Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence

Abdessamad Imine, Michaël Rusinowitch,
Gérald Oster and Pascal Molli

LORIA, INRIA - Lorraine
Campus Scientifique, 54506 Vandœuvre-Lès-Nancy Cedex, France

Abstract

Distributed groupware systems provide computer support for manipulating objects such as a text document or a filesystem, shared by two or more geographically separated users. Data replication is a technology to improve performance and availability of data in distributed groupware systems. Indeed, each user has a local copy of the shared objects, upon which he may perform updates. Locally executed updates are then transmitted to the other users. This replication potentially leads, however, to divergent (*i.e.* different) copies. In this respect, Operational Transformation (OT) algorithms are applied for achieving convergence of all copies, *i.e.* all users view the same objects. Using these algorithms users can exchange their updates in any order since the convergence should be ensured in all cases. However, the design of such algorithms is a difficult and error-prone activity since building the correct updates for maintaining good convergence properties of the local copies requires examining a large number of situations. In this paper, we present the modelling and deductive verification of OT algorithms with algebraic specifications. We show in particular that many OT algorithms in the literature do not satisfy convergence properties unlike what was stated by their authors.

Key words: Distributed Groupware Systems, Replication, Operational Transformation, Algebraic Specification, Automated Verification.

Email addresses: `imine@loria.fr` (Abdessamad Imine), `rusi@loria.fr` (Michaël Rusinowitch), `oster@loria.fr` (Gérald Oster), `molli@loria.fr` (Pascal Molli).

1 Introduction

Distributed groupware systems allow two or more users (sites) to simultaneously manipulate objects (*i.e.* text, image, graphic, etc.) without the need for physical proximity and enable them to synchronously observe each other's changes. In order to achieve an unconstrained group work, the shared objects are *replicated* at the local memory of each participating user. Every operation is executed locally first and then *broadcasted* for execution at other sites. So, the operations are applied in different orders at different *replicas* (or copies) of the object. This potentially leads to *divergent* (or different) replicas – an undesirable situation for replication-based distributed groupware systems [22].

Operational Transformation is an approach which has been proposed to overcome the divergence problem, especially for building real-time groupware [5,20]. This approach consists of an algorithm which transforms an operation – previously executed by some other site – according to local concurrent ones in order to achieve convergence. It has been used in several group editors [5,16,20,18,24,21], and more recently it is employed in other replication-based groupware distributed systems such as a generic synchronizer [14]. The advantages of this approach are: (i) it is independent of the replica state and depends only on concurrent operations; (ii) it enables an unconstrained concurrency, *i.e.* no global order on operations is required; (iii) it ensures a good responsiveness in real-time interaction context. However, if OT algorithms are not correct then the consistency of shared data is not ensured. Accordingly, it is critical to verify such algorithms in order to avoid the loss of data when broadcasting operations. According to [16,19], the OT algorithm needs to fulfill two convergence conditions C_1 and C_2 that will be detailed in Section 2. Finding such an OT algorithm and proving that it satisfies C_1 and C_2 is not an easy task. This proof is often difficult – even impossible – to produce by hand and unmanageably complicated.

Our solution. To overcome this problem, it is necessary to encourage OT algorithm designers to write a formal specification, *i.e.* a description about the replica behaviour, and then verify the correctness of the OT algorithm *w.r.t.* convergence conditions by using a theorem prover. However, effective use of a theorem prover typically requires expertise that is uncommon among software engineers. So, our work is aimed at designing and implementing techniques underlying the design of OT algorithms which meet the following requirements: (i) *Writing formal specifications must be effortless.* (ii) *High degree of automation must be provided in the proof process.* The designers should use the theorem prover as a (push-button) probing tool to verify convergence conditions.

Using Observational Semantics, we treat a replica object as a black box [8].

We specify interactions between a replica object and a user. Operations for modifying the replica states are called *methods* and operations for observing the states are called *attributes*. We only access to the current state by observing its predecessor states modified by methods through attributes. We have implemented our approach in a tool which enables a developer to define all replica operations (methods and attributes) and the associated OT algorithm. From this description, our tool generates an algebraic specification described in terms of conditional equations. As verification back-end we use `SPIKE`, a first-order implicit induction prover, which is suitable for reasoning about conditional theories [3,4].

The main contribution of this paper is that it shows with lightweight formal verification techniques, it is feasible (i) to write easily a formal specification of a replica object, and (ii) to have its OT checked *w.r.t.* convergence conditions so as to guarantee the correctness of the OT algorithm. Moreover, using our theorem-proving approach we have obtained unexpected results. Indeed, we have detected bugs in several OT-based distributed groupware systems designed by specialists from the domain [9,10].

Related work. To our best knowledge, there is no other work on formal verification of OT algorithms. In [10], we have represented the replica as an abstract data type, but the proof effort was increased with complex data structures (*e.g.* an XML tree). Indeed, a proof property involving data may call for numerous sub-proofs of properties about its logical structure. In [12,11], we have used the Situation Calculus for hiding the internal state of replica but this formalism turned out to be inappropriate to accurately model the notion of state. In this work, we defend the thesis that the observational semantics is well-suited to abstract away from the internal replica structure. It describes the behaviour of a replica object as it is viewed by an external user. Since OT algorithms rely on replica methods, then the verification process becomes easier.

Plan of the paper. This paper is organized as follows: in Section 2 we give the basic concepts of the OT approach and a model for distributed groupware systems based on transformation. The ingredients of our formalization for specifying the replica object and OT algorithm are given in Section 3. In Section 4, we present how to express convergence conditions in our algebraic framework. Section 5 briefly describes our tool and numerous bugs that we have found in some replication-based groupware distributed systems. Finally, we give conclusions and present future work.

2 Operational Transformation Approach

Distributed groupware systems allow a group of users to simultaneously manipulate the same object (*i.e.* a text, an image, a graphic, etc.) from physically dispersed sites (or users) that are interconnected by a supposed reliable network [6]. There are two kinds of groupware: *synchronous* and *asynchronous* systems. In synchronous groupware, people interact with each other at the same time and the response time must be short. Group editors are examples of people editing a shared document at the same time [5,16,20,21]. In asynchronous ones, users usually collaborate by accessing and modifying shared information without immediate knowledge about the actions of other users (either because users work at different times or simply because they do not have access to each other's actions). Version control systems [17] and data synchronizers [14] are examples where users modify a copy of the shared object at different times and have to merge later their modifications in order to obtain the same object state.

As human users are an integrated part of them, the distributed groupware systems have in general the following characteristics [5,20,14]:

- *Distribution*: users may reside on different computers connected by different communication networks with nondeterministic latency.
- *Unconstrained interaction*: multiple users are allowed to concurrently and freely modify any part of the shared object at any time, in order to facilitate free and natural information flow among multiple users.

However, these requirements are particularly difficult to achieve in wide-area and mobile wireless networks where high communication latencies are common. Thus a *replicated* architecture is used: the shared objects are replicated on the local memory of each participating user. The operations of each user are executed on the local replica immediately without being blocked or delayed, and then are propagated to remote users to be executed again.

