séquences.

Comme propriétés de $PERM$, nous énonçons les lemmes suivants :

Les lemmes 3.7 et 3.8 montrent que la permutation des requêtes dans une histoire n'affecte pas l'état résultant de l'exécution de cette histoire.

Lemme 3.7 *Supposons que IT satisfait la condition TP1. Si* $PERM(r_2, r_1) = [r'_2; r'_1]$ *alors* $[r_1; r_2] \equiv [r'_2; r'_1]$. ■

Preuve. Nous avons $ET(r_2, r_1) = r'_2$, $IT(r'_2, r_1) = r''_2$ (où $r''_2 \approx r'_2$ selon le lemme 3.5) et $IT(r_1, r_2) = r'_1$. Puisque IT satisfait $TP1$ alors nous avons $[r'_2; r'_1] \equiv [r_1; r''_2]$ et $[r_1; r''_2] \equiv [r_1; r_2]$. ■

Lemme 3.8 *Soit* $H = [r_1; r_2; \ldots; r_i; r_{i+1}; \ldots; r_n]$ *une histoire. Supposons que* $PERM(r_{i+1}, r_i) = [r'_{i+1}; r'_i]$. *Alors l'histoire* $H' = [r_1; r_2; \ldots; r'_{i+1}; r'_i; \ldots; r_n]$ *est équivalente à* H. ■

Preuve. Soient $H_1 = [r_1; r_2; \ldots; r_i; r_{i+1}]$, $H_2 = [r_{i+2}; \ldots; r_n]$ et $H'_1 = [r_1; r_2; \ldots; r'_{i+1}; r'_i]$ En utilisant le lemme 3.7, nous avons $H_1 \equiv H'_1$. Puisque $[H_1; H_2] \equiv [H'_1; H_2]$ alors nous pouvons conclure que $H' \equiv H$. ■

Dans le lemme suivant, nous montrons comment les dépendances sémantiques sont préservées par permutation dans une histoire à trois requêtes.

Lemme 3.9 *Soit* $H = [r_1; r_2; r_3]$ *une histoire formée de trois requêtes. Alors les assertions suivantes sont vraies :*

(i) Supposons que r_2 *et* r_3 *sont de même type. Si* $r_1 \xrightarrow{s} r_2$ *et* $r_2 \not\xrightarrow{s} r_3$ *alors* $[r_1; r'_3; r'_2] \equiv_t H$ *tels que* $PERM(r_2, r_3) = [r'_3; r'_2]$ *et* $r_1 \xrightarrow{s} r'_2$ *;*

(ii) Supposons que r_1 *et* r_2 *sont de même type. Si* $r_1 \not\xrightarrow{s} r_2$ *et* $r_2 \xrightarrow{s} r_3$ *alors* $[r'_2; r'_1; r_3] \equiv_t H$ *tels que* $PERM(r_1, r_2) = [r'_2; r'_1]$ *et* $r'_2 \xrightarrow{s} r_3$ *;*

(iii) Supposons que r_1 *et* r_2 *sont de même type. Si* $r_1 \not\xrightarrow{s} r_2$ *et* $r_1 \xrightarrow{s} r_3$ *alors* $[r'_2; r'_1; r_3] \equiv_t H$ *tels que* $PERM(r_1, r_2) = [r'_2; r'_1]$ *et* $r'_1 \xrightarrow{s} r_3$ *;*

(iv) $(ii) \iff (iii)$.

■

Preuve. *Cas* (i) : D'après notre définition de dépendance sémantique nous avons forcément $r_1.o = Ins(p_1, e_1, \omega_1)$. Comme r_2 et r_3 sont de même type, nous avons deux cas à considérer :

1. $r_2.o = Del(p_2)$ et $r_3.o = Del(p_3)$: Comme $r_1 \xrightarrow{s} r_2$ alors $p_1 = p_2$. Si $p_3 \geq p_2$ alors nous avons $r'_3.o = Del(p_3 + 1)$, $r'_2 = Del(p_2)$ et $r_1 \xrightarrow{s} r'_2$. Sinon, nous avons $r'_3.o = Del(p_3)$, $r'_2 = Del(p_2 - 1)$ et $r_1 \xrightarrow{s} r'_2$.
2. $r_2.o = Ins(p_2, e_2, \omega_2)$ et $r_3.o = Ins(p_3, e_3, \omega_3)$: Comme $r_1 \xrightarrow{s} r_2$, nous avons soit $p_1 = p_2$ et $r_1.u \leq r_2.u$, ou soit $p_2 = p_1 + 1$ et $r_1.u \geq r_2.u$. Sans perdre de généralité, nous supposons que $\omega_1 = \omega_2 = \omega_3 = \epsilon$. Si $p_3 < p_2$ ou $p_3 = p_2$ et $r_3.u < r_2.u$ alors nous avons $r'_3.o = Ins(p_3, e_3, \omega_3)$, $r'_2 = Ins(p_2{+}1, e_2, [p_2])$ et $r_1 \xrightarrow{s} r'_2$. Par contre, si $p_3 > p_2{+}1$ ou $p_3 = p_2{+}1$ et $r_3.u > r_2.u$ alors $r'_3.o = Ins(p_3 - 1, e_3, \omega_3)$, $r'_2 = Ins(p_2, e_2, \omega_2)$ et $r_1 \xrightarrow{s} r'_2$.

Cas (ii) : Comme r_1 et r_2 sont de même type et $r_2 \xrightarrow{s} r_3$ alors nous avons forcément $r_1.o = Ins(p_1, e_1, \omega_1)$ et $r_2.o = Ins(p_2, e_2, \omega_2)$. Sans perdre de généralité, nous supposons que $\omega_1 = \omega_2 = \epsilon$. Nous avons deux cas à considérer :

1. $r_3.o = Del(p_3)$: Nous avons $p_2 = p_3$ puisque $r_2 \xrightarrow{s} r_3$.
   (a) $p_2 < p_1$ ou $(p_2 = p_1$ et $r_2.u < r_1.u)$: Nous avons donc $r'_2.o = Ins(p_2, e_2, \omega_2)$ et $r'_1.o = Ins(p_1 + 1, e_1, [p_1])$. Comme $p_3 < p_1 + 1$ alors $ET(r_3, r'_1) = r'_3$ tels que $r'_3.o = Del(p_3)$ et $r'_2 \xrightarrow{s} r'_3$.
   (b) $p_2 > p_1 + 1$ ou $(p_2 = p_1 + 1$ et $r_2.u > r_1.u)$: Nous avons donc $r'_2.o = Ins(p_2 - 1, e_2, \omega_2)$ et $r'_1.o = Ins(p_1, e_1, \omega_1)$. Comme $p_3 > p_1$ alors $ET(r_3, r'_1) = r'_3$ tels que $r'_3.o = Del(p_3 - 1)$ et $r'_2 \xrightarrow{s} r'_3$.
2. $r_3.o = Ins(p_3, e_3, \omega_3)$ et $, \omega_3 = \epsilon$:
   (a) $p_2 = p_3$ et $r_3.o \geq r_2.o$: Nous considérons deux cas.
      i. $p_2 < p_1$ ou $(p_2 = p_1$ et $r_2.u < r_1.u)$: Nous avons donc $r'_2.o = Ins(p_2, e_2, \omega_2)$ et $r'_1.o = Ins(p_1 + 1, e_1, [p_1])$. Comme $p_3 < p_1 + 1$ alors $ET(r_3, r'_1) = r'_3$ tels que $r'_3.o = Ins(p_3, e_3, \omega_3)$ et $r'_2 \xrightarrow{s} r'_3$.
      ii. $p_2 > p_1 + 1$ ou $(p_2 = p_1 + 1$ et $r_2.u > r_1.u)$: Nous avons donc $r'_2.o = Ins(p_2 - 1, e_2, \omega_2)$ et $r'_1.o = Ins(p_1, e_1, \omega_1)$. Comme $p_3 > p_1$ alors $ET(r_3, r'_1) = r'_3$ tels que $r'_3.o = Ins(p_3 - 1, e_3, \omega_3)$ et $r'_2 \xrightarrow{s} r'_3$.
   (b) $p_3 = p_2 + 1$ et $r_3.o \leq r_2.o$: Deux cas sont possibles :

i. $p_2 < p_1$ ou ($p_2 = p_1$ et $r_2.u < r_1.u$) : Nous avons donc $r'_2.o = Ins(p_2, e_2, \omega_2)$ et $r'_1.o = Ins(p_1 + 1, e_1, [p_1])$. Comme $p_3 = p_2 + 1$ et $p_2 \leq$ alors nous en déduisons que $p_3 \leq p_1 + 1$ et $r_3.u < r_1.u$. Ainsi $ET(r_3, r'_1) = r'_3$ tels que $r'_3.o = Ins(p_3, e_3, \omega_3)$ et $r'_2 \xrightarrow{s} r'_3$.

ii. $p_2 > p_1 + 1$ ou ($p_2 = p_1 + 1$ et $r_2.u > r_1.u$) : Nous avons donc $r'_2.o = Ins(p_2 - 1, e_2, \omega_2)$ et $r'_1.o = Ins(p_1, e_1, \omega_1)$. Comme $p_3 > p_1 + 1$ alors $ET(r_3, r'_1) = r'_3$ tels que $r'_3.o = Ins(p_3 - 1, e_3, \omega_3)$ et $r'_2 \xrightarrow{s} r'_3$.

■

3.3.4 Réordonner les histoires

Dans cette sous-section, nous allons étudier une classe de séquences de requêtes. L'objectif de cette étude est de pouvoir utiliser cette classe de séquences pour construire des chemins de transformation qui permettent sans faille d'assurer la convergence dans objets collaboratifs linéaires.

