INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Proving correctness of transformation functions in collaborative editing systems

Gérald Oster — Pascal Urso — Pascal Molli — Abdessamad Imine

N° 5795

Décembre 2005

Thèmes COG et SYM

ISSN 0249-6399 ISRN INRIA/RR--5795--FR+ENG

Proving correctness of transformation functions in collaborative editing systems

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° 5795 — Décembre 2005 — 45 pages

Abstract: Operational transformation (OT) is an approach which allows to build real-time groupware tools. This approach requires *correct* transformation functions regarding two conditions called TP_1 and TP_2. Proving *correctness* of these transformation functions is very complex and error prone. In this paper, we show how a theorem prover can address this serious bottleneck. To validate our approach, we verified correctness of state-of-art transformation functions defined on strings of characters with surprising results. Counter-examples provided by the theorem prover helped us to design the tombstone transformation functions. These functions verify TP_1 and TP_2, preserve intentions and ensure multi-effect relationships.

Key-words: Collaborative editing, Operational Transformation, Optimistic replication, Formal methods

[*] GlobIS Group, Department of Computer Science, ETH Zurich, osterg@inf.ethz.ch
[†] ECOO Team, INRIA Lorraine, urso@loria.fr and molli@loria.fr
[‡] CASSIS Team, INRIA Lorraine, imine@loria.fr

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

Conception et vérification des fonctions de transformation pour les systèmes d'édition collaborative

Résumé : L'approche des transformées opérationelles (OT) permet de construire des éditeurs collaboratifs temps réel. Ce modèle repose sur l'utilisation de fonctions de transformation qui doivent satisfaire deux conditions dénommées respectivement TP_1 et TP_2. Il est difficile et long de vérifier la satisfaction de ces deux conditions. Pour répondre à ce problème, nous utilisons, dans cet article, une approche reposant sur un démonstrateur de théorème. Afin de valider notre approche, nous avons spécifié les fonctions de transformation, issues de l'état de l'art, définies pour une chaîne de caractères. Nous avons vérifié leur correction et nous avons trouvé des contre-exemples pour toutes les propositions. Au regard de ces contre-exemples fournis par notre démonstrateur, nous avons défini de nouvelles fonctions de transformation dénommées "Tombstone Transformation Functions". Ces fonctions vérifient les conditions TP_1 et TP_2, préservent les intentions et également les relations "multi-effets".

Mots-clés : Edition collaborative, Transformées opérationnelles, Réplication optimiste, Calcul formel

1 Introduction

A real time group editor[3, 23] allows multiple users to edit the same document 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 is no locking or serialization protocols. Any user can edit the document at any time. If two users generate concurrent operations, the system has to ensure that replicas will converge while preserving effects of concurrent operations.

Operational transformation (OT) is an approach [3, 22] which allows to build real-time groupware like shared editors. Algorithms like adOPTed [17], GOTO [24], SOCT 2,3,4 [21, 25] are used to maintain consistency of shared data. These algorithms rely on the definition of transformation functions. If these functions are not correct regarding two specific conditions named TP_1 and TP_2 then these algorithms cannot ensure consistency of shared data.

Proving correctness of transformation functions even on a simple typed object like a string of characters is a complex task. Moreover, if we manage more complex typed objects with more operations defined on, the proofs are almost impossible to establish without a computer. This is a serious bottleneck for building more complex real-time groupware software.

In this paper, we use an automatic theorem prover to verify correctness of transformation functions. This approach allows us to quickly determine if proposed transformation functions are correct or not. Theorem proving requires to specify formally transformation functions. However, if hand-written proving is error-prone, specifying can be also error-prone. Nevertheless, even if transformation functions are relatively small, they generate huge number of cases. Consequently, it is easier to find an error in specification than in the proof.

First, in this paper, we describe how we can specify data types, transformation functions, and properties and how the proof process works. Next, to validate our approach, we formalized existing propositions and verified their correctness. All existing transformation functions are not correct. In this paper, we propose a new set of correct transformation functions called Tombstones Transformation Functions (TTF). TTF verify TP_1 and TP_2 properties, verify intention preservation as defined by Sun et al. [24] and also verify the CSM model as defined by Li et al. [9].

The paper is organized as follows. Section 2 introduces the operational transformation model as defined by Ressel [17]. Section 3 describes the principles of our verification process. Section 4 details state-of-art transformation functions defined on a string of characters, and gives the counter-examples we found. In section 5, we describe the Tombstones Transformation Functions that verify TP_1 and TP_2 conditions. Section 6 evaluates the TTF functions in the two others OT models and presents open issues. Finally, section 7 concludes the paper with some final remarks.

2 OT approach: The Ressel's model

We use the model of Operational Transformation defined by Ressel [17]. It considers n sites, each site owns a copy of shared data. When a site performs an update, it generates a corresponding operation. Every operation is processed in four steps: (i) executed on one site, (ii) broadcasted to other sites, (iii) received by other sites, (iv) executed on other sites.

OT frameworks distinguish two main components:

- an *integration algorithm*. This algorithm is in charge of reception, diffusion and execution of operations. When necessary, it calls transformation functions. This algorithm does not depend on type of replicated data ;
- a set of *transformation functions*. These functions merge concurrent modifications in serializing two concurrent operations. These functions are specific to a particular type of replicated data like string of characters, XML document or file system.

As every optimistic replication algorithms, OT approach ensures eventual consistency i.e. when the system is idle, all the copies converge to a same value.

2.1 Causality preservation

Considering two operations op_1 and op_2, operation op_1 is said to *precede* op_2 if and only if op_2 is generated on a copy after op_1 was executed on this copy. Subsequently, op_2 may depend on effects of execution of op_1. *Causality preservation* criterion ensures that all operations ordered by a precedence relation will be executed in the same order on every copy. Generally, this relation is maintained using state vectors [12, 4] associated to replicated objects and operations. It can also be maintained using continuous timestamps delivered by a sequencer like in Vidot et al. [25].

2.2 Copies convergence

When two operations are not causally connected by a precedence relation, they are concurrent. Two concurrent operations can be executed in different order on two different copies. Consequently, when an operation is received on one site, the current state of shared object may be different from the one where the operation has been generated. Thus, executing this operation in its generated form on a remote site, may not preserve its effects and the copies may not converge as indicated in figure 1(a).

In order to solve these consistency problems, Ellis et al. [3] introduced a transformation function T. This function is used to transform remote operations when they arrive

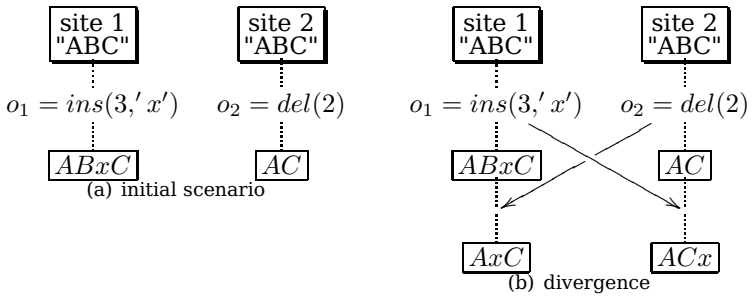


Figure 1: Divergence problem

on a site. Remotes operations are transformed regarding concurrent operations that were already executed on local copy.

For instance, if we consider our previous example, op_1 is not any more executed as soon as it arrives on site 2, but it is transformed regarding concurrent operations, in our case operation op_2. op_2 removed a character located before the insertion position of op_1. Thus, the insertion position of op_1 is decreased of one position to take into account previous execution of op_2. Consequently, on site 2, operation $op'_1 = ins(2, x)$ has to be executed (see figure 2). Intuitively, we can define the transformation indicated in figure 3.

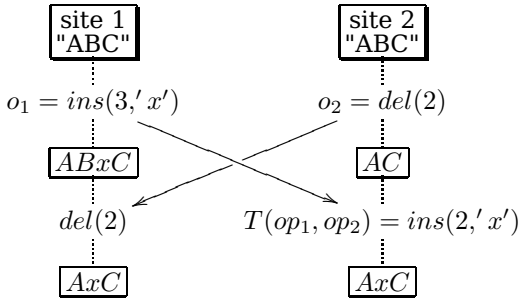


Figure 2: Convergence using transformation function

$T(Ins(p_1, c_1,), Del(p_2)) :-$
if $(p_1 < p_2)$ **return** $Ins(p_1, c_1)$
else return $Ins(p_1 - 1, c_1)$

Figure 3: One naive transformation function

Definition 2.1 (Transformation function T) *A transformation function T takes two concurrent operations as parameters. These two operations, namely op_1 and op_2, must be defined on a same state S. Transformation function computes a new operation $T(op_1, op_2)$ that is equivalent to op_1 – i.e. has the same effects – but defined on the state $S \odot op_2$. $S \odot op_2$ is the state resulting from the execution of op_2 on state S.*

Later, Ressel et al. [17] show that transformation functions that satisfy the two conditions TP_1 and TP_2 ensure convergence whatever reception order of concurrent operations.

Definition 2.2 (TP_1 condition) *For every two concurrent operations op_1 and op_2 defined on the same state, the transformation function T satisfy TP_1 condition if and only if :*

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

where $op_i \circ op_j$ denotes the sequence of operation formed of op_i followed by op_j ; and where $\equiv$ denotes equivalence of the two sequences of operations.

This first condition, TP_1, expresses an *equivalence* (noted $\equiv$) between two sequences. Each sequence consists of two operations. Given two concurrent operations op_1 and op_2, the execution of the sequence of op_1 followed by $T(op_2, op_1)$ on a state S must produce the same state as the execution of the sequence of op_2 followed by $T(op_1, op_2)$. Thus, TP_1 is equivalently expressed as :

$$\forall S, op_1, op_2 : S \odot op_1 \odot T(op_2, op_1) = S \odot op_2 \odot T(op_1, op_2)$$

Definition 2.3 (TP_2 condition) *For every three concurrent operations op_1, op_2 and op_3 defined on the same state, the transformation function T satisfy TP_2 condition if and only if :*

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

This second condition TP_2 stipulates an *equality* between two operations transformed with regards to two equivalent[1] sequences of operations. Given three operations op_1, op_2 and op_3, the transformation of op_3 with regards to the sequence formed by op_2 followed by $T(op_1, op_2)$ must give the same operation as the transformation of op_3 with regards to the sequence formed by op_1 followed by $T(op_2, op_1)$. By definition, expression $T(op_x, op_y \circ op_z)$ is also equals to expression $T(T(op_x, op_y), op_z)$. Therefore, TP_2 is equivalently expressed as :

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

[1] These two sequences are equivalent by transformation.