2.1 Convergence Problems

One of the significant issues when building distributed groupware systems with a replicated architecture and an arbitrary communication of messages between users is the *consistency maintenance* (or *convergence*) of all replicas. To illustrate this problem, consider the following example:

Example 2.1 *Consider the following group text editor scenario (see Figure 1): there are two users (sites) working on a shared document represented by a sequence of characters. These characters are addressed from* 0 *to the end* *of the document. Initially, both copies hold the string "*`efecte`*". User* 1 *executes operation* $op_1 = Ins(1, \text{“}f\text{”})$ *to insert the character "*`f`*" at position* 1*. Concurrently, user* 2 *performs* $op_2 = Del(5)$ *to delete the character "*`e`*" at position* 5*. When* op_1 *is received and executed on site* 2*, it produces the expected string "*`effect`*". But, when* op_2 *is received on site* 1*, it does not take into account that* op_1 *has been executed before it and it produces the string "*`effece`*". The result at site* 1 *is different from the result of site* 2 *and it apparently violates the intention of* op_2 *since the last character "*`e`*", which was intended to be deleted, is still present in the final string. Consequently, we obtain a* divergence *between sites* 1 *and* 2*. It should be pointed out that even if a serialization protocol [5] was used to require that all sites execute* op_1 *and* op_2 *in the same order to obtain an identical result "*`effece`*", this identical result is still inconsistent with the original intention of* op_2*.*

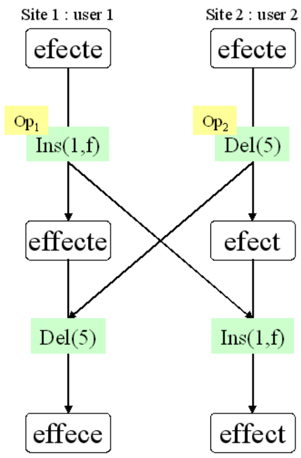

Fig. 1. Incorrect integration.

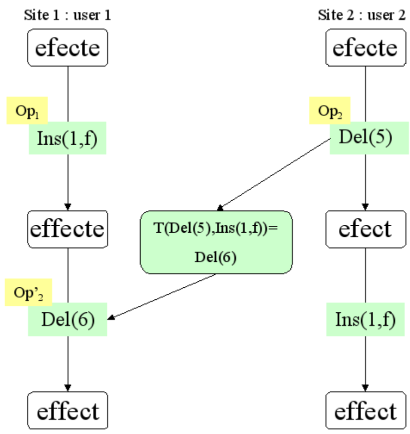

Fig. 2. Integration with transformation.

To maintain convergence, an OT approach has been proposed in [5] where a user X might get an operation *op* that was previously executed by some other user Y on the replica of the shared object. User X does not necessarily integrate *op* by executing it as it is on its replica. Instead, he might execute a variant of *op*, denoted by op' (called *transformation* of *op*) that intuitively intends to achieve the same effect as *op*. This approach is based on an algorithm which takes two concurrent operations that are defined on the same object state. We denote this algorithm by a function T.

Example 2.2 *In Figure 2, we illustrate the effect of* T *on the previous example. When* op_2 *is received on site* 1*,* op_2 *needs to be transformed according to* op_1 *as follows:* $T((Del(5), Ins(1, \text{“}f\text{”})) = Del(6)$*. The deletion position of* op_2 *is incremented because* op_1 *has inserted a character at position* 1*, which is before the character deleted by* op_2*. Next,* op_2' *is executed on site* 1*. In the same way, when* op_1 *is received on site* 2*, it is transformed as follows:* $T(Ins(1, \text{“}f\text{”}), Del(5)) = Ins(1, \text{“}f\text{”})$*;* op_1 *remains the same because "*`f`*" is inserted before the deletion position of* op_2*.*

In the following, we consider a distributed groupware system as a *group of users* (or sites), each communicating with one another through a shared object. The shared object is replicated among a group of sites where every site has its own replica. Every shared object has:

(1) a type (*i.e.* a text, an XML, a file system, etc.) which defines a set of possible *states*, denoted by $\mathcal{S}$;
(2) a set of primitive operations, denoted by $\mathcal{O}$, where each operation is given with a pre-condition under which it is enabled on an object state;
(3) a transition function $\bullet : \mathcal{S} \times \mathcal{O} \to \mathcal{S}$

Definition 2.3 (Local and Remote Operations).
Given a site, a local *operation is an operation generated on this site whereas a* remote *operation is one that is generated on another site.*

Each site generates operations sequentially and stores these operations in a data structure called *history*:

Definition 2.4 (Histories).
A history *is a sequence of operations. We model histories as elements of the set $\mathcal{H}$ which are defined by the following syntax:*

$$h \ ::= \ \Lambda \,|\, op \,|\, h; h$$

where $op \in \mathcal{O}$. The symbol Λ denotes the empty history – a history with no operations. We denote the length of a history h by $|h|$.

The expression $(st)h$ represents the object state obtained by executing history h on object state st. It is recursively defined as follows:

$$(st)\Lambda = st \text{ and}$$
$$(st)(op_1; op_2; \ldots; op_n) = (((st \bullet op_1) \bullet op_2) \bullet \ldots) \bullet op_n$$

Definition 2.5 (Legality).
A history h is legal *from every object state st if the pre-condition of each operation of h is satisfied.*

Definition 2.6 (History Equivalence).
Two histories h_1 and h_2 are equivalent *for every object state st if the following conditions are satisfied:*

(1) h_1 and h_2 are legal from st;
(2) $|h_1| = |h_2|$;

(3) $(st)h_1 = (st)h_2$;

This equivalence is denoted by $\equiv_{st}$.

Lemma 2.7 *History equivalence* $\equiv_{st}$ *is a congruence.*

Definition 2.8 (Convergence Property). *The* convergence property *states all replicas are identical after that all generated operations have been executed at all sites.*

Every site uses an OT algorithm for transforming remote operations in order to correctly integrate them in its own history.

Definition 2.9 (OT Function).
An OT algorithm is a function $T : \mathcal{O} \times \mathcal{O} \rightarrow \mathcal{O}$ *defined for all* $(op_1, op_2) \in \mathcal{O} \times \mathcal{O}$ *such that:*

(1) op_1 *and* op_2 *are concurrent and defined on the same object state, and;*
(2) the pre-condition of $T(op_1, op_2)$ *is satisfied on the state resulting from the execution of* op_2.

In $T(op_1, op_2)$, op_1 is the remote operation whereas op_2 is the local operation. The OT function is used as follows: let op_i and op_j be two concurrent operations defined on the same object state. Suppose that op_i and op_j are generated on sites i and j respectively (with $i \neq j$). Given $op_i' = T(op_i, op_j)$ and $op_j' = T(op_j, op_i)$. Then, site i executes the history $(op_i; op_j')$ and site j performs the history $(op_j; op_i')$. In fact, when a remote operation arrives at a site it is transformed to include the effect of other operations (those which it did not see in its original site) in order to correctly integrate it in the local history.

Example 2.10 *Consider the group text editor GROVE designed by Ellis and Gibbs [5] who are the pioneers of the OT approach. There are two editing operations:* $Ins(p, c, pr)$ *to insert a character* c *at position* p *and* $Del(p, pr)$ *to delete a character at position* p. *Operations* Ins *and* Del *are extended with a new parameter* pr [1]. *This one represents a priority scheme that is used to solve a conflict occuring when two concurrent insert operations were originally intended to insert different characters at the same position. In Figure 3, we give the four transformation cases for* Ins *and* Del *proposed by Ellis and Gibbs. There are two interesting situations in the first case. Indeed, when the arguments of both insert operations are equal (i.e.* $p1 = p2$ *and* $c1 = c2$*) the function* T *returns the idle operation* Nop *that has a null effect on text state* [2].