Nous avons vu auparavant que nous pouvons réordonner successivement deux requêtes d'une histoire et ce sans altérer l'état résultant de l'exécution de cette histoire. Ainsi, nous pouvons donc définir une relation d'équivalence entre les histoires comme suit :

Définition 3.13 *Une histoire $(H, \ll_H)$ est* équivalente par transformation *à une histoire $(H', \ll_{H'})$, notée $H \equiv_t H'$, ssi les conditions suivantes sont satisfaites :*

(i) $|H| = |H'|$;

(ii) $H \equiv H'$;

(iii) $(H', \ll_{H'})$ *peut être obtenu à partir de* $(H, \ll_H)$ *en appliquant un nombre fini de permutations.*

■

Nous pouvons voir cette équivalence par transformation comme une application bijective $f : H \rightarrow H'$, de telle sorte que f peut être considérée comme la composition d'un nombre fini de permutations. Chaque permutation a la forme suivante :

$$c_i : H \rightarrow H' \text{ (avec } 1 < i \leq |H| - 1) \text{ tels que}$$

$$\begin{cases} c_i(r_i) = r'_i & \text{où } r'_i = ET(r_{i-1}, r_i) \text{ et } (r_{i-1}, r_i) \in \ll_H \\ c_i(r_{i-1}) = r'_{i-1} & \text{où } r'_{i-1} = IT(r_{i-1}, r'_i) \\ c_i(r) = r & \text{pour tout } r \neq r_i \text{ et } r \neq r_{i-1} \end{cases}$$

Par abus de notation, nous écrivons $f(H) = H'$.

La préservation de la dépendance sémantique dans une histoire H signifie que toute requête r_j dépendant d'une autre requête r_i doit impérativement apparaître après celle-ci, *i.e.* $i < j$.

Définition 3.14 *Une histoire H est dite* préservant la dépendance sémantique *(PDS) ssi l'une des conditions suivantes est satisfaite :*

(i) $|H| \leq 1$;

(ii) $|H| > 1$ *et pour toutes les requêtes* $H[i]$ *et* $H[j]$ *avec* $i, j \in \{0, \ldots, |H|-1\}$: *si* $H[i] \xrightarrow{s} H[j]$ *alors* $i < j$.

■

Nous définissons maintenant une classe d'histoires se caractérisant par le fait que chaque histoire peut être considérée comme la concaténation d'une séquence de requêtes d'insertion avec une autre séquence de requêtes d'effacement.

Définition 3.15 *Une histoire* H *est dite* canonique *ssi (i)* H *est une histoire PDS ; et, (ii)* $H = [H_i; H_d]$ *avec* H_i *est une sous-histoire contenant des requêtes d'insertion et* H_d *est une sous-histoire contenant des requêtes d'effacement.* ■

Il faut souligner que H_d contiennent des requêtes qui sont concurrentes entre-elles. Par conséquent, seule H_i doit être individuellement une PDS.

Dans ce qui suit, nous allons énoncer des propriétés propres aux histoires canoniques.

Théorème 3.2 *Etant données deux histoires canoniques* H_1 *et* H_2 *telle que* $H_1 \equiv_t H_2$. *Pour toute requête* r *nous avons :*

$$IT^*(r, H_1) \approx IT^*(r, H_2).$$

■

Preuve. Soit $f : H_1 \rightarrow H_2$ une application bijective. Supposons que f est la composition de p permutations. Ensuite, nous procédons par induction sur p.

(i) *base d'induction :* Soit $p = 1$ et $H_1 = [r_1; \ldots; r_{i-1}; r_i; \ldots; r_n]$. Supposons que la permutation est réalisée à l'indice i $(1 < i < n)$ et, r_{i-1} et r_i sont de même type. Par conséquent, nous obtenons $H_2 = [r_1; \ldots; r'_i; r'_{i-1}; \ldots; r_n]$ à partir de H_1 tel que $PERM(r_{i-1}, r_i) = [r'_i; r'_{i-1}]$. De cette façon nous avons :

$$H_1 = [H_1[1, i-2]; r_{i-1}; r_i; H_1[i+1, n]] \text{ et}$$
$$H_2 = [H_1[1, i-2]; r'_i; r'_{i-1}; H_1[i+1, n]].$$

Soit $r' = IT^*(r, H_1[1, i-2])$. Ainsi, nous avons $IT^*(r, H_1) = IT^*(IT^*(r', [r_{i-1}; r_i]), H_1[i+1, n])$ et $IT^*(r, H_2) = IT^*(IT^*(r', [r'_i; r'_{i-1}]), H_1[i+1, n])$. Trois cas sont possibles :

(a) r' est une requête d'effacement : dans ce cas $IT^*(r', [r_{i-1}; r_i]) = IT^*(r', [r'_i; r'_{i-1}]) = r''$ (voir le lemme 2.4) et par conséquent $IT^*(r, H_1) = IT^*(r, H_2)$.

(b) r, r_i et r_i des requêtes d'insertion : en utilisant le lemme 3.2, nous avons $IT^*(r', [r_{i-1}; r_i]) = IT^*(r', [r'_i; r'_{i-1}]) = r''$; par conséquent $IT^*(r, H_1) = IT^*(r, H_2)$.

(c) r est une requête d'insertion et, r_i et r_i sont des requêtes d'effacement : soient $r'' = IT^*(r', [r_{i-1}; r_i])$ et $r''' = IT^*(r', [r'_i; r'_{i-1}])$. En utilisant le lemme 3.3, nous avons $r'' \approx r'''$. Comme H_1 et H_2 sont canoniques alors $H_1[i+1, n]$ contient seulement des requêtes d'effacement. Par conséquent, $IT^*(r'', H_1[i+1, n]) \approx IT^*(r''', H_1[i+1, n])$ (voir le théorème 3.1) et $IT^*(r, H_1) \approx IT^*(r, H_2)$.

(ii) *Hypothèse d'induction :* Supposons que ce théorème est vrai pour $p \geq 1$.

(iii) *Pas d'induction :* Nous prouvons maintenant qu'il est pour $p+1$. Etant donné une histoire canonique H construite à partir de H_1 en appliquant p permutations. Soit H_2 une histoire canonique obtenue en réalisant une permutation sur H. Comme $IT^*(r, H_1) \approx IT^*(r, H)$ (hypothèse d'induction) et $IT^*(r, H) \approx IT^*(r, H_2)$ (base d'induction) alors $IT^*(r, H_1) \approx IT^*(r, H_2)$.

■

La dépendance sémantique est préservée par transformation inclusive par rapport à toute requête.

Lemme 3.10 *Soient i_1 et i_2 deux requêtes concurrentes d'insertion. Pour toute requête r, Si $i_1 \xrightarrow{s} r$ et $r \parallel i_2$ alors $IT(i_1, i_2) \xrightarrow{s} IT(r, i'_2)$ où $i'_2 = IT(i_2, i_1)$.* ■

Preuve. Soient $i_1.o = Ins(p_1, e_1, \omega_1)$, $i_2.o = Ins(p_2, e_2, \omega_2)$, $i'_1 = IT(i_1, i_2)$, $i'_2 = IT(i_2, i_1)$ et $r' = IT(r, i'_2)$. Trois cas sont possibles :

1. $r.o = Ins(p, e, \omega)$, $p = p_1$ et $r.u \geq i_1.u$: Selon les relations entre les paramètres de i_1 et i_2 nous avons deux cas :
   (a) ($p_1 = p_2$ et $i_1.u < i_2.u$) ou ($p_1 < p_2$) : nous avons $i'_1 = i_1$, $i'_2.o = Ins(p_2 + 1, e_2, p_2\omega_2)$ et $r' = r$.
   (b) ($p_1 = p_2$ et $i_1.u > i_2.u$) ou ($p_1 > p_2$) : nous avons $i'_1.o = Ins(p_1 + 1, e_1, p_1\omega_1)$, $i'_2 = i_2$ et $r'.o = Ins(p+1, e, p\omega)$.
2. $r.o = Ins(p, e, \omega)$ $p = p_1 + 1$ et $r.u \leq i_1.u$: Selon les relations entre les paramètres de i_1 et i_2 nous avons deux cas :
   (a) ($p_1 = p_2$ et $i_1.u < i_2.u$) ou ($p_1 < p_2$) : nous avons $i'_1 = i_1$, $i'_2.o = Ins(p_2 + 1, e_2, p_2\omega_2)$ et $r' = r$.
   (b) ($p_1 = p_2$ et $i_1.u > i_2.u$) ou ($p_1 > p_2$) : nous avons $i'_1.o = Ins(p_1 + 1, e_1, p_1\omega_1)$, $i'_2 = i_2$ et $r'.o = Ins(p+1, e, p\omega)$.
3. $r.o = Del(p)$ et $p = p_1$: Nous avons également deux cas.
   (a) ($p_1 = p_2$ et $i_1.u < i_2.u$) ou ($p_1 < p_2$) : dans cas nous avons $i'_1 = i_1$, $i'_2.o = Ins(p_2 + 1, e_2, p_2\omega_2)$ et $r' = r$.
   (b) ($p_1 = p_2$ et $i_1.u > i_2.u$) ou ($p_1 > p_2$) : dans ce cas nous avons $i'_1.o = Ins(p_1 + 1, e_1, p_1\omega_1)$, $i'_2 = i_2$ et $r'.o = Del(p+1)$.

■

La dépendance sémantique est préservée par transformation inclusive par rapport à toute histoire canonique.

Théorème 3.3 *Supposons que H est une histoire canonique contenant seulement des requêtes d'insertion. Soient r_1 et r_2 deux requêtes concurrentes avec H. Si $r_1 \xrightarrow{s} r_2$ alors $IT^*(r_1, H) \xrightarrow{s} IT^*(r_2, H')$ où $H' = IT^*(H, u_1)$.* ■

Preuve. Par induction sur $|H|$. ■

En ajoutant une requête d'insertion à une histoire canonique, il est toujours possible de construire une nouvelle histoire canonique.

Lemme 3.11 *Supposons que H est une histoire qui contient seulement des requêtes d'effacement. Soit i une requête d'insertion qui est générée sur l'état $s = Do(H, s_0)$. Alors il existe une histoire canonique H' telle que $H' \equiv_t [H; i]$.* ■

Preuve. Soit $|H| = n$. Nous procédons par induction sur n.

