INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Real time group editors without Operational transformation

Gérald Oster — Pascal Urso — Pascal Molli — Abdessamad Imine

N° 5580

Mai 2005

Thèmes COG et SYM

ISSN 0249-6399 ISRN INRIA/RR--5580--FR+ENG

Real time group editors without Operational transformation

Gérald Oster [*] , Pascal Urso [†] , Pascal Molli [‡] , Abdessamad Imine [§]

Thèmes COG et SYM — Systèmes cognitifs et Systèmes symboliques
Projets ECOO et CASSIS

Rapport de recherche n° 5580 — Mai 2005 — 24 pages

Abstract: A real time group editor allows multiple users to edit the same text at the same time from multiple sites across Internet. The real time group editors community has developed a framework called Operational Transformation (OT) for maintaining consistency of shared data. OT differs from other optimistic replication systems by not only ensuring content consistency but also intention consistency.

In this paper, we describe the WOOT (WithOut Operational Transformation) framework that ensures intention consistency without following the OT approach. However, thanks to its new viewpoint, WOOT is drastically simpler, more efficient and does not require vector clocks or central sites. The WOOT framework is particularly adapted to very large peer-to-peer networks.

Key-words: Real time groupware, optimistic replication, operational transformation, replicated data consistency

[*] ECOO
[†] ECOO
[‡] ECOO
[§] CASSIS

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Editeurs synchrones sans transformées opérationnelles

Résumé : Un éditeur synchrone permet à plusieurs utilisateurs de modifier le même document au même moment depuis plusieurs sites à travers Internet. La communauté des éditeurs synchrones a développé un environnement appelé transformées opérationnelles (OT) pour maintenir la cohérence des données partagées. OT se différencie des autres approches de réplication optimiste en assurant en plus de la convergence des données, le respect de l'intention. Dans cet article, nous décrivons l'environnement WOOT qui garantit la cohérence des intentions mais sans adopter l'approche des transformées opérationnelles. Cette nouvelle approche est largement plus simple, plus efficace et ne requiert ni vecteurs d'horloge ni sites centraux. L'environnement WOOT est particulièrement adapté aux réseaux pair à pair à large échelle

Mots-clés : Editeurs temps rééls, réplication optimiste, transformées opérationnelles, cohérence des données réparties

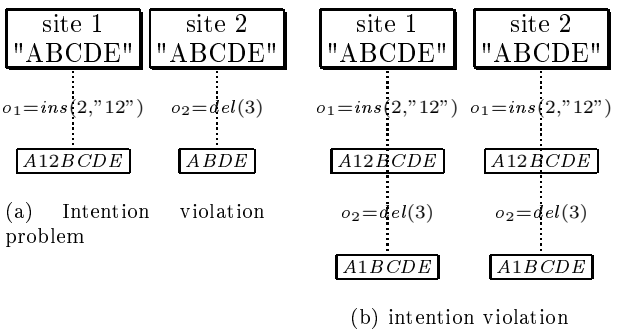


Figure 1: Intention violation problem

1 Introduction

A real time group editor [1, 11] allows multiple users to edit the same text at the same time from multiple sites across Internet. In order to achieve high responsiveness, shared data are replicated on all sites. In order to achieve unconstrained interactions, there are no locking or serialization protocols. Any user can edit the text at any time. If two users generate concurrent operations, the systems have to ensure that replicate will converge to the best possible state.

Group editors are based on the Operational Transformation (OT) model. In this model, an operation generated on one site is executed locally immediately and next broadcasted to other sites to be re-executed. OT model traditionally ensures the consistency model defined by [12]. OT algorithms ensure Causality, Convergence and Intention preservation. If Causality and convergence are very common in optimistic replication[7], intention preservation is more unusual. Intention preservation as defined in [12] ensures that (1) the effect of executing any operation o at remote site achieves the same effect that executing o when it was generated (2) execution effects of concurrent operations do not interfere.

Intention-violation problem has been introduced in the REDUCE approach [10]. Two sites share a string containing "ABCDE" (cf figure 1(a)). Site 1 inserts "12" at position 2 and obtains "A12BCDE". Site 1 has executed operation $o_1 = ins(2, "12")$ with the intention to insert "12" between A and B. Site 2 deletes one character at position 3 and obtains "ABDE". Site 2 has executed operation $o_2 = del(3)$ with the intention to delete the character 'C'.

If we execute both operations and preserve intentions, we must obtain "A12BDE". But if we use a serialization protocol, it can decide to serialize o_1 before o_2. The final result in this case is "A1BCDE" (cf figure 1(b)). Convergence is achieved but intentions are not preserved.

Recently, Li et al[4, 5] introduced the notion of operation effects relation preservation. For example if a user inserts a character 'b' between 'a' and 'c', we must ensure that in

the convergence state, 'b' will be between 'a' and 'c'. If it exists an order relation between two operations effects on a state, then this order relation must be preserved when both operations are re-executed on any state.

Currently, only the SDT algorithm ensures intention consistency as defined in [5]. SDT does not require a central site. However, SDT is a complex algorithm and requires vector clocks. Vector clocks do not scale because their size are proportional to sites count. Using vector clocks is a major drawback for deploying Real time groupware on large peer-to-peer networks.