[1] This priority is the site identifier where operations have been generated. Two operations generated from different sites have always different priorities.
[2] The definition of T is completed by: $T(Nop, op) = Nop$ and $T(op, Nop) = op$ for

The second interesting situation is when only the insertion positions are equal (i.e. $p1 = p2$*). Such conflicts are resolved by using the priority order associated with each insert operation. The insertion position will be shifted to the right (*$p1 + 1$*) when Ins has a higher priority. The remaining cases of* T *are quite simple.*

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

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

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

T(Del(p1,pr1),Del(p2,pr2)) =
 if (p1 < p2) return Del(p1,pr1)
 else if (p1 > p2) return Del(p1-1,pr1)
      else return Nop()
```

Fig. 3. Transformation function defined by Ellis and Gibbs [5].

In order to ensure that the system remains convergent under application of T, this function has to satisfy the following two conditions [16,19]:

Definition 2.11 (Condition C_1).
T is said to satisfy C_1 if for all operations op_1, $op_2 \in \mathcal{O}$, if $op_1' = T(op_1, op_2)$ and $op_2' = T(op_2, op_1)$ then:

$$(op_1; op_2') \equiv_{st} (op_2; op_1')$$

Definition 2.12 (Condition C_2).
T is said to satisfy C_2 if for all operations op,op_1, and $op_2 \in \mathcal{O}$ if $op_1' = T(op_1, op_2)$ and $op_2' = T(op_2, op_1)$ then:

$$T(T(op, op_1), op_2') = T(T(op, op_2), op_1')$$

---

every operation *op*.

C_1 defines a *state identity* and ensures that if op_1 and op_2 are concurrent, the effect of executing op_1 before op_2 is the same as executing op_2 before op_1. This condition is necessary but not sufficient when the number of concurrent operations is greater than two. As for C_2, it ensures that transforming op along equivalent and different histories will give the same result. In [18,13], the authors have proved that conditions C_1 and C_2 are sufficient to ensure the convergence property for *any number* of concurrent operations which can be executed in *arbitrary order*.

It should be noted that the function T of Figure 3 contains some not obvious bugs that lead to divergence situations. These situations will be detailed in Section 4.

2.3 History Transformation

We begin by extending transformation T to work over histories of operations.

Definition 2.13 (Extension of T).
We define $T^* : \mathcal{H} \times \mathcal{H} \rightarrow \mathcal{H}$ as follows:

$$T^*(h, \Lambda) = h \tag{1}$$
$$T^*(\Lambda, h) = \Lambda \tag{2}$$
$$T^*(h_1, (h_2; h_3)) = T^*(T^*(h_1, h_2), h_3) \tag{3}$$
$$T^*((h_1; h_2), h_3) = T^*(h_1, h_3)\ ;\ T^*(h_2, T^*(h_3, h_1)) \tag{4}$$

for all legal histories h, h_1, h_2 and h_3.

Let h_1 and h_2 be two concurrent legal histories from the same object state. If $h_1' = T^*(h_1, h_2)$, then T^* is used to add a legal history h_1' to h_2. The first two equations in Definition 2.13 are trivial. Equation (3) means that transforming h_1 against $(h_2; h_3)$ consists in first transforming h_1 with respect to h_2 (producing $T^*(h_1, h_2)$) and then transforming $T^*(h_1, h_2)$ against h_2. As for Equation (4), the transformation of history $(h_1; h_2)$ with respect to h_3 begins by transforming h_1 against h_3. Next, h_2 is not directly transformed with respect to h_3, because h_2 follows h_1 and the operations of h_1 are not in h_3. Instead, h_3 must be transformed against h_1 (to include its effect) and then h_2 may be transformed against the result.

In the following, we assume that the OT function T satisfies the convergence conditions C_1 and C_2 and we will show that these conditions can be extended to histories. Note that we replace $T^*(h_1, h_2)$ by $T(h_1, h_2)$ when h_1 and h_2 contain only one operation.

Theorem 2.14 *Given h_1 and h_2 two legal histories. Then, we have:*

$$h_1; T^*(h_2, h_1) \equiv_{st} h_2; T^*(h_1, h_2)$$

PROOF. Assume that $h_1; T^*(h_2, h_1)$ and $h_2; T^*(h_1, h_2)$ are legal and $|h_1| = |T^*(h_1, h_2)|$ (resp. $|h_2| = |T^*(h_2, h_1)|$). Let n and m be the lengths of h_1 and h_2, respectively. We proceed by double induction on n and m.
Basis step: If $n = 0$ or $m = 0$ the result is trivial.
Induction hypothesis: for $n \geq 0$ and $m \geq 0$, $h_1; T^*(h_2, h_1) \equiv h_2; T^*(h_1, h_2)$.
Induction step: Let $n+1$ and $m+1$ be the lengths of h'_1 and h'_2, respectively, where $h'_1 = (op_1; h_1)$ and $h'_2 = (op_2; h_2)$ for some operations op_1 and op_2. Assume that h'_1 and h'_2 are legal. Let $H_1 = h'_1; T^*(h'_2, h'_1)$ and $H_2 = h'_2; T^*(h'_1, h'_2)$:

$$\begin{array}{lll}
H_1 & = & op_1; h_1; T^*(op_2; h_2, op_1; h_1) \\
 & & \text{[by rewriting } h'_1 \text{ and } h'_2] \\
 & = & op_1; h_1; T^*(T(op_2, op_1), h_1); \\
 & & \quad T^*(T^*(h_2, T(op_1, op_2)), T^*(h_1, T(op_2, op_1))) \\
 & & \text{[Definition of } T^*] \\
 & \equiv_{st} & op_1; T(op_2, op_1); T^*(h_1, T(op_2, op_1)); \\
 & & \quad T^*(T^*(h_2, T(op_1, op_2)), T^*(h_1, T(op_2, op_1))) \\
 & & \text{[Induction hypothesis and } C_1] \\
H_2 & = & op_2; h_2; T^*(op_1; h_1, op_2; h_2) \\
 & & \text{[by rewriting } h'_1 \text{ and } h'_2] \\
 & = & op_2; h_2; T^*(T(op_1, op_2), h_2); \\
 & & \quad T^*(T^*(h_1, T(op_2, op_1)), T^*(h_2, T(op_1, op_2))) \\
 & & \text{[Definition of } T^*] \\
 & \equiv_{st} & op_2; T(op_1, op_2); T^*(h_2, T(op_1, op_2)); \\
 & & \quad T^*(T^*(h_1, T(op_2, op_1)), T^*(h_2, T(op_1, op_2))) \\
 & & \text{[Induction hypothesis and } C_1]
\end{array}$$

We can conclude that $H_1 \equiv_{st} H_2$ by using condition C_1 and induction hypothesis. □

Theorem 2.15 *Given h_1, h_2 and h_3 three legal histories. Then, we have:*

$$T^*(h_3, h_1; T^*(h_2, h_1)) = T^*(h_3, h_2; T^*(h_1, h_2))$$

PROOF. Let n, m and p be the lengths of h_1, h_2 and h_3, respectively. We proceed by triple induction on n, m and p.
Basis step: If $n = 0$, $m = 0$ or $p = 0$ the result is trivial.
Induction hypothesis: for $n \geq 0$, $m \geq 0$ and $p \geq 0$ $T^*(h_3, h_1; T^*(h_2, h_1)) = T^*(h_3, h_2; T^*(h_1, h_2))$.
Induction step: Let $n+1$, $m+1$ and $p+1$ be the lengths of h'_1, h'_2 and h'_3, respectively, where $h'_1 = (op_1; h_1)$, $h'_2 = (op_2; h_2)$ and $h'_3 = (op_3; h_3)$ for

some operations op_1, op_2 and op_3. Let $H_1 = T^*(h'_3, h'_1; T^*(h'_2, h'_1))$ and $H_2 = T^*(h'_3, h'_2; T^*(h'_1, h'_2))$.

By using definition of T^* and Theorem 2.14, $H_1 = H'_1; H''_1$ where :

$$\begin{array}{rcl}
H'_1 & = & T^*(T(T(op_3, op_1), T(op_2, op_1)), T^*(h_1, T(op_2, op_1)); \\
 & & \quad T^*(T^*(h_2, T(op_1, op_2)), T^*(h_1, T(op_2, op_1)))) \\
H''_1 & = & T^*(T^*(h_3, T(op_1, op_3); T(T(op_2, op_3), T(op_1, op_3))), \\
 & & \quad T^*(T^*(h_1, T(op_2, op_1)), T(op_3, op_1; T(op_2, op_1))); \\
 & & \quad T^*(T^*(T^*(h_2, T(op_1, op_2)), T(op_3, op_1; T(op_2, op_1))), \\
 & & \qquad T^*(T^*(h_1, T(op_2, op_1)), T(op_3, op_1; T(op_2, op_1)))))
\end{array}$$

In the same way, by using definition of T^* and Theorem 2.14, $H_2 = H'_2; H''_2$ where :

$$\begin{array}{rcl}
H'_2 & = & T^*(T(T(op_3, op_2), T(op_1, op_2)), T^*(h_2, T(op_1, op_2)); \\
 & & \quad T^*(T^*(h_1, T(op_2, op_1)), T^*(h_2, T(op_1, op_2)))) \\
H''_2 & = & T^*(T^*(h_3, T(op_2, op_3); T(T(op_1, op_3), T(op_2, op_3))), \\
 & & \quad T^*(T^*(h_2, T(op_1, op_2)), T(op_3, op_2; T(op_1, op_2))); \\
 & & \quad T^*(T^*(T^*(h_1, T(op_2, op_1)), T(op_3, op_2; T(op_1, op_2))), \\
 & & \qquad T^*(T^*(h_2, T(op_1, op_2)), T(op_3, op_2; T(op_1, op_2)))))
