HAL
open science

Peer to Peer Optimistic Collaborative Editing on XML-like trees

Denis Lugiez, Stéphane Martin

To cite this version:

Denis Lugiez, Stéphane Martin. Peer to Peer Optimistic Collaborative Editing on XML-like trees. 2008. hal-00343484

HAL Id: hal-00343484
https://hal.science/hal-00343484

Preprint submitted on 27 Jan 2009

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

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

Peer to Peer Optimistic Collaborative Editing on XML-like Trees

Denis Lugiez and Stéphane Martin

Laboratoire d'Informatique Fondamentale 39 rue F. Joliot-Curie, 13013 Marseille, France { denis.lugiez, stephane.martin}@lif.univ-mrs.fr

Abstract. Collaborative editing consists in editing a common document shared by several independent sites. Conflicts occurs when different users perform simultaneous uncompatible operations. Centralized systems solve this problem by using locks that prevent some modifications to occur and leave the resolution of conflicts to users. Optimistic peer to peer (P2P) editing doesn't allow locks and uses a Integration Transformation IT that reconciliates conflicting operations and ensures convergence (all copies are identical on each site). Two properties TP1 and TP2, relating the set of allowed operations Op and the transformation IT, have been shown to ensure convergence. The choice of the set Op is crucial to define an integration operation that satisfies TP1 and TP2. Many existing algorithms don't satisfy these properties and are incorrect. No algorithm enjoying both properties is known for strings and little work has been done for XML trees in a pure P2P framework. We focus on editing XML-like trees, i.e. unranked-unordered labeled trees also considered in the Harmony project. We show that no transformation satisfying TP1 and TP2 exists for a first set of operations but that TP1 and TP2 hold for a richer set of operations, provided that some decoration is added to the tree. We show how to combine our approach with any convergent editing process on strings to get a convergent process. We have implemented our transformation using a P2P algorithm inspired by Ressel et al. whose correctness relies on underlying partial order structure generated by the dependence relation on operations.
Keywords: Peer to Peer, Concurrent Processes, Collaborative Editing, Optimistic reconciliation, XML

1 Introduction

Collaborative edition is a concurrent process that allows separate users -sites- to work on the same data called *a collaborative object* using a set of defined operations. Distinct authors working on the same article, shared calendar, on-line encyclopedia are example of such processes. This activity can be centralized by a distinguished site that coordinates and resolves the conflicts that can arise from

concurrent access to the same resource -for instance two sites want to insert two distinct character at the same position in a word-, like in the subversion system (svn). A more liberal approach relies on a peer to peer process (in short P2P) where the set of users in not fixed in advance and where no central site coordinates the process. Therefore conflict resolution is much more complex, especially when one has an optimistic approach that considers that each operation is meaningful and must be taken into account. A simpler solution that relies on priority attributed to users and undoing conflicting operations can lead to a situation where only the operations of one user are performed and all other operations are discarded, which is the opposite of a cooperative work. Therefore a main issue in collaborative edition is to ensure convergence (i.e. each user gets the same copy of the shared data) in the optimistic framework. The Integration Transformation approach uses a operator IT that combines concurrent operations to get a new operation merging the effect of these concurrent operations to resolve the conflicts. Convergence is proved when this transformation enjoys two properties TP1 and TP2. The problem is hard for linear structures like words and most algorithms proposed [10,11,4] are non-trivial. Unfortunately recent works [3] show that these algorithm don't have the convergence property. Furthermore, few results have been obtained for tree-like structures in a pure P2P optimistic framework which is the basis for collaborative edition on XML-documents (the solution in [8] uses time-stamp, i.e. a central server). In this paper we concentrate on labeled unranked-unordered trees, called XML-like trees- which are already considered in the Harmony project [9] and also provides a close approximation to XML-documents (in many applications, the ordering on siblings on XML document is not relevant). Our first results states that no IT transformation can exist for a first basic set of operations. Then we refine the data structure and we give a rich set of operations that allows to define an IT transformation satisfying TP1 and TP2. The proof has been automated with the Vote system [2] which uses Spike, a theorem prover based on term rewriting. Then we show how to combine this data structure with another date structure for which a convergent algorithm exists to get convergence for the composed data structure. This results allows collaborative

editing on a complex data structure combining a tree-like structure and other basic structure like words.

Section 2 gives the basic definitions, section 3 describes the main basic data structures words and trees. Then we give the negative results for these collaborative objects in section 4. The new tree-like collaborative object is given in section 5 as well as an integration transformation that ensures convergence. Combination of convergent algorithm are given in Section 6 and Section 7 discusses implementation issues.

2 The Framework

2.1 Collaborative Editing and Convergence

A collaborative object consists of a type (calendar, XML document,...) that defines the set of *states*, a set Op of operations and an operator Do that applies an operation op to a state s (i.e. an element of the type) to get another state $op(s)$ that is denoted by $Do(s, op)$. For instance, the collaborative object *word* consists of Σ^* the set of words on an alphabet Σ, operations *InsCh(p,c)* to insert character c at position p, and *DelCh(p)* to delete the character at position p and Do operation simply applies these deletion or insertion to the current state (which is some word). A sequence of operations is called an *history* and denoted by $[op_1; op_2; \ldots; op_n]$ and we use the notation $[op_1; op_2; \ldots; op_n](s)$ to denote $Do(\ldots, Do(Do(s, op_1), op_2), \ldots, op_n)$ (apply op_1 first, then op_2,...).

Collaborative editing is a special kind of concurrent programming on a shared collaborative object shared by distinct sites. Centralized systems like svn have a system of locks that prevent conflicts[1], but pure P2P systems have no centralization process that enforce each site to have the same data. The optimistic approach assumes that no operation is lost and the main issue is to ensure convergence, i.e. all sites eventually have the same copy of the shared object.

Requests and computations. Each site generate local requests that consists of some operation op to execute on the shared object plus additional information (site identifier, operation number, history,...).

[1] a user can be in conflict with the master copy, but conflict resolution is under user's responsibility

Each local request is broadcast to all other sites and we assume that no messages is lost and that the execution ordering doesn't exchange messages. Requests generated and received by each site are queued and extracted from the queue to be executed, i.e. the operation is performed on the current copy of the collaborative object. Local requests are linearly ordered and the execution of requests respects this ordering. Therefore requests can be causally related or concurrent (requests generated independently by distinct sites)

The causality relation and concurrent request. Let r_1^i be generated by site i and r_2^j generated by site j. The causality relation $\succ$ is defined by $r_1^i \succ r_2^j$ iff either $i = j$ and r_1^i is generated before r_2^j or $i \neq j$ and the request r_1^i is executed on site j before r_2^j. The relation $\succ$ is a partial order and we say that two requests r and r' are *concurrent*, denoted by $r \parallel r'$, iff $r \not\succ r'$ and $r' \not\succ r$. In the following, we identity a request and the operation it conveys, and we extend $\succ$ to operations.

Concurrency may lead to conflicts: For instance two distinct sites insert different characters at the same position. These conflicts are solved using a transformational approach. Assume that a site s has performed operation op and that it receives an request containing operation op' that has been issued by another site s' concurrently to op (i.e. $op \parallel op'$). Instead of executing op', the site s executes $IT(op', op)$, the transformation of operation op' according to op. Meanwhile site s', which has executed op' and receives a request to execute op will execute $IT(op, op')$.

The *convergence property* states that all sites share the same copy of the collaborative object after they have processed all requests.