(i) *Base d'induction :* pour $n = 1$ nous avons $H = [d]$ où d est une requête d'effacement. Ainsi, en utilisant le lemme 3.7 nous avons $[d; i] \equiv_t [i'; d']$ où $[i'; d'] = PERM(i, d)$. Ainsi $H' = [i'; d']$ est canonique.

(ii) *Hypothèse d'induction :* pour $n > 1$ il existe une histoire canonique $H' \equiv_t [H; i]$.

(iii) *Pas d'induction :* Soit $H = [d_1; d_2; \ldots; d_n; d_{n+1}]$. En utilisant l'hypothèse d'induction, nous avons : $H = [d_2; \ldots; d_n; d_{n+1}; i] \equiv_t [i'; d'_2; \ldots; d'_n; d'_{n+1}]$ où i' est le résultat de la permutation de i avec les requêtes d_2, …, d_{n+1}. Selon le lemme 3.7, $[i''; d'_1] \equiv_t [d_1; i']$ puisque $PERM(i', d_1) = [i''; d'_1]$. Par conséquent, $H' = [i''; d'_1; d'_2; \ldots; d'_n; d'_{n+1}; i] \equiv_t [H; i]$ et H' est canonique.

■

3.4 OPTIC : Un algorithme d'intégration de requêtes

Nous proposons un nouvel algorithme, nommé OPTIC (abréviation de OPerational Transformation with Intense Concurrency), pour la gestion de concurrence dans les éditeurs collaboratifs. Cet algorithme repose à la fois sur un modèle de concurrence où les accès aux données sont concomitants (grâce à la réplication) et sur un modèle de cohérence causale. Par ailleurs, pour restaurer la cohérence des données OPTIC utilise l'approche des transformées opérationnelles. Cette approche confère un très haut degré de concurrence quant à la modification des objets répliqués.

OPTIC se situe dans la famille des algorithmes optimistes dont la restauration de la cohérence du système se fait de manière automatique (sans intervention manuelle des utilisateurs). Au regard de certains algorithmes de gestion de concurrence basés l'approche des transformées opérationnelles, OPTIC revêt un certain nombre d'avantages :

1. Grâce à sa politique optimiste, il permet un accès simultané aux données avec une restauration automatique de la cohérence des données.

2. Il met en œuvre une causalité minimale entre les requêtes grâce à un mécanisme de détection de dépendances qui se base essentiellement sur la sémantique de l'objet collaboratif.
3. Il n'est pas tributaire du nombre de participants comme c'est la cas des éditeurs collaboratifs dont la causalité est implantée par des vecteurs d'état. En effet, le nombre des participants peut être arbitraire ce qui facilite le déploiement de `OPTIC` sur un réseau pair-à-pair.

Nous considérons un éditeur collaboratif comme un groupe de N sites (où N est arbitraire) qui démarrent une session de collaboration à partir du même état initial s_0. Chaque site possède une histoire canonique H (les insertions avant les effacement) contenant toutes les requêtes exécutées sur ce site. `OPTIC` est détaillé dans l'algorithme 11 qui doit se dérouler sur chaque site.

3.4.1 Génération d'une requête locale

Chaque fois qu'une opération o est générée localement, nous l'exécutons *immédiatement* sur son état de génération, à savoir $s = Do(H, s_0)$. Après avoir formé la requête $r = (u, k, null, o)$ [14], nous appelons la fonction ComputeBF(r,H) (voir l'algorithme 12) pour calculer le contexte minimal de génération de r. En d'autres termes, au lieu de considérer H comme étant le contexte de génération de r, nous allons tenter de minimiser ce contexte en excluant autant que possible certaines requêtes de H (par l'intermédiaire de la transformation exclusive ET). Pour bien comprendre cette étape, considérons l'ensemble $Dep(r) = \{r'' \in H | r'' \xrightarrow{s} r\}$ qui est construit en partant de la définition 3.11. Si $Dep(r) = \emptyset$, alors la nouvelle requête obtenue r' est indépendante de H (dans ce cas $r \approx IT^*(r', H)$). Sinon, $Dep(r) \neq \emptyset$, alors r' doit être exécutée sur tous les sites après les requêtes de $Dep(r)$. Dans ce cas, $r'.a$ contiendra seulement l'identifiant de la requête dont elle dépend directement plus la forme de dépendance fd_j (voir la définition 3.5).

L'intégration de r après H peut produire une histoire non canonique. Pour rendre $[H; r]$ canonique, nous utilisons la fonction Canonize(r,H) qui est présentée dans l'algorithme 13. Cette fonction procède par des permutations successives en utilisant $PERM$.

Enfin, nous diffusons la requête r' (qui est le résultat de la fonction ComputeBF) vers les autres sites pour qu'elle soit exécutée sur les autres copies.

3.4.2 Intégration d'une requête distante

Chaque site dispose d'une file d'attente Q pour stocker les requêtes distantes qui proviennent des autres sites. Une fois reçue, la requête est ajoutée à Q. Pour assurer la préservation de la causalité, une requête distante est invoquée quand elle est causalement prête. Etant donnée une requête r générée sur le site i. Dans `OPTIC` , r est causalement prête sur le site j si la requête dont elle dépend a été reçue et exécutée sur ce site [15].

[14] *null* signifie que r ne dépend d'aucune requête.

[15] Si $r' \xrightarrow{s} r$ alors il suffit de chercher si l'histoire locale H contient r'.

```
1: Programme principal :
2: Initialisation
3: tant que présent sur le réseau faire
4:    si il y a une opération o alors
5:       Generate_Request(o)
6:    sinon
7:       Receive_Request
8:       Integrate_Remote_Requests
9:    fin si
10: fin tant que

11: Initialisation :
12: Q ← []
13: H ← []
14: s ← s_0
15: k ← 1
16: u ← Identification de l'utilisateur local

17: Generate_Request(o) :
18: s ← Do(o, s)
19: r ← (u, k, null, o)
20: r' ← ComputeBF(r,H)
21: H ← Canonize(r,H)
22: diffuser r' aux autres utilisateurs

23: Receive_Request :
24: si il y a une requête r qui provient du réseau alors
25:    Q ← Q + r
26: fin si

27: Integrate_Remote_Requests :
28: si il y a r dans Q qui est causalement prête pour l'exécution alors
29:    Q ← Q − r
30:    r' ← ComputeFF(r,H)
31:    s ← Do(r'.o, s)
32:    H ← Canonize(r',H)
33: fin si
```

Algorithme 11: Contrôle de Concurrence

Nous balayons Q de la gauche vers la droite pour extraire la première requête r qui est causalement prête. Ensuite, nous appelons la fonction ComputeFF(r,H) (voir l'algorithme 14) pour calculer la forme transformée r' qui sera exécutée sur l'état $s = Do(H, s)$. Soit n est la longueur de H. Deux cas sont donc possibles :

1. Si $r'.a = null$ alors r' est transformée par rapport à H.
2. Si $r'.a \neq null$ alors il existe une requête $H[j]$ ($j \in \{0, \ldots, n-1\}$) dont r' dépend ; $H[j]$ est trouvée grâce aux informations contenues dans $r'.a$. Contrairement aux autres algorithmes d'intégration, OPTIC ne réorgani-

```
1: COMPUTEBF(r,H) : r'
2: r' ← r
3: pour (i = |H| − 1 ; i ≥ 0 ; i − −) faire
4:   si r' est indépendante de H[i] alors
5:     r' ← ET(r', H[i])
6:   sinon
7:     r'.a = (H[i].u, H[i].k, fd_j) {fd_j avec j = 1, 2, ou 3 selon la forme de
       dépendance}
8:     retourner r'
9:   fin si
10: fin pour
11: retourner r'
```

Algorithme 12: Détection de dépendance sémantique

```
1: CANONIZE(r,H) : H'
2: H' ← [H; r]
3: si H' est canonique alors
4:   retourner H'
5: sinon
6:   pour (i = |H'| − 1 ; i ≥ 0 ; i − −) faire
7:     < H'[i − 1], H'[i] >← PERM(H'[i], H'[i − 1])
8:     si H' est canonique alors
9:       retourner H'
10:    fin si
11:  fin pour
12: fin si
```

Algorithme 13: Canonisation des histoires.

sera pas H en deux sous-histoires contenant respectivement les requêtes précédant r' et les requêtes concurrentes à r'. Au lieu, il met à jour $r'.o$ en fonction de $H[j]$ et la forme de dépendance et ensuite il la transforme par rapport à $H[j+1, n-1]$.

Finalement, la forme transformée r est exécutée après H. Un appel à la fonction CANONIZE est fait pour rendre $[H; r']$ canonique.

3.4.3 Exemple illustratif

Nous allons montrer le fonctionnement de `OPTIC` sur un scénario compliqué qui a été publié dans [52]. Supposons que nous avons trois sites qui démarrent une session de collaboration à partir du même état initial s_0 = "abc". Les trois sites génèrent respectivement trois requêtes concurrentes $r_1.o = Del(2)$, $r_2.o = Ins(3, x, \epsilon)$ et $r_3.o = Ins(2, y, \epsilon)$. Le site 1 génère $r_4.o = Del(1)$ après l'intégration de r_1, r_2 et r_3. Après avoir intégré r_2 et r_1, le site 2 génère $r_5.o = Del(1)$. Quant au site 3, il génère $r_6.o = Ins(3, z)$ après l'intégration de r_3, r_2 et r_1. A l'état de repos, les trois sites vont converger vers l'état "$yzxc$". L'ordre d'intégration de toutes les requêtes est illustré à la figure 3.2.

```
1: ComputeFF(r,H) : r'
2: r' ← r
3: j ← −1
4: si r'.a ≠ null alors
5:    Soit j l'indice dans H de la requête dont dépend r'
6:    Mettre à jour r'.o à partir de H[j].o selon la forme de dépendance
7: fin si
8: pour (i = j + 1 ; i ≤ |H| − 1 ; i + +) faire
9:    r' ← IT(r', H[i])
10: fin pour
11: retourner r'
```

Algorithme 14: Forme transformée à exécuter.