In this paper, we present a new framework called W00T (WithOut Operational Transformation) that ensures Intention consistency as defined by Li[5] but without operational transformations, without vector clocks and without central sites. W00T is particularly adapted to very large peer-to-peer networks, drastically simpler than SDT and easy to implement.

2 WOOT approach

Traditional Optimistic replication approaches ensure convergence of replicas [7]. The OT approach ensures more than convergence and defines the notion of intention consistency [5]. Intention consistency means that operation effects on generation state are preserved when operation is re-executed on remote sites. But what is the operation effect of a simple insert operation on a string? Sun describes it when he presented the intention violation problem[12]. When a user observes the string "ABCDE" and inserts "12" at position 2, he inserts "12" between A and B. Intention consistency means on this example that if "12" has been inserted between A and B, this ordering $A \prec$ "12" $\prec B$ must be preserved on any further states. Currently, only the SDT algorithm can ensure that these orderings are preserved. If we analyze SDT, we can see that the main problem is to determine these ordering relations just by analyzing the log of operations.

The WOOT approach is fairly simple. Instead of re-computing orderings at reception, we send orderings because we know this information when operations are generated.

The immediate effect is straightforward. Instead of executing and broadcasting $insert(2,"12")$ as in OT algorithms, we execute $insert(2,"12")$ and broadcast $insert('A' \prec "12" \prec' B')$. The first problem is what to do if we receive $insert('A' \prec "12" \prec' B')$ and 'B' has been locally deleted. The WOOT approach is drastically simple: 'B' will exist because we do not delete character, we mark it as invisible.

Of course, if we do not delete characters, it requires more memory and generates bigger files. But we show in this paper that in fact, with the WOOT approach, we don't need to keep the log of operations with vector clocks. Finally, the space complexity of WOOT is lesser than comparable OT algorithms.

Each insert operation generates two new order relationships, but it does not generate a total order, just a partial order. For example, consider three sites and each site generates one operation as presented in figure 2. We represent order relationships between characters in the Hasse diagram of figure 3.

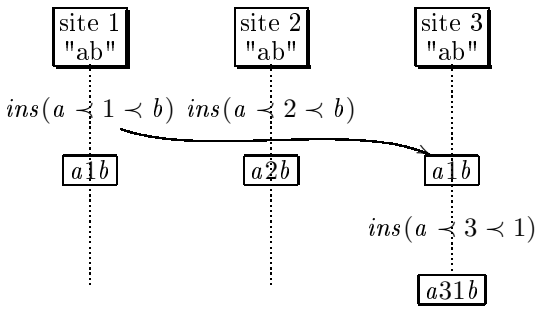


Figure 2: Generating partial orders

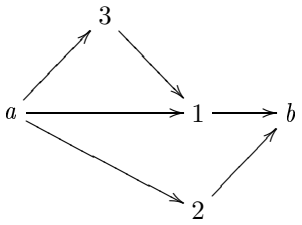
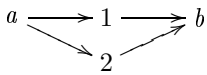


Figure 3: Hasse diagram of order relation between characters

Acceptable states are all linear extensions of this partial order i.e. a312b, a321b, a231b. To achieve convergence, all sites must find the same linear extension. Traditionally, a topological sort is used to find a linear extension of a partial ordered set. However, we cannot use it in our context. Each time an operation is received, linearization must be done and a new state can be observed by the user. A new linearization on a site must be compatible with all previous one. It means that if a linearization determined that 'a' is before 'b', next linearization cannot linearize 'b' before 'a'.

We apply a topological sort on:



First, the only node without predecessor is 'a'. We push it on the result stack and remove it with its edges from the graph. Then next nodes without predecessors are '1' and '2'. We need a choice axiom and we take the least. So we push '1' on the result stack and remove it with its edges from the graph. We repeat these operations until the graph is empty. The result of this topological sort is "a12b".

The operation $insert(a \prec 3 \prec 1)$ is received on the same site and we re-execute the topological sort on the graph of figure 3. We obtain "a231b" which is incompatible with the previous linearization i.e. 2 is now before 1.

The challenge of the WOOT framework is to ensure convergence with a monotonic linearization function. We focus on this paper on sequence of characters but it is clear that the WOOT framework supports any linear structure. A linear structure can be complex i.e. an ordered tree is a linear structure.

3 WOOT Framework

In this section, we present the whole WOOT model and algorithms. We formally define the data structure used by WOOT and the order relations taken in account to linearize the characters. Thus, we explain the algorithms used by the WOOT framework. Then, the framework is evaluated in terms of correctness and complexity.

3.1 Definition of the PCI Consistency model

A group editor is consistent iff :

Precondition Preservation Operations are integrated if their preconditions are true.

Convergence When the same set of operations has been executed at all sites, all copies of the shared document are identical.

Intention Preservation For any operation O, the effects of executing O at all sites are the same of executing O on generation state.

Compared to the original consistency model [1], we do not require causality. An operation can be integrated as soon as its preconditions are true. For example, a user executes locally $o_1 = ins(a \prec' 1' \prec b)$, broadcasts it and next $o_2 = ins(c \prec' 2' \prec d)$ and broadcasts it. Another site can receive o_2 and next o_1. Clearly, if causality is required, o_2 must be executed after o_1. We allow executing o_2 immediately iff preconditions of o_2 are verified i.e. if 'c' and 'd' exist locally.