Ressel et al. [17] demonstrated these two conditions TP_1 and TP_2 are sufficient to ensure convergence of copies whatever the order in which concurrent operations are transformed.

Without a correct set of transformation functions, the integration algorithm cannot ensure consistency and the resulting groupware tools would not be reliable. Currently, as we demonstrate it in section 4, none transformation functions verifying TP_1 and TP_2 is known. The problem comes from the huge number of different cases to verify even in case small and simple transformation functions. With an automated theorem prover, we found an error in all existing transformation function. In the next section we describes how automated theorem proving works, and what are the benefits and liabilities of automated theorem proving for the OT approach.

3 Automated verification of OT functions

In this section, we present our framework to prove correctness of transformation functions regarding the TP_1 and TP_2 conditions. Our framework relies on an automated theorem prover which delivers a formal proof of correctness. The verification is entirely based on syntactic rules originating from mathematical logic. Such automation makes the validation process reliable and excludes careless mistakes. Also, validation performance are greatly improved by reducing this time consuming step.

Our validation process is composed of three steps :

1. A modelling step to formalize transformation functions we want to validate ;
2. A validation step to prove their correctness regarding the TP_1 and TP_2 conditions ;
3. An analysis step to interpret results obtained by the theorem prover.

If the theorem prover terminates with all conjectures proven, then the transformation functions are correct regarding the TP_1 and TP_2 conditions. On the contrary, there are two cases that can make the theorem prover stops with unproven conjectures :

- Some lemma or definition must be added to helps the prover to achieve;
- One or more counter-examples were found. In this case, transformation functions must be fixed regarding these scenarios.

In the following, we illustrate each step of the validation process. As an example, we take the transformation functions from Ellis et al. [3]. Hence, we consider a string of characters as shared object. This structure is a list of characters where positions in the list are mapped to indexes of characters in the string (positions start at index 1).

Two operations can update a string of characters :

- $Ins(p, c)$ to insert the character c at position p.
- $Del(p)$ to remove the character located at position p.

$T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) :-$
  if $(p_1 < p_2)$ **return** $Ins(p_1, c_1, pr_1)$
  else if $(p_1 > p_2)$ **return** $Ins(p_1 + 1, c_1, pr_1)$
  else if $(c_1 == c_2)$ **return** $Id()$
  else if $(pr_1 > pr_2)$ **return** $Ins(p_1 + 1, c_1, pr_1)$
  else return $Ins(p_1, c_1, pr_1)$

$T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) :-$
  if $(p_1 < p_2)$ **return** $Ins(p_1, c_1, pr_1)$
  else return $Ins(p_1 - 1, c_1, pr_1)$

$T(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1, pr_1)$
  else return $Del(p_1 + 1, pr_1)$

$T(Del(p_1, pr_1), Del(p_2, pr_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1, pr_1)$
  else if $(p_1 > p_2)$ **return** $Del(p_1 - 1, pr_1)$
  else return $Id()$

Figure 4: Transformation functions proposed by Ellis et al.

Figure 4 gives definition of transformation functions proposed by Ellis et al. [3]. Authors put an additional parameter pr on their operations. This parameter is a priority based on site identifier which is used to break the tie when conflicting situation occurs (i.e. two insertions at the same position).

3.1 Step 1: Formal modelling

There are two categories of theorem provers. On the one hand, proof assistants interact at many step of the proof with user, and help him to build a rigorous proof. On the other hand, automated theorem prover build a proof automatically without user action by applying prebuilt strategies. This kind of tools are particularly adequate to demonstrate proposition that are not complex but with combinatorial issues. This is the case with OT approach. Transformation functions are relatively small functions but generates a huge number of different cases. Verifying all cases by hand is error-prone. A hand-written proof was been made for [21] and in [8] but we found the counter-examples described in sections 4.4 and 4.5.

On the other hand, specifying is also error-prone. A specification error leads to claim proof of incorrect transformation functions. We made this mistake in [5]. In this paper, we correct the specification and found the counter-examples described in sections 4.3 and 4.4.

However, with a theorem prover, the proof is safe. Reviewers have only to check that the specification is correct. Then, users can concentrate on transformation functions and let the theorem prover finds counter examples. By this way, we can leverage the bottleneck of proof stage for the OT approach. The proof can scale. It means that we can handle more complex types or more operations on one type.

In our framework, we use the SPIKE [1, 19] automated theorem prover which builds proof by induction.

Theorem provers rely on a specification language based on a formal logic. They provide a support to verify properties expressed using logical formulas. Transformation functions presented in figure 4 are already expressed in a formal way. However, this formalism is not the same as the one used by SPIKE. SPIKE needs a formal specification expressed in terms of rewriting rules of first order equational logic [2]. We must translate specifications from one formalism to the other one. This translation is done automatically by a tool we have developed [6]. For instance, specification resulting from translation of Ellis's transformation functions is given in figure 5. This tools allows to express transformation function in a more human readable language. It also extracts counter example from the SPIKE proof trace.

3.1.1 Modelling of the TP_1 condition

The TP_1 condition defines a *sequence equivalence* for two concurrent operations op_1 and op_2 :

$$TP_1 : op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$$

where $\circ$ operator constructs a sequence of operations from two operations.

Two sequences are said equivalent (noted $\equiv$) if executed from the same state they give the same resulting state. This equivalence can be expressed formally as follows :

$$\begin{array}{l} \forall op_i \in Op, \forall op_j \in Op, \forall st \in State, \\ enabled(op_i, st) \wedge enabled(op_j, st) \wedge conc(op_i, op_j) \Rightarrow \\ \qquad (st \odot op_i) \odot T(op_j, op_i) = (st \odot op_j) \odot T(op_i, op_j) \end{array}$$

where :

- Op is the set of operations defined on a replicated object.
- $State$ is the set of states that can happen for a replicated object.

(RT1) $(p_1 < p_2) = true \Rightarrow T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = Ins(p_1, c_1, pr_1)$;
(RT2) $(p_1 < p_2) = false, (p_1 > p_2) = true \Rightarrow$
$T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = Ins(p_1 + s(0), c_1, pr_1)$;
(RT3) $(p_1 < p_2) = false, (p_1 > p_2) = false, c_1 = c_2 \Rightarrow$
$T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = Id$;
(RT4) $(p_1 < p_2) = false, (p_1 > p_2) = false, c_1 \neq c_2, (pr_1 > pr_2) = true \Rightarrow$
$T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = Ins(p_1 + s(0), c_1, pr_1)$;
(RT5) $(p_1 < p_2) = false, (p_1 > p_2) = false, c_1 \neq c_2, (pr_1 > pr_2) = false \Rightarrow$
$T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = Ins(p_1, c_1, pr_1)$;

(RT6) $(p_1 < p_2) = true \Rightarrow T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = Ins(p_1, c_1, pr_1)$;
(RT7) $(p_1 < p_2) = false \Rightarrow T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = Ins(p_1 - s(0), c_1, pr_1)$;

(RT8) $(p_1 < p_2) = true \Rightarrow T(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)) = Del(p_1, pr_1)$;
(RT9) $(p_1 < p_2) = false \Rightarrow T(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)) = Del(p_1 + s(0), pr_1)$;

(RT10) $(p_1 < p_2) = true \Rightarrow T(Del(p_1, pr_1), Del(p_2, pr_2)) = Del(p_1, pr_1)$;
(RT11) $(p_1 < p_2) = false, (p_1 > p_2) = true \Rightarrow T(Del(p_1, pr_1), Del(p_2, pr_2)) = Del(p_1 - s(0), pr_1)$;
(RT12) $(p_1 < p_2) = false, (p_1 > p_2) = false \Rightarrow T(Del(p_1, pr_1), Del(p_2, pr_2)) = Id$;

Figure 5: Transformation functions from Ellis and al. expressed in SPIKE formalism.

- $s \odot op$ denotes the state resulting from the execution of an operation op on a particular state s.
- Predicate $enabled(op_i, st)$ checks if preconditions of operation op_i are fulfilled, i.e. conditions under which operation op_i is allowed to be executed on state st. For instance, preconditions of operation $Ins(p_1, c_1, pr_1)$ must check if $1 < p_1 \leq length(st) + 1$ to ensure that an insertion can only be made at a proper location in the string of characters st.
- Predicate $conc(op_i, op_j)$ determines if two operations op_i et op_j can be concurrent. For instance, for two operations $op_1 = Ins(p_1, c_1, pr_1)$ and $op_2 = Ins(p_2, c_2, pr_2)$, this predicate must stipulate $pr_1 \neq pr_2$ because two concurrent operations are always generated on two different sites. In most of the specifications, including the TTF specification, we simply states that every pair of operation can be concurrent.

In order to prove that transformation functions satisfy the condition TP_1, we must be able to describe state st and all other states of the replicated object. One solution is to model this state using algebraic data types. But, structures of replicated objects can be as complex as an XML document modelled by an ordered tree where nodes are labelled with values. The algebraic specification for this kind of replicated object is too

complex. Consequently, proof of TP_1 condition implies to prove a lot of properties on the structure of the manipulated object ; these additional proofs increase considerably the time and the expertise required to achieve the proof. We have chosen another approach based on situation calculus [13]. We used it to prove TP_1 for transformation functions manipulating blocks of text, File system and XML data in [14]. Other approach can be used such as observational semantics as in [7]

In the situation calculus, the state of a replicated object is represented by a *situation* built by induction from operations that change the system. The set of situations is denoted Sit. Effects of an operation on a situation are seen through modifications of some characteristics. Each characteristic is modelled by an *observation function* obs_n where its last parameter is a situation st.

In our example, the replicated object is a string of characters. In order to describe its state, we define an observation function $car(i, st)$ which observes the i^{th} character of the string in the situation st. We define how this observation function perceives execution of each operation. In other words, in our example, how are perceived changes implied by execution of an operation $Ins(p, c, pr)$ or by an operation $Del(p, pr)$ on a situation st.

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

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

Figure 6: Definition of observation function $car(pos, st)$.