\end{array}$$

Using condition C_2 and induction hypothesis, we can conclude that $H'_1 = H'_2$ and $H''_1 = H''_2$. □

Using the function T^* (the extended definition of T), combined with Theorems 2.14 and 2.15, we provide an interesting procedure for building more complex scenarios in distributed groupware systems based on OT approach. However, this procedure is useless if the OT algorithm does not satisfy the convergence conditions. Proving the correctness of OT algorithms, *w.r.t* C_1 and C_2 is very complex and error prone even on a simple string object. Consequently, to be able to develop the transformational approach and to safely use it in other replication-based distributed systems with simple or more complex objects, proving conditions on OT algorithms must be assisted by an automatic theorem prover. In this respect, we present in this work a formal framework for correctly designing OT algorithms.

3 Formal Specification

We present in this section the theoretical background of our framework. We first briefly review the basics of algebraic specification. Then, we give the ingredients of our formalization for specifying and reasoning on OT algorithms.

We assume that the reader is familiar with the basic concepts of algebraic specification [25], term rewriting and equational reasoning [23]. A *many-sorted signature* Σ is a pair (S, F) where S is a set of *sorts* and F is a $S^* \times S$-sorted set (of *function symbols*). Here, S^* is the set of finite (including empty) sequences of elements of S. Saying that $f : s_1 \times \ldots \times s_n \to s$ is in $\Sigma = (S, F)$ means that $s_1 \ldots\ s_n \in S^*$, $s \in S$, and $f \in F_{s_1 \ldots s_n, s}$. We assume that we have a partition of F in two subsets: the first one C contains the *constructor symbols* and the second one D is the set of *defined symbols*, such that C and D are disjoint. Let X be a family of sorted variables and let $T(F, X)$ be the set of sorted terms. When a term does not contain variables, it is called *ground* term. The set of all ground terms is $T(F)$.

A *substitution* η assigns terms of appropriate sorts to variables. If t is a term, then $t\theta$ denotes the application of substitution θ to t. If η applies every variable to ground term, then η is a ground substitution. We denote by $\equiv$ the syntactic equivalence between objects. An *equation* is a formula of the form $l = r$. A *conditional* equation is a formula of the following form: $\bigwedge_{i=1}^{n} a_i = b_i \implies l = r$. It will be written $\bigwedge_{i=1}^{n} a_i = b_i \implies l \to r$ and called a *conditional rewrite rule* when using an order on terms. The term l is the *left-hand side* of the rule. A set of conditional rewrite rules is called a *rewrite system.* A constructor is *free* if it is not the root of a left-hand side of a rule. A term is *strongly irreducible* if none of its non-variable subterms matches a left-hand side of a rule in a rewrite system. A symbol $f \in F$ is *completely defined* if all ground terms with root f are reducible to terms in $T(C)$. A rewrite system is *sufficiently complete* if all symbols in D are completely defined.

An *algebraic specification* is a pair $(\Sigma, \mathcal{A})$ where Σ is a many-sorted signature and $\mathcal{A}$ is a rewrite system called the set of *axioms* of $(\Sigma, \mathcal{A})$. A *clause* Δ is an expression of the form: $\bigwedge_{i=1}^{n} a_i = b_i \implies \bigvee_{j=1}^{m} a'_j = b'_j$. The clause Δ is a *Horn clause* if $m \leqslant 1$. The clause Δ is a *logical consequence* of $\mathcal{A}$ if Δ is valid in any model of $\mathcal{A}$, denoted by $\mathcal{A} \models \Delta$. The clause Δ is said to be *inductively valid* in $\mathcal{A}$, denoted by $\mathcal{A} {\models}_{Ind} \Delta$, if for any ground substitution σ:

$$[(\text{for all i, } \mathcal{A}\models a_i\sigma = b_i\sigma) \text{ implies (there exists } j,\ \mathcal{A}\models a'_j\sigma = b'_j\sigma)].$$

For instance, consider the following algebraic specification on the natural numbers: $S = \{Nat\}$, the set of constant constructor symbols is $C_{\epsilon,Nat} = \{0 :\to Nat\}$, the set of non constant constructor symbols is $C_{Nat,Nat} = \{succ : Nat \to Nat\}$, the set of defined function symbols is $D_{Nat\,Nat,Nat} = \{+ : Nat \times Nat \to Nat\}$ and the set of axioms $\mathcal{A} = \{0 + x = x, succ(x) + y = succ(x + y)\}$. The set C is used to define every term in $T(F)$, *i.e.* using axioms of $\mathcal{A}$ we can replace the term $0 + (succ(0) + 0)$ by the term $succ(0)$. So we can state

$0 + (succ(0) + 0) = succ(0)$ is a logical consequence of $\mathcal{A}$ whereas $x + 0 = 0$ is not a logical consequence (but an inductive consequence) of $\mathcal{A}$.

An *observational signature* is a many-sorted signature $\Sigma = (S, S_{obs}, F, X)$ where $S_{obs} \subseteq S$ is the set of *observable* sorts. An *Observational Specification* is a pair $(\Sigma, \mathcal{A})$ where Σ is an observational signature and $\mathcal{A}$ is a set of axioms.

3.2 Replica Specification

The main component in replication-based distributed groupware system is the replica. Every replica has a set of operations. The *methods* are operations which modify the replica state. The *attributes* are operations which extract informations from the replica state. In some cases, the replica state is small, like a text document. In other cases, it can be large, like a database, an XML tree or a filesystem. So, representing and directly reasoning on the replica state is an expensive task and requires an expertise for proving properties relevant to the replica structure. In this work, we use an observational technique which conceals the internal state of the replica by extracting only relevant information from the sequence of methods executed on it.

We use the `State` sort for representing the domain of replica state. This sort has two constructor functions: (i) the constant constructor S_0 (the initial state), and (ii) a constructor Do which given a method and a state gives the resulting state provided that the execution of this method is possible. The sort `Meth` represents the set of methods. Every method type has its own constructor. These constructors are free since methods are assumed to be distinct. For every method, we should indicate conditions under which it is enabled. For this we use a boolean function $Poss$ defined by a set of conditional equations. We introduce a constant constructor Nop to represent an *idle* method which has null effect on the replica state. As to attributes, we express them by monadic function symbols on the `State` sort. These attribute functions are used as observers and are inductively defined upon the `State` sort. The OT algorithm is denoted by the function symbol T which takes two methods as arguments and produces another method. We then formally define a replica specification:

Definition 3.1 (Replica Specification). *Given S the set of all sorts, $S_{bs} = \{\texttt{State}, \texttt{Meth}\}$ is the set of* basic sorts *and $S_{ds} = S \setminus S_{bs}$ is the set of* data sorts. *A* replica specification *RS is an observational specification $(\Sigma^{RS}, \mathcal{A}^{RS})$ such that:*

(1) Σ^{RS} is an observational signature which has a single non-observable sort `State`. *The set of function symbols F is defined as $C \cup D$ such that:*