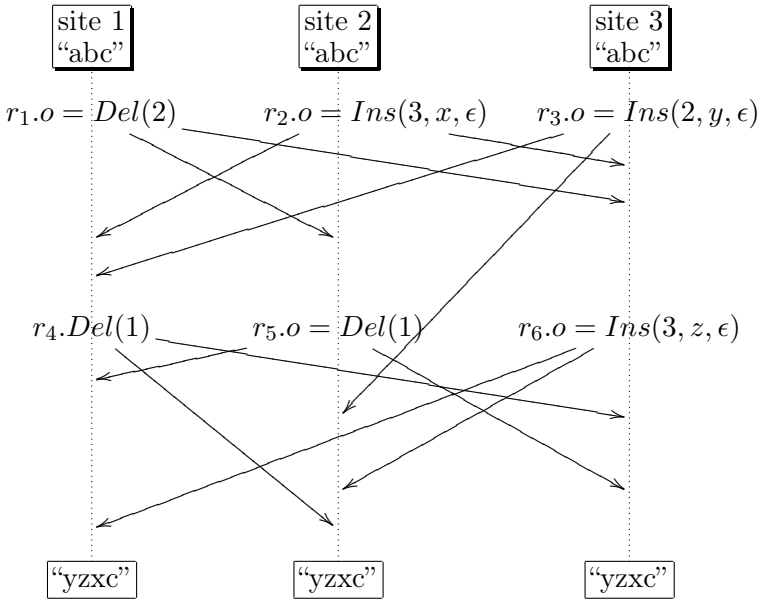


FIG. 3.2 – Scénario d'une session de collaboration entre trois sites.

L'intégration des requêtes est décrite en trois étapes :

Etape 1. Initialement, les histoires des sites sont vide ($H_i^0 = []$ pour $i = 1, 2, 3$). Dans ce cas, les requêtes sont considérées comme concurrentes puisqu'elles ne dépendent d'aucune autre requête.

Sur le site 1, après l'exécution locale de r_1, l'état obtenu est $s_1^1 =$ "ac" et la nouvelle histoire est $H_1^1 = [r_1]$. Lorsque r_2 est reçue, comme r_1 et r_2 sont concurrentes, la forme transformée $r_2' = IT(r_2, r_1)$ (où $r_2'.o = Ins(2, x, [3])$ est exécutée pour produire l'état $s_1^2 =$ "axc". L'histoire $[H_1^1; r_2']$ est transformée pour devenir canonique comme $H_1^2 = [r_2; r_1]$. L'intégration de r_3 passe par une transformation par rapport à H_1^2. Nous obtenons d'abord $r_3'' = IT(r_3, r_2) = r_3$; ensuite nous obtenons $r_3' = IT(r_3, r_1)$ avec $r_3'.o = Ins(2, y, [2])$. L'exécution de r_3' produit l'état $s_1^3 =$ "$ayxc$". La canonisation de $[H_1^2; r_3']$ produit $H_1^3 = [r_2; r_3; r_1']$ avec $r_1'.o = Del(3)$.

Sur le site 2, l'exécution locale de r_2 produit l'état $s_2^1 =$ "$abxc$" et l'histoire

$H_2^1 = [r_2]$. L'intégration de r_1 passe par la transformation $r'_1 = IT(r_1, r_2) = r_1$. L'état $s_2^2 =$ "axc'' et l'histoire $H_2^2 = [r_2; r_1]$ sont le résultat de l'exécution de r_1.

Sur le site 3, après l'exécution locale de r_3, l'état résultant est $s_3^1 =$ "$aybc''$ et l'histoire obtenue est $H_3^1 = [r_3]$. Lorsque r_2 arrive, elle est transformée en $r'_2 = IT(r_2, r_3)$ avec $r'_2.o = Ins(4, x, [3])$. L'exécution de r'_2 produit $s_3^2 =$ "$aybxc''$ et $H_3^2 = [r_3; r_2]$. L'intégration de r_1 passe par la transformation $r'_1 = IT^*(r_1, H_3^2)$ avec $r'_1.o = Del(3)$, dont l'exécution mène à l'état $s_3^3 =$ "$ayxc''$ et l'histoire $H_3^3 = [r_3; r'_2, r'_1]$.

Etape 2. Trois requêtes concurrentes r_4, r_5 et r_6 sont générées respectivement sur les sites 1, 2 et 3. Elles sont diffusées aux autres sites comme suit.

Sur le site 1, r_4 est générée après $H_1^3 = [r_2; r_3; r'_1]$ avec $r'_1.o = Del(3)$. L'exécution de r_4 donne l'état $s_1^4 =$ "yxc''. Pour diffuser r_4 avec un contexte minimal de génération, elle est exclusivement transformée (en utilisant la fonction ComputeBF) par rapport à H_1^3 pour devenir $r'_4 = ET^*(r_4, H_1^3) = r_4$. Ainsi, r_4 est concurrente à H_1^3 et elle est donc propagée aux autres sites. L'intégration de r_4 produit l'histoire $H_1^4 = [r_2; r_3; r'_1; r_4]$.

Sur le site 2, r_5 est générée sur l'état $s_2^2 =$ "axc'' et après l'histoire $H_2^2 = [r_2; r_1]$. Son exécution produit $s_2^3 =$ "xc''. La même requête est propagée aux autres sites puisque $r_5 = ET^*(r_5, H_2^2)$; r_5 est donc concurrente à H_2^2. L'histoire obtenue est $H_2^3 = [r_2; r_1; r_5]$.

Sur le site 3, r_6 est générée après l'histoire $H_3^3 = [r_3; r'_2; r'_1]$ (avec $r'_2.o = Ins(4, x, [3])$ et $r'_1.o = Del(3)$) et son exécution mène vers l'état $s_3^4 =$ "$ayzxc''$. La transformation exclusive de r_6 par rapport à $[r'_2; r'_1]$ donne $r'_6 = r_6$ avec $r_6.o = Ins(3, z, \epsilon)$. Comme $r_3.o = Ins(2, y, \epsilon)$, il est clair que $r_3 \xrightarrow{s} r_6$ (deuxième forme de dépendance fd_2). Ainsi, r_6 sera propagée aux autres avec comme information qu'elle dépend de r_3, *i.e.* $r_6.a = (3, 1, fd_2)$. Enfin, $[H_3^3; r_6]$ est transformée sous la forme canonique $H_3^4 = [r_3; r'_2; r_6; r''_1]$ avec $r''_1.o = Del(4)$.

Etape 3. Nous allons voir maintenant comment intégrer les requêtes r_4, r_5 et r_6 sur les autres sites.

Sur le site 1, lorsque r_5 arrive, elle est concurrente à $H_1^4 = [r_2; r_3; r'_1; r_4]$ avec $r'_1.o = Del(3)$. Dans ce cas, elle est transformée en $r'_5 = IT^*(r_5, H_1^4)$ avec $r'_5.o = Nop()$. L'exécution de r'_5 produit l'état $s_1^5 = s_1^4 =$ "yxc'' et l'histoire $H_1^5 = [r_2; r_3; r'_1; r_4; r'_5]$. Lorsque r_6 arrive, elle est causalement prête ($r_3 \xrightarrow{s} r_6$) puisque r_3 a été déjà exécutée. Son intégration nécessite qu'elle soit transformée en $r'_6 = IT(r_6, [r_3; r'_1; r_4; r'_5])$ avec $r'_6.o = Ins(2, z, [3.3])$. L'exécution de r'_6 produit l'état $s_1^6 =$ "$yzxc''$. La canonisation de $[H_1^5; r'_6]$ donne l'histoire $H_1^6 = [r_2; r_3; r_6; r''_1; r_4; r'_5]$ avec $r''_1.o = Del(4)$.

Sur le site 2, lorsque r_3 arrive, l'état actuel est $s_2^3 =$ "xc'' et l'histoire courante est $H_2^3 = [r_2; r_1; r_5]$. La requête r_3 est donc transformée en $r'_3 = IT^*(r_3, H_2^3)$ avec $r'_3.o = Ins(1, y, [2.2])$. L'exécution de r'_3 mène vers l'état $s_2^4 =$ "yxc'' et l'histoire canonique $H_2^4 = [r_2; r_3; r'_1; r_5]$ avec $r'_1.o = Del(3)$. Comme r_6 est causalement prête, son intégration passe par la transformation $r'_6 = IT^*(r_6, [r'_1; r_5])$ avec $r'_6.o = Ins(2, z, [3.3])$, dont l'exécution produit l'état $s_2^5 =$ "$yzxc''$ ainsi que l'histoire canonique $H_2^5 = [r_2; r_3; r_6; r''_1; r_5]$ avec $r''_1.o = Del(4)$. Comme vu dans l'étape 2, r_4 ne dépend d'aucune requête. Elle est donc transformée en $r'_4 =$

$IT^*(r_4, H_2^5)$ avec $r'_4.o = Nop()$. L'état final est $s_2^6 = s_2^5 =$ "$yzxc''$ et l'histoire finale est $H_2^6 = [r_2; r_3; r_6; r''_1; r_5; r'_4]$.

Sur le site 3, l'état actuel est $s_3^4 =$ "$ayzxc''$ et l'histoire courante est $H_3^4 = [r_3; r'_2; r_6; r''_1]$ avec $r'_2.o = Ins(4, x, [3])$ et $r''_1.o = Del(4)$. Lorsque r_4 arrive, elle est transformée en $r'_4 = IT^*(r_4, H_3^4)$ avec $r'_4.o = Del(1)$. L'exécution de r'_4 produit l'état $s_3^5 =$ "$yzxc''$ et l'histoire $H_3^5 = [r_3; r'_2; r_6; r''_1; r'_4]$. De la même manière, l'intégration de r_5 se fait par transformation par rapport à H_3^5 avec $r'_5 = IT^*(r_5, H_3^5)$ et $r_5.o = Nop()$. L'état final est $s_3^6 = s_3^5 =$ "$yzxc''$ et l'histoire finale est $H_3^6 = [r_3; r'_2; r_6; r''_1; r'_4, r'_5]$.