Figure 6 gives complete definition of the observation function $car(pos, st)$ regarding the two operations defined on a string of characters. $car'(n)/Ins(p, c, pr)$ denotes the function that computes the value $car'(n)$ of the observation function on the situation S_n regarding its value $car(n)$ on the previous situation S_{n-1} ; situation S_n is resulting from the execution of an operation $Ins(p, c, pr)$ on the situation S_{n-1}.

For instance, we consider the string S_{n-1}="abcd". We assume this string was updated by on operation ins(3,x), so it becomes S_n ="abxcd". Using our observation function, we can deduce which character is located at any position in the string S_n. If we watch at position 3, then the character is 'x'. If we watch at a position lower than 3, the insertion had no impact, and thus the character, we are looking for, is located at the same position in the string S_{n-1}. Finally, if we watch at a position higher than 3, the

insertion had an impact. Consequently, we have to decrease the position we are looking at of one position to find the right character in string S_{n-1}.

Figure 7 presents definition of the observation function translated in the formalism used by SPIKE. $S(0)$ means 1 in the Peano's arithmetic used in SPIKE.

(RO1) $n = p \Rightarrow car(n, xSt \odot Ins(p, c, pr)) = c$;
(RO2) $n \neq p, (n > p) = true \Rightarrow car(n, xSt \odot Ins(p, c, pr)) = car(n - s(0), xSt)$;
(RO3) $n \neq p, (n > p) = false \Rightarrow car(n, xSt \odot Ins(p, c, pr)) = car(n, xSt)$;

(RO4) $(n \geq p) = true \Rightarrow car(n, xSt \odot Del(p, pr)) = car(n + s(0), xSt)$;
(RO5) $(n \geq p) = false \Rightarrow car(n, xSt \odot Del(p, pr)) = car(n, xSt)$;

Figure 7: Observation function $car(pos, st)$ in SPIKE formalism.

Using observation functions, we define an equivalence between two situations.

Definition 3.1 (Equivalence of situations) *Two situations st_1 et st_2 are equivalent (denotes $=_{Obs}$) if and only if :*

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

where Obs denotes the set of all observation functions defined on shared object.

We give a definition for two equivalent sequences of operations.

Definition 3.2 (Equivalence of sequences of operation) *Two sequences of operations seq_1 and seq_2 are equivalent if and only if :*

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

Using definition 3.1, we express the TP_1 condition as the following conjecture :

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

This conjecture can be rewritten using definition 3.2 as follows :

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

As we define only one observation function in our example, to prove correctness of transformation functions regarding the TP_1 condition, our theorem prover has only to prove the following conjecture :

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

3.1.2 Modelling of the TP_2 condition

Modelling of the TP_2 condition is straightforward. The TP_2 condition is the following equality involving three concurrent operations op_1, op_2 and op_3 :

$$TP_2 : T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$$

By definition, for all operations op_i, op_j et op_k, transformation $T(op_i, op_j \circ op_k)$ is equals to $T(T(op_i, op_j), op_k)$. Consequently, TP_2 condition can be formally expressed as follows :

$\forall op_i \in Op, \forall op_j \in Op, \forall st \in Sit,$
$enabled(op_i, st) \wedge enabled(op_j, st) \wedge enabled(op_k, st) \wedge$
$conc(op_i, op_j, op_k) \Rightarrow$
$\quad T(T(op_k, op_i), T(op_j, op_i)) = T(T(op_k, op_j), T(op_i, op_j))$

Therefore, after translation of the TP_2 condition in the SPIKE formalism, SPIKE must prove the following conjecture :

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

3.2 Step 2 : Verification

In this section, we explain in details the proof of the TP_1 condition for transformation functions, proposed by Ellis et al. [3], as our theorem prover does it.

3.2.1 Generating conjectures

In section 3.1.1 we shown that our theorem prover has to prove the following conjecture in order to verify correctness of transformation functions regarding TP_1 condition :

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

SPIKE starts its proof by replacing induction variables with inductions terms in the conjecture presented above. In this conjecture, op_1, op_2 and p_x are the induction variables. op_1 and op_2 variables are substituted by the induction terms $Ins(p_i, c_i, pr_i)$ and $Del(p_i, pr_i)$. In producing all the combinations, our theorem prover has now to prove the four following conjectures :

(C1) $conc(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = true,$
$\quad enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Ins(p_2, c_2, pr_2), xSt) = true \Rightarrow$
$\quad\quad car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot T(Ins(p_2, c_2, pr_2), Ins(p_1, c_1, pr_1)))$
$\quad = car(p_x, xSt \odot Ins(p_2, c_2, pr_2) \odot T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)));$

(C2) $conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot T(Del(p_2, pr_2), Ins(p_1, c_1, pr_1)))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)))$;

(C3) $conc(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)) = true$,
$enabled(Del(p_1, pr_1), xSt) = true, enabled(Ins(p_2, c_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Del(p_1, pr_1) \odot T(Ins(p_2, c_2, pr_2), Del(p_1, pr_1)))$
$= car(p_x, xSt \odot Ins(p_2, c_2, pr_2) \odot T(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)))$;

(C4) $conc(Del(p_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Del(p_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Del(p_1, pr_1) \odot T(Del(p_2, pr_2), Del(p_1, pr_1)))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot T(Del(p_1, pr_1), Del(p_2, pr_2)))$;

We limit our study to the proof of the (C3) conjecture. Other conjectures are proven in the same manner.

3.2.2 Rewriting and simplification

During this step, SPIKE uses the rewriting rules related to definition of transformation functions and definition of observation functions. It simplifies current conjectures and derives new conjectures from current conjectures. The final result does not depend on the order in which rewriting rules are applied. Consequently, at each rewriting step, SPIKE can freely choose which rule to apply.

For instance, let's it apply rewriting rules (RT6) and (RT7) (see figure 5 page 10) coming from definition of the transformation function $T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2))$. Our current conjecture (C3) is rewritten into two new conjectures where changes are highlighted in bold :

(C5) $(\mathbf{p_1} < \mathbf{p_2}) = \mathbf{true}$, $conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot T(Del(p_2, pr_2), Ins(p_1, c_1, pr_1)))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot \mathbf{Ins(p_1, c_1, pr_1)})$;

(C6) $(\mathbf{p_1} < \mathbf{p_2}) = \mathbf{false}$, $conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot T(Del(p_2, pr_2), Ins(p_1, c_1, pr_1)))$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot \mathbf{Ins(p_1 - s(0), c_1, pr_1)})$;

Then, we apply rules (RT8) and (RT9) coming from definition of the transformation function $T(Del(p_2, pr_2), Ins(p_1, c_1, pr_1))$. The two conjectures presented above are rewritten into four new conjectures :

(C7) $(\mathbf{p_2} < \mathbf{p_1}) = \mathbf{true}$, $(p_1 < p_2) = true, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot \mathbf{Del(p_2, pr_2)})$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1, c_1, pr_1))$;

(C8) $(\mathbf{p_2} < \mathbf{p_1}) = \mathbf{false}$, $(p_1 < p_2) = true, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot \mathbf{Del(p_2 + s(0), pr_2)})$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1, c_1, pr_1))$;

(C9) $(\mathbf{p_2} < \mathbf{p_1}) = \mathbf{true}$, $(p_1 < p_2) = false, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot \mathbf{Del(p_2, pr_2)})$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1))$;

(C10) $(\mathbf{p_2} < \mathbf{p_1}) = \mathbf{false}$, $(p_1 < p_2) = false, conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot \mathbf{Del(p_2 + s(0), pr_2)})$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1))$;

After each step of rewriting, SPIKE performs a step of simplification in order to eliminate conjectures with inconsistent hypotheses. For instance, during such step, (C7) conjecture is eliminated because it's impossible to satisfy the two hypotheses $(p_2 < p_1) = true$ and $(p_1 < p_2) = true$.

The proof process continues. We limit our explanation to the proof of (C10) conjecture. We assume that our theorem prover chooses to rewrite the term $car(p_x, xSt \odot Ins(p_1, c_1, pr_1) \odot Del(p_2 + s(0), pr_2))$ by applying rules (RO4) and (RO5) related from observation function (see figure 7 page 12).

(C11) $(\mathbf{p_x} \geq \mathbf{p_2 + s(0)}) = \mathbf{true}$, $(p_2 < p_1) = false, (p_1 < p_2) = false$,
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$\mathbf{car(p_x + s(0), xSt \odot Ins(p_1, c_1, pr_1))}$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1))$;

(C12) $(\mathbf{p_x} \geq \mathbf{p_2 + s(0)}) = \mathbf{false}$, $(p_2 < p_1) = false, (p_1 < p_2) = false$,
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true$,
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$\mathbf{car(p_x, xSt \odot Ins(p_1, c_1, pr_1))}$
$= car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1))$;

Then, it continues to construct its proof in rewriting others terms where observation function is used. We limit our study to the proof of the (C12) conjecture. In rewriting the term $car(p_x, xSt \odot Del(p_2, pr_2) \odot Ins(p_1 - s(0), c_1, pr_1))$ with rules (RO1), (RO2) and (RO3), we obtain :

(C13) $\mathbf{p_x = p_1 - s(0)}$, $(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) = \mathbf{c_1}$;

(C14) $\mathbf{p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true}$, $(p_x \geq p_2 + s(0)) = false,$
$(p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) = \mathbf{car(p_x - s(0), xSt \odot Del(p_2, pr_2))}$;

(C15) $\mathbf{p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = false}$, $(p_x \geq p_2 + s(0)) = false,$
$(p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) = \mathbf{car(p_x, xSt \odot Del(p_2, pr_2))}$;

Afterwards, we interest ourselves in the proof of the (C14) conjecture. We assume that SPIKE rewrites the term $car(p_x - s(0), xSt \odot Del(p_2, pr_2))$ using rules (RO4) and (RO5).

(C16) $\mathbf{(p_x - s(0) \geq p_2) = true}$, $p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true,$
$(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) = \mathbf{car(p_x - s(0) + s(0), xSt})$;

(C17) $\mathbf{(p_x - s(0) \geq p_2) = false}$, $p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true,$
$(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$car(p_x, xSt \odot Ins(p_1, c_1, pr_1)) = \mathbf{car(p_x - s(0), xSt})$;

Subsequently, we assume SPIKE rewrite the term $car(p_x, xSt \odot Ins(p_1, c_1, pr_1))$ in applying rules (RO1), (RO2) and (RO3). It obtains six conjectures, but for space reason we only show two of them.