On one hand, we allow higher concurrency, and it is important for the high-responsiveness requirement, on the other hand, if a hidden dependency exists between o_1 and o_2, it will be violated. We have chosen higher concurrency considering that it is acceptable if intentions are preserved. This choice is not so important. It is possible to use WOOT with a causal reception. If causal reception often involves vector clocks, vector clocks are not required for ensuring effect preservation and convergence.

The PCI consistency model is similar to the Sun CCI consistency model[12] except that causality has been replaced by preconditions. Compared to the Li CSM consistency model [5], we do not adopt the multi-operation effect relation preservation. We think that relation between operation effects are already captured by the intention preservation rule. Li wrote that intention preservation and multi-operation effect preservation imply convergence. We think that intention preservation and convergence imply also multi-operation effects preservation. At last, we are convinced that CSM, CCI and PCI consistency models are

designed to converge to the same final state. We just differ from CCI and CSM by not requiring traditional causality rule.

3.2 Data model

Each site s gets a unique identifier $numSite_s$, a logical clock H_s, a sequence $string_s$ of W-characters and a pool of pending operations $pool_s$.

Definition 1 *A W-character c is a five-tuple $< id, v, \alpha, id_{cp}, id_{cn} >$ where*

- *id is the identifier of the character.*
- *$v \in \{True, False\}$ indicates if the character is visible,*
- *α is the alphabetical value of the effect character,*
- *id_{cp} is the identifier of the previous W-character of c.*
- *id_{cn} is the identifier of the next W-character of c.*

The previous and next W-characters of c are the W-characters between which c has been generated.

Definition 2 *The previous W-character of a W-character c is denoted $C_P(c)$. The next W-character of a W-character c is denoted $C_N(c)$.*

To identify in a unique way the characters we use an identifier based on a site number and a local clock.

Definition 3 *A character identifier is a couple (ns, ng) where ns is the identifier of a site and ng is a natural number. When a character is generated on a site s, its identifier is fixed at $(numSite_s, H_s)$.*

Each time a W-character is generated on a site s, the local clock H_s is incremented. Since $numSite$ is unique, the couple $(numSite_s, H_s)$ forms a unique identifier for a character.

$string_s$ is a W-string. It contains all the integrated W-characters.

Definition 4 *A W-string is an ordered sequence of W-characters $C_b\, c_1\, c_2 \ldots c_n\, C_e$ where C_b and C_e are special W-characters (with special identifiers) marking the beginning and the ending of the sequence.*

We define the following functions for a sequence S:

- $|S|$ denotes the length of the sequence S.
- $S[p]$ denotes the element at the position p in S. We state that the first element of a sequence S is at position 0 and the last element is at position $|S| - 1$.
- $pos(S, c)$ returns the position of the element c in S as a natural number.

- $insert(S, c, p)$ inserts the element c in S at position p.
- $subseq(S, c, d)$ returns the part of the sequence S between the elements c and d, both not included.
- $contains(S, c)$ returns true if c can be found in S

We also need the following functions to link the W-string and the string the user sees.

- $value(S)$ is the representation of S (i.e. the sequence of visible alphabetical values).
- $ithVisible(S, i)$ is the i^{th} visible character of S.

Two operations update a W-string:

ins(c) inserts the W-character c between its previous and next characters. The precondition is previous and next characters exist.

del(c) deletes the W-character c. The precondition of $del(c)$ is c exists.

3.3 Orders

Definition 5 *Let a and b two W-characters. $a \prec b$ if and only if there exists a set of characters $c_0, c_1, ...c_i$ such that $a = c_0, b = c_i$ and $C_N(c_j) = c_{j+1}$ or $c_j = C_P(C_{j+1})$ for all $0 \leq j \leq i$.*

$\prec$ is a binary relation over the set of W-characters. $\prec$ is irreflexive, transitive and asymmetric. $\prec$ is a strict partial order.

To obtain a string from this partial order, we have to find a linear extension (i.e. a total order).

Definition 6 *Let S be a sequence, the relation $\leq_S$ is defined as $a \leq_S b$ if and only if $pos(S, a) \leq pos(S, b)$*

When no precedence relation can be established between two characters, we have to order them. To ensure convergence, this order must be set independently from the state of the site. We use the characters identifier.

Definition 7 *Let a and b two W-characters with their respective identifiers (ns_a, ng_a) and (ns_b, ng_b). $a <_{id} b$ if and only if (1) $ns_a < ns_b$ or (2) $ns_a = ns_b$ and $ng_a < ng_b$.*

3.4 Algorithms

When a site generates an operation, the operation is integrated locally, broadcasted and then integrated by all other sites. The reason to the local integration is to take into account the possible invisible characters i.e. previously deleted characters.

Generation. For an operation op, $type(op)$ denotes the type of the operation: *del* or *ins*. $char(op)$ denotes the character manipulated by the operation.

When user interacts with the framework, he only sees $value(S)$. So, when an insert operation is generated the user-interface only knows the visible position and the alphabetical value of the character to insert. For instance, $ins(2, a)$ in "xyz" is translated into $ins(y \prec a \prec z)$.

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

Similarly, when a delete operation is generated we have to retrieve the W-character at this position.