3.4.4 Correction

Selon la définition 3.9, un éditeur collaboratif doit préserver la relation de dépendance et assurer la convergence des copies. Dans ce qui suit, nous allons montrer que OPTIC satisfait les deux propriétés.

Préservation de la relation de dépendance

S'il y a une relation de dépendance entre deux requêtes alors cette relation est préservée par transformation.

Lemme 3.12 *Soit $H = [r_1; r_2; \ldots; r_n]$ une histoire canonique. Pour toute histoire canonique H' telle que $H \equiv_t H'$, si $r_i \xrightarrow{s} r_j$ alors $f(r_i) \xrightarrow{s} f(r_j)$ où $f : H \to H'$ est une application bijective.* ■

Preuve. Supposons que f est la composition de p permutations. Alors nous procédons par induction sur p.

(i) *Base d'induction :* Soit $p = 1$ et $H = [r_1; \ldots; r_i; \ldots; r_j; \ldots]$ tel que $i < j$. Sans perdre de généralité,nous considérons le cas où $r_i \xrightarrow{s} r_j$ est une dépendance directe *i.e.* il n'y a pas de r_k tels que $i < k < j$ et $r_i \xrightarrow{s} r_k \xrightarrow{s} r_j$. Soient $r'_j = ET(r_j, H[i+1, j-1])$ et $r''_j = ET(r_j, H[i+2, j-1])$. Par définition, $r_i \xrightarrow{s} r_j$ implique que $r_i \xrightarrow{s} r'_j$ et $r_i \xrightarrow{s} r''_j$. Soit k la position où la permutation a eu lieu sur H. Nous avons les cas suivants :

(a) $k < i$ ou $k > j + 1$:
$PERM(r_{k-1}, r_k)$ n'affecte pas les requêtes entre r_i et r_j ;

(b) $i + 1 < k < j$:
Comme H' est canonique alors r_{k-1} et r_k doivent être de même type. De $H = [\ldots; r_i; \ldots; r_{k-1}; r_k; \ldots; r_j; \ldots]$ nous obtenons $H' = [\ldots; r_i; \ldots; r'_k; r'_{k-1}; \ldots; r_j; \ldots]$ tel que $PERM(r_{k-1}, r_k) = [r'_k; r'_{k-1}]$. Comme $H[i+1, j-1] \equiv_t H'[i+1, j-1]$, alors $r'_j = ET(r_j, H[i+1, j-1]) = ET(r_j, H'[i+1, j-1])$ en utilisant le lemme 3.6. D'où $r_i \xrightarrow{s} r_j$ est préservée.

(c) $k = i$:
De $H = [\ldots; r_{i-1}; r_i; \ldots; r_j; \ldots]$ nous obtenons $H' = [\ldots; r'_i; r'_{i-1}; \ldots; r_j; \ldots]$. Comme $r_{i-1} \not\xrightarrow{s} r_i$ et $r_i \xrightarrow{s} r'_j$ alors $r'_i \xrightarrow{s} r_j$ en utilisant le lemme 3.9(ii).

(d) $k = i + 1$:
De $H = [\ldots; r_i; r_{i+1}; \ldots; r_j; \ldots]$ nous obtenons $H' = [\ldots; r'_{i+1}; r'_i; \ldots; r_j; \ldots]$. Comme $r_i \overset{s}{\not\rightarrow} r_{i+1}$ et $r_i \overset{s}{\rightarrow} r''_j$ alors $r'_i \overset{s}{\rightarrow} r_j$ en utilisant le lemme 3.9(iii).

(e) $k = j$:
De $H = [\ldots; r_i; \ldots; r_{j-1}; r_j; \ldots]$ nous obtenons $H' = [\ldots; r_i; \ldots; r'_j; r'_{j-1}; \ldots]$. Comme $r_i \overset{s}{\rightarrow} r_j$ alors $r_i \overset{s}{\rightarrow} ET(r_j, r_{j-1})$.

(f) $k = j + 1$:
Si r_j et r_{j+1} sont de même type alors de $H = [\ldots; r_i; \ldots; r_j; r_{j+1}; \ldots]$ nous obtenons $H' = [\ldots; r_i; \ldots; r'_{j+1}; r'_j; \ldots]$. Comme $r_i \overset{s}{\rightarrow} r_j$ et $r_j \overset{s}{\not\rightarrow} r_{j+1}$ alors $r_i \overset{s}{\rightarrow} r'_j$ en utilisant le lemme 3.9(i).

(ii) *Hypothèse d'induction :* Ce lemme est vrai pour $p \geq 1$.

(iii) *Pas d'induction :* Montrons que ce lemme est vrai pour $p+1$. Soient H'' une histoire canonique construite à partir de H en appliquant p permutations et H' une histoire canonique obtenue en faisant une permutation à H''. Alors ce lemme est vrai pour H' puisque il est vraie pour H'' (hypothèse d'induction) et il est vrai pour une seule permutation (base d'induction).

■

Toutes les dépendances contenues dans une histoire canonique sont préservées par transformation.

Théorème 3.4 *Soient H_1 et H_2 deux histoires canoniques telle que $H_1 \equiv_t H_2$. Si H'_1 est une sous-histoire fermée de H_1 alors $f(H'_1)$ est aussi une sous-histoire fermée de H_2 où $f : H_1 \rightarrow H_2$ est une application bijective.* ■

Preuve. Soit $n = |H'_1|$. Nous procédons par induction sur n.

(i) *Base d'induction :* pour $n = 2$ nous avons $H'_1 = [r_i; r_j]$ tel que $r_i \overset{s}{\rightarrow} r_j$ avec $i, j \in \{1, \ldots, |H_1|\}$ et $i < j$. En utilisant le lemme 3.12, nous pouvons dire que $f(H'_1)$ est une sous-histoire fermée de H_2.

(ii) *Hypothèse d'induction :* $f(H'_1)$ est une sous-histoire fermée de H_2 pour $n \geq 2$.

(iii) *Pas d'induction :* Soient $H'_1 = [r_{i_1}; \ldots; r_{i_n}]$ et $H''_1 = [H'_1; r]$ où r est une requête et $r_{i_n} \overset{s}{\rightarrow} r$. Comme $f(r_{i_j}) \overset{s}{\rightarrow} f(r_{i_{j+1}})$ pour $j \in \{1, \ldots, n\}$ (hypothèse d'induction) et $f(r_{i_n}) \overset{s}{\rightarrow} f(r)$ (base d'induction) alors $f(H''_1)$ est une sous-histoire fermée de H_2.

■

Comme conséquence du théorème précédent, une sous-histoire fermée ne contenant que des insertions peut être mise au devant d'une histoire canonique.

Corollaire 3.1 *Soit H'_1 une sous-histoire fermée d'une histoire canonique H_1 telle que H'_1 contient seulement des requêtes d'insertion. Il doit exister une histoire canonique H_2 tel que (i) $H_1 \equiv_t H_2$, et (ii) $H_2 = [H'_2; H''_2]$ où $H'_2 = f(H'_1)$ pour toute application bijective $f : H_1 \rightarrow H_2$.* ■

Convergence

Le théorème suivant énonce que lors de la génération d'une requête il est toujours possible de reconstruire une histoire canonique.

Théorème 3.5 *Supposons que H est une histoire canonique. Soit r une requête générée sur l'état $s = Do(H, s_0)$. Il doit exister une histoire canonique H' telle que : $H' \equiv_t [H; r]$.* ■

Preuve. Nous considérons deux cas : soit r est une requête d'insertion, soit r est une requête d'effacement.

1. r est une requête d'effacement : Nous obtenons H' en ajoutant simplement r à H, *i.e.* $H' = [H; r]$.
2. r est une requête d'insertion : Comme H est canonique alors $H = [H_i; H_d]$ où H_i contient des insertions et H_d contient des effacements. En utilisant le lemme 3.11, nous pouvons permuter r avec n'importe quelle requête dans H_d et nous obtenons $[u'; H_d] \equiv_t [H_d; r]$ ($r' = ET^*(r, H_d)$). Ainsi, nous obtenons $H' = [H_i; r'; H'_d]$ telle que $H' \equiv_t [H; r]$; H' est donc canonique.

■

Le théorème suivant concerne l'intégration des requêtes distantes. Il stipule que l'intégration d'une requête sur deux sites équivalents (ayant des histoires équivalentes) préserve toujours cette équivalence entre les deux sites.

Théorème 3.6 *Tous les deux sites ont des histoires canoniques équivalentes à l'état de repos.* ■

Preuve. Soient H_1 et H_2 deux histoires canoniques de deux sites différents, telle que $H_1 \equiv_t H_2$. Soit r une requête distante à intégrer dans les deux sites. Ainsi, deux cas sont possibles :

(i) *r est indépendante :* ceci veut dire que $r.a = null$. Soir $r' = IT^*(r, H_1)$ et $r'' = IT^*(r, H_2)$. Comme $H_1 \equiv_t H_2$ alors $r' \approx r''$ selon le théorème 3.2. Ainsi, $Do([H_1; r'], s_0) = Do([H_2; r''], s_0)$.

(ii) *u dépend d'une autre requête :* ceci veut dire que $r.a \neq null$. Soit $H'_1 = [r_{i_1}; r_{i_2}; \ldots; r_{i_n}]$ une sous-histoire fermée de H_1 telle que $r_{i_n} \xrightarrow{s} r$. Notons que H'_1 contient seulement des insertions. Dans le diagramme suivant nous esquissons les histoires canoniques et équivalentes que nous pouvons construire à partir de H_1 et H_2 en utilisant des applications bijectives f, f', g_1 et g_2 :

$$\begin{array}{ccc} H_1 & \xrightarrow{f} & H_2 \\ \downarrow{\scriptstyle g_1} & & \downarrow{\scriptstyle g_2} \\ H_3 & \xrightarrow{f'} & H_4 \end{array}$$

Comme $H'_2 = f(H'_1)$ est aussi une sous-histoire fermée de H_2 alors nous en déduisons que $f(r_{i_n}) \xrightarrow{s} r$. En utilisant le corollaire 3.1, nous avons $H_3 = [H'_3; H''_3]$ tel que $H'_3 = g_1(H'_1)$. De la même manière, $H_4 = [H'_4; H''_4]$ tel que $H'_4 = g_1(H'_2)$. Il est facile de voir que $f'(g_1(H'_1)) = g_2(f(H'_1))$ et

$H'_3 = H'_4$ puisqu'elles contiennent respectivement les n premières requêtes de H_3 et H_4, et elles sont fermées. Dans ce cas, l'intégration de r consiste à la transformer par rapport à H''_3 et H''_4. Soit $r' = IT^*(r, H''_3)$ et $r'' = IT^*(r, H''_4)$. Puisque $H''_3 \equiv_t H''_4$ alors $r' \approx r''$ selon le théorème 3.2. Ainsi, $Do([H_3; r'], s_0) = Do([H_4; r''], s_0)$.