(C18) $\mathbf{p_x = p_1}$, $(p_x - s(0) \geq p_2) = true, p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true,$
$(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$
$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$\mathbf{c_1} = car(p_x - s(0) + s(0), xSt);$

(C19) $\mathbf{p_x \neq p_1, (p_x > p_1) = true}$, $(p_x - s(0) \geq p_2) = true, p_x \neq p_1 - s(0),$
$(p_x > p_1 - s(0)) = true, (p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$conc(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) = true,$

$enabled(Ins(p_1, c_1, pr_1), xSt) = true, enabled(Del(p_2, pr_2), xSt) = true \Rightarrow$
$\mathbf{car(p_x - s(0), xSt)} = car(p_x - s(0) + s(0), xSt);$

Conjecture (C19) is eliminated during step of simplification since its hypotheses are inconsistent. It remains one conjecture (C18). This conjecture will be rewritten using rules coming from definitions of $enabled(op, xSt)$ and $conc(op_i, op_j)$ as follows :

(C24) $p_x = p_1, (p_x - s(0) \geq p_2) = true, p_x \neq p_1 - s(0), (p_x > p_1 - s(0)) = true,$
$(p_x \geq p_2 + s(0)) = false, (p_2 < p_1) = false, (p_1 < p_2) = false,$
$\mathbf{pr_1 \neq pr_2, 0 < p_1, 0 < p_2} \Rightarrow$
$c_1 = car(p_x - s(0) + s(0), xSt);$

This conjecture cannot be rewritten any more. When the theorem prover has to prove only this kind of conjectures, it stops.

3.3 Step 3 : Results analysis

The theorem prover can stop for several reasons :

- All conjectures have been successfully proven. Thus, the transformation functions are correct.
- The proof stops on a failure. It can have two meanings :
  1. One or more refutations are found. It means that a counter-example has been found, and consequently that transformation functions do not satisfy the condition.
  2. No refutation are found, but the theorem prover cannot apply rules any more. In this case, we have to add definitions or lemma to our system specification.

For example, We consider the conjecture (C24) obtained in the previous section. There is a simplification for this conjecture that our theorem prover is unable to apply : $p_x - s(0) + s(0) = p_x$. If we add such a lemma, SPIKE simplify this expression to obtain :

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

No more simplification can be done on this conjecture. Moreover, there is no inconsistency in its hypotheses. As a result, this conjecture is a counter-example which leads to violation of the TP_1 condition.

To get the scenario, we have just :

- to take the operations on which we were applying the verification, in this case the two concurrent operations are : $Ins(p_1, c_1, pr_1)$ and $Del(p_2, pr_2)$;

- to instantiate variables with values regarding hypotheses of our counter-example conjecture. In our case, we can set $p_1 = p_2 = 2$, $pr_1 = 1$, $pr_2 = 2$, et $c_1 = x$;
- to execute the scenario.

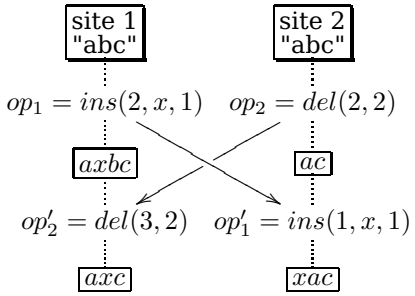


Figure 8: Counter-example found by SPIKE.

The conjecture leads us the well known counter-example [17] depicted at figure 8. This counter-example required two users $user_1$ and $user_2$ who own a copy of the string of characters `'abc'`.

1. $user_1$ produces operation op_1 to insert the character `'x'` at position 2. Concurrently, $user_2$ removes the character `'b'`, located at position 2, by applying operation op_2.
2. When operation op_2 is received by $user_1$, it must be transformed according to operation op_1, which is concurrent. Thus, transformation $T(Del(2, 2), Ins(2,'x', 1))$ is computed ; it gives $Del(3, 2)$. Execution of this operation updates the string of characters to `'axc'`
3. In the same way, when op_1 is received by $user_2$, this operation is transformed according to op_2. $T(Ins(2, x, 1), Del(2, 2))$ is called and gives $Ins(1, x, 1)$. Execution of this operation gives the string of characters `'xac'`.

The TP_1 condition is violated. Consequently, copies do not converge.

4 Verifying existing transformation functions

In this section, we invalidate all previous propositions of transformation functions for string of characters by giving a counter-example for each.

4.1 Ellis's Transformation Functions

Ellis and Gibbs [3] are the pioneers of the operational transformation. They defined the transformation functions shown below. Operations Ins and Del are extended with a new parameter pr representing the priority. Priorities are based on the site identifier where operations are generated [2]. $Id()$ is the Identity operation, which does not affect state.

$T(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) :-$
  if $(p_1 < p_2)$ **return** $Ins(p_1, c_1, pr_1)$
  else if $(p_1 > p_2)$ **return** $Ins(p_1 + 1, c_1, pr_1)$
  else if $(c_1 == c_2)$ **return** $Id()$
  else if $(pr_1 > pr_2)$ **return** $Ins(p_1 + 1, c_1, pr_1)$
  else return $Ins(p_1, c_1, pr_1)$

$T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2)) :-$
  if $(p_1 < p_2)$ **return** $Ins(p_1, c_1, pr_1)$
  else return $Ins(p_1 - 1, c_1, pr_1)$

$T(Del(p_1, pr_1), Ins(p_2, c_2, pr_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1, pr_1)$
  else return $Del(p_1 + 1, pr_1)$

$T(Del(p_1, pr_1), Del(p_2, pr_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1, pr_1)$
  else if $(p_1 > p_2)$ **return** $Del(p_1 - 1, pr_1)$
  else return $Id()$

Figure 9: Ellis et al.'s transformation functions.

It is well known that these transformation functions are not correct [17, 24, 21]. Nevertheless, we submitted them to SPIKE in order to verify if the problem can be automatically detected. SPIKE found the counter-example depicted in figure 8 in a few seconds. SPIKE detected that condition TP_1 is violated.

The error comes from the definition of T for couple of operation (Ins, Del). The condition $p_1 < p_2$ should be rewritten $p_1 \leq p_2$. But if we re-submit this version to the theorem prover, it is still not correct with the counter-example detailed in the next section.

[2] This priority becomes even more complex since it is also used like a list.

4.2 Ressel's Transformation Functions

Ressel et al. [17] modified Ellis's transformation functions in order to satisfy TP_1 and TP_2. Priorities are replaced by the parameter $u_i \in 1, 2, ..., n$. This parameter represents the user who generates the operation.

The definition of T for couple (Ins, Ins) as follows : when two insert operations have the same position p, the character produced by the site with the lower range is inserted at p ; the other one will be inserted at position $p + 1$.

$T(Ins(p_1, c_1, u_1), Ins(p_2, c_2, u_2)) :-$
  if $(p_1 < p_2)$ **or** $(p_1 == p_2$ **and** $u_1 < u_2)$ **return** $Ins(p_1, c_1, u_1)$
  else return $Ins(p_1 + 1, c_1, u_1)$

$T(Ins(p_1, c_1, u_1), Del(p_2, u_2)) :-$
  if $(p_1 \leq p_2)$ **return** $Ins(p_1, c_1, u_1)$
  else return $Ins(p_1 - 1, c_1, u_1)$

$T(Del(p_1, u_1), Ins(p_2, c_2, u_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1, u_1)$
  else return $Del(p_1 + 1, u_1)$

$T(Del(p_1, u_1), Del(p_2, u_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1, u_1)$
  else if $(p_1 > p_2)$ **return** $Del(p_1 - 1, u_1)$
  else return $Id()$

Figure 10: Ressel et al.'s transformation functions.

This strategy satisfy TP_1 but SPIKE found the TP_2 counter-example given in figure 11. This counter-example requires three sites where operations $op_1 = Ins(3, x)$, $op_2 = Del(2)$ and $op_3 = Ins(2, y)$ are concurrent.

Copies on site 2 and 3 do not converge. Consequently, transformation functions of Ressel et al. do not verify TP_2.

4.3 IMOR transformation functions

In [5], we proposed to add a new parameter ip_i to every Ins operation. This parameter represents the *initial position* of character c_i. Suppose one user inserts a character x at position 3, thus an operation $Ins(3, 3, x)$ is generated. If this operation is transformed, only the position (first parameter) will change. The initial position parameter is not affected. Figure 12 gives whole definition of these functions.

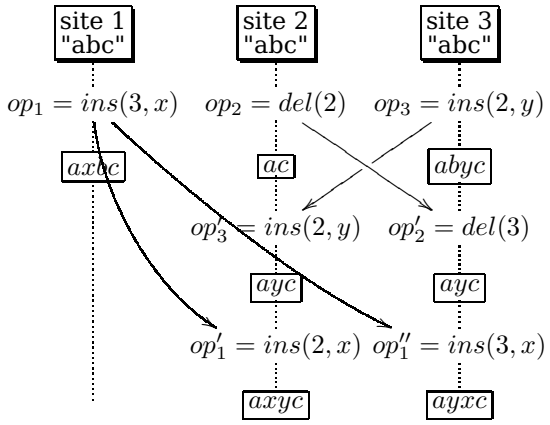


Figure 11: Counter example violating condition TP_2.