(a) $C_{\epsilon,\texttt{State}} = \{S_0\}$*,* $C_{\texttt{Meth State,State}} = \{Do\}$ *and* $C_{\omega,s} = \emptyset$ *for all other cases where* $\omega \in S^*_{bs}$ *or* s *is* `State`*;*

(b) $D_{\texttt{Meth Meth,Meth}} = \{T\}$*,* $D_{\texttt{Meth State,Bool}} = \{Poss\}$ *and* $D_{\omega,s} = \emptyset$ *for all other cases where* $\omega \in S^*_{bs}$ *and* $s \in S_{bs}$*.*

(2) The set of axioms (written as conditional equations) $\mathcal{A}^{RS}$ *is the union of the following sets:*

(a) the set of method precondition axioms $\mathcal{D}_P$*;*

(b) the set of attribute axioms $\mathcal{D}_A$*;*

(c) the set of axioms defining $\mathcal{D}_T$ *the transformation function* T*.*

Example 3.2 *Consider the group text editor of Example 2.10. The replica string has two methods:* $Ins(p,c,pr)$ *and* $Del(p,pr)$*. We define two attributes: Length for extracting the length of the string and* Car *for giving the character of the string at given position and state. The replica specification is given in Figure 4. The set* $C_{\omega,\texttt{Meth}}$ *(*$\omega \in S^*_{ds}$*) contains all constructor methods which represents the method types of a replica. All the necessary conditions for executing a method are given by* $\mathcal{D}_P$ *(lines* $1-2$*). The set* $D_{\omega\ \texttt{State},s}$ *contains all replica attributes where* $\omega \in S^*_{ds}$ *and* $s \in S_{ds}$*.* $\mathcal{D}_A$ *is illustrated in lines* $3-9$*. Note that* Car *and* $Length$ *are defined at the initial state* S_0 *as follows: (i)* $Car(x,S_0) = null$ *where null represents the character null value; (ii)* $Length(S_0) = 0$*. Lines* $10-25$ *gives the equational definition of* T*.*

We will choose from all interpretations for the signature Σ^{RS}, the ones that reflect the desired properties described in our model of the distributed groupware system (see Subsection 2.2). We use an observational semantics which is based on weakening the satisfaction relation [4,2,8,7]. Informally speaking, the replica objects which cannot be distinguished by *experiments* are considered as observationally equal. When using algebraic specifications, such experiments can be formally defined by *contexts* of observable sorts and operators over the signature of the specification.

Definition 3.3 (Context). *Let* RS *be a replica specification and* $T^{RS}(F,X)$ *its term algebra.*

(1) A Σ^{RS}-context *of sort* `State` *is a term* $c \in T^{RS}(F,X)$ *with a distinguished linear variable* $z_{\texttt{State}}$ *of sort* `State`*. This variable is called the* context variable *of* c*. To indicate the context variable occurring in* c *we often write* $c[z_{\texttt{State}}]$*.*

(2) A Σ^{RS}*-context* c *is called an* observable Σ^{RS}-context *if the sort of* c *is in* S_{ds}*, and a* state Σ^{RS}-context *if the sort of* c *is* `State`*.*

(3) A Σ^{RS}*-context* c *is* appropriate *for a term* $t \in T^{RS}(F,X)$ *iff the sort of* t *matches that of* $z_{\texttt{State}}$*.* $c[t]$ *defines the replacement of* $z_{\texttt{State}}$ *by* t *in* $c[z_{\texttt{State}}]$*.*

(4) $ObsCt_{\Sigma^{RS}}$ *denotes the set of all observable* Σ^{RS}*-contexts.*

Sorts: `State`, `Meth`, `bool`, `nat`, `char`

Constructors

$S0$	:	$\to$ `State`
Do	:	`Meth` $\times$ `State` $\to$ `State`
Ins	:	`nat` $\times$ `char` $\times$ `nat` $\to$ `Meth`
Del	:	`nat` $\times$ `nat` $\to$ `Meth`
Nop	:	$\to$ `Meth`

Defined Operations

$Poss$	:	`Meth` $\times$ `State` $\to$ `bool`
$Length$	:	`State` $\to$ `nat`
Car	:	`nat` $\times$ `State` $\to$ `char`
T	:	`Meth` $\times$ `Meth` $\to$ `Meth`

Variables

x, $p1$, $p2$, $pr1$, $pr2$	:	`nat`;
st	:	`State`;
m	:	`Meth`;
$c1$, $c2$	:	`char`;

Axioms

1. $Poss(Ins(p1, c1, pr1), st) = (p1 \leq Length(st));$
2. $Poss(Del(p1, pr1), st) = (p1 < Length(st));$

3. $Length(Do(Ins(p1, c1, pr1), st)) = Length(st) + 1;$
4. $Length(Do(Del(p1, pr1), st)) = Length(st) - 1;$
5. $x = p1 \quad \Rightarrow \quad Car(x, Do(Ins(p1, c1, pr1), st)) = c1;$
6. $(x > p1) = true \quad \Rightarrow \quad Car(x, Do(Ins(p1, c1, pr1), st)) = Car(x - 1, st);$
7. $(x < p1) = true \quad \Rightarrow \quad Car(x, Do(Ins(p1, c1, pr1), st)) = Car(x, st);$
8. $(x \geq p1) = true \quad \Rightarrow \quad Car(x, Do(Del(p1), st)) = Car(x + 1, st);$
9. $(x < p1)) = true \quad \Rightarrow \quad Car(x, Do(Del(p1), st)) = Car(x, st);$

10. $T(Nop, m) = Nop;$
11. $T(m, Nop) = m;$
12. $(p1 < p2) = true \quad \Rightarrow \quad T(Ins(p1, c1, pr1), Ins(p2, c2, pr2)) = Ins(p1, c1, pr1);$
13. $(p1 > p2) = true \quad \Rightarrow \quad T(Ins(p1, c1, pr1), Ins(p2, c2, pr2)) = Ins(p1 + 1, c1, pr1);$
14. $p1 = p2 \wedge c1 = c2 \quad \Rightarrow \quad T(Ins(p1, c1, pr1), Ins(p2, c2, pr2)) = Nop;$
15. $p1 = p2 \wedge c1 \neq c2 \wedge (pr1 > pr2) = true \ \Rightarrow$
16. $\qquad T(Ins(p1, c1, pr1), Ins(p2, c2, pr2)) = Ins(p1 + 1, c1, pr1);$
17. $p1 = p2 \wedge c1 \neq c2 \wedge (pr1 < pr2) = true \ \Rightarrow$
18. $\qquad T(Ins(p1, c1, pr1), Ins(p2, c2, pr2)) = Ins(p1, c1, pr1);$
19. $(p1 < p2) = true \quad \Rightarrow \quad T(Ins(p1, c1, pr1), Del(p2, pr2)) = Ins(p1, c1, pr1);$
20. $(p1 \geq p2) = true \quad \Rightarrow \quad T(Ins(p1, c1, pr1), Ins(p2, c2, pr2)) = Ins(p1 - 1, c1, pr1);$
21. $(p1 < p2) = true \quad \Rightarrow \quad T(Del(p1, pr1), Del(p2, pr2)) = Del(p1, pr1);$
22. $(p1 > p2) = true \quad \Rightarrow \quad T(Del(p1, pr1), Del(p2, pr2)) = Del(p1 - 1, pr1);$
23. $p1 = p2 \quad \Rightarrow \quad T(Del(p1, pr1), Del(p2, pr2)) = Nop;$
24. $(p1 < p2) = true \quad \Rightarrow \quad T(Del(p1, pr1), Ins(p2, c2, pr2)) = Del(p1, pr1);$
25. $(p1 \geq p2) = true \quad \Rightarrow \quad T(Del(p1, pr1), Ins(p2, c2, pr2)) = Del(p1 + 1, pr1);$

Fig. 4. Replica specification for the group text editor GROVE [5].

A *State* Σ^{RS}-context can be regarded as a sequence of methods. An *observable* Σ^{RS}-context is the sequence formed by an attribute on the top of a *State* Σ^{RS}-context.

Example 3.4 *Consider the replica specification in Figure 4. There are infinitely many observable* Σ^{RS}*-contexts:* $Length(z_{\texttt{State}})$, $Car(x, z_{\texttt{State}})$, $Car(x, Do(Ins(p, c, pr), z_{\texttt{State}}))$, $\ldots$, $Length(Do^n(Del(p), z_{\texttt{State}}))$.

Our notion of observational validity is based on the idea that two replica objects in the given algebra are observationally equal if they cannot be distinguished by computation with observable results.

Definition 3.5 (Observational Validity). *Two terms* t_1 *and* t_2 *are* observationally equal *if for all* $c \in ObsCt_{\Sigma^{RS}}$ $\mathcal{A}^{RS} \models_{ind} c[t_1] = c[t_2]$. *We denote it by* $\mathcal{A}^{RS} \models_{obs} t_1 = t_2$ *or simply* $t_1 =_{obs} t_2$.

Theorem 3.6 *The relation* $=_{obs}$ *is a congruence on* $T(F)$.

The proof of Theorem 3.6 is given in [4].

Definition 3.7 (State Property). *Let* $P \equiv \bigwedge_{i=1}^{n} a_i = b_i \implies t_1 = t_2$. *We say that* P *is a* state property *(or observationally valid) and we denote it by* $\mathcal{A}^{RS} \models_{obs} P$ *if for all ground substitutions* σ, $(\forall\, i \in [1..n]\, \mathcal{A}^{RS} \models_{obs} a_i\sigma =_{obs} b_i\sigma)$ *implies* $\mathcal{A}^{RS} \models_{obs} t_1\sigma =_{obs} t_2\sigma$.

Our purpose is to propose a technique to prove and disprove (or refute) state properties. Note that our state properties are Horn clauses and therefore in the scope of observational properties mentionned in [4]. In this work, the authors have introduced the concept of *critical contexts*. These ones are sufficient to prove observational theorems by reasoning on the ground irreducible observable contexts rather than on the whole set of observable contexts. In the following, we denote by $\mathcal{R}$ a conditional rewrite system which is obtained by orienting the axioms of $\mathcal{A}^{RS}$.