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

Reception Sites may receive operations with unverified preconditions. The *isExecutable* function checks preconditions of an operation.

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

To deal with pending operations each site maintains a pool of operations.

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

For instance, a site executes $del(c)$ only if c is present. If c is not present, the integration of the operation is delayed until c is present.

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

Example 1 *According to the scenario of figure 1, site 3 delays the integration of o_2 until reception of o_1.*

However, site 1 can execute o_3 before receiving o_2 even if o_2 happened before o_3.

In traditional approach, that uses vector clocks, the execution is delayed in both cases.

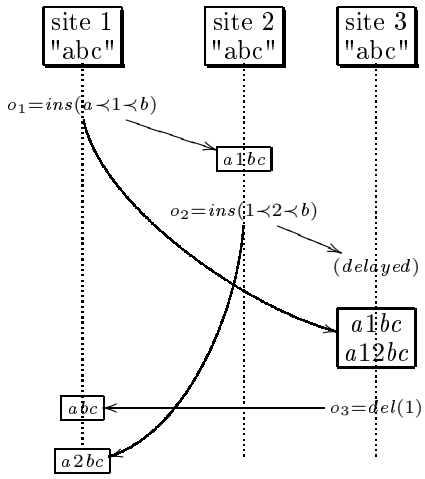


Figure 4: Precondition preservation

Integration To integrate an operation $del(c)$, we only need to set the visible flag of the character c to *false*, whatever the previous value.

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

To integrate an operation $ins(c)$ in $string_s$, we need to place c among all the characters between c_p and c_n. These characters can be previously deleted characters or characters inserted by concurrent operations. When an operation $ins(c)$ is executable on a site, the procedure $IntegrateIns(c, c_p, c_n)$ can be called.

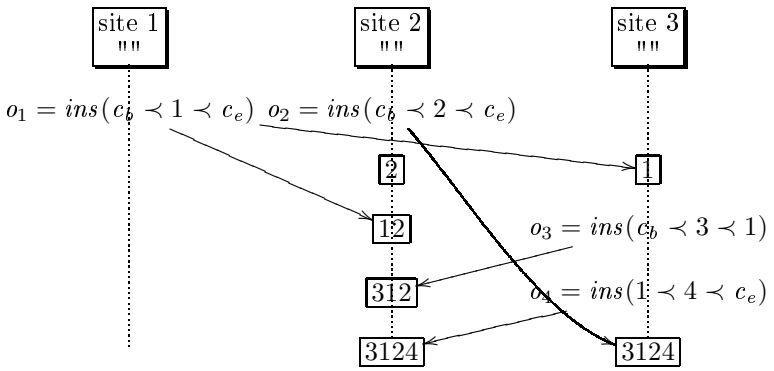
IntegrateIns(c, c_p, c_n)
  let $S := string_s$
  let $S' := subseq(S, c_p, c_n)$
  if $S' = \varnothing$ **then** $insert(S, c, pos(S, c_n))$
  else
    let $L := c_p\, d_0 d_1 \ldots d_m\, c_n$ where $d_0 \ldots d_m$ are the W-chars in S' s.t. $C_P(d_i) \leq_S c_p$ and $c_n \leq_S C_N(d_i)$
    let $i := 1$
    while $(i < |L| - 1)$ and $(L[i] <_{id} c)$ **do**
      $i := i + 1$
    IntegrateIns$(c, L[i-1], L[i])$
  endif

The algorithm orders characters with $<_{id}$ when no precedence relation $\prec$ is available. Thus, the algorithm removes from S' the characters that have a previous or next character in S'. Indeed, such characters are ordered by the precedence relation and not by the identifier ordering.

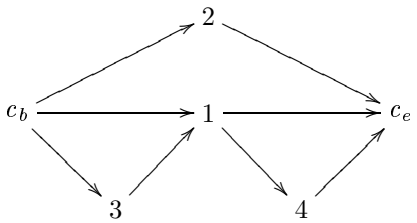
Because characters in $d_0 d_1 ... d_m$ are ordered by the relation $<_{id}$, the algorithm inserts c at its place according to $<_{id}$. However, there may be some characters in S between $L[i-1]$ and $L[i]$. Thus, we call recursively the integration function.

3.5 Examples

Let site 1, site 2 and site 3 be three sites in the initial state "$c_b\, c_e$". We consider the following scenario:



This scenario generates the following Hasse Diagram:



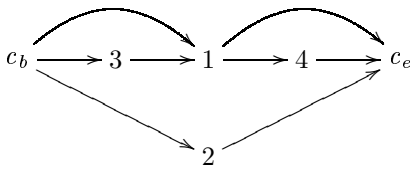
Lets assume that '1' $<_{id}$ '2' $<_{id}$ '3' $<_{id}$ '4'. Site 2 integrates o_1, o_3, o_4 in this order:

1. *IntegrateIns*$(1, c_b, c_e)$: $S' = L =$ "2" and $1 <_{id} 2$, WOOT integrates 1 between c_b and 2. During the recursive call *IntegrateIns*$(1, c_b, 2)$, we get $S' = \varnothing$. Thus we compute "$c_b 12 c_e$".

2. *IntegrateIns*$(3, c_b, 1)$: $S' = \varnothing$, WOOT inserts 3 between c_b and 1.