$T(Ins(p_1, ip_1, c_1), Ins(p_2, ip_2, c_2)) :-$
  if $(p_1 < p_2)$ **return** $Ins(p_1, ip_1, c_1)$
  else if $(p_1 > p_2)$ **return** $Ins(p_1 + 1, ip_1, c_1)$
  else if $(ip_1 < ip_2)$ **return** $Ins(p_1, ip_1, c_1)$
  else if $(ip_1 > ip_2)$ **return** $Ins(p_1 + 1, ip_1, c_1)$
  else if $(code(c_1) < code(c_2))$ **return** $Ins(p_1, ip_1, c_1)$
  else if $(code(c_1) > code(c_2))$ **return** $Ins(p_1 + 1, ip_1, c_1)$
  else return $Id()$

$T(Ins(p_1, ip_1, c_1), Del(p_2)) :-$
  if $(p_1 > p_2)$ **return** $Ins(p_1 - 1, ip_1, c_1)$
  else return $Ins(p_1, ip_1, c_1)$

$T(Del(p_1, pr_1), Ins(p_2, ip_2, c_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1)$
  else return $Del(p_1 + 1)$

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

Figure 12: IMOR transformation functions

In [5], we claimed that this set of transformation functions satisfied TP_2. Unfortunately, we made a mistake in the specification of transformation functions. We described operations as:

operation
$(p == o)$ **and** $(p \leq length()) : Ins(nat\, p, nat\, o, char\, c)$;
$p < length() : Del(nat\, p)$;

By specifying $p == o$ as one of the preconditions of operation ins, SPIKE does not generate TP_1 or TP_2 instances with $p \neq o$. However, as presented in figure 13, such a case is possible. Thus, we rewrite the specification as:

operation
$p \leq length() : Ins(nat\, p, nat\, o, char\, c)$;
$p < length() : Del(nat\, p)$;

We run the prover on this specification and SPIKE returns the counter example presented figure 13.

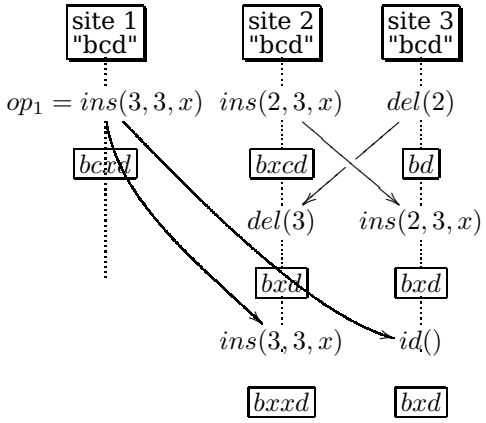


Figure 13: IMOR counter example (3 sites)

In contrast to the other counterexamples, we noticed the concurrent operations, involved in this scenario, have been previously transformed. For instance, the operation $ins(2,3,x)$ on site 2 was originally an $ins(3,3,x)$ operation which had been transformed according to a $del(1)$ or a $del(2)$ operation. On the other hand, the operation $ins(3,3,x)$ from site 1, has been transformed yet. From theses observations, we can assume that the original situation that lead to the counterexample should be the situation depicted in figure 14.

4.4 Suleiman's transformation functions

Suleiman et al. propose a set of transformation functions in [20]. They add two new parameters to operation Ins. Ins is defined as follows : $Ins(p_i, c_i, b_i, a_i)$ where b_i (a_i respectively) is the set of concurrent operations to this insertion operation and that

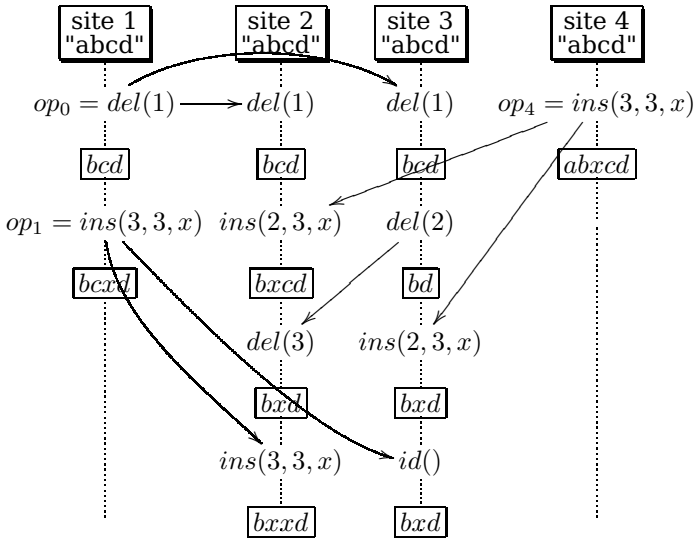


Figure 14: IMOR counter example

have deleted a character before (after respectively) the position p_i. Hence, for two concurrent operations $Ins(p_1, c_1, b_1, a_1)$ and $Ins(p_2, c_2, b_2, a_2)$ defined on the same state, the following cases are given:

- if $(b_1 \cap a_2) \neq \emptyset$ then c_2 was inserted before c_1,
- if $(a_1 \cap b_2) \neq \emptyset$ then c_2 was inserted after c_1,
- if $(b_1 \cap a_2) = (a_1 \cap b_2) = \emptyset$ then c_1 and c_2 were inserted at same position. Hence, we can use the *code* [3] of character c_i to determine which character we have to insert at this position.

Suleiman et al. made the full proof of correctness by hand, and argued that their transformation functions are correct regarding the TP_1 and TP_2 conditions. In [5] we make a proof of these transformation functions. In fact, we made the same mistake than the one explained in previous section 4.3. We specified the ins operation as follows :

operation
$p \leq length()$ **and** $av = \{\}$ **and** $ap = \{\}$: $Ins(nat\, p, char\, c, setop\, av, setop\, ap)$;
$p < length() : Del(nat\, p, opid\, i)$;

Then, SPIKE generates ins operation with empty sets. It means these operations could not have been previously transformed against operations of deletion. This as-

[3] *code* function permits to compare and to sort two characters according to their alphabetic value

$T(Ins(p_1, c_1, b_1, a_1), Ins(p_2, c_2, b_2, a_2)) :-$
  if $(p_1 < p_2)$ **return** $Ins(p_1, c_1, b_1, a_1)$
  else if $(p_1 > p_2)$ **return** $Ins(p_1 + 1, c_1, b_1, a_1)$
  else if $(b_1 \cap a_2) \neq \varnothing$ **return** $Ins(p_1 + 1, c_1, b_1, a_1)$
  else if $(a_1 \cap b_2) \neq \varnothing$ **return** $Ins(p_1, c_1, b_1, a_1)$
  else if $(code(c_1) > code(c_2))$ **return** $Ins(p_1, c_1, b_1, a_1)$
  else if $(code(c_1) < code(c_2))$ **return** $Ins(p_1 + 1, c_1, b_1, a_1)$
  else return $Id()$

$T(Ins(p_1, c_1, b_1, a_1), Del(p_2)) :-$
  if $(p_1 > p_2)$ **return** $Ins(p_1 - 1, c_1, b_1 \cup \{Del(p_2)\}, a_1)$
  else return $Ins(p_1, c_1, b_1, a_1 \cup \{Del(p_2)\})$

$T(Del(p_1, pr_1), Ins(p_2, c_2, b_2, a_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1)$
  else return $Del(p_1 + 1)$

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

Figure 15: Suleiman transformation functions

sumption was wrong. The before set and after set of an operation can be arbitrary filled. We rewrite the specification as follows :

operation
  $p \leq length() : Ins(nat\, p, char\, c, setop\, av, setop\, ap);$
  $p < length() : Del(nat\, p, opid\, i);$

With this specification, SPIKE gives the counter example of figure 16.

As in the counterexample for IMOR, the operations involved in the scenario are not original ones. The operation $ins(3, x, \varnothing, \{del(3)\})$ on site 2 is resulting from the transformation according to a $del(3)$ operation. And, as the $del(3)$ operation is in its "after a_i" set, we can assume that the original operation was $ins(3, x, \varnothing, \varnothing)$. In the same way, the $ins(3, y, \{del(3)\}, \varnothing)$ is resulting also from the transformation according to a $del(3)$ operation. But this time, the $del(3)$ is in its "before b_i" set, thus it means that the original operation should be $ins(4, y, \varnothing, \varnothing)$. And finally, the operation $ins(3, x, \varnothing, \varnothing)$ has not been transformed according any $del()$ operation. From all of these observations, we can assume that the original situation might be the situation depicted in figure 17.

Thus, transformation functions from Suleiman et al. do not verify TP_2. This example demonstrates how hand-written proof is error prone and how specification is error

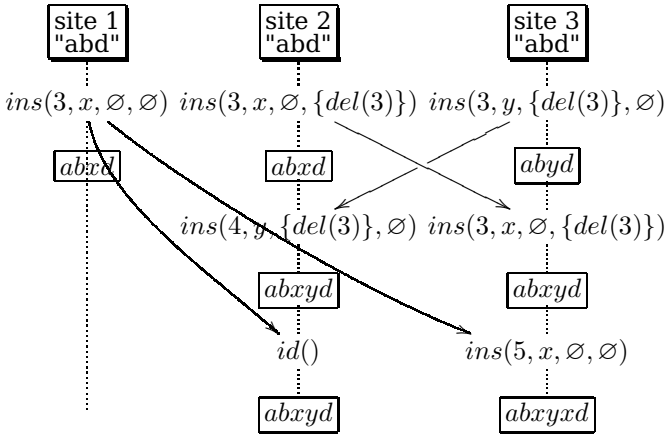


Figure 16: Suleiman counter example (3 sites)

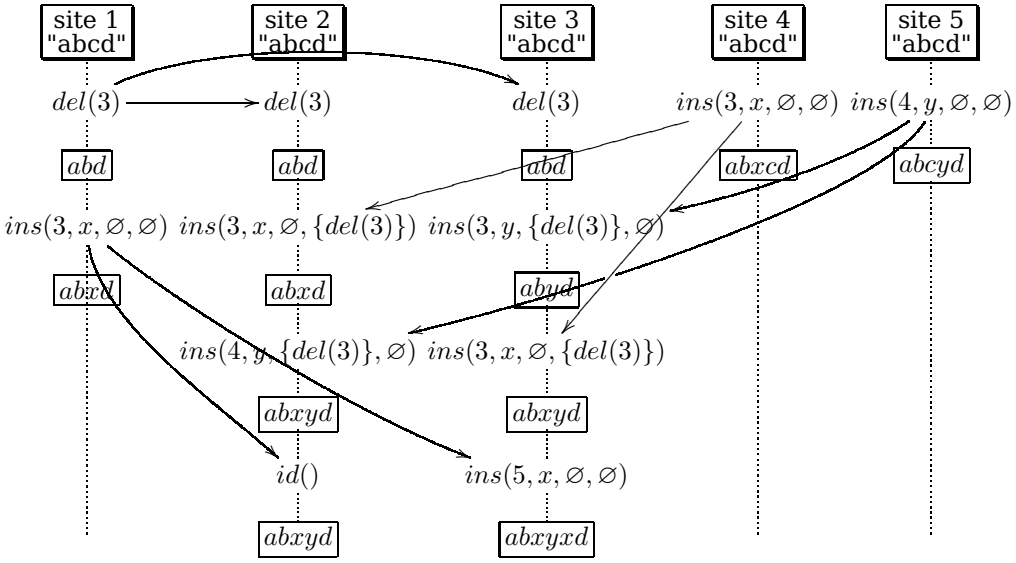


Figure 17: Suleiman counter example

prone too. Nevertheless, it was easier to find the specification error in twenty lines of specification, than to find the proof error in ten pages of proof of Suleiman's thesis.