2.2 The Integration Transformation and the Convergence Theorem

The Integration function IT takes two operations op_2 issued by $site_2$ and op_1 issued by $site_1$ and returns a operation $IT(op_2, op_1) \in Op$ that $site_1$ executes. Meanwhile $site_2$ executes $IT(op_1, op_2)$. This integration function IT is extended to integrate an operation with a set of concurrent operations (see [2]). The classical properties required for ensuring convergence are:

– $TP1$ property states an *equality on states*
$[op_1; IT(op_2, op_1)](t) = [op_2; IT(op_1, op_2)](t)$

$$\begin{tikzcd} & t_1 \arrow[r, "IT(op_2, op_1)"] & t'_1 \\ t \arrow[ur, "op_1"] \arrow[dr, "op_2"'] & & \\ & t_2 \arrow[r, "IT(op_1, op_2)"] & t'_2 \end{tikzcd} \qquad TP1 \text{ implies } t'_1 = t'_2$$

– $TP2$ property states an *identity of operations*:
$IT(IT(op, op_1), IT(op_2, op_1)) = IT(IT(op, op_2), IT(op_1, op_2))$

Theorem 1 ([10]). *If IT satisfies TP1 and TP2 then the convergence property holds.*

A main issue in collaborative editing is, given a collaborative object, design an IT function that satisfies TP1 and TP2. A related issue is to design the most expressive set of operations, such that there exists an IT satisfying TP1 and TP2. The larger the set of operations, the better but extending the set of operations results in a combinatorial explosion when proving TP1 and TP2. At the present time, no set of operations has been designed to handle XML-like documents in a pure P2P approach.

2.3 An Abstract Description of Editing Algorithm

Each site has a set of local variables $i, s, h, \ldots$ site identifier, current state of the shared object, history,... and an environment E is a set of values of these variables (for all sites). A request is a tuple of values $\langle i, opnb, op, \ldots \rangle$ (site identifier, operation numbering, operation,.... The set of environment is $\mathcal{E}nv$ and the set of request is $\mathcal{R}eq$.

Local transitions are described by a transition function $\varphi_l : Op \times \mathcal{E}nv \rightarrow \mathcal{E}nv$ that given an operation $op \in Op$, a current environment E computes the new environment E' corresponding to the execution of op. The request r_l sent to other sites is the value of some of the local variables. This process is described as $\varphi_l.!r_l$.

An *external request* r_e is followed by a local computation $\varphi_e : \mathcal{R}eq \times \mathcal{E}nv \rightarrow \mathcal{E}nv$ updating the local variables (using the IT function but possibly other functions). This process is described as $?r_e.\varphi_e$. A collaborative editing algorithm on a collaborative object is described by $\mathcal{E}nv, \mathcal{R}eq, \varphi_l, \varphi_e$ (assuming that transformations like IT and possibly other functions are already defined).

Each site performs a non deterministic choice between the two processes and repeats this computation until all messages are processed. A computation is a sequence of $\varphi_l(op, E).!r_l$ and $?r_e.\varphi_e(E)$ that results from an interleaving of the computations on each site respecting the causality relation.

3 Words and Tree-like Data Structures

In this section, we recall some known facts on words and set up a first approach for XML-like trees.

The collaborative object *word* is given by the set of words on a finite alphabet Σ and the operations $InsCh(p, c)$ that inserts a character $c \in \Sigma$ at position $p \in Pos$, $DelCh(p)$ that deletes the character at position $p \in Pos$ and $Nop()$ where Pos is the set of positions i.e. sequences of integers. Several Transformations IT have been defined but none satisfies both $TP1$ and $TP2$ (see section 4). Some variants of this object use slightly a more elaborated data type and operations to keep track of operations performed at a given position or for a given character.

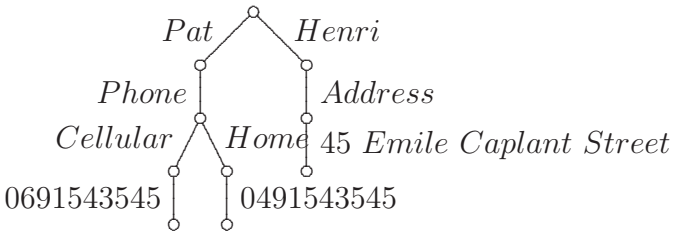
The *tree* data structure that we define is already used in the Harmony project [9]. Let $\mathcal{N}$ be a set of names, the set T of unordered unranked edge labeled trees is defined by the grammar:

$$\begin{aligned} T ::= & \{\} && //Empty\ tree \\ & \mid \{n_1(T), ..., n_m(T)\} \quad n_i \in \mathcal{N}, n_i \neq n_j \text{ if } i \neq j && //Set\ of\ tree \end{aligned}$$

The definition ensures that two edges issued from the same node have different labels: i.e. a given label occurs at most once on siblings. Trees are unordered i.e., for any permutation σ, we have that $\{n_1(t_1), \ldots, n_m(t_m)\} = \{n_{\sigma(1)}(t_{\sigma(1)}), \ldots, n_{\sigma(m)}(t_{\sigma(m)})\}$. In figures, we draw $\{\}$ as a node, and we add a root node to a tree $\{n_1(t_1), \ldots, n_m(t_m)\}$.

Example 1.

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



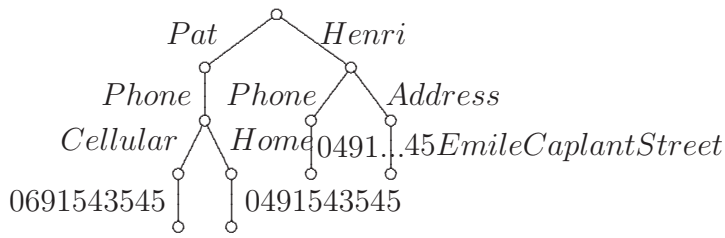
A path is a sequence of names, ϵ is the empty path and $p.p'$ is the concatenation of paths p and p'. The set of paths is written $\mathcal{P}$. The projection of tree t along a path p, written $t_{|_p}$, is defined by $t_{|_\epsilon} = t$ and $t_{|_{n.p}} = t_{|_{n\,|_p}}$, $n \in \Sigma, p \in \mathcal{P}$. We write $p_1 \lhd p_2$, when a path p_1 is a prefix of another path p_2.

The operations that we consider are:

- $Add(p, n)$: Add a edge labeled n at end of path p.

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

  Example : $t' = Add(Henri.Phone, 0491835469)(t)$



  $Add(Henri, Phone)(t') = t'$ since $Henri.Phone$ already exists.
- $Nop()$: Do nothing. $Nop()(t) = t$
- $Del_1(p, n)$: Replace a edge labeled n at end of path p by the set of its successors.

  $Del_1(n'.p, n)(t) = t$, if $n \notin Dom(t)$
  $Del_1(n_i.p, n)(\{n_1(t_1), ..., n_i(t_i), ..., n_q(t_q)\}) =$
  $\{n_1(t_1), ..., n_i(Del_1(p, n)(t_i)), ..., n_q(t_q)\}$
  $Del_1(\epsilon, n)(t) = t$, if $n \notin Dom(t)$
  $Del_1(\epsilon, n_i)(\{n_1(t_1), ..., n_i(t_i), ..., n_q(t_q)\}) = \{n_1(t_1), ..., n_q(t_q)\} \oplus t_i$

4 Negative Results for Words and Trees

The Word Case. Imine's work [2,3] contains counter-examples for the convergence property for the algorithms presented in [10,11,4] and discusses this issue. He defines a weaker property TP2' which requires the identity on states instead of operations. Then he gives an algorithm ensuring convergence relying on TP2' but this algorithm needs the reordering of histories. Therefore, we can state:

Proposition 1 ([2]). *No transformation IT for words described in the literature satisfies TP1 and TP2.*

Unordered Unranked Trees. Let $Op = \{Nop(), Add(p,n), Del_1(p,n)\}$ $p \in \mathcal{P}, n \in \Sigma$. We say that an operation $op(x_1, \ldots, x_n)$ is defined from Op iff $op(x_1, \ldots, x_n)$ is some element of Exp according to the grammar:

$$\begin{array}{l} Exp \ ::= op(y_1, \ldots, y_p) \mid \text{ if } Cond \text{ then } Exp \text{ else } Exp2 \text{ fi } op \in Op \\ Cond ::= x \bowtie y \mid Cond \wedge Cond \mid \neg Cond \end{array}$$

where $\bowtie$ denotes $=$ or $\lhd$, x, y are variables or expressions $p.n$. This grammar capture the natural definitions of any operation on trees from the basic operations of Op excepting iteration and recursion which are out of scope in our framework.