■

3.5 Etude Comparative

Depuis que le problème du "$TP2$ puzzle" a été découvert, plusieurs travaux de recherche ont vu le jour pour tenter de résoudre ce problème. Nous scindons ces travaux en deux catégories.

La première catégorie essaye d'éviter le scénario du "$TP2$ puzzle". Ceci est possible en posant des contraintes sur l'ordre des communications entre les différentes copies d'un objet collaboratif, et ce afin de réduire l'espace des exécutions possibles. A titre d'exemple, SOCT4 [89] utilise un estampilleur pour forcer un ordre total sur les mises-à-jours. Cet ordre global peut être également obtenu en utilisant un procédé undo/do/redo comme dans GOT [85]. Néanmoins de telles solutions ne peuvent pas passer à l'échelle puisque elles se basent sur des solutions client-serveur. Par exemple, SOCT4 requiert un site central pour héberger l'estampilleur.

La seconde catégorie traite de la résolution du "$TP2$ puzzle". Dans ce cas, les mises-à-jours concurrentes peuvent s'exécuter dans n'importe quel ordre (pas d'ordre global). Cependant, les algorithmes de transformation doivent satisfaire la condition $TP2$. Dans cette catégorie, nous pouvons citer aDOPTed [73], SOCT2 [81], GOTO [84] et SDT [50; 51]. Malheureusement, nous avons montré dans le chapitre 1 que les algorithmes de transformation de ces systèmes échouent à satisfaire $TP2$.

Une première réflexion a été menée dans LBT [54] pour construire un ordre total sur tous les caractères manipulés durant une session de collaboration. Un tel ordre nécessite une identification unique de chaque caractère. Pour assurer la convergence sur tous les sites, l'intégration d'une mise-à-jour distante consiste à déduire l'ordre total entre les caractères. Pour ce faire, les auteurs font appel à (i) une table de hashage contenant toutes les relations entre les différents caractères ; (ii) une relation causale entre les mises-à-jours implantée par des vecteurs d'état ; (iii) des transformations bidirectionnelles (exclusive et inclusive) ; (iv) des chemins de transformation particuliers comme ceux que nous utilisons dans `OPTIC` à savoir des histoires contenant les insertions avant les effacements. L'approche s'avère à la fois très compliquée vu le nombre de moyens déployés (table de hashage, vecteurs d'état, transformation, ...) pour assurer la convergence, et moins pratique même pour des éditeurs collaboratifs ayant un nombre de participants petit. Par ailleurs, LBT ne passe pas à l'échelle puisque l'approche elle-même est tributaire d'un nombre fixe de sites et ce à cause de l'utilisation des vecteurs d'état.

Dans ABT [53; 52], les auteurs ont pallié les inconvénients de LBT en utilisant peu de moyens pour assurer la convergence. Chaque site dispose d'une

histoire $H = [H_i; H_d]$ dans laquelle les insertions (H_i) devancent les effacements (H_d). La relation causale est implantée par des vecteurs d'état. Avant de diffuser une opération locale *op*, les effets de H_d sont exclus de *op* en utilisant une transformation exclusive. Par contre, l'intégration d'une opération distante *op* nécessite la réorganisation de H_i en $H = [H_{ih}; H_{ic}]$ de telle façon que H_{ih} contient les opérations précédant causalement *op* et H_{ic} contient les opérations concurrentes à *op*. Ainsi, l'exécution de *op* requiert seulement l'inclusion des effets de $H = [H_{ic}; H_d]$.

Il y a une similarité entre ABT et `OPTIC` quant à l'utilisation des histoires canoniques. Néanmoins, `OPTIC` confère un degré de concurrence plus important que celui de ABT. En effet, nous propageons une opération locale avec un contexte de génération minimal en excluant autant que possible toutes les opérations de H. Quant à l'intégration d'une opération distante *op*, elle ne nécessite pas une réorganisation de H. Elle inclut directement dans *op* les effets des opérations qui suivent l'opération qui précède causalement *op*. De surcroît, grâce à la relation de dépendance que nous avons définie entre les insertions et les effacements, nous nous sommes déchargés d'utiliser les vecteurs d'état. En conséquence, contrairement à ABT, `OPTIC` peut facilement passer à l'échelle et être implanté sur un réseau pair-à-pair.

3.6 Conclusion

Dans ce chapitre, nous avons présenté un nouvel environnement pour l'édition collaboratif basé sur la réplication et la transformation des opérations. Nous permettons un accès simultané aux données ainsi qu'une restauration automatique de la cohérence des données grâce aux algorithmes de transformation. Une causalité minimale entre les opérations est mise en oeuvre par le biais d'une relation de dépendance basée sur la sémantique de l'objet collaboratif. Ce qui nous décharge d'utiliser les vecteurs d'état, permettant donc que le nombre des sites soit variable pendant les sessions de collaboration. Ainsi, nous pourrons déployer notre environnement sur un réseau pair-à-pair.

En perspective, nous envisagerons de traiter du problème de l'annulation des opérations (UNDO) [72; 83; 29]. Nous projetons également de réutiliser cette approche pour réaliser un éditeur Wiki [15].

Conclusion Générale

Bilan

Les réseaux Pair-à-Pair (P2P) se caractérisent par (i) une réplication massive des données ; (ii) Une topologie variable du réseau (un site peut joindre ou quitter le réseau à tout moment) ; et, (iii) L'absence de la notion client/serveur (un site peut être à la fois client et serveur). Ainsi, il ressort de cette thèse que l'édition collaborative sur un réseau P2P doit traiter deux problèmes majeurs : la convergence des copies et le passage à l'échelle.

En ce qui concerne la convergence des copies, nous avons utilisé l'approche des transformées opérationnelles. Cette approche est utilisée pour sérialiser par transformation des opérations concurrentes. Chaque objet collaboratif doit avoir son propre algorithme de transformation. Pour garantir la convergence des copies, un algorithme de transformation doit satisfaire deux conditions $TP1$ et $TP2$. L'écriture d'un algorithme de transformation devient donc une tâche ardue puisqu'elle nécessite des vérifications qui sont très difficiles (voire même impossibles) à effectuer à la main.

Aussi, nous avons proposé une méthodologie formelle pour spécifier et vérifier des objets collaboratifs synchronisés par une transformation opérationnelle. Cette méthodologie est basée sur des techniques avancées de déduction automatique, et dont l'exploitation a été conséquente. En effet, nous avons détecté des situations de divergence de copies dans plusieurs éditeurs collaboratifs connus dans la littérature. Nous avons également proposé une méthode compositionnelle qui permet de concevoir un algorithme de transformation pour un objet composite en réutilisant les algorithmes de ses objets composants.

La conception d'un algorithme de transformation pour des objets linéaires (tels que la liste, le texte, etc) reste toujours un problème ouvert quant à la satisfaction de $TP2$. Théoriquement, l'acquisition d'une telle condition va assurer la convergence dans un contexte où les collaborateurs peuvent échanger et intégrer les opérations sans la moindre contrainte. Or, à notre connaissance, il n'existe aucun algorithme de transformation qui vérifie $TP2$. Aussi, nous avons proposé un nouvel algorithme de transformation basé sur une forme affaiblie de $TP2$. Cet algorithme garantit la convergence pour une classe particulière de séquences d'opérations.

Les vecteurs d'état sont très utilisés dans les systèmes collaboratifs pour détecter la concurrence entre les opérations [26; 73; 81; 85]. Cependant, la taille d'un vecteur d'état est fixe et proportionnelle au nombre de sites. Il n'est donc pas adapté aux réseaux P2P, puisqu'il ne permet pas le passage à l'échelle. Pour nous affranchir de l'utilisation des vecteurs d'état, nous avons défini une relation minimale de précédence entre les opérations d'insertion et d'effacement. Cette relation est basée sur la sémantique de l'objet collaboratif.

Tous les résultats mentionnés dans les paragraphes précédents ont permis de réunir les conditions nécessaires pour faire de l'édition collaborative sur un réseau P2P. Nous avons donc conçu un nouvel environnement qui supporte un travail collaboratif sans contrainte et des mises-à-jour simultanées des données partagées avec une convergence des copies automatique grâce à la transformation des opérations. Cet environnement peut également passer à l'échelle.

Perspectives

A l'issue de cette thèse, plusieurs points s'avèrent intéressants à explorer et à développer :

Convergence sémantique. Comme nous l'avons souligné dans cette thèse, la réplication peut entraîner après synchronisation une divergence de données. La convergence des copies peut donc se présenter selon deux formes : (i) la convergence *syntaxique* signifie que les états des copies sont identiques mais leur contenu peut être dépourvu de sens pour la collaboration ; (ii) la convergence *sémantique* signifie que les états sont identiques et significatifs. Pour comprendre les deux types de convergence, considérons l'exemple suivant : deux utilisateurs corrigent de manière simultanée le mot partagé $''vill''$. Le premier utilisateur insère à la fin du mot le caractère $'e'$ pour obtenir $''ville''$. Quant au deuxième utilisateur, il ajoute à la fin du mot le caractère $'a'$ pour produire $''villa''$. Après réconciliation des modifications, les deux utilisateurs vont converger vers soit $''vallae''$, soit $''vallea''$. Il s'agit en l'occurrence d'une convergence purement syntaxique puisque les deux mots n'ont pas de sens en français. Une convergence sémantique préconiserait soit $''villa''$, soit $''ville''$. A notre connaissance, il n'y aucun système collaboratif qui résout de tels problèmes. Nous pensons qu'il serait intéressant de traiter la convergence sémantique en amont de la convergence syntaxique et de la considérer comme un problème de consensus [57].