4.5 SDT Transformation functions

Li proposes a new set of transformation function based on the State Difference Transformation (SDT) approach. We call them SDT transformation functions and they are described in figure 18.

$T(o_1 = Ins(p_1, c_1), o_2 = Ins(p_2, c_2)) :-$
if $\beta_{lsp}(o_1) < \beta_{lsp}(o_2)$ **return** o_1
else if $\beta_{lsp}(o_1) > \beta_{lsp}(o_2)$ **return** $ins(p_1 + 1, c_1)$
else
  if $p_1 < p_2$ **return** $ins(p_1, c_1)$
  else if $p_1 > p_2$ **return** $ins(p_+1, c_1)$
  else
    if $id(o_1) < id(o_2)$ **return** o_1
     else return $ins(p_1 + 1, c_1)$

Figure 18: SDT transformation functions

SDT transformation functions introduce a function $\beta_{lsp}(op)$. This function computes the position of op on state called Last Synchronization Point (LSP). LSP is the state identified by the state vector $V_{min} = min(v(o_1), v(o_2))$. $v(o_1), v(o_2)$ are state vectors of o_1 and o_2. V_{min} is formed with minimal value of each component of the state vector of o_1 and o_2.

In order to compute the position of op on the LSP state, SDT computes the sequence SQ of operations from LSP to s. Then, SDT computes another sequence SD which is equivalent to SQ but that only contains the net effect between LSP and s. Next, SDT excludes effects of SD from o_1 and o_2 and obtains positions of o_1, o_2 on state LSP. Comparing $\beta_{lsp}(o_1)$ and $\beta_{lsp}(o_2)$ allows to break the tie.

We specified the SDT transformation function described in figure 18with SPIKE without specifying the $\beta()$ function. So SPIKE check all possible *scenarii* with all possible return value of $\beta()$. Finally, SPIKE gives the counter-example described in figure 19.

On site 1 on state s_0, during $T(op_1, op_2)$, $\beta(op_1) = \beta(op_2)$ and $(p_1 < p_2)$. Thus positions determine the result of trnasformation. On site 2, on state s_1 during $T(op_1, op_2)$, $\beta(op_1) = \beta(op_2)$ and $p_1 == p_2$. so this time, site identifiers break the tie. This couter-example illustrates that a tie on β can be broken by two different methods; positions or site identifiers.

Since we do not specified the β function, such a case may not arrive. The complete scenario presented in figure 20 show that this case is possible.

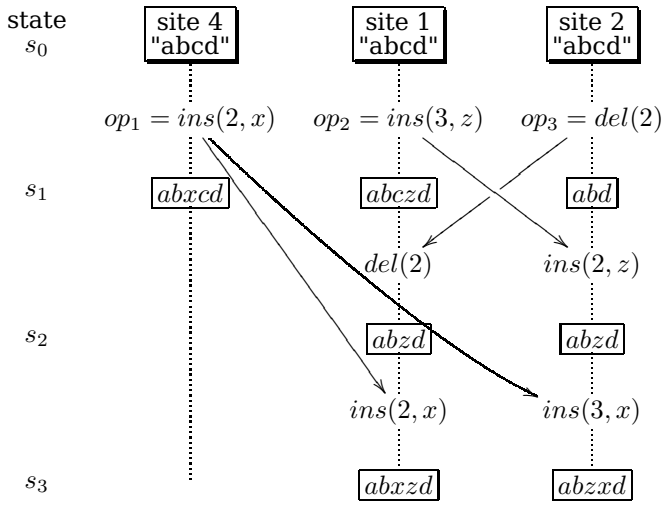


Figure 19: SDT potential counter-example

The scenario of figure 20 shows the following problem:

- During $T(op_{41}, op_3)$ on state s_2 on site 1, if $\beta_{lsp}(op_{41}) = \beta_{lsp}(op_3) = 2$, then positions of insertion break the tie.
- During $T(op_{41}, op_3)$ on state s_3 on site 2, if $\beta_{lsp}(op_{41}) = \beta_{lsp}(op_3) = 2$, then site identifiers break the tie.

If the tie is broken by two different methods, it diverges. We have verified that this counter example leads to divergence in the implementation of SDTO [4].

This counter-example demonstrates the need of automated theorem proving. Transformation functions are small, but it generates a huge number of cases to verify. According to the authors of SDT [10]: "Due to the huge number of cases to consider, the proof scales poorly to a more sophisticated operation set". We think that automated theorem proving helps to make the proof scaling.

5 Tombstones Transformation Functions

All the transformation functions presented above are designed to manage a very simple tie : three concurrent operations : $ins(2, x)$, $del(2)$ and $ins(3, y)$ as presented in figure 21.

[4] Authors of SDTO made their implementation available online at the following location http://cocasoft.csdl.tamu.edu/~lidu/projects/CE/

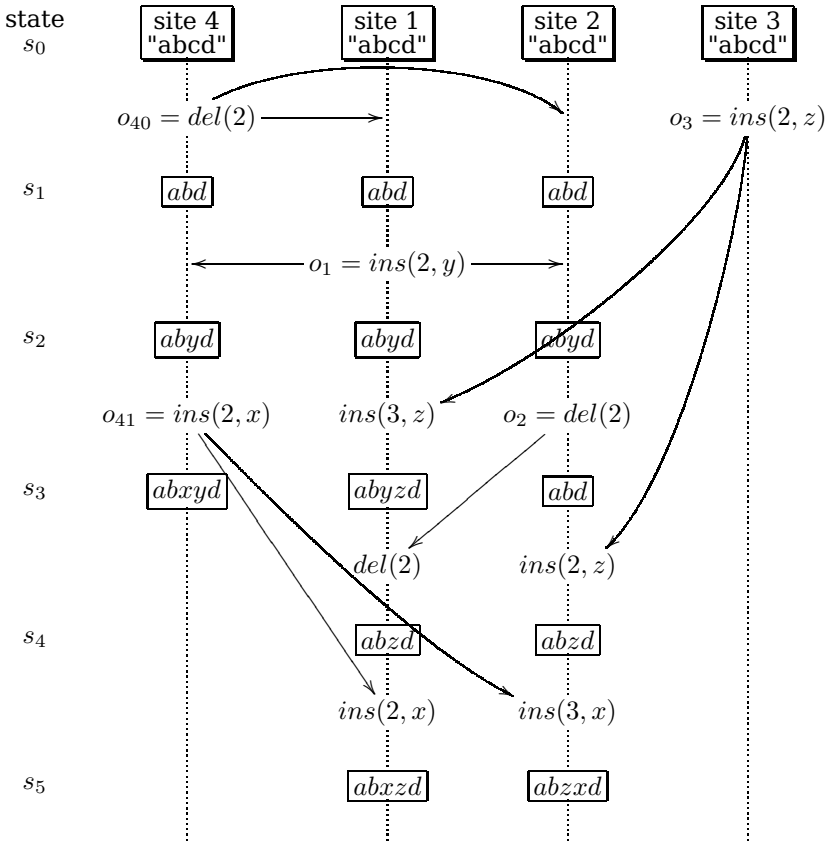


Figure 20: SDT counter-example

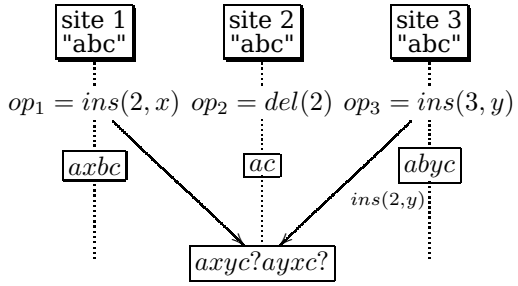


Figure 21: Common problem

To break the tie between $ins(2, x)$ ad $ins(2, y)$ on site 2, all the approaches choose a different way : initial position for IMOR, sets for Suleiman et al. or state difference for Li et al.. Unfortunately, all these approaches, even complicated ones, fail to order correctly x and y. And all the counter-examples are only instances of this tie. Nevertheless, character b obviously separates x and y.

Our idea is to keep the deleted character b as a tombstone. This is a transposition of the WOOT approach [16] in the OT framework. Tombstones are well known in distributed system and are heavily used in Usenet, Active Directory [18]. In Usenet, tombstones are used to make conflicts update/delete non ambiguous. If a character is deleted, we maintain useful informations about its former position but not its whole content. For a character string, it is equivalent to keep the character in its position and mark the character as invisible. If we manage lines instead of characters, it means that we maintain the identity of the line, but not its content.

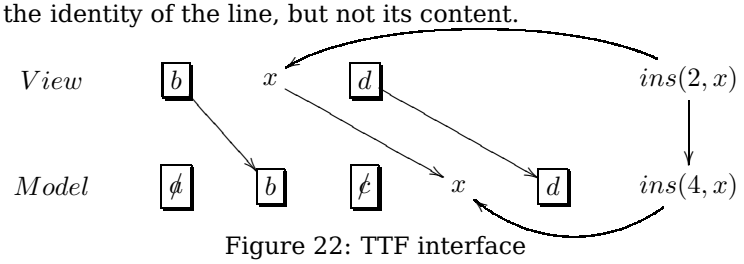


Figure 22: TTF interface

Consequently, hidden characters are present in the model of the string on local site. They are present in the model but they are not visible in the view of this model. We illustrate this in figure 22. The operation $ins(2, x)$ is generated on the view but executed on the model as $ins(4, x)$. This is also the operation $ins(4, x)$ which is broadcasted and will be transformed in all other sites.

This approach is quite new for OT. Traditionally, there is no difference between the generated operation and the executed operation. Make this distinction is the heart of the TTF approach. The effect of this strategy leads to trivial transformation functions presented in figure 23.