3. *IntegrateIns*$(4, 1, c_e)$: $S' = L =$ "2" and $2 <_{id} 4$, WOOT integrates 4 between 2 and c_e . We obtain "$c_b 3124 c_e$".

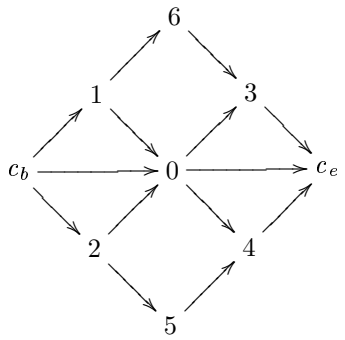
Site 3 integrates o_2.



During *IntegrateIns*$(2, c_b, c_e)$: $S' =$ "314" and $C_N(3) = C_P(4) = 1$, we get $L =$ "1". As $1 <_{id} 2$, WOOT integrates 2 between c_1 and c_e. During the recursive call *IntegrateIns*$(2, 1, c_e)$: $S' = L =$ "4" and $2 <_{id} 4$, WOOT integrates 2 between 1 and 4. Site 3 obtains "$c_b 3124 c_e$".

On site 1, whatever the order of arrival of o_2, o_3, o_4, WOOT computes $c_b 3124 c_e$. In every case, the final string is "3124" and PCI consistency is ensured.

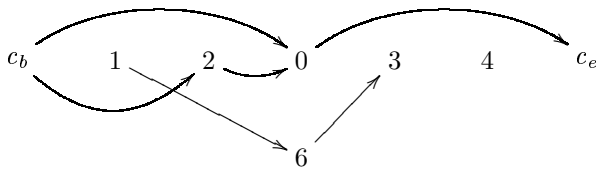
However, it is possible to generate more complex scenario for the WOOT integration procedure. We can produce a scenario with 7 sites. Each site generates a character containing its site identifier. We can generate the following Hasse diagram:



Suppose characters 0,1,2,3,4 already received on a site. First, we simulate reception of 6 then 5. Next, we will simulate reception of 5 then 6 and verify that we find the same result.

The integration procedure produced the string "12034". 1 is linearized before 2 because the $1 <_{id} 2$ and 3 is linearized before 4 because $3 <_{id} 4$. 2 is before 0 and 3 is after 0.

We integrate now $ins(1 \prec 6 \prec 3)$ in "12034". $S' =$ "20" and $L =$ "0". On the following schema, we can see that only 0 has $C_P(0) \leq_S C_P(6)$ and $C_N(6) \leq_S C_N(0)$.



As $0 <_{id} 6$, we linearize 6 between 0 and 3. We obtain "120634".

Now, the same site receives $ins(2 \prec 5 \prec 4)$ and integrates 5 in string "120634". $S' =$ "063" and $L =$ "0". As $0 <_{id} 5$, we integrate recursively 5 between 0 and 4. $S' =$ "63" and $L =$ "3". As $3 <_{id} 5$, 5 is between 3 and 4. So the final result is "1206354".

We restart the scenario on the "12034" state but now we receives $ins(2 \prec 5 \prec 4)$ first. $S' =$ "03" and $L =$ "0". As $0 <_{id} 5$, we integrate recursively 5 between 0 and 4. $S' = L =$ "3" and $3 <_{id} 5$, we obtain "120354".

Now we integrate $ins(1 \prec 6 \prec 3)$ on "120354". $S' =$ "20" and $L =$ "0". As $0 <_{id} 6$, 6 is between 0 and 3. We obtain "1206354" as expected.

4 Correctness and Complexity

First, we check that WOOT ensures the PCI consistency model. Next we evaluate the time and space complexity of WOOT.

4.1 Correctness

Theorem 1 *The algorithm of integration terminates.*

Proof 1 *Proof by contradiction.*

The algorithm does not terminate if and only if the recursive call is not done on a strictly smaller subsequence. This can happens only if we get, a non-empty S' and an $L = c_p c_n$. If $L = c_p c_n$, every character in this S' have its predecessor or its successor in S'. Since the characters have been generated in a strict order, at least the firstly integrated character between c_p and c_n has its previous and its next character outside S'.

Thus, there are at least 3 characters in L. So the recursive call is done on a strictly smaller subsequence and the algorithm terminates.

Precondition preservation It is obvious to show that, since an operation *op* is integrated only if $isExecutable(op)$ is true, precondition preservation is ensured.

Intention preservation Our linearization order must respect the precedence order defined when operations are generated.

Theorem 2 *The relation $\leq_s$ induced by WOOT on each site is a linear extension of the relation $\prec$.*

Proof 2 *The $\prec$ relation is never modified after the generation of an operation. The linearization $\leq_S$ is only modified through IntegrateIns. The integration of a character c is always done by the insertion of c between $C_P(c)$ and $C_N(c)$. Thus $\leq_s$ is a linear extension of $\prec$.*

However, ensuring intention preservation is not enough. For instance, if two sites insert respectively 'a' and 'b' between the same characters 'x' and 'y', we can obtain two different strings on the two sites; respectively "axyb" and "ayxb". These two linear extensions respect intention preservation but they do not converge.