Synchroniseur de données. Le travail mobile a démocratisé l'utilisation des synchroniseurs de données, comme Unison [67]. Ainsi, un utilisateur peut manipuler des copies des ses propres objets (agenda, fichier) sur des terminaux différents (poste fixe, portable, PDA, etc) et à des instants différents, avec cependant un risque de divergence de ces copies en cas de mises-à-jour multiples. Les synchroniseurs existants sont limités car soit ils imposent une synchronisation ne dépassant pas deux copies à la fois, soit ils nécessitent un serveur central. Il serait intéressant de réutiliser notre environnement collaboratif pour lever de telles limitations.

Annulation. Dans les éditeurs collaboratifs, la possibilité d'annuler des opérations est souhaitable car elle permet à l'utilisateur de corriger ses éventuelles erreurs [73; 29; 88]. Aussi, il serait intéressant de pourvoir notre environnement d'une fonction d'annulation.

Annexe

Preuve de terminaison

En utilisant l'outil AProVE, nous avons montré la terminaison du système d'équations suivant :

$$
\begin{aligned}
IT^*(h, \Lambda) &= h && (3.1)\\
IT^*(\Lambda, h) &= \Lambda && (3.2)\\
IT^*(h_1, [h_2; h_3]) &= IT^*(IT^*(h_1, h_2), h_3) && (3.3)\\
IT^*([h_1; h_2], h_3) &= [IT^*(h_1, h_3)\,;\, IT^*(h_2, IT^*(h_3, h_1))] && (3.4)\\
IT^*([op_1], [op_2]) &= [IT(op_1, op_2)] && (3.5)
\end{aligned}
$$

Voici la trace donnée par AProVE :

```
Term Rewriting System R:

   [h, h1, h2, h3, op1, op2]
   ite(h, V) -> h
   ite(V, h) -> V
   ite(h1, f(h2, h3)) -> ite(ite(h1, h2), h3)
   ite(f(h1, h2), h3) -> f(ite(h1, h3), ite(h2, ite(h3, h1)))
   ite(g(op1), g(op2)) -> g(it(op1, op2))

Termination of R to be shown.




   R     ->Dependency Pair Analysis



R contains the following Dependency Pairs:


   ITE(h1, f(h2, h3)) -> ITE(ite(h1, h2), h3)
   ITE(h1, f(h2, h3)) -> ITE(h1, h2)
   ITE(f(h1, h2), h3) -> ITE(h1, h3)
   ITE(f(h1, h2), h3) -> ITE(h2, ite(h3, h1))
   ITE(f(h1, h2), h3) -> ITE(h3, h1)

Furthermore, R contains one SCC.



   R     ->DPs

       ->DP Problem 1
         ->Negative Polynomial Order
```

```
Dependency Pairs:
ITE(f(h1, h2), h3) -> ITE(h3, h1)
ITE(f(h1, h2), h3) -> ITE(h2, ite(h3, h1))
ITE(f(h1, h2), h3) -> ITE(h1, h3)
ITE(h1, f(h2, h3)) -> ITE(h1, h2)
ITE(h1, f(h2, h3)) -> ITE(ite(h1, h2), h3)
Rules:

   ite(h, V) -> h
   ite(V, h) -> V
   ite(h1, f(h2, h3)) -> ite(ite(h1, h2), h3)
   ite(f(h1, h2), h3) -> f(ite(h1, h3), ite(h2, ite(h3, h1)))
   ite(g(op1), g(op2)) -> g(it(op1, op2))



The following Dependency Pairs can be strictly oriented using the given order.


   ITE(f(h1, h2), h3) -> ITE(h3, h1)
   ITE(f(h1, h2), h3) -> ITE(h2, ite(h3, h1))
   ITE(f(h1, h2), h3) -> ITE(h1, h3)
   ITE(h1, f(h2, h3)) -> ITE(h1, h2)
   ITE(h1, f(h2, h3)) -> ITE(ite(h1, h2), h3)


Moreover, the following usable rules (regarding the implicit AFS) are oriented.

   ite(h, V) -> h
   ite(V, h) -> V
   ite(h1, f(h2, h3)) -> ite(ite(h1, h2), h3)
   ite(f(h1, h2), h3) -> f(ite(h1, h3), ite(h2, ite(h3, h1)))
   ite(g(op1), g(op2)) -> g(it(op1, op2))


Used ordering:
Polynomial Order with Interpretation:

POL( ITE(x_1, x_2) ) = x_1 + x_2

POL( f(x_1, x_2) ) = x_1 + x_2 + 1

POL( ite(x_1, x_2) ) = x_1

POL( V ) = 0

POL( g(x_1) ) = 0


This results in one new DP problem.
```

```
   R     ->DPs

       ->DP Problem 1
         ->Neg POLO

           ->DP Problem 2
             ->Dependency Graph


Dependency Pair:

Rules:

   ite(h, V) -> h
   ite(V, h) -> V
   ite(h1, f(h2, h3)) -> ite(ite(h1, h2), h3)
   ite(f(h1, h2), h3) -> f(ite(h1, h3), ite(h2, ite(h3, h1)))
   ite(g(op1), g(op2)) -> g(it(op1, op2))



Using the Dependency Graph resulted in no new DP problems.

Termination of R successfully shown.

Duration:
0:00 minutes
```

Liste des figures

Liste des tableaux

Liste des algorithmes

Bibliographie

[1] Microsoft ActiveSync.
http ://www.microsoft.com/windowsmobile/activesync/default.mspx. Octobre 2006.

[2] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4) :335–371, 2004.

[3] Alessandro Armando, Michaël Rusinowitch, and Sorin Stratulat. Incorporating decision procedures in implicit induction. *Journal of Symbolic Computation*, 34(4) :241–258, 2001.

[4] Gilles Barthe and Sorin Stratulat. Validation of the javacard platform with implicit induction techniques. In *RTA*, pages 337–351, 2003.

[5] Narjes Ben Rajeb. *Preuves par induction implicite : cas des théories associatives-commutatives et observationnelles*. Thèse de doctorat, Université Henri Poincaré, Nancy, mai 1997.

[6] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[7] Michel Bidoit and Rolf Hennicker. Behavioural theories and the proof of behavioural properties. *Theor. Comput. Sci.*, 165(1) :3–55, 1996.

[8] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated Mathematical Induction. *Journal of Logic and Computation*, 5(5) :631–668, 1995.

[9] Adel Bouhoula. *Preuves Automatiques par Récurrence dans les Théories Conditionnelles*. Thèse de doctorat, Université Henri Poincaré, Nancy, mars 1994.

[10] Adel Bouhoula. Using induction and rewriting to verify and complete parameterized specifications. *Theor. Comput. Sci.*, 170(1-2) :245–276, 1996.

[11] Adel Bouhoula and Michael Rusinowitch. Observational proofs by rewriting. *Theoretical Computer Science*, 275(1–2) :675–698, 2002.

[12] Per Cederqvist. *Version Management with CVS*. Network Theory Ltd., décembre 2002.

[13] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP*, pages 44–57, 1993.

[14] Allison Colin. Concurrency control for real time groupware. In *Proceedings of the Concurrent Engineering : Research and Applications. A Global perspectives - CE'94*, pages 163–170, Pittsburgh, Pennsylvania, USA, august 1994.

[15] Ward Cunninghan. Wikiwikiweb history. `http://c2.com/wiki?WikiHistory`, 2006.

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

[17] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. Generalizing operational transformation to the standard general markup language. In *CSCW '02 : Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 58–67, New York, NY, USA, 2002. ACM Press.

[18] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms : Taxonomy and survey. *ACM Comput. Surv.*, 36(4) :372–421, 2004.

[19] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, pages 1–12, 1987.

[20] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17 :279–301, 1982.

[21] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, pages 243–320. 1990.

[22] Razvan Diaconescu. Behavioural specification for hierarchical object composition. *Theor. Comput. Sci.*, 343(3) :305–331, 2005.

[23] Razvan Diaconescu and Kokichi Futatsugi. Logical foundations of cafeobj. *Theor. Comput. Sci.*, 285(2) :289–318, 2002.

[24] Razvan Diaconescu, Kokichi Futatsugi, and Shusaku Iida. Component-based algebraic specification and verification in cafeobj. In *World Congress on Formal Methods*, pages 1644–1663, 1999.

[25] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer-Verlag New York, Inc., 1985.

[26] Clarence A. Ellis and Simon J. Gibbs. Concurrency Control in Groupware Systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[27] Clarence A. Ellis, Simon J. Gibbs, and Gail L. Rein. Groupware : Some issues and experiences. *Commun. ACM*, 34(1) :39–58, 1991.

[28] Patrick Th. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5) :60–67, 2004.

[29] Jean Ferrié, Nicolas Vidot, and Michelle Cart. Concurrent undo operations in collaborative environments using operational transformation. In *CoopIS/DOA/ODBASE (1)*, pages 155–173, 2004.

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

[31] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Automatic termination proofs in the dependency pair framework. In *IJCAR*, pages 281–286, 2006.

[32] Joseph Goguen and Razvan Diaconsecu. Towards an algebraic semantics for the object paradigm. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, pages 1–29, 1994.

[33] Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1) :55–101, 2000.

[34] Joseph A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1) :49–67, 1991.

[35] P. Krishna Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP*, pages 314–329, 2003.

[36] Palm HotSync. http ://www.palmone.com/us/support/hotsync.html. Octobre 2006.

[37] Claudia-Lavinia Ignat and Moira C. Norrie. Codoc : Multi-mode collaboration over documents. In *CAiSE*, pages 580–594, 2004.