According to the theorem prover, these transformation functions verify TP_1 and TP_2. Figure 24 describes how TTF behaves on the SDT counter-example

The complete specification is presented in the appendix A. The complete verification of TP_1 and TP_2 is totally automatic and takes 1mn30s on a laptop. The main benefits of TTF approach is its simplicity. The idea is simple and with the help of the theorem prover, the proof can scale to more complex operations or more complex data types.

TTF functions can be used with the adOPTed integration algorithm. adOPTed has been proved correct in [11] if transformation functions verify TP_1 and TP_2. We provide for the first time such transformations.

Even if exclusion transformations are not required, we can defined such functions. Figure 25 describes reverse TTF transformation functions. These functions can be used

$T(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) :-$
  if $(p_1 < p_2)$ **return** $Ins(p_1, c_1)$
  else if $(p_1 == p_2$ **and** $sid_1 < sid_2)$ **return** $Ins(p_1, c_1, sid_1)$
  else return $Ins(p_1 + 1, c_1, sid_1)$

$T(Ins(p_1, c_1, sid_1), Del(p_2, sid_2)) :-$
  return $Ins(p_1, c_1, sid_1)$

$T(Del(p_1, sid_1), Ins(p_2, c_2, sid_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1, sid_1)$
  else return $Del(p_1 + 1, sid_1)$

$T(Del(p_1, sid_1), Del(p_2, sid_2)) :-$
  return $Del(p_1, sid_1)$

Figure 23: TTF transformation functions.

safely with SOCT2 and GOTO. We have proven that these reverse transformation function are correct by proving the following reversibility property with the theorem prover.

Definition 5.1 (Reversibility property) *For all concurrent operations op_1 and op_2 defined on the same state*

$$T^{-1}(T(op_1, op_2), op_2) = op_1$$

.

TTF is the only transformation functions that verify TP_1 and TP_2. Automated theorem proving ensures that the proof of these properties is correct according to the specification. The complete specification is presented in the appendix A and can be easily verified. Previous specification constraints (see section 4.3) are not present in this specification.

If tombstones is a very easy solution for the OT approach, TTF retains a tombstone to mark a deleted character. So, the space overhead of tombstones grows indefinitely. If we use an expiration period for garbaging deleted characters, this method is unsafe. If we use a two-phase protocol to purge safely the tombstones as in [18], all sites must be alive for the algorithm to make progress.

We can solve the problem by choosing another model for the local string. In Figure 26(a), tombstones are represented in the string as $\not{a}b\not{c}d$. We can just keep visible characters of this string with their absolute positions as in figure 26(b). By this way, the space overhead of tombstones does not grows any more.

To resume, Tombstones Transformation functions are the only transformation functions that ensure TP_1 and TP_2. They can be used with adOPTed, GOTO and SOCT2.

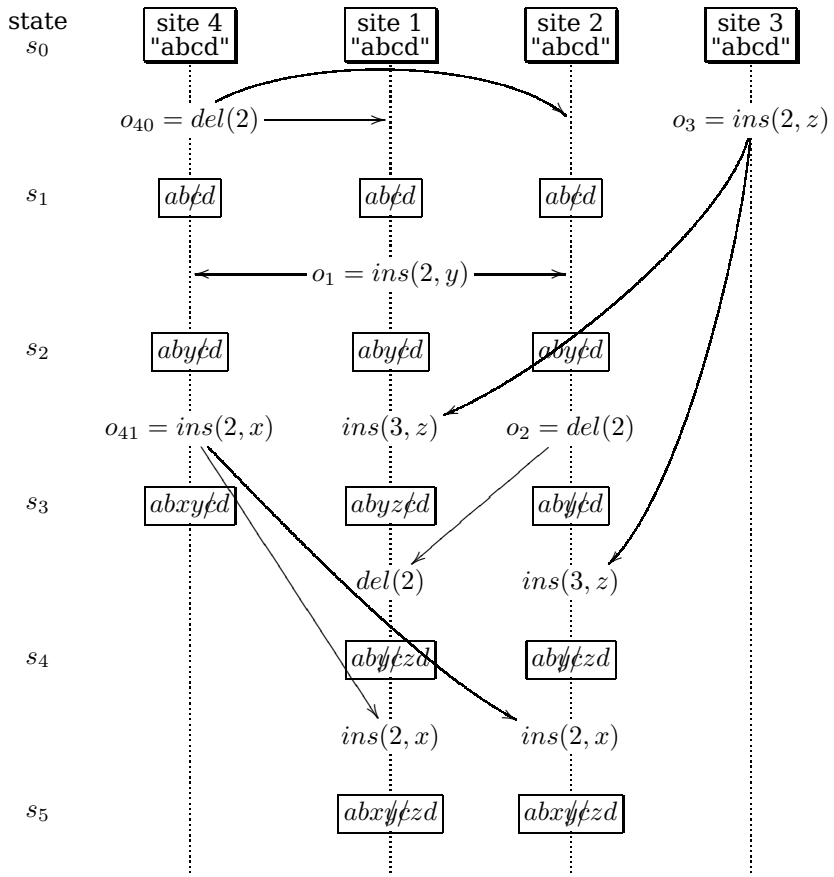


Figure 24: TTF behaviour on SDT counter-example

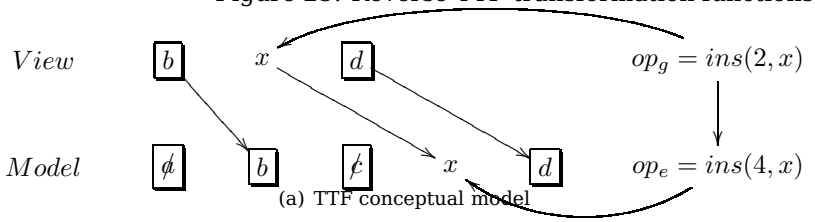
$T^{-1}(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) :-$
  if $(p_1 < p_2)$ **return** $Ins(p_1, c_1, sid_1)$
  else if $(p_1 == p_2$ **and** $sid_1 < sid_2)$ **return** $Ins(p_1, c_1, sid_1)$
  else return $Ins(p_1 - 1, c_1, sid_1)$

$T^{-1}(Ins(p_1, c_1, sid_1), Del(p_2, sid_2)) :-$
  return $Ins(p_1, c_1, sid_1)$

$T^{-1}(Del(p_1, sid_1), Ins(p_2, c_2, sid_2)) :-$
  if $(p_1 < p_2)$ **return** $Del(p_1, sid_1)$
  else return $Del(p_1 - 1, sid_1)$

$T^{-1}(Del(p_1, sid_1), Del(p_2, sid_2)) :-$
  return $Del(p_1, sid_1)$

Figure 25: Reverse TTF transformation functions.



(a) TTF conceptual model

(b) TTF space preserverving model

6 Related Work

The TTF transformation functions are defined in the Ressel's OT model [17]. There are two other OT models: the CCI model defined by Sun [24] and the CSM model defined by Li [9]. In this part, we evaluate TTF in the two other models and we describe open issues of TTF.

Sun introduces the Intention preservation problem in [24]. It considers two sites, each site owns a copy of a string of characters "ABCDE" (cf. figure 26(c)). Site 1 inserts "12" at position 2 and obtains "A12BCDE"'. Site 1 has executed operation

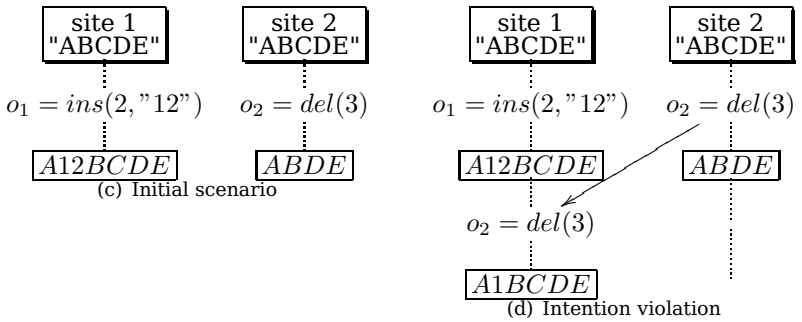


Figure 26: Intention violation problem

$op_1 = ins(2, "12")$ with the intentions to insert "12" between A and B. Site 2 deletes one character at position 3 and gets "ABDE". Site has executed operation op_2 with the intention to delete the character "C". If we execute both operations and preserve intentions, we must obtain "A12BDE". But if we just send operation op_2 to Site 1 and execute on its current states, we get "A1BCDE" (see figure 26(d)) from which character '2' has been removed. On this site, the intentions of the operations have not been preserved.

In this example, intention preservation means that if "12" has been inserted between 'A' and 'B', then this ordering $'A' \prec "12" \prec' B'$ must be preserved for any further states.

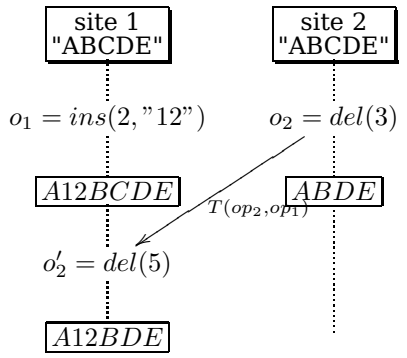


Figure 27: Intention violation problem fixed

According to Sun et al. [24], intention preservation is defined as follows :

1. For any operation op, the effects of executing op at all sites are the same as the intention of op
2. the effects of executing op does not change the effects of independent operations.

The intention preservation definition is not a formal property. First, we define intention of our operation as follows:

- As in [9, 15], for ins operations, we express the intention by the partial order relation $\prec$. We get that $x \prec c \prec y$ if one user generates $ins(c, p)$ on a site where x is visible at a position less than p and y is visible at a position greater or equal than p.
- The effect of an operation $del(p)$ on a string S is to delete the character $S[p]$.

Preserving the intentions of ins operations means that $\prec$ relations hold on all futhers states.

The TTF approach respects intentions $x \prec c \prec y$ on the generation site at the generation time. Since operations can only inserts characters, the order $x \prec c \prec y$ is always preserved on generation site. Since our approach ensures convergence, the order will eventually be the same in all sites, and intentions $x \prec c \prec y$ will be preserved on every site.

The CSM model [9] stands for Causality, Single-operation effects preservation and Multi-operation effects relation preservation.

- Causality is defined as in the Ressel's model.
- Single-operation effect preservation: the effect of executing any operation in any execution state achieves the same effect as in its generation state.
- Multi-operation effects relation preservation: the effects relation of any two operations maintains after they are executed in any states.