Definition 3.8 (Inconsistent State Property). *We say that the state property* $P \equiv \bigwedge_{i=1}^{n} a_i = b_i \implies t_1 = t_2$ *is* provably inconsistent *iff there exists a substitution* σ *and a critical context* c *such that: (i)* $\forall\, i \in [1..n]$, $a_i\sigma = b_i\sigma$ *is an inductive theorem w.r.t.* $\mathcal{R}$, *and, (ii)* $c[t_1 = t_2]$ *is strongly irreducible by* $\mathcal{R}$.

Provably inconsistent state properties are not observationally valid when $\mathcal{R}$ is ground convergent. The computation of critical contexts requires that axioms are sufficiently complete [25]. More details on how to compute critical context and refute observational theorems can be found in [4].

In this work we rely on the inference I system proposed in [4]. This one consists of a set of transition rules applied to $(\mathcal{E},\mathcal{H})$, where $\mathcal{E}$ is the set of conjectures to prove and $\mathcal{H}$ is the set of induction hypotheses. Given a set of conditional rewriting rules, an *I-derivation* is a sequence of states: $(\mathcal{E}_0,\emptyset)\vdash_I(\mathcal{E}_1,\mathcal{H}_1)\vdash_I \ \ldots(\mathcal{E}_n,\mathcal{H}_n)$. An I-derivation *fails* when $\mathcal{E}_n$ is not empty and no rule can be applied to this set. An I-derivation *succeeds* if $\mathcal{E}_n$ is empty, *i.e.* all conjectures are proved. We consider this proof machinery as a function, denoted $\textsc{Proof}(E)$, which takes as argument a set of conjectures to be proved and returns a set of lemmas remaining to be proved in order to show E is observationally valid. Thus, if $\textsc{Proof}(E)$ returns an empty set then E is observationally valid.

4 Proving Convergence Properties

Before stating the properties that a replica object has to satisfy for ensuring convergence, we introduce some notations. Let m_1, m_2, $\ldots$, m_n and st be terms of sorts `Meth` and `State` respectively:

(1) As in Definition 2.4, we denote a sequence of methods (or history) as:

$$(st)m_1;m_2;\ldots;m_n \equiv Do(m_n,\ldots,Do(m_2,Do(m_1,st))\ldots)$$

(2) $Legal(m_1;m_2;\ldots;m_n,st) \equiv Poss(m_1,st) \wedge Poss(m_2,(st)m_1) \wedge \ldots \wedge Poss(m_n,(st)m_1;m_2;\ldots;m_{n-1})$.

(3) The expression $Occ(m,h)$ represents the number of occurrences of method m in history h.

We redefine our notion of history equivalence (see Definition 2.6) as follows:

Definition 4.1 (Equivalence of histories). *Given two histories h_1 and h_2. For every replica state st, we say that h_1 and h_2 are* equivalent *if the following conditions are satisfied:*

(1) $Legal(h_1,st)$ *and* $Legal(h_2,st)$ *are true;*
(2) $|h_1| = |h_2|$;
(3) $(st)h_1 =_{obs} (st)h_2$;
(4) $Occ(Nop,h_1) = Occ(Nop,h_2)$.

The fourth condition enables us to eliminate among equivalent histories the ones that do not have a practical interest, *i.e.* that represent scenarios that are not reachable in distributed groupware systems based on OT approach. For instance, consider Example 3.2. Both histories $(Ins(1,x,1);Nop)$ and $(Ins(1,x,2);Nop)$ are equivalent and they represent histories executed by sites 1 and 2 after broadcasting and transformation steps. On the other hand, the histories $(Ins(1,x,1);Del(1,1))$ and $(Nop;Nop)$ are not equivalent according to our definition (though they produce the same state) because this scenario is not possible.

In the following, we show how to express the satisfaction of conditions C_1 and C_2 as properties to be checked in our algebraic framework. Let $(\Sigma^{RS}, \mathcal{A}^{RS})$ and $\mathcal{M}^{RS}$ be a replica specification and the method set respectively, corresponding to a replica object RS.

4.1 Condition C_1

C_1 expresses a state identity between two method sequences. As mentioned before, we use an observational approach for comparing two states. Accordingly, we define the condition C_1 by the following state property (where the variable st is universally quantified):

$$\begin{aligned}\Phi_1(m_1, m_2) \equiv\ & (Legal(m_1; T(m_2, m_1), st) = true \wedge \\ & Legal(m_2; T(m_1, m_2), st) = true) \\ & \implies (st)m_1; T(m_2, m_1) =_{obs} (st)m_2\, T(m_1, m_2)\end{aligned} \tag{5}$$

The first convergence property is formulated as a conjecture to be proved from the replica specification. It means that: for all methods m_1 and m_2 and for every state st, such that m_1 and m_2 are enabled on st, then the states $((st)m_1; T(m_2, m_1))$ and $((st)m_2; T(m_1, m_2))$ are observationally equal. This conjecture is defined as follows:

Conjecture 1 (Convergence Property $CP1$). *A replica object RS satisfies the condition C_1 iff $\Phi_1(m_1, m_2)$ is a state property for all m_1 and m_2.*

According to Definition 4.1, the convergence property $CP1$ means that the histories $m_1; T(m_2, m_1)$ and $m_2; T(m_1, m_2)$ are equivalent.

Definition 4.2 (CP1-scenario). *A* $CP1$-scenario *is a triple (M_1,M_2,$\mathcal{E}$) where M_1 and M_2 are two methods and $\mathcal{E}$ is the set of conjectures generated by the function* PROOF$(\{\Phi_1(M_1, M_2\})$.

In Figure 5 we present an algorithm for verifying the convergence property $CP1$ by detecting all $CP1$-scenarios that violate this property. The $CP1$-scenarios simply consist of methods and conditions over argument methods which may lead to potential divergence situations.

Example 4.3 *Consider the group editor of Example 3.2. When applying our algorithm to replica specification of Figure 4, we have detected that convergence property $CP1$ is violated by giving the $CP1$-scenario depicted in Figure 6.*

Input : A replica specification RS.
Output : $\mathcal{S}$ a set of $CP1$-scenarios.

$\mathcal{S} \leftarrow \emptyset$;
foreach method M_1 in $\mathcal{M}^{RS}$
  foreach method M_2 in $\mathcal{M}^{RS}$
    $E \leftarrow \{\Phi_1(M_1, M_2)\}$;
    $E \leftarrow \textsc{Proof}(E)$;
    if $E \neq \emptyset$ **then** $\mathcal{S} \leftarrow \mathcal{S} \cup \{(M_1, M_2, E)\}$;
  endfor
endfor

Fig. 5. Algorithm for Checking Convergence Property $CP1$.

From this scenario, we can extract the following informations: (i) the methods ($Ins(u1, u2, u3)$ and $Del(u4, u5)$) that cause divergence problem; (ii) the observation (the attribute Car) that distinguishes the resulting states, and; (iii) the conditions over method arguments (Preconditions) which lead to divergence situation. The counter-example is simple (as illustrated in Figure 7; for clarity we have omitted the priority parameter): (i) $user_1$ inserts x in position 2 (op_1) while $user_2$ concurrently deletes the character at the same position (op_2). (ii) When op_2 is received by site 1, op_2 must be transformed according to op_1. So $T(Del(2), Ins(2, x))$ is called and $Del(3)$ is returned. (iii) In the same way, op_1 is received on site 2 and must be transformed according to op_2. $T(Ins(2, x), Del(2))$ is called and returns $Ins(3, x)$. Condition C_1 is violated. Accordingly, the final results on both sites are different.

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

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

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

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

Preconditions:
(u1 <= Length(u5))=true /\
(u4 < Length(u5))=true /\
u1 = u4;
```