Theorem 2. *There is no definition of $IT(op_1, op_2)$ from Op such that IT satisfies $TP1$.*

We can restore $TP1$ and $TP2$ using a stronger notion of deletion (See Appendix 8.3). Let Del_2 be the operation deleting the entire subtree and let $Op' = \{Nop(), Add(p,n), Del_2(p,n)\}$ $p \in \mathcal{P}, n \in \Sigma$.

Theorem 3. *There is a IT for Op' that satisfies $TP1$ and $TP2$.*

5 Unordered Unranked Trees Revisited

In collaborative edition each site is identified by its number and numbers the operations that it performs. This ordering is linear and unambiguous. When a tree is constructed from the empty tree, one

can uniquely label each edge by the site number and the numbering of the operation that has created this edge. Since we can also add labels like those of XML-documents, we have a data structure that corresponds to unordered XML documents where the edges are labeled by an item occurring once in the tree.

5.1 The Data Structure

A identifier is either one of the reserved names *doc* (for document) or *mem* (for memory) or a pair of natural numbers $(site, nbop)$ where the *site* denotes a site number and *nbop* denotes an operation number. ID denotes the set of identifiers. A label l is an element of a set of labels Ł (for instance *section, paragraph,...*).

We consider trees defined as in section 3 on the set of names $\mathcal{N} = Ł \times ID$ assuming that each identifier occurs once in the tree.

$$T ::= \{\} \mid \{(l_1, id_1)(T), ..., (l_m, id_m)(T)\}$$

where each id_i's occurs once in the whole tree.
Example 2.

$$t = \left\{ \begin{array}{l} (Pat, 1; 1)\left(\left\{ (Phone, 2; 1)\left(\left\{ \begin{array}{l} (Home, 3; 1)(\{(0491543545, 4; 1)(\{\})\}) \\ (Cellular, 5; 1)(\{(0691543545, 6; 1)(\{\})\}) \end{array} \right\}\right)\right\}\right) \\ (Henri, 2; 2)(\{(Address, 3; 2)(\{(45\ Emile\ Caplant\ Street, 4; 2)(\{\})\})\}) \end{array} \right\}$$

From now on, $\oplus$ denotes the union of multisets. Actually, we use this operation only for disjoint sets, computing a set (not a multiset). We define two projection operations:

The projection $t_{|_{id}}$ of t in id is defined by $\{\}_{|_{id_i}} = \{\}$ and
$\{(l_1, id_1)(t_1), ..., (l_i, id_i)(t_i), ..., (l_m, id_m)(t_m)\}_{|_{id_i}} = t_i$
$\{(l_1, id_1)(t_1), ..., (l_m, id_m)(t_m)\}_{|_{id}} = t_{1|_{id}} \oplus ... \oplus t_{m|_{id}}$
and the second projection $t_{\lceil_{id_i}}$ is defined by $\{\}_{\lceil_{id_i}} = \{\}$ and
$\{(l_1, id_1)(t_1), ..., (l_i, id_i)(t_i), ..., (l_m, id_m)(t_m)\}_{\lceil_{id_i}} = \{(l_i, id_i)(t_i)\}$
$\{(l_1, id_1)(t_1), ..., (l_m, id_m)(t_m)\}_{\lceil_{id}} = \{t_{1\lceil_{id}}, ..., t_{m\lceil_{id}}\}$

Example 3. Let t be as above, then:

$$t_{|_{3;1}} = \{(0491543545, 4; 1)(\{\})\}$$
$$t_{\lceil_{3;1}} = \{(Home, 3; 1)(\{(0491543545, 4; 1)(\{\})\})\}$$

Each tree can be transformed into an (unordered) XML tree by the tree morphism defined by $\varphi(\{\}) = \{\}$ and
$\varphi(\{(l_1, id_1)(t_1), \ldots, (l_m, id_m)(t_m)\}) = \{(l_1)(\varphi(t_1)), \ldots, (l_m)(\varphi(t_m))\}$

5.2 Gluing Memory and Tree in a Single Tree

As already mentioned, the collaborative object that we use consists in two parts: one is a tree that represents the document that we edit and the other one is a memory where we keep some previous parts of the document that have been erased. The memory is needed because solving conflicts may require to fetch parts of the trees in the memory to update the document part (this comes from the move operation Mv). To get a uniform definition for operations, we represent the memory and the document in a single tree, so-called well-formed tree. A *well-formed tree* is a tree of the form $\{(\perp, data)(t_d), (\perp, mem)(t_m)\}$ where $\perp$ is some new label.

The Set of Operations Op. Firstly, we define two auxiliary functions:

- $Erase(id, t)$ deletes the node having identifier id in t.

$$
\begin{aligned}
&Erase(id, \{\}) = \{\} \\
&Erase(id, \{(l_1, id_1)(t_1), ..., (l_q, id_q)(t_q)\}) = \\
&\quad \{(l_1, id_1)(Erase(id, t_1)), ..., (l_q, id_q)(Erase(id, t_q))\} \\
&Erase(id, \{(l_1, id_1)(t_1), ..., (l_{id}, id)(t_{id}), ..., (l_q, id_q)(t_q)\}) = \\
&\quad \{(l_1, id_1)(t_1), \ldots, (l_{i-1}, id_{i-1})(t_{i-1}, (l_{i+1}, id_{i+1})(t_{i+1}), \ldots, (l_q, id_q)(t_q)\}
\end{aligned}
$$

- $AddTree(id, s, t)$ adds s under identifier id in t (performing union of s and of the subterm in t).

$$
\begin{aligned}
&AddTree(id_p, t', \{\}) = \{\} \\
&AddTree(id_p, t', \{(l_1, id_1)(t_1), ..., (l_q, id_q)(t_q)\}) = \\
&\quad \{(l_1, id_1)(AddTree(id_p, t', t_1)), ..., (l_q, id_q)(AddTree(id_p, t', t_q))\} \\
&AddTree(id, t', \{(l_1, id_1)(t_1), ..., (l_{id}, id)(t_i d), ..., (l_q, id_q)(t_q)\}) = \\
&\quad \{(l_1, id_1)(t_1), ..., (l_{id}, id)(t_{id} \oplus t'), ..., (l_q, id_q)(t_q)\}
\end{aligned}
$$

Let $Op = \{Add(id_p, n, id), Del(id), Mv(id, id_p), Ren(id, n), Nop()\}$, where $id \in ID \setminus \{data, mem\}, id_p \in ID, n \in \Sigma$ be the new set of operations.

- $Add(id_p, id)$: Add a edge labeled $noValue$ with identifier id under a node whose identifier is id_p.

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

– $Del(id)$: Delete a node id and store deleted subtree in memory.

$$Del(id)(t) = AddTree(mem, t_{|_{id}}, Erase(id, t))$$

– $Mv(id, id_p)$: Move node id under node id_p

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

– $Ren(id, n)$: Change label of node id

$$\begin{aligned}
&Ren(id, l)(\{\}) = \{\} \\
&Ren(id, l)(\{(l_1, id_1)(t_1), ..., (l_q, id_q)(t_q)\}) = \\
&\qquad \{(l_1, id_1)(Ren(id, l)(t_1)), ..., (l_q, id_q)(Ren(id, l)(t_q))\} \\
&Ren(id, l)(\{(l_1, id_1)(t_1), ..., (n', id)(t_i), ..., (l_q, id_q)(t_q)\}) = \\
&\qquad \{(l_1, id_1)(t_1), ..., (n, id)(t_i), ...(l_q, id_q)(t_q)\}
\end{aligned}$$

– $Nop()$: Do nothing. $Nop()(t) = t$

Besides basic operations for adding and deleting edges, we add two useful operations, one for renaming labels (change a `\section` to a `\subsection` for instance) and another one for moving parts of a tree (let's move the *theorem* before the *corollary* for instance). This last operation is the reason why we need a memory part in the tree.