We have already proved that TTF preserve single-operation effect. TTF also preserve multi-operation effects by preserving the partial order $\prec$. The figure 28 illustrates how TTF transformation functions behaves on the Effect Relation Violation puzzle (ERV puzzle). The figure 28(a) presents the classical TP2 puzzle. Finally, copies do not converge. In [9], authors point out that there is also a Effect Relation Violation. When o_1 is generated, character '1' is inserted between 'b' and 'c'. So, we have $b \prec 1 \prec c$. Next, o_2 generates $a \prec 2 \prec b$. Finally on site 3, user deletes character 'b'. The final state must preserve $a \prec 2 \prec \not{b} \prec 1$. It is not the case on site 1. The problem comes from the deletion of the landmark character 'b'. It confuses transformation functions. TTF do not delete character, so the landmark character is not deleted. So TTF solve nicely the ERV puzzle as illustrated in 28(b)

To resume, TTF are defined in the Ressel's model and combined with the adOPTed, GOTO or SOCT2 integration algorithms, they ensures Causality and Convergence. However, If we use TTF in the Sun's OT model where intentions preservation are defined, TTF preserve intentions. If we use TTF in the CSM model as defined in [9], TTF preserve effect relationships.

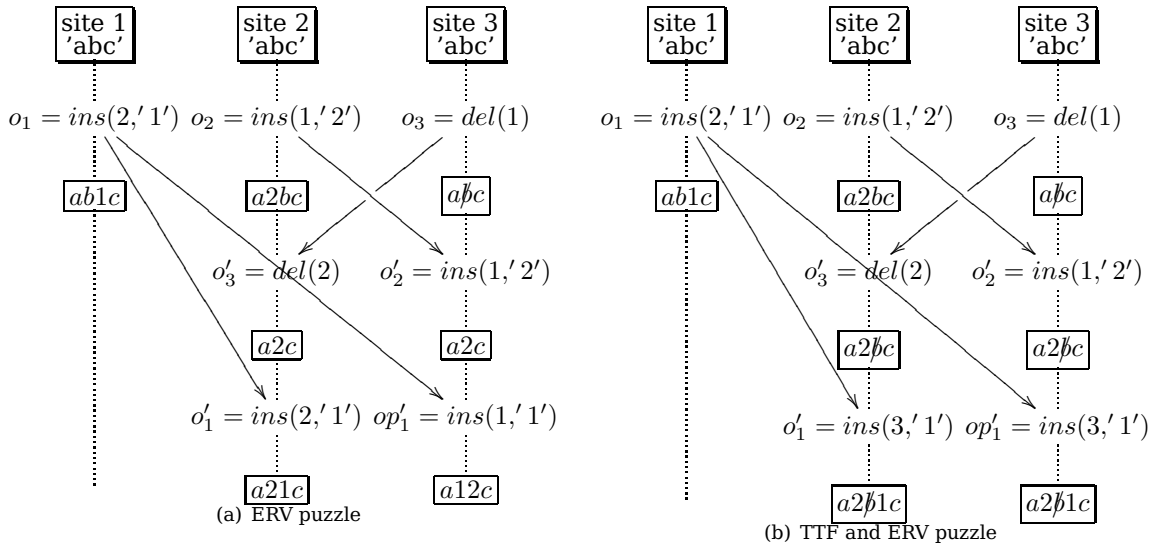


Figure 28: TTF behaviour and the ERV problem

7 Conclusion

We have presented how a theorem prover can help transformation functions designers for writing transformation functions.

Automated theorem proving is well adapted for proving transformation functions. If the proof of TP_2 generates a huge number of different cases, the proof of one case is easy. Then the theorem prover is used to explore the combinatorial explosion of possibilities. The proof of TP_1 is in fact more difficult because it requires to express the state of shared objects. If the state of a String is easy to represent, the state of an XML tree is more complex. In this paper, we used the situation calculus to simplify the representation of states. We have already validated this approach by verifying TP_1 on strings, blocks of text, File system and XML in [14].

Using a theorem prover has some liabilities; it proves what is specified. It does not prevent specification errors. We have done such errors in [5]. Nevertheless, the transformation functions specification is generally small and it is easier to find an error in the specification than in the proof.

We used automated theorem proving to invalidate all existing transformation functions. These counter-examples leads us to propose the Tombstone Transformation Functions. These functions verifies TP_1, TP_2. The tombstone approach allows also TTF to preserve intentions and effect relations.

In this paper, we used the theorem prover to verify TP_1 and TP_2 in the context of the Ressel Model. In fact, these properties ensure convergence like every optimistic replication approach [18]. The intention preservation property is specific to the OT approach. Unfortunately, this property is not well formalized and cannot be formally proven. We think that The CSM model makes one step in the formalization of the intention. We plan to use the theorem prover to verify a formal version of intention preservation property.

A VOTE specification of TTF functions

```
specification ttf;

type nat, char;

observator
        char car(nat);
        bool visible(nat);
        nat length();

operation
        [p > 0 and p ≤ length()+1 ] Ins(nat p, char c, nat siteid);
        [p > 0 and p ≤ length() ] Del(nat p, nat siteid);

transform
        T(Ins(p1,c1,sid1), Ins(p2,c2,sid2)) =
                if (p1 < p2) then Ins(p1,c1,sid1)
                elseif ((p1 = p2) and (sid1 < sid2)) then Ins(p1,c1,sid1)
                else Ins(p1+1,c1,sid1) ;

        T(Ins(p1,c1,sid1), Del(p2,sid2)) =
                Ins(p1,c1,sid1) ;

        T(Del(p1,sid1), Ins(p2,c2,sid2)) =
                if (p1 < p2) then Del(p1,sid1)
                else Del(p1+1,sid1) ;

        T(Del(p1,sid1), Del(p2,sid2)) =
                Del(p1,sid1) ;

reverse
        Trev(Ins(p1,c1,sid1), Ins(p2,c2,sid2)) =
                if (p1 < p2) then Ins(p1,c1,sid1)
                elseif ((p1 = p2) and (sid1 < sid2)) then Ins(p1,c1,sid1)
                else Ins(p1−1,c1,sid1) ;

        Trev(Ins(p1,c1,sid1), Del(p2,sid2)) =
                Ins(p1,c1,sid1) ;

        T(Del(p1,sid1), Ins(p2,c2,sid2)) =
                if (p1 < p2) then Del(p1,sid1)
                else Del(p1−1,sid1) ;

        T(Del(p1,sid1), Del(p2,sid2)) =
                Del(p1,sid1) ;

definition
        conc(Ins(p1,c1,sid1), Ins(p2,c2,sid2)) =
                if (sid1 ≠ sid2) then true
                else false ;

        conc(Ins(p1,c1,sid1), Del(p2,sid2)) =
                if (sid1 ≠ sid2) then true
                else false ;

        conc(Del(p1,sid1), Del(p2,sid2)) =
                if (sid1 ≠ sid2) then true
                else false ;
```

```
        conc(Del(p1,sid1), Del(p2,sid2)) =
                if (sid1 ≠ sid2) then true
                else false ;

        car'(n)::Ins(p, c, sid) =
                if (n = p) then c
                elseif (n > p) then car(n−1)
                else car(n) ;

        car'(n)::Del(p, sid) =
                car(n) ;

        visible'(n)::Ins(p, c, sid) =
                if (n = p) then true
                elseif (n > p) then visible(n−1)
                else visible(n) ;

        visible'(n)::Del(p, sid) =
                if (n = p) then false
                else visible(n) ;

        length'()::Ins(p, c, sid) =
                length()+1 ;

        length'()::Del(p, sid) =
                length() ;


property
        // TP1 property
        conc(x,y)=true, enabled(x, xSt) = true, enabled(y, xSt) = true, ⇒
                Obs(do(T(y, x), do(x, xSt))) = Obs(do(T(x, y), do(y, xSt))) ;

        // TP2 property
        conc(x,y)=true, conc(x,z)=true, conc(y,z)=true,
        enabled(x, xSt) = true, enabled(y, xSt) = true, enabled(z, xSt) = true ⇒
                T(T(z, x), T(y, x)) = T(T(z, y), T(x, y));

        // Reversibility property
        Trev(T(x,y),y) = x;
```

B SPIKE specification of TTF functions

specification : ttf
use : nats;

sorts : Op State char;

constructors :
do__ : Op State → State;
S0 : → State;
Ins___ : nat char nat → Op;
Del__ : nat nat → Op;

c0 : → char;
nc_ : char → char;

defined functions :
T__ : Op Op → Op;
Trev__: Op Op → Op;
poss__ : Op State → bool;
ex_ : State → bool;
conc__ : Op Op → bool;
visible__ : nat State → bool;
length_ : State → nat;
car__ : nat State → char;

axioms :

// Preconditions
(p > 0) = true, ($p \leq$ (length(xSt) + $s(0)$)) = true ⇒ poss(Ins(p, c, sid), xSt) = true;
(p > 0) = false ⇒ poss(Ins(p, c, sid), xSt) = false;
($p \leq$ (length(xSt) + $s(0)$)) = false ⇒ poss(Ins(p, c, sid), xSt) = false;

(p > 0) = true, ($p \leq$ length(xSt)) = true ⇒ poss(Del(p, sid), xSt) = true;
(p > 0) = false ⇒ poss(Del(p, sid), xSt) = false;
($p \leq$ length(xSt)) = false ⇒ poss(Del(p, sid), xSt) = false;

poss(a,xSt)=true ⇒ ex(do(a, xSt)) = ex(xSt);
poss(a,xSt)=false ⇒ ex(do(a, xSt)) = false;
ex(S0) = true;

$sid_1 = sid_2$⇒ conc(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = false;
$sid_1 \neq sid_2$⇒ conc(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = true;
$sid_1 = sid_2$⇒ conc(Ins(p_1, c_1, sid_1), Del(p_2, sid_2)) = false;
$sid_1 \neq sid_2$⇒ conc(Ins(p_1, c_1, sid_1), Del(p_2, sid_2)) = true;
$sid_1 = sid_2$⇒ conc(Del(p_1, sid_1), Ins(p_2, c_2, sid_2)) = false;
$sid_1 \neq sid_2$⇒ conc(Del(p_1, sid_1