Convergence We do not have a hand-written proof of convergence. To verify the correctness of our algorithm we have used the model-checker TLC on a specification modelled on the TLA+ specification language [15]. The model-checking techniques are particularly suited to verify concurrent systems. With the TLC model checker we verified a bounded version of our framework. Due to the famous state explosion problem, it is impossible in practice to test a system with a big amount of sites and characters. We made complete verifications up to four sites and fives characters. It took about two weeks with a Pentium(R) 4 CPU $2.80GHz$. Section A contains the complete TLA specification.

Convergence requires that two characters on two sites are linearized in the same order. This conjecture and the fact that every generated character will be inserted in every site ensure the convergence criteria.

Conjecture 1 *Let S_1 and S_2 two W-strings maintained by two different sites. For every pair $\{c, d\}$ of characters whose appear on both sites, we get*

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

To reduce design complexity of the specification and to accelerate model-checking, we made two slight generalizations. The *del* operation and its integration do not appear in the model since this operation does not affect the linearization order. However, to simulate deletion of characters, we allow generating a $ins(c_p \prec c \prec c_n)$ in S requiring only $c_p \leq_S c_n$ i.e. as if the characters between c_p and c_n were deleted.

The second generalization consists in representing characters simply by an identifier. This is also a generalization since in the model any sites can generate any character identifiers.

The TLC model-checker found an error in a previous naive version of the integration algorithm. In this version, we did not filter S' to obtain L. We thought that since all the characters between c_p and c_n were concurrent to c we simply have to order them according to $<_{id}$. The model checker found the counter example presented in figure 3. This counter-example helped us to design the current version of the integration algorithm.

4.2 Complexity

We evaluate algorithmic complexity of WOOT in function of n the operations count already generated by all sites. The following theorem express that size required by WOOT is proportional to n.

Theorem 3 *WOOT space complexity is $O(n)$.*

Proof 3 *The size of the local clock and site identifier are constant (0(1))*

- *The pool contains k operations. We get that $k \leq n$.*
- *The size of a W-character is constant $O(1)$. there are $m \leq (n-k)$ already integrated insert operations. The W-string begins with two characters (c_b and c_e). Delete operations do not affect the size of the W-string and each insert operation adds a unique W-character. Thus, the size of a W-string is proportional to $m+2$.*

Thus, the WOOT space complexity is proportional to $k+m$ and, since $k+m \leq n$, the WOOT space complexity is $O(n)$.

Theorem 4 *The worst case time complexity of integration is $O(n^3)$.*

Proof 4 *Let m be the size of a string$_s$; The integration of a $del(c)$ operation takes a linear time $O(m)$. We have to scan the W-string to find the identifier of c.*

For an $ins(c)$ operation, we have the following statements and their worst-case time complexity:

- *$S' = subseq(\ldots)$ $(O(m))$*
- *either $insert(\ldots)$ $(O(m))$*
- *or*
  - *filter S' to obtain L : since $\leq_S$ is in $O(m)$, it takes $O(m^2)$.*
  - **while** *loop to find i $(O(m))$*

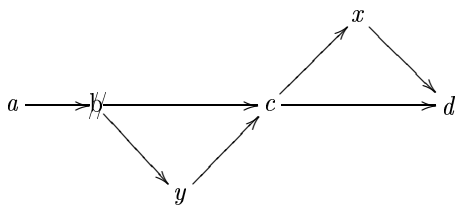
– *recursive call.*

There are at most m recursive calls to IntegrateIns. Thus, time complexity is $O(m(m + max(m, m^2 + m))) = O(m^3) = O(n^3)$ *because* $m \leq n$.

We supposed that finding a W-character or comparing two W-characters position in W-String is in $O(n)$. If we maintain an index of identifiers in a W-String, we can improve the WOOT average time complexity to $O(n^2)$. It will take more space but the space complexity will be still in $O(n)$.

5 Related work

In the OT approach, just few algorithms ensures the intention consistency model as defined in [5]. Often, they fail on the famous TP2 puzzle. The starting point of TP2 puzzle is very simple. Three sites share a string "abcd". Site 1 executes $o_1 = ins(3,'x')$, site 2 executes $o_2 = del(2)$ and site 3 performs $o_3 = ins(2, y)$. Everybody see immediately that 'x' is after 'c', 'y' before 'c' and 'b' is destroyed. Then, the convergence state should be "aycxd". Why is it so complex to solve this problem with OT ? The problem looks simple because we can observe all sites simultaneously, an OT algorithm run on each site and must build its state with just the knowledge of its log. When operation o_1 is received on site 2, o_1 do not contain the information: "x should be after c". The OT algorithm has to recompute this information with its log. This leads OT algorithms to solve complex problems and propose complex solutions. If we take the WOOT viewpoint on TP2 puzzle, we obtain the following Hasse diagram:



The partial order is already a total order. The only linear extension of this graph is "abycxd". As 'b' is not visible, the final string is "aycxd" as expected.

SOCT4[14] maintains the same transformation path on all sites by using a continuous global total order built with a centralized timestamper and deferred broadcast. Li in [5] wrote that SOCT4 cannot ensure operation effects relation preservation and gives an example p458. We redo the example and found that, on this example, operation effect relation preservation is preserved. We claim that it is not proven that SOCT4 cannot ensure the intention consistency model. However, it is clear that the centralized timestamper is not compatible with peer-to-peer environment. If we compare WOOT and SOCT4, WOOT do not require a central site.