Proposition 2. *Let t be a well-formed tree, let $op \in Op$, then $op(t)$ is a well-formed tree.*

Remark 1. By definition an identifier id is created once since it is equal to $(site, nbop)$ where site is the number of the site which has created it and $nbop$ is the numbering of the creation operation. Therefore if the edge corresponding to this identifier is created, and deleted later on, it cannot be re-created (since the numbering or the site number is different). An edge can be created at the "same" place[2], but with a different identifier.

5.3 The *IT* Transformation

Theorem 4. *The transformation IT defined in figure 1 satisfies TP1 and TP2.*

Proof. The proof relies on a highly combinatorial case analysis and was double checked using the Vote tool [2].

[2] we use the intuitive notion of *same* here

$$
\begin{aligned}
&IT(Add(id_p, id), Add(id'_p, id')) = Add(id_p, id),\\
&IT(Add(id_p, id), Del(id')) = \begin{cases} Nop()\ if\ id = id' \\ Add(mem, id)\ if\ id_p = id' \\ Add(id_p, id)\ otherwise \end{cases}\\
&IT(Del(id), Add(id'_p, id')) = Del(id)\\
&IT(Del(id), Del(id')) = Del(id)\\
&IT(Ren(id_1, l_1), Ren(id_2, l_2)) = \begin{cases} Nop()\ if\ s_2 < s_1 \wedge id_1 = id_2 \\ Ren(id_1, l_1)\ otherwise. \end{cases}\\
&IT(Ren(id_1, l_1), op) = Ren(id_1, l_1)\ if\ op \neq Ren\\
&IT(op, Ren(id_1, l_1)) = op\ if\ op \neq Ren\\
&IT(Mv(id_1, id_p), Mv(id_2, id'_p)) = \begin{cases} Nop(),\ \text{ if }\ s_2 < s_1 \wedge id_1 = id_2 \\ Mv(id_1, id_p)\ otherwise \end{cases}\\
&IT(Mv(id_1, id_p), Del(id')) = \begin{cases} Mv(id_1, mem)\ if\ id_p = id' \\ Nop()\ if\ id_1 = id' \\ Mv(id_1, id_p)\ otherwise \end{cases}\\
&IT(Mv(id_1, id_2), op) = Mv(id_1, id_2)\ if\ op \neq Mv, Del\\
&IT(op, Mv(id_1, id_2)) = op\ if\ op \neq Mv\\
&IT(op_1, Nop()) = op_1\\
&IT(Nop(), op_2) = Nop();
\end{aligned}
$$

where $id_p, id'_p \in ID, id \in ID \setminus \{data, mem\}$.

Fig. 1. The Transformation IT

6 Combining XML-like Trees and Words

Composition of Trees and Words. Let (T, Op_T, Do_T) be the collaborative object obtained from trees and the set of operations defined in section 5. Let $Dom(t)$ be the set of identifier occurring in $t \in T$. Let $Data = (D, Op_D, Do_D,)$ be another collaborative object. We assume that $d_0 \in D$ is the default initial value for elements of type D. Let $\delta : ID \rightarrow D$ be a labelling function that associates to each $id \in ID$ some element $d = \delta(id)$ of $Data$. A labeled tree is a pair t, δ and $T(D)$ denotes the set of labeled trees. For instance the labelling can associate to each identifier id a string that can be the information stored at the terminal node of the edge labeled by id, we call this data structure XML-like trees.

We define the collaborative object $T(Data)$, the trees parameterized by $Data$, as follows:

- The set of states is $T(D)$,
- The set Op of operations is composed of op_{id} for $id \in ID$, $op_{id} \in Op_D$, and op where $op \in Op_T$.

- The Do function is defined by
  $Do((t,\delta), op_{id}) = (t,\delta')$ where the labelling δ is identical to δ except that $\delta'(id) = op(\delta(id))$.
  $Do((t,\delta), op) = (t',\delta')$ where $t' = Do(t, op)$ and δ' is identical to δ except that $\delta(id) = d_0$ (the default value of D) if id is an identifier not occurring in t.

Composition of Convergent Algorithms. Let $\mathcal{A}_T$ be a convergent collaborative editing algorithm for T defined by $\mathcal{E}nv_T, \mathcal{R}eq_T, ?r_e^T.\varphi_e^T, \varphi_l^T.!r_l^T$ and let let $\mathcal{A}_D$ be a convergent collaborative editing algorithm for Data defined by $\mathcal{E}nv_D$,$\mathcal{R}eq_D$, $?r_e^D.\varphi_e^D$, $\varphi_l^D.!r_l^D$. We define a collaborative editing algorithm for $T(D)$ by composing both algorithm in a product-like way. Environments have the form $\langle E_T, E_D\rangle$ where $E_T \in \mathcal{E}nv_T$ and E_D is a partial function $ID \to \mathcal{E}nv_D$. The function is defined for $id \in Dom(s)$ where $s \in E_T$ is the state of the collaborative object. Similarly requests have the form $\langle r_T, \perp\rangle$ or $\langle \perp, r_D\rangle$ where $\perp$ stands for undefined, $r_T \in \mathcal{R}eq_T$ and r_D is a pair (id, r) with $id \in ID, r \in \mathcal{R}eq_D$. The set of environment is denoted by $\mathcal{E}nv$, the set of requests is denoted by $\mathcal{R}eq$. The composition is defined by

- Local computation $\phi_l : Op, \mathcal{E}nv \to \mathcal{E}nv$ where
  $\phi_l(op, \langle E_T, E_D\rangle) = \varphi_l^T(op, E_T)$ and $r_l = \langle r_l^T, \perp\rangle$ if $op \in Op_T$
  $\phi_l(op, \langle E_T, E_D\rangle) = \varphi_l^D(op_{id}, E_D(id))$ and $r_l = \langle \perp, (id, r_l^D)\rangle$
  if $op = op_{id} \in Op_D$
- Computation following external requests $\phi_e : \mathcal{R}eq, \mathcal{E}nv \to \mathcal{E}nv$ where $\phi_e(r_e, \langle E_T, E_D\rangle) = \varphi_e^T(r_e^T, E_T)$ if $r_e = \langle r_e^T, \perp\rangle$ and
  $\phi_e(r_e, \langle E_T, E_D\rangle) = \varphi_e^D(r_e^D, E_D(id))$ if $E = \langle E_T, E_D\rangle$, $r_e = \langle \perp, (id, r_e^D)\rangle$

The initial state is the empty tree, labeled by d_0 and the current state is the tree which is the current state s_T computed by $\mathcal{A}_T$ and for each $id \in Dom(s_T)$ the labelling is the state computed by A_D.

Theorem 5. *If A_D and A_T are convergent, then their composition is convergent.*

Let XML-like documents be labeled unranked-unordered trees decorated with strings. Since convergent algorithms for words exist (more complex than algorithms using IT, see [5,2] for instance) and since the transformation IT of section 5 is TP1 and TP2, we have:

Theorem 6. *There exists a convergent editing algorithm for XML-like documents.*

7 Algorithm and Implementation

The algorithm follows the lines given at section 2.3. It is similar to [10,7], but we replace the explicit vector dependency by sending the set of (minimal dependencies) of the operation sent by the site. This amounts to giving an slightly modified version of the translate function that computes the integration of an operation with respect to a set of dependencies. Therefore the set of sites is not fixed in advance and can evolve during the editing process. As mentioned in [7], the correctness of this algorithm relies on the partial ordering structure underlying the set of requests.