[38] Abdessamad Imine. Component-based Specification of Collaborative Objects. In *The Second International Workshop on Views On Designing Complex Architectures (VODCA)*, Bertinoro, Italy, September 2006.

[39] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving Correctness of Transformation Functions in Real-Time Groupware. In *in 8th European Conference of Computer-supported Cooperative Work*, Helsinki, Finland, 14.-18. September 2003.

[40] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Deductive verification of distributed groupware systems. In *AMAST*, pages 226–240, 2004.

[41] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Towards synchronizing linear collaborative objects with operational transformation. In *FORTE*, pages 411–427, 2005.

[42] Abdessamad Imine, Pascal Molli, Gérald Oster, and Pascal Urso. Vote : Group editors analyzing tool. In Ingo Dahn and Laurent Vigneron, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[43] Abdessamad Imine, Michael Rusinowitch, Gérald Oster, and Pascal Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 351(2) :167–183, 2006.

[44] Abessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Achieving convergence with operational transformation in distributed groupware systems. Research Report RR-5188, INRIA, 2004.

[45] Apple iSync. more ways to sync your digital life. http ://www.apple.com/macosx/features/isync/. Octobre 2006.

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

[47] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[48] Emmanuel Kounalis and Michaël Rusinowitch. On word problems in horn theories. *J. Symb. Comput.*, 11(1/2) :113–127, 1991.

[49] Harry R. Lewis, Christos H. Papadimitriou, and Christos Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, 1997.

[50] Du Li and Rui Li. Ensuring Content Intention Consistency in Real-Time Group Editors. In *the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, March 2004. IEEE Computer Society.

[51] Du Li and Rui Li. Preserving operation effects relation in group editors. In *CSCW*, pages 457–466, 2004.

[52] Du Li and Rui Li. A lightweight approach to operational transformation based concurrency control. Technical report, Center for the Study of Digital Libraries (CSDL), Texas A&M University, 2006.

[53] Rui Li and Du Li. Commutativity-based concurrency control in groupware. In *the First IEEE International Conference on Collaborative Computing : Networking, Applications and Worksharing.(CollaborateCom'05)*, 2005.

[54] Rui Li and Du Li. A landmark-based transformation approach to concurrency control in group editors. In *GROUP*, pages 284–293, 2005.

[55] Rui Li, Du Li, and Chengzheng Sun. A time interval based consistency control algorithm for interactive groupware applications. In *ICPADS*, pages 429–436, 2004.

[56] Brad Lushman and Gordon V. Cormack. Proof of correctness of ressel's adopted algorithm. *Information Processing Letters*, 86(6) :303–310, 2003.

[57] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[58] Olav Lysne. Proof by consistency in constructive systems with final algebra semantics. In *ALP*, pages 276–290, 1992.

[59] Friedemann Mattern. Virtual time and global states of distributed systems. In Michel Cosnard et al., editor, *Proceedings of the International Workshop*

on Parallel and Distributed Algorithms, pages 215–226, Château de Bonas, France, october 1989. Elsevier Science Publishers B. V.

[60] Pascal Molli, Gérald Oster, Hala Skaf-Molli, and Abdessamad Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, pages 212–220. ACM Press, 2003.

[61] Pascal Molli, Hala Skaf-Molli, Gérald Oster, and Sébastien Jourdain. SAMS : Synchronous, Asynchronous, Multi-Synchronous Environments. In *The Seventh International Conference on CSCW in Design*, Rio de Janeiro, Brazil, September 2002.

[62] JeanFrançois Monin. *Comprendre les méthodes formelles. Panorama et outils logiques.* Masson, Paris, 1996.

[63] M. H. A. Newman. On theories with combinatorial definition of equivalence. *Annals of Mathematics*, 43(2) :223–243, 1942.

[64] Gérald Oster. *Réplication optimiste et cohérence des données dans les environnements collaboratifs répartis.* Thèse de doctorat, Université Henri Poincaré, Nancy, Novembre 2005.

[65] P. Padawitz. *Computing in Horn Clause Theories.* Springer, Berlin, Heidelberg, 1988.

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

[67] Benjamin C. Pierce and 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.

[68] Atul Prakash and Michael J. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4) :295–330, 1994.

[69] Atul Prakash and Michael J. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4) :295–330, 1994.

[70] Atul Prakash and Hyong Sop Shim. Distview : Support for building efficient collaborative applications using replicated objects. In *CSCW*, pages 153–164, 1994.

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

[72] Matthias Ressel and Rul Gunzenhäuser. Reducing the problems of group undo. In *GROUP*, pages 131–139, 1999.

[73] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhauser. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'96)*, pages 288–297, Boston, Massachusetts, USA, November 1996.

[74] Grigore Rosu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.

[75] M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical Verification of an Ideal ABR Conformance Algorithm. *Journal of Automated Reasoning*, 30(2) :153–177, February 2003.

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

[77] R. Salz. Internetnews : Usenet transport for internet sites. In *Proc. of the Summer 1992 USENIX Conference*, pages 93–98, San Antionio, Texas, 1992.

[78] Marc Shapiro, Nuno M. Preguiça, and James O'Brien. Rufis : Mobile data sharing using a generic constraint-oriented reconciler. In *Proceedings of 5th IEEE International Conference on Mobile Data Management - MDM'04*, pages 146–151. IEEE Computer Society, january 2004.

[79] Haifeng Shen and Chengzheng Sun. Flexible notification for collaborative systems. In *CSCW*, pages 77–86, 2002.

[80] Maher Suleiman. Sérialisation des opérations concurrentes dans les systèmes collaboratifs répartis. In *Thèse de Doctorat de l'Université de Montpellier II*, 1998.

[81] Maher Suleiman, Michèle Cart, and Jean Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work : The Integration Challenge (GROUP'97)*, pages 435–445. ACM Press, November 1997.

[82] Maher Suleiman, Michèle Cart, and Jean Ferrié. Concurrent Operations in a Distributed and Mobile Collaborative Environment. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 36–45. IEEE Computer Society, 1998.

[83] Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Hum. Interact.*, 9(4) :309–361, 2002.

[84] Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors : Issues, algorithms, and achievements. In *CSCW*, pages 59–68, 1998.

[85] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving Convergence, Causality-preservation and Intention-preservation in real-time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1) :63–108, March 1998.

[86] David Sun, Steven Xia, Chengzheng Sun, and David Chen. Operational transformation for collaborative word processing. In *CSCW*, pages 437–446, 2004.

[87] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP'95*, pages 172–182. ACM Press, 1995.

[88] Nicolas Vidot. Convergence des copies dans un environnements collaboratifs répartis. In *Thèse de Doctorat de l'Université de Montpellier II*, 2002.

[89] Nicolas Vidot, Michèle Cart, Jean Ferrié, and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'00)*, Philadelphia, Pennsylvania, USA, December 2000.

[90] Martin Wirsing. Algebraic Specification. *Handbook of theoretical computer science (vol. B) : formal models and semantics*, pages 675–788, 1990.

[91] N. H. Xuong. *Mathématiques Discrètes et Informatique.* Masson, 1992.

Résumé

Les systèmes d'édition collaborative permettent la manipulation d'objets, comme des documents texte, par plusieurs personnes qui sont réparties dan le temps et dans l'espace. La réplication des objets est indispensable dans de tels systèmes mais elle peut entraîner un problème de divergence de données. Pour pallier ce problème, une approche optimiste, appelée *transformation opérationnelle*, est utilisée dans ce domaine. L'objectif de cette thèse est de proposer un cadre formel pour la conception d'algorithmes de transformation corrects qui peuvent être déployés sur des systèmes d'édition collaborative en vue de garantir la convergence des données. Dans un premier temps, nous avons proposé une méthodologie formelle pour spécifier et vérifier des objets collaboratifs synchronisés par une transformation opérationnelle. Cette méthodologie repose sur l'utilisation de techniques avancées de déduction automatique. Son exploitation était conséquente puisqu'elle nous a permis de détecter des situations de divergence dans des systèmes collaboratifs bien connus dans la littérature. Assurer La convergence pour des objets linéaires (tels que la liste, le texte, l'arbre ordonné XML, etc) demeure toujours un défi. A ce titre, nous avons constaté que les conditions de convergence connues dans la littérature sont très difficiles à satisfaire. Aussi, nous avons proposé un nouvel algorithme de transformation basé sur une forme relaxée de ces conditions. Comme complément à cet algorithme de transformation, nous avons conçu un environnement d'intégration pour l'édition collaborative basée sur des objets linéaires. L'originalité de cet environnement est le fait qu'il peut être déployé sur un réseaux pair-à-pair (P2P). Enfin, nous avons proposé une technique permettant la composition d'objets simples pour former des objets complexes tout en préservant des critères de convergence imposés sur les algorithmes de transformation.

Mots-clés: Méthodes Formelles, Réplication Optimiste, Edition Collaborative, Convergence des données, Transformées Opérationnelles

Abstract

Collaborative editing systems provide computer support for manipulating objects such as a text document, shared by two or more users that are temporally and spatially distributed. Object replication is essential in such systems, but it can leads to a data divergence problem. To overcome this problem, an optimistic approach, called the *operational transformation*, is used. This thesis is aimed to propose a formal framework for designing correct transformation algorithms that can be embedded in collaborative editing systems for achieving data convergence. Firstly, we have proposed a formal methodology for specifying and verifying collaborative objects synchronized by operational transformation approach. This methodology relies on using advanced automated deduction techniques. Thanks to our formal framework, we have detected divergence situations in many well-known systems. Ensuring convergence for linear objects (such as a list, a text, an ordered XML tree) still remains challenging. In this respect, we have noticed that the known convergence conditions are hardly to be satisfied. So, we have proposed a new tranformation algorithm based on relaxed form of these conditions. Moreover, we have designed an integration environment for collaborative edition based on linear objects. The novelty of this environment is that it can be deployed in peer-to-peer networks (P2P). Lastly, we have proposed a compositional technique enabling construction of complex objects from primitive objects by preserving convergence criterias imposed on transformation algorithms.

Keywords: Formal Methods, Optimistic Replication, Collaborative Editing, Data Convergence, Operational Transformation.