SOCT3[9] also maintains a continuous global total order using a centralized timestamper but does not require deferred broadcast. SOCT3 uses a separation procedure for resolving

partial concurrency [8] that requires backward transformations. Using backward transformation without requiring TP2 on forward transformation is unsafe. So SOCT3 mandates TP2 [13] and TP2 transformation functions are not provided (see figure 6).

GOT [12] maintains a non-continuous global total order without using a centralized timestamper. Non-continuous global order forces the GOT algorithm to undo/redo some operations. Undo/redo is not really adequate for the high-responsiveness requirement of real-time editor. GOT uses exclusion transformations to solve partial concurrency. Unlike SOCT3, GOT calls exclusion transformation on two causally dependent operations. Exclusion transformation is not always defined in this case. This can lead to inconsistencies.

Other OT algorithms rely on satisfaction of TP1 and TP2 as defined in [6] for their transformation functions. Currently, existing transformation functions from Ressel[6], Sun[12] and IMOR[3] violate TP2. Counter example for IMOR appears in [5], p 465 and counter examples for Ressel and Sun appear in [3]. The scenario for IMOR is illustrated in figure 5.

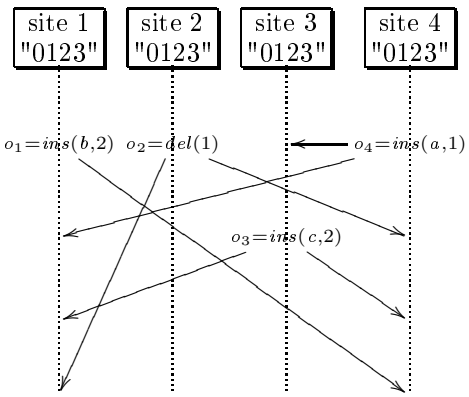
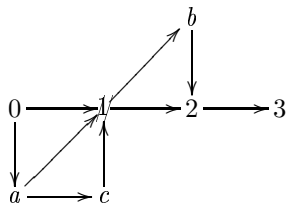


Figure 5: Scenario from Li for IMOR

If we take the WOOT viewpoint on this scenario, we obtain the following Hasse diagram:



All characters are already totally ordered, so all sites will converge to "0acb23".

There is no counter-example for [9]. Figure 6 presents a counter-example for Suleiman transformation functions. This scenario violates both TP2 and intention preservation. Complete transformation functions from Suleiman appear in [9, 8]. In fact, we generate a classical TP2 puzzle between sites 1,2 and 3. Sites 4 and 5 are just here to fill the before and after sets of operations defined by Suleiman.

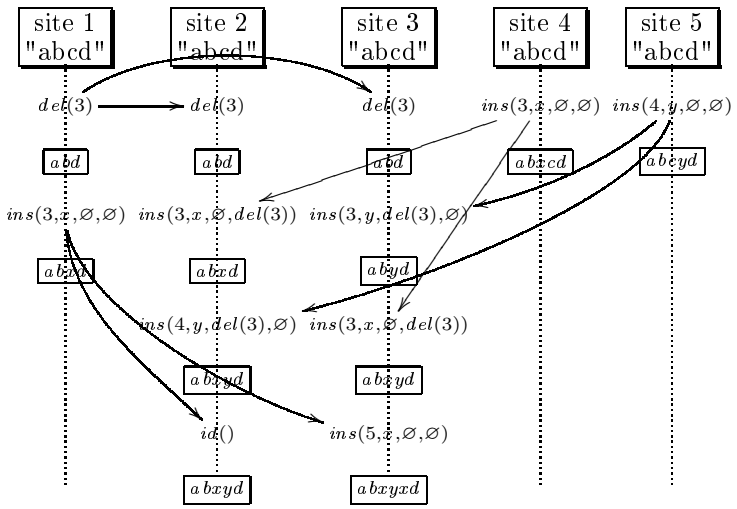
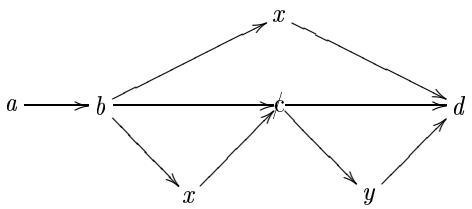


Figure 6: Suleiman counter example

The WOOT viewpoint on this counter-example is presented in the following Hasse diagram:



On site 2, when $ins(b \prec x \prec d)$ from site 1 is received, we integrate the 'x' in the current state "abx¢yd". $S' = xcy$ and $L = c$. We assume that $c <_{id} x$. We recursively call integration of x between c and d. $S' = L = y$ and $x <_{id} y$, so we obtain "abx¢xyd". The same integration scenario is executed on site 3 and returns the same result.