The implementation has been done in Java and performs well in practice. Examining random execution of the algorithm shows that most of the computations are implicitly independent: operations on nodes of distinct identifiers don't interfere. The operations that may cause actual conflicts are renaming of labels (on the same identifier). In many other cases, the integration $IT(op, op')$ returns op.

We plan to investigate further the algorithm and its properties to give theoretical bases for a set of optimizations that can improve its efficiency. For instance, we have proved that integrating an operation with pairwise disjoint operations always return the same operation, therefore some memoization techniques could be used to save computation time.

8 Conclusion

We have proposed a first approach to deal with XML-like trees in a P2P Collaborative Editing framework using a rich set of operations and a transformation enjoying the key properties to ensure convergence (when none of existing algorithms for words achieve this goal). We are currently investigating several issues. The first one is to deal with ordered unranked trees but, since this case contains the word case, the problem is hard and the existence of a simple integration transformation is still pending. Another issue is to deal with typing issues, where the relevant notion of type is regular tree languages for unordered-unranked tree languages (that generalizes DTD and XML-Schemas to this data-structure) like in [1]. The first results in this direction shows that requiring to use transformations that

respect types strongly restrict the class of well-typed trees. Finally, trees have a structure which is inherently concurrent (branches are independent up to their common root) and can be exploited to improve the computational aspects of our algorithm.

References

1. J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. A logic your typechecker can count on: Unordered tree types in practice. In *Workshop on Programming Language Technologies for XML (PLAN-X), informal proceedings*, January 2007.
2. A. Imine. *Conception Formelle d'Algorithmes de Réplication Optimiste. Vers l'Edition Collaborative dans les Réseaux Pair-à-Pair*. PhD thesis, Université Henri Poincaré, Nancy, déc. 2006.
3. A. Imine, M. Rusinowitch, G. Oster, and P. Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theor. Comput. Sci.*, 351(2):167–183, 2006.
4. Du Li and Rui Li. Ensuring content and intention consistency in real-time group editors. In *ICDCS*, pages 748–755. IEEE Computer Society, 2004.
5. Rui Li and Du Li. Commutativity-based concurrency control in groupware. In *CollaborateCom*. IEEE, 2005.
6. D. Lugiez and S.Martin. P2P optimistic colaborative editing on XML-like trees. http://, 2008.
7. Brad Lushman and Gordon V. Cormack. Proof of correctness of Ressel's adOPTed algorithm. *Information Processing Letter*, 2003.
8. G. Oster, H. Skaf-Molli, P. Molli, and H. Naja-Jazzar. Supporting collaborative writing of XML documents. *unpublished*, 2007.
9. B.C. Pierce, A. Schmitt, and M. B. Greenwald. Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Tech. Report MS-CIS-03-42, University of Pennsylvania, 2003. Superseded by MS-CIS-05-02.
10. M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 288–297, New York, NY, USA, 1996. ACM.
11. M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *GROUP '97: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 435–445, New York, NY, USA, 1997. ACM.

Appendix

8.1 Proof of Theorem 2

¡¡¡¡¡¡¡ main.tex

Proof. The proof is by induction on n.

- Base case $n = 1$. The result is obvious (the only substitution is the identity.
- Inductive step. We assume that for all $op, op_1, \ldots, op_{n-1}$, σ permutation of $\{1, \ldots, n-1\}$ we have $IT^*(op, [op_1, \ldots, op_{n-1}]) = IT^*(op, [op_{\sigma(1)}, \ldots, op_{\sigma(n-1)}])$.
  Let $op, op_1, \ldots, op_n \in Op$ and let σ be a permutation of $\{1, \ldots, n\}$. We distinguish several cases:
  - $\sigma(n) = n$. Then σ is a permutation of $\{1, \ldots, n-1\}$.

$$\begin{aligned}
IT^*(op, [op_{\sigma(1)}, \ldots, op_{\sigma(n)}]) &= IT^*(op, [op_{\sigma(1)}, \ldots, op_{\sigma(n-1)}, op_n]) \\
&= IT(\, IT^*(op, [op_{\sigma(1)}, \ldots, op_{\sigma(n-1)}]), \\
&\qquad IT^*(op_n, [op_{\sigma(1)}, \ldots, op_{\sigma(n-1)}]) \\
&= IT(\, IT^*(op, [op_{\sigma 1}, \ldots, op_{n-1}]), \\
&\qquad IT^*(op_n, [op_1, \ldots, op_{n-1}])) \\
&\quad (\textit{by induction hypothesis}) \\
&= IT^*(op, [op_1, \ldots, op_{n-1}, op_n])
\end{aligned}$$

  - σ exchanges n and $n-1$ and $\sigma(i) = i$ for $i \neq n, n-1$.

$$\begin{aligned}
IT^*(op, [op_{\sigma(1)}, \ldots, op_{\sigma(n)}]) &= IT^*(op, [op_1, \ldots, op_{n-2}, op_n, op_{n-1}]) \\
&= IT(\, IT^*(op, [op_1, \ldots, op_{n-2}, op_n]), \\
&\qquad IT^*(op_{n-1}, [op_1, \ldots, op_{n-2}, op_n])) \\
&= IT(\, IT\ (IT^*(op, [op_1, \ldots, op_{n-2}]), \\
&\qquad\quad IT^*(op_n, [op_1, \ldots, op_{n-2}])), \\
&\qquad IT\ (IT^*(op_{n-1}, [op_1, \ldots, op_{n-2}]), \\
&\qquad\quad IT^*(op_n, [op_1, \ldots, op_{n-2}]))) \\
&= IT(IT(op, op_1), IT(op_2, op_1) \\
&= IT(IT(op, op_2), IT(op_1, op_2)\ (\textit{by } TP2) \\
&= IT(\, IT(IT^*(op, [op_1, \ldots, op_{n-2}]), \\
&\qquad\quad IT^*(op_{n-1}, [op_1, \ldots, op_{n-2}])), \\
&\qquad IT(IT^*(op_n, [op_1, \ldots, op_{n-2}]), \\
&\qquad\quad IT^*(op_{n-1}, [op_1, \ldots, op_{n-2}])))) \\
&= IT^*(op, [op_1, \ldots, op_{n-2}, op_{n-1}, op_n])
\end{aligned}$$

- $\sigma(n) \neq n, n-1$. Then σ can be composed as three substitutions $\sigma_1, \sigma_2, \sigma_3$:
  σ_1 exchanges $n-1$ and $\sigma(n)$ and leave other element unchanged (hence $\sigma_1(n) = n$ since $\sigma(n) \neq n$). σ_2 exchanges $n-1$ and n. $\sigma_3(n) = n$ and σ_3 is such that $\sigma(i) = \sigma_3(\sigma_2(\sigma_1(i)))$.
  By the first case

  $$IT^*(op, [op_1, \ldots, op_n]) = IT^*(op, [op_{\sigma_1(1)}, \ldots, op_{\sigma_1(n)}])$$

  By the second case

  $$IT^*(op, [op_{\sigma_1(1)}, \ldots, op_{\sigma_1(n)}]) = IT^*(op, [op_{\sigma_2(\sigma_1(1))}, \ldots, op_{\sigma_2(\sigma_1(n))}])$$

  By the first case again

  $$IT^*(op, [op_{\sigma_2(\sigma_2(\sigma_1(1))}, \ldots, op_{\sigma_2(\sigma_1(n))}]) = IT^*(op, [op_{\sigma_3(\sigma_2(\sigma_2(\sigma_1(1)))}, \ldots, op_{\sigma_3(\sigma_2(\sigma_1(n)))}])$$

  Therefore

  $$IT^*(op, [op_1, \ldots, op_n]) = IT^*(op, [op_{\sigma(1)}, \ldots, op_{\sigma(n)}])$$

Proposition 3.