Fig. 6. Output of our algorithm.

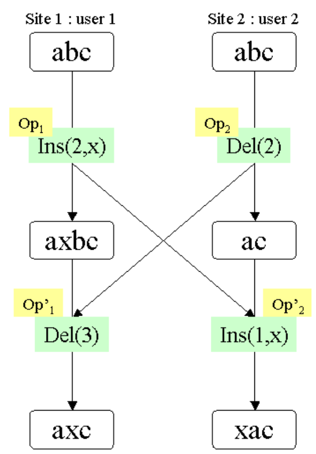

Fig. 7. Scenario violating $CP1$.

The error comes from the definition of $T(Ins(p1, c1, pr1), Del(p2, pr2))$. The

Input : A replica specification RS.
Output : $\mathcal{S}$ a set of $CP2$-scenarios.

$\mathcal{S} \leftarrow \emptyset$;
foreach method M_1 in $\mathcal{M}^{RS}$
  foreach method M_2 in $\mathcal{M}^{RS}$
    foreach method M_3 in $\mathcal{M}^{RS}$
      $E \leftarrow \{\Phi_2(M_1, M_2, M_3)\}$;
      $E \leftarrow \textsc{Proof}(E)$;
      if $E \neq \emptyset$ **then** $\mathcal{S} \leftarrow \mathcal{S} \cup \{(M_1, M_2, M_3, E)\}$;
    endfor
  endfor
endfor

Fig. 8. Checking Algorithm of Convergence Property $CP2$.

condition $p_1 < p_2$ should be rewritten $p_1 \leq p_2$. Other bugs have been detected in other string-based group editors [16,20]. More details can be found in [10].

4.2 Condition C_2

C_2 stipulates a *method identity* between two equivalent sequences. Given three methods m_1, m_2 and m_3, transforming m_3 with respect to two histories $(m_1; T(m_2, m_1))$ and $(m_2; T(m_1, m_2))$ must give the same method. We define C_2 by the following property:

$$\Phi_2(m_1, m_2, m_3) \equiv T^*(m_3, m_1; T(m_2, m_1)) = T^*(m_3, m_2; T(m_1, m_2))$$

The second convergence property is formulated as a conjecture to be proved from the replica specification.

Conjecture 2 (Convergence Property $CP2$). *A replica object RS satisfies the condition C_2 iff: $\mathcal{A}^{RS} \models_{obs} \Phi_1(m_1, m_2, m_3)$ for all methods m_1, m_2 and m_3.*

Definition 4.4 ($CP2$-scenarios). *A $CP2-$*scenario *is represented by a quadruple $(M_1, M_2, M_3, \mathcal{E})$ where M_1, M_2 and M_3 are three methods and $\mathcal{E}$ is the set of conjectures obtained by the function* $\textsc{Proof}(\Phi_2(M_1, M_2, M_3))$.

A $CP2$-scenario simply gives methods and conditions that may lead to potential divergence situations. In Figure 8, we present an algorithm for checking the convergence property $CP2$.

Example 4.5 *Consider the replica specification of Figure 4 with the modifi-*

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

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

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

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

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

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

Fig. 9. Output of our algorithm.

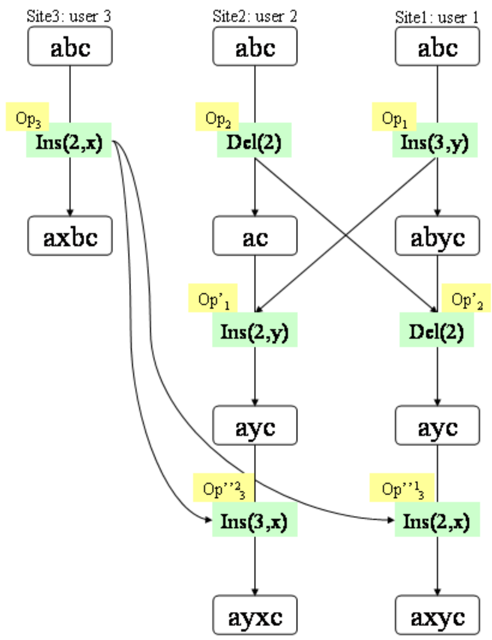

Fig. 10. Scenario violating $CP2$.

cation regarding T for satisfying the convergence property $CP1$ (see Example 4.3). Using our algorithm, we have detected that convergence property $CP2$ is not satisfied. In Figure 9 we give one of the $CP2$-scenarios output by our algorithm. When analyzing this scenario, we notice that transforming op1 along sequences $S1$ and $S2$ produces different methods (i.e. $Ins(u1+1, u2, u3) \neq Ins(u1, u2, u3)$. There is a divergence problem caused by the triple (Ins, Del, Ins). Consider for instance in Figure 10, three sites 1, 2, 3 start from the same initial state "abc". They generate operations $op_1 = Ins(3, y, 1)$, $op_2 = Del(2, 2)$ and $op_3 = Ins(2, x, 3)$ concurrently, which change their states to "abyc", "ac" and "axbc" respectively. At site 1, when op_2 is received, it is transformed against op_1 resulting in $op'_2 = Del(2, 2)$. After executing op'_2 the state becomes "ayc". When op_3 arrives, it is transformed against $op_1; op'_2$ resulting in $op''^1_3 = Ins(2, x, 3)$ whose execution leads to state "axyc".

At site 2, op_1 arrives first and is transformed against op_2 resulting in $op'_1 = Ins(2, y, 1)$. After op'_1 is executed, the state becomes "ayc". And when op_3 arrives it is transformed first against op_2 resulting in $op'^2_3 = Ins(2, x, 3)$. Then op'^2_3 is transformed against op'_1. Since the priority of op'^2_3 is greater than that of op'_1, it is shifted and we obtain $op''^2_3 = Ins(3, x, 3)$. After executing $op''^2_3 = Ins(3, x, 3)$, the state of site 2 becomes "ayxc" which is not identical to the state ("axyc") of site 1. Consequently, this OT algorithm does not verify convergence property $CP2$.