In [5], Li defines transformation functions that satisfy TP1, TP2 and a new integration algorithm SDT. When SDT transforms two concurrent insert operations o_1 and o_2 defined on the same state s, SDT computes $\beta_{lsp}(o_1)$ and $\beta_{lsp}(o_2)$ which are the position of o_1 and o_2 on a state called Last Synchronization Point (LSP). LSP is the state identified by the vector clock $V_{min} = min(v(o_1), v(o_2))$. $v(o_1), v(o_2)$ are vector clocks of o_1, o_2. V_{min} is built with the minimal value of each component of the vector clocks of o_1, o_2. First, SDT computes the sequence SQ of operations that generates state s starting from state LSP. Then, SDT computes another sequence SD which is sequence equivalent to SQ but that only contains the net effect between LSP and s. Next, SDT excludes SD from o_1 and $o_2$and obtains positions of o_1, o_2 on state LSP. Comparing $\beta(o_1)$ and $\beta(o_2)$ allow breaking the tie while ensuring intention consistency.

If we compare WOOT and SDT, it is clear that WOOT is much simpler. Unlike SDT, WOOT does not require vector clocks. At first sight, WOOT seems to require more space than SDT, but it is false. SDT implies that each site keep a log of executed operations. Each operation keeps a vector clock. For n generated operations, SDT has a log of size n. if m is the number of sites, a vector clock size is $O(m)$. So the space complexity of SDT is $O(n * m)$. The WOOT space complexity is $O(n)$.

Tendax [1] [2] is a collaborative editor relying on a database. WOOT and Tendax have the same operations profile. Also, thanks to its centralized approach, it does not require vector clocks. In Tendax, each character is stored with a unique identifier, the identifier of the previous character and the identifier of the next character. However, Tendax removes characters and Tendax behavior is not formally described in this case. Compared to Tendax, WOOT does not require a central server.

6 Conclusion

Currently, WOOT is the only framework that preserves intention consistency on linear structures without central sites and vector clocks. Unlike OT algorithms, WOOT scales and is particularly adapted to large peer-to-peer networks.

Although WOOT makes drastic choices on its data model i.e. no deletion of elements, unique identifiers for characters, the WOOT space complexity is less or equal to comparable OT frameworks. If we have not yet a complete proof, the WOOT framework has been formally verified with model checking tools.

We have realized a simple implementation with threads in java that allows simulating any scenario. It can be downloaded from `http://www.loria.fr/~molli/woot`.

We are working now on a complete proof of WOOT correctness, group undo features , and WOOT support for XML tree.

7 Acknowledgments

Special thanks to Florent Jouille for his implementation of WOOT in few days. Thanks to Hala Skaf for reviewing.

References

[1] Clarence A. Ellis and Simon J. Gibbs. Concurrency Control in Groupware Systems. In SIGMOD Conference, volume 18, pages 399–407, 1989.

[2] T. B. Hodel, M. Dubacher, and K. R. Dittrich. Using database management systems for collaborative text editing. In ACM European Conference of Computer-supported Cooperative Work (ECSCW CEW 2003), Helsinki (Finnland), 2003.

[1] http://www.tendax.net/

[3] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In Proceedings of the 8th European Conference on Computer-Supported Cooperative Work, Helsinki, Finland, September 2003. ACM.

[4] Du Li and Rui Li. Ensuring content and intention consistency in real-time group editors. In ICDCS, pages 748–755, 2004.

[5] Du Li and Rui Li. Preserving operation effects relation in group editors. In CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work, pages 457–466, New York, NY, USA, 2004. ACM Press.

[6] 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.

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

[8] 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.

[9] 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.

[10] C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms and achievements. In Computer Supported Cooperative Work, 1998.

[11] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In Proceedings of the 1998 ACM conference on Computer supported cooperative work, pages 59–68. ACM Press, 1998.

[12] 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.

[13] Nicolas Vidot. Convergence des Copies dans les Environnements Collaboratifs Répartis. PhD thesis, Université Montpellier II, 20 septembre 2002.

[14] 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.

[15] Panagiotis Manolios Yuan Yu and Leslie Lamport. Model checking TLA+ specifications. In Proceedings of Correct Hardware Design and Verification Methods (CHARME'99), pages 54–66, 1999.

A WOOT TLA specification

MODULE *woot*

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

CONSTANTS

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

VARIABLES

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

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

$integrate(c, icp, icn, seq) \triangleq$
  LET $int[cp, cn \in Chars] \triangleq$
    LET $i1 \triangleq find(seq, cp)$
      $i2 \triangleq find(seq, cn)$
      $comp(d) \triangleq (find(seq, ccp[d]) \leq i1) \wedge (i2 \leq find(seq, ccn[d]))$
    IN
      IF $(i2 - i1) = 1$ THEN
        $[i \in 1 \,..\, (Len(seq) + 1) \mapsto$ IF $(i < i2)$ THEN $seq[i]$ ELSE
          IF $(i = i2)$ THEN c ELSE $seq[i - 1]]$
       ELSE
        LET $ic \triangleq$ CHOOSE $i \in i1 \,..\, i2 - 1 :$
          $(i = i1 \vee (comp(seq[i]) \wedge less(seq[i], c))) \wedge$
          $\forall j \in i + 1 \,..\, i2 - 1 : comp(seq[j]) \Rightarrow less(c, seq[j])$
            $id \triangleq$ CHOOSE $i \in ic + 1 \,..\, i2 :$
          $(i = i2 \vee (comp(seq[i]) \wedge less(c, seq[i]))) \wedge$
          $\forall j \in i1 + 1 \,..\, i - 1 : comp(seq[j]) \Rightarrow less(seq[j], c)$

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

raz ≜
```

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

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

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

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

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399