Proof.

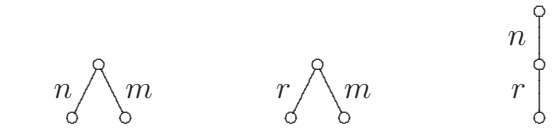
8.2 Proof of Theorem 2

======= We prove that no IT exists for our first set of operations on trees. ¿¿¿¿¿¿¿ 1.3

Proof. We assume that $TP1$ holds and we prove that $IT(op_1, op_2)$ can't be defined on an operation of Op. Let $t_1 = op_1(t)$, $t_2 = op_2(t), t_1' = op_2'(t_1)$ with $op_2' = IT(op_2, op_1)$, $t_2' = op_1'(t_2)$ with $op_1' = IT(op_1, op_2)$. We assume that $IT(op_1, op_2)$ is another operation of Op. The extension to a boolean combination of operation is straightforward.

- $op_2' = Nop()$
  - $op_1' = Nop()$: Trivial because $t_1 \neq t_2$
  - $op_1' = Add(_, _)$
    Then there is at least one more edge on t_2'.

- $op'_1 = Del(x, y)$ we get :



$(x = n, y = r)_{or}(x = \epsilon, y = n)_{or}(x = \epsilon, y = m)$

Any possible operation leaves t_2 unchanged.

In all case $t'_1 \neq t'_2$

– $op'_2 = Add(_, _)$
  - $op'_1 = Nop()$
    We have r under m on t_1 and under n on t'_2.
  - $op'_1 = Add(_, _)$
    The number of edges on t'_1 and on t'_2 are different.
  - $op'_1 = Del(_, _)$ same case

– $op'_2 = Del(_, _)$

$t'_1 = m$ or r or we return on $Nop()$ case.

  - $op'_1 = Nop()$: The number of nodes are different, therefore $t'_1 \neq t'_2$
  - $op'_1 = Add(_, _)$ idem
  - $op'_1 = Del(_, _)$ idem

□

8.3 A Stronger Deletion ensures TP1, TP2 for trees

The New Set of Operations and *IT*. Let us define a new deletion operation.

$Del_2(p, n)$: Delete the subtree accessed from the edge labeled by n at the end of path p.

$Del_2(n'.p, n)(t) = t$, if $n \notin Dom(t)$
$Del_2(n_i.p, n)(\{n_1(t_1), ..., n_i(t_i), ..., n_q(t_q)\}) =$
$\{n_1(t_1), ..., n_i(Del_2(p, n)(t_i)), ..., n_q(t_q)\}$
$Del_2(\epsilon, n)(t) = t$, if $n \notin Dom(t)$
$Del_2(\epsilon, n_i)(\{n_1(t_1), ..., n_i(t_i), ..., n_q(t_q)\}) = \{n_1(t_1), ..., n_q(t_q)\}$

Let Op be the set of operations $\{Nop(), Add(p,n), Del_2(p,n)\}$ $p \in \mathcal{P}, n \in \Sigma$ and let IT be defined by:

The IT function is defined by:

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

Proof of TP1 and TP2 with Strong Deletion

Theorem 7. *IT satisfies $TP1$ and $TP2$.*

Proving TP1 $\forall op_1, op_2 \in Op, s \in State,$
$(t)[op_1; IT(op_2, op_1)] = (t)[op_2; IT(op_1, op_2)]$

We perform a case analysis on op_1 and op_2 :

1. $op_1 = Add(p,n)$ and $op_2 = Add(p',n')$
   $(t)[Add(p,n); IT(Add(p',n'), Add(p,n))] =$
   $(t)[Add(p,n); Add(p',n')]$

   $(t)[Add(p',n'); IT(Add(p,n), Add(p',n'))] =$
   $(t)[Add(p',n'); Add(p,n)]$
   We prove :
   $Do(Do(t, Add(p,n)), Add(p',n')) =$
   $Do(Do(t, Add(p',n')), Add(p,n))$.

   We perform an induction on path length.
   (a) Empty path :
   - If $n, n' \notin Dom(t)$ and $n \neq n'$
     $Add(\epsilon, n')(Add(\epsilon, n)(\{n_1(T_1), ..., n_q(T_q)\}))$
     $= \{n_1(T_1), ..., n_q(T_q), n(\{\}), n'(\{\})\}$

$Add(\epsilon, n')(Add(\epsilon, n)(\{n_1(T_1), ..., n_q(T_q)\}))$
$= \{n_1(T_1), ..., n_q(T_q), n'(\{\}), n(\{\})\}$
which are equal.

- If $n = n'$
  We obtain : $= \{n_1(T_1), ..., n_q(T_q), n(\{\})\}$
  Because we use the third choice of function $Add(\epsilon, n)(t)$ and first operation add $n(\{\})$.
- If $n \in Dom(t)$
  We have $\{n_1(T_1), ..., n_q(T_q), n'(\{\})\}$ Third we use the second case of definition
- idem if $n' \in Dom(t)$ with n.
- If $n, n' \in Dom(t)$ the tree is unchanged.

(b) if $p.n \lhd p' : \exists p'', p.n.p'' = p$ if $n \in dom(p)$ then $Add(p, n)$ do nothing.
else
We have
$t_{|_p} = \{n_1(T_1), ..., m_1(T'_1), ...n_q(T_q)\}$ $^{(1)}$ $= \{n_1(T_1), ..., m_1(T'_1), ...n_q(T_q), n(\{\})\}$
$\{n_1(T_1), ..., m_1(T'_1), ...n_q(T_q), n(Add(p'', n')(\{\})\}$

By definition :
$Add(p', n')(t) = \{n_1(T_1), ..., m_1(T'_1), ...,$
$n_q(T_q, n(Add(p'', n')(\{\})\}$ therefor $n \in dom(t_{|_p})$ and $Add(p, n)$ do nothing.
so $^{(2)}$ $= \{n_1(T_1), ..., m_1(T'_1), ...n_q(T_q), n(Add(p'', n')(\{\})\}$

(c) idem for $p'.n' \lhd p$

(d) if $p \lhd p'$ We have : $p' = p.m_1.p_1$
We have $t_{|_p} = \{n_1(T_1), ..., m_1(T'_1), ..., n_{q-1}(T_{q-1})\}$

Two cases occurs, by recurrence definition :
$t'_{|_p} =$
$\{n_1(T_1), ..., m_1(Add(p'_1, n')(T'_1)), ..., n_{q-1}(T_{q-1}), n(\{\})\}$

(e) idem for $p' \lhd p$

(f) p, p' not empty (general case) $\exists p \in \mathcal{P} | p = p_{comon}.p'_1$ and $p' = p_{comon}.p'_2$ We have two non-empty paths then :
$p'_1 = m_1.p''_1$ and $p'_2 = m_2.p''_2$

We have
$t_{|_p} = \{n_1(T_1), ..., m_1(T'_1), ..., m_2(T'_2), ...n_{q-2}(T_{q-2})\}$

We have by definition :
$t'_{|_p} = \{n_1(T_1), ..., m_1(Add(p''_1, n)(T'_1), ...$
$, m_2, (Add(p''_2, n')(T_2)), ..., n_{q-2}(T_{q-2})\}$