5 Implementation

We have implemented the observational approach in our tool `VOTE` (Validation of Operational Transformation Environment) [11]. This tool is designed to automatically check the convergence properties $CP1$ and $CP2$. It builds an algebraic specification based on conditional equations. As a verification back-end (implementing the PROOF function) we use `SPIKE` [4], an automated induction-based theorem prover. `SPIKE` was employed for the following reasons: (i) its high automation degree; (ii) its ability to perform case analysis (to deal with multiple methods and many transformation cases); (iii) its ability to find counter-examples; (iv) its incorporation of *decision procedures* (to automatically eliminate arithmetic tautologies produced during the proof attempt) [1].

When `SPIKE` is called, either the convergence properties proof succeed and OT algorithm is validated, or the `SPIKE`'s proof-trace is used for extracting all scenarios which may lead to potential divergence situations. There are two possible scenarios: the first one is meaningless because conjectures are valid but it comes from a failed proof attempt by `SPIKE` [3]. Such cases can be overcome by simply introducing new lemmas. The second one concerns cases violating convergence properties. `VOTE` gives all necessary informations (methods and conditions) to understand the divergence origin. Consequently, these informations help developer to correct its OT algorithm.

We have detected a lot of bugs in well-known group editors such that GROVE [5], Joint Emacs [16], REDUCE [4] [20], SAMS [5] [15] and CoWord [6] [21] which are based on transformational approach for maintaining consistency of shared data. The results of our experiments are reported in Table 1. GROVE, Joint Emacs, REDUCE and CoWord are group text editors whereas SAMS is an XML document-based group editor. The system $So6$ [7] is a file synchronizer which uses an OT algorithm for synchronizing many file system replicas [14].

6 Conclusion

We have presented our formal approach which is intended to automatically detect copies divergence in distributed groupware systems. To meet conver-

[3] like $Car((p+1)-1, st) = Car((p-1)+1, st)$.

[4] `http://www.cit.gu.edu.au/~scz/projects/reduce`

[5] `http://woinville.loria.fr/sams`

[6] `http://www.cit.gu.edu.au/~scz/projects/coword/`

[7] `http://libresource.inria.fr/projects/so6`

Groupware Systems	C_1	C_2
GROVE	violated	violated
Joint Emacs	violated	violated
REDUCE	correct	violated
CoWord	correct	violated
SAMS	correct	violated
*So*6	correct	violated

Table 1
Case studies.

gence requirement, the OT algorithm of these systems must be checked *w.r.t.* the convergence conditions C_1 and C_2. This task is difficult – even impossible – to carry out by hand due to the numerous cases to test. To overcome this problem, we have proposed an algebraic framework to assist the design of correct OT algorithms. Thanks to our framework, we have detected bugs in many well-known systems. So, we think that our approach is very valuable because: (i) it can help significantly to increase confidence in an OT algorithm; (ii) having the theorem prover ensures that all cases are considered and quickly produces counter-example scenarios; (iii) formalization is very easy and effortless. A drawback of this framework is that the user have to identify which set of characteristics gives a complete observation of the replica object. However, this can also be viewed as an advantage because the complexity of the proof is highly reduced.

Future work. Many features are planned to be investigated effectively with large systems. We plan to ensure the correct composition of OT algorithms for handling composed objects. Finally, we intend to integrate in our framework the generation of Java classes from correct OT algorithms.

References

[1] Armando, A., Rusinowitch, M., Stratulat, S.: Incorporating Decision Procedures in Implicit Induction, *Journal of Symbolic Computation*, **34**(4), 2001, 241–258.

[2] Bidoit, M., Hennicker, R., Wirsing, M.: Behavioural and abstractor specifications, *Science of Computer Programming*, **25**(2-3), 1995, 149–186.

[3] Bouhoula, A., Kounalis, E., Rusinowitch, M.: Automated Mathematical Induction, *Journal of Logic and Computation*, **5(5)**, 1995, 631–668.

[4] Bouhoula, A., Rusinowitch, M.: Observational proofs by rewriting, *Theoretical Computer Science*, **275**(1–2), 2002, 675–698.

[5] Ellis, C. A., Gibbs, S. J.: Concurrency Control in Groupware Systems, *SIGMOD Conference*, 18, 1989.

[6] Ellis, C. A., Gibbs, S. J., Rein, G.: Groupware: some issues and experiences, *Commun. ACM*, **34**(1), 1991, 39–58, ISSN 0001-0782.

[7] Goguen, J., Lin, K., Roşu, G.: Circular Coinductive Rewriting, *Proceedings, 15th International Conference on Automated Software Engineering (ASE'00)*, Institute of Electrical and Electronics Engineers Computer Society, 2000, Grenoble, France, 11-15 September 2000.

[8] Goguen, J., Malcolm, G.: A hidden agenda, *Theoretical Computer Science*, **245**(1), 2000, 55–101.

[9] Imine, A., Molli, P., Oster, G., Rusinowitch, M.: Development of Transformation Functions Assisted by a Theorem Prover, *Fourth International Workshop on Collaborative Editing (ACM CSCW'02), Collaborative Computing in IEEE Distributed Systems Online*, November 2002.

[10] Imine, A., Molli, P., Oster, G., Rusinowitch, M.: Proving Correctness of Transformation Functions in Real-Time Groupware, *in 8th European Conference of Computer-supported Cooperative Work*, Helsinki, Finland, 14.-18. September 2003.

[11] Imine, A., Molli, P., Oster, G., Urso, P.: VOTE: Group Editors Analyzing Tool, *Electronic Notes in Theoretical Computer Science* (I. Dahn, L. Vigneron, Eds.), 86, Elsevier, 2003.

[12] Imine, A., Urso, P.: Automatic Detection of Copies Divergence in Collaborative Editing Systems, *Electronic Notes in Theoretical Computer Science* (T. Arts, W. Fokkink, Eds.), 80, Elsevier, 2003.

[13] Lushman, B., Cormack, G. V.: Proof of correctness of Ressel's adOPTed algorithm, *Information Processing Letters*, **86**(3), 2003, 303–310.

[14] Molli, P., Oster, G., Skaf-Molli, H., Imine, A.: Using the transformational approach to build a safe and generic data synchronizer, *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, ACM Press, 2003, ISBN 1-58113-693-5.

[15] Molli, P., Skaf-Molli, H., Oster, G., Jourdain, S.: SAMS: Synchronous, Asynchronous, Multi-Synchronous Environments, *The Seventh International Conference on CSCW in Design*, Rio de Janeiro, Brazil, September 2002.

[16] Ressel, M., Nitsche-Ruhland, D., Gunzenhauser, R.: An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors, *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'96)*, Boston, Massachusetts, USA, November 1996.

[17] Shen, H., Sun, C.: Flexible Merging for Asynchronous Collaborative Systems, *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, Springer-Verlag, 2002, ISBN 3-540-00106-9.

[18] Suleiman, M., Cart, M., Ferrié, J.: Concurrent Operations in a Distributed and Mobile Collaborative Environment, *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, IEEE Computer Society, 1998, ISBN 0-8186-8289-2.

[19] Sun, C., Ellis, C.: Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements, *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, ACM Press, 1998, ISBN 1-58113-009-0.

[20] Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving Convergence, Causality-preservation and Intention-preservation in real-time Cooperative Editing Systems, *ACM Transactions on Computer-Human Interaction (TOCHI)*, **5**(1), March 1998, 63–108, ISSN 1073-0516.

[21] Sun, D., Xia, S., Sun, C., Chen, D.: Operational Transformation for Collaborative Word Processing, *to appear in Proceedings of ACM 2004 Conference on Computer Supported Cooperative Work, Nov 6-10, Chicago, IL USA.*

[22] Tanenbaum, A. S.: *Distributed operating systems*, Prentice-Hall, Inc., 2002, ISBN 0-13-219908-4.

[23] Terese: *Term Rewriting Systems*, Cambridge University Press, 2003.

[24] Vidot, N., Cart, M., Ferrié, J., Suleiman, M.: Copies convergence in a distributed real-time collaborative environment, *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'00)*, Philadelphia, Pennsylvania, USA, December 2000.

[25] Wirsing, M.: Algebraic Specification, *Handbook of theoretical computer science (vol. B): formal models and semantics*, 1990, 675–788.