2. $op_1 = Add(p, n)$ and $op_2 = Del(p', n')$
   $(t)[Add(p, n); IT(Del(p', n'), Add(p, n))]^{(1)}$
   $(t)[Del(p', n'); IT(Add(p, n), Del(p', n'))]^{(2)}$
   - $p = p'$ and $n = n'$
     $^{(1)} = (t)[Add(p, n), Del(p, n)]$
     $^{(2)} = (t)[Del(p, n; Nop()]$
     - if $n \in Dom(p)$ then $Add(p, n)(t)$ do nothing. Therefore $^{(1)} =^{(2)}$
     - if $n \notin Dom(p)$ then $Add(p, n)(t)$ create a node who delete by $Del(n, p)$ in $^{(1)}$. and $Del(n, p)$ do nothing in $^{(2)}$
       Therefore $^{(1)} =^{(2)}$
   - $p'.n' \lhd p$
     $^{(1)} = (t)[Add(p, n); Del(p', n')]$
     $^{(2)} = (t)[Del(p', n'); Nop()]$
     if p=p'.n'.p"
     We take : $t_{|_{p'}} = \{n_1(T_1), ..., n'(T), ..., n_{q-1}(T_{q-1})\}$ We have :
     $^{(1)}{}_{|_p} = \{n_1(T_1), ..., n'(Add(p'', n)(T)), ..., n_{q-2}(T_{q-2})\} = \{n_1(T_1), ..., n_{q-2}(T_{q-2})\}$
     $^{(2)}{}_{|_p} = \{n_1(T_1), ..., n_{q-2}(T_{q-2})\}$
   - else : same demo of 1f.
3. idem for $op_1 = Del(p, n)$ and $op_2 = Add(p', n')$
4. $op_1 = Del(p, n)$ and $op_2 = Del(p', n')$
   $(t)[Del(p, n); IT(Del(p', n'), Del(p, n))]^{(1)}$
   $(t)[Del(p', n'); IT(Del(p, n), Del(p', n'))]^{(2)}$
   - p.n=p'.n' :
     $^{(1)} = (t)[Del(p, n), Nop()]$
     $^{(2)} = (t)[Del(p, n), Nop()]$
   - $p.n \lhd p'$
     We have p'=p.n.p";
     We take $t_{|_p} = \{n_1(T_1), ..., n(T), ..., n_{q-1}(T_{q-1})\}$
     $^{(1)} = (t)[Del(p, n); Nop()]$
     $^{(2)} = (t)[Del(p', n'); Del(p, n)]$
     $^{(1)}{}_{|_p} = \{n_1(T_1), ..., n_{q-1}(T_{q-1})\}$
     first time : $Del(p, n)(t)_{|_p}\{n_1(T_1), ...,$
     $n(Del(p'', n')(T)), ..., n_{q-1}(T_{q-1})\}$
     therefore $^{(2)}{}_{|_p} = \{n_1(T_1), ..., n_{q-1}(T_{q-1})\}$

- idem for $p'.n' \triangleleft p'$
- else: same 1f we have two independant subtree.

5. case Nop() is trivial. $\square$

Proving TP2 $IT(IT(Op, Op_1), IT(Op_2, Op_1))^{(1)} = IT(IT(Op, Op_2), IT(Op_1, Op_2))^{(2)}$

We will explore every case :

- $Op = Add(p, n), Op_1 = Add(p_1, n_1)$ and $Op_2 = Add(p_2, n_2)$ therefore $^{(1)} = Add(p, n)$ and $^{(2)} = Add(p, n)$
- $Op = Add(p, n), Op_1 = Add(p_1, n_1)$ and $Op_2 = Del(p_2, n_2)$
  $IT(IT(Add(p, n), Add(p_1, n_1)), IT(Del(p_2, n_2), Add(p_1, n_1)))^{(1)}$
  $IT^{(b)}(IT(Add(p, n), Del(p_2, n_2),$
  $IT^{(a)}(Add(p_1, n_1), Del(p_2, n_2)))^{(2)}$
  $^{(1)} = IT(Add(p, n), Del(p_2, n_2))$
  $^{(2)} = IT^{(b)}(IT(Add(p, n), Del(p_2, n_2)), Add(X, n_1)) = IT(Add(p, n), Del(p_2, n_2))$
  or $^{(2)} = IT^{(b)}(IT(Add(p, n), Del(p_2, n_2)), Nop()) = IT(Add(p, n), Del(p_2, n_2))$
  Because $^{(a)}$ give a Add() or a Nop() the second argument of $^{(b)}$ is a *Add* or a *Nop*.
- Idem for $Op = Add(p, n), Op_1 = Del(p_1, n_1)$ and $Op_2 = Add(p_2, n_2)$
- $Op = Add(p, n), Op_1 = Del(p_1, n_1)$ and $Op_2 = Del(p_2, n_2)$
  $IT(IT(Add(p, n), Del(p_1, n_1)), IT(Del(p_2, n_2), Del(p_1, n_1)))^{(1)}$
  $IT(IT(Add(p, n), Del(p_2, n_2)), IT(Del(p_1, n_1), Del(p_2, n_2)))^{(2)}$
  - If $p_1 = p_2$ and $n_1 = n_2$
    $^{(1)} = IT(IT(Add(p, n), Del(p_1, n_1)), Nop())$
    $^{(2)} = IT(IT(Add(p, n), Del(p_1, n_1)), Nop())$
  - If $p_2.n_2 \triangleleft p_1$
    $^{(1)} = IT(IT(Add(p, n), Del(p_1, n_1)), Del(p_2, n_2)))$
    because $p_1.n_1 \neq p_2.n_2$
    $^{(2)} = IT(IT(Add(p, n), Del(p_2, n_2)), Nop())$
    * if $p_2.n_2 \triangleleft p$
      $^{(1)} = IT(IT(Add(p, n), Del(p_1, n_1)), Del(p_2, n_2)))$
      $^{(2)} = Nop()$
      · if $p_1.n_1 \triangleleft p$
        $^{(1)} = Nop()$
        $^{(2)} = Nop()$
      · else :
        $^{(1)} = IT(Add(p, n), Del(p_2, n_2)) = Nop()$
        $^{(2)} = Nop()$

    - idem for $p_1.n_1 \lhd p$
    - else :
      $^{(1)} = IT(Add(p,n), Del(p_2,n_2)) = Add(p,n)$ because
      $p_2.n_2 \not\lhd p \wedge p_1.n_1 \not\lhd p$
      $^{(2)} = IT(IT(Add(p,n), Del(p_2,n_2)), Nop()) =^{(1)}$
  - idem if $p_1.n_1 \lhd p_2$
  - Else :
    $^{(1)} = IT(IT(Add(p,n), Del(p_1,n_1)), Del(p_2,n_2)))$
    $^{(2)} = IT(IT(Add(p,n), Del(p_2,n_2)), Del(p_1,n_1)))$

    - if $p = p_1 \wedge n = n_1$
      $^{(1)} = IT(Nop(), Del(p_2,n_2))) = Nop()$
      $^{(2)} = IT(IT(Add(p_1,n_1), Del(p_2,n_2)), Del(p_1,n_1)))$
      By hypothese :
      $^{(2)} = IT(Add(p_1,n_1), Del(p_1,n_1))) = Nop()$
    - idem if $p = p_2 \wedge n = n_2$
    - if $p_1.n_1 \lhd p$ $^{(1)} = IT(Nop(), Del(p_2,n_2)))$
      $^{(2)} = IT(IT(Add(p,n), Del(p_2,n_2)), Del(p_1,n_1)))$

      We have $p_2 \neq p \vee n_2 \neq n$ and $p_2.n_2 \not\lhd p$ because
      $p_2.n_2 \lhd p \wedge p_1.n_1 \lhd p \Rightarrow p_1.n_1 \lhd p_2.n_2 \vee p_2.n_2 \lhd p_1.n_1$
      $^{(2)} = IT(Add(p,n), Del(p_1,n_1))) = Nop() =^{(1)}$
    - idem if $p_2.n_2 \lhd p$
    - else :
      $^{(1)} = IT(Add(p,n))$
      $^{(2)} = IT(Add(p,n))$
- Trivial for $Op = Del(p,n), Op_1 = Add(p_1,n_1)$ and $Op_2 = Add(p_2,n_2)$
- if $Op = Del(p,n), Op_1 = Del(p_1,n_1)$ and $Op_2 = Add(p_2,n_2)$

  $^{(1)} = IT(IT(Del(p,n), Del(p_1,n_1)),$
  $IT(Add(p_2,n_2), Del(p_1,n_1))$
  $^{(1)} = IT(Del(p,n), Del(p_1,n_1))$ because the first argument will be a 'Del' and the second will be a 'Add'.
  $^{(2)} = IT(IT(Del(p,n), Add(p_2,n_2)),$
  $IT(Del(p_1,n_1), Add(p_2,n_2))$
  $^{(2)} = IT(Del(p,n), Dell(p_1,n_1))$
- idem for $Op = Del(p,n), Op_1 = Add(p_1,n_1)$ and $Op_2 = Del(p_2,n_2)$
- if $Op = Del(p,n), Op_1 = Del(p_1,n_1)$ and $Op_2 = Del(p_2,n_2)$

- Trivial If $Op = Nop()$
- if $Op = X(n,p), Op_1 = Nop()$ and $Op_2 = X'(n_2,p_2)$
  $^{(1)} = IT(IT(X(p,n), Nop()), IT(X'(p_2,n_2), Nop())$
  $= IT(X(p,n), X'(p_2,n_2))$
  $^{(2)} = IT(IT(X(p,n), X'(p_2,n_2)), IT(Nop(), X'(p_2,n_2))$
  $= IT(X(p,n), X'(p_2,n_2))$
- idem $Op = X(n,p), Op_1 = X(p_1,n_1)$ and $Op_2 = Nop()$
- Trivial, if $Op = X(n,p), Op_1 = Nop()$ and $Op_2 = Nop()$

□

8.4 Proof of Theorem 4

The proof is similar to the previous proof and has been checked by Vote using the following specification:

```
%VOTE file for proving TP1/TP2 on XML like trees

type node(mem,data),lbl(novalue),nat;
observator
%test node existence
 bool exist(node);
%relation between son and father
 bool childof(node, node);
%returns the label of a node
 lbl getLbl(node);
auxiliary
%returns tree if there is a path between nodes
 bool childofp(node, node);

operation
%add a node n, if it doesn't exists,
%it becomes a son of p that must exist
 not(exist(n)) and exist(p) and (n!=mem) and (n!=data) : Add(node p,node n)

%delete an existing node that must be different
%from the two initial nodes mem and data
 exist(n) and (n!=mem) and (n!=data) : Del(node n);
```

```
%site t moves node n under node p if n exists and is different
%from mem and data
exist(n) and exist(p) and (n!=mem) and (n!=data) and (n != p): Move(node n,

%site t renames a node  n with  label l if n exists and is
%different from mem and data
 exist(n) and (n!=mem) and (n!=data): Ren(node n,lbl l,nat t);

transform
%definition of the IT transformation
 T(Add(p1,n1),Del(n2)) = if (p1==n2) then
return Add(mem,n1)
else
return Add(p1, n1)
endif;
T(Ren(n1,l1,s1), Del(n2)) = if (n1==n2) then
return nop
else
return Ren(n1,l1,s1)
endif;

T(Ren(n1,l1,s1),Ren(n2,l2,s2))= if (n1==n2 and s1 > s2) then
return nop
else
return Ren(n1,l1,s1)
endif;

T(Move(n1,p1,s1),Move(n2,p2,s2)) = if (n1==n2 and s1 > s2) then
return nop
else
return Move(n1,p1,s1)
endif;
T(Move(n1,p1,s1),Del(n2)) = if(n1==n2) then
return nop
elseif (p1 == n2) then
return Move(n1,mem,s1)
else
return Move(n1,p1,s1)
```

```
endif;
definition
exist'(n1)/Add(p2,n2) = if (n1 == n2) then return true
elseif (n1==mem or n1==data) then
return true
else return exist(n1)
endif;
exist'(n1)/Del(n2) =  if(n1==mem or n1==data) then
return true
elseif (n1 == n2) then return false
  else return exist(n1)
endif;
childof'(n1,p1)/Add(p2,n2) = if (n1 == n2 and p2==p1) then return true
else return childof(n1,p1)
endif;
childof'(n1,p1)/Del(n2) = if (n2 == n1) then
return false
elseif (n2==p1) then
return false
elseif (p1==mem and childof(n1,n2)) then
return true
  else
return childof(n1,p1)
endif;

childof'(n1,p1)/Move(n2,p2,s1) = if(n1 == n2 and p1==p2) then
return true
elseif (n1==n2 and p1!=p2) then
return false
else
return childof(n1,p1)
endif;
getLbl'(n1)/Add(p2,n2) = if (n1==n2) then return novalue
  else
return getLbl(n1)
  endif;
getLbl'(n1)/Del(n2) = if (n2 == n1) then return novalue
  else
```

```
return getLbl(n1)
endif;
getLbl'(n1)/Ren(n2,l2,s2) =if (n1==n2) then
return l2
  else
return getLbl(n1)
endif;
lemma
%basic lemmas needed for the proof

%all trees have node meme and data
=>exist(mem);
=>exist(data);

%assume no auto-concurrency
s1>=s2 and s2>=s1 =>;
not( s1>s2) and not(s2>s1) =>;

%Axioms for trees
childof(x,y) and childof(x,z) and (z!=y) =>;
childofp(x,y) and childofp(y,z)=>childofp(x,z);
childof(x,y)=> childofp(x,y);
childofp(x,x)=>;
```

The output of Vote is:

```
Elapsed time: -704.857296 s

--- Global statistics of the main successful operations ---

- contextual_rewriting   : 0 of 0 tries.
- equational_rewriting   : 0 of 0 tries.
- conditional_rewriting  : 334 of 85455 tries.
- partial_case_rewriting : 0 of 0 tries.
- total_case_rewriting   : 675 of 675 tries.
- induction              : 0 of 0 tries.
- subsumption            : 165 of 63888 tries.
- tautology              : 71 of 245580 tries.
```

```
-----------
  Total clauses: 30428

  Max depth    : 1


All sets of conjectures were successfully processed
```

8.5 Proof of Theorem 5

We give the proof of the combination theorem.

Proof. Given a sequence of computations $Comp$ i.e. a sequence of expressions $\phi_l(E).!r_l$ or $r_e.\phi_e(E)$ respecting causality, we extract $Comp_T$ and $Comp_{id}$ the respective computations of $\mathcal{A}_T$ and A_D for each id:

$\Pi_T(\phi_l(\langle E_T, E_D\rangle).!r_l) = \varphi_l^T(E_T).!r_l^T$ if $r_l = \langle r_l^T, \bot\rangle$

$\Pi_{id}(\phi_l(\langle E_T, E_D\rangle).!r_l) = \varphi_l^D(E_D(id)).!r_l^D$ if $r_l = \langle \bot, (id, r_l^D)\rangle$

$\Pi_T(?r_e.\phi_e(\langle E_T, E_D\rangle) = ?r_e^T.\varphi_l^T(E_T)$ if $r_e = \langle r_e^T, \bot\rangle$

$\Pi_{id}(?r_e.\phi_e(\langle E_T, E_D\rangle) = ?r_e^D.\varphi_e^D(E_D(id))$ if $r_e = \langle \bot, (id, r_e^D)\rangle$

and for all other cases $\Pi_T(\ldots) = \Pi_{id}(\ldots) = 0$ where 0 is the null process that does nothing. By construction $Comp_T$ respects the causality relations restricted to the operations of Op_T. The same holds for $Comp_{id}$. the causality relation (but the reverse doesn't necessarily holds). Therefore $Comp_T$ is a legal computation of $\mathcal{A}_D$ and by the convergence of $\mathcal{A}_D$ each site has he same state s_T. For each $id \in Dom(s_T)$, the sequence $Comp_{id}$ is legal computation of $\mathcal{A}_{id}$, therefore each site has the same state s_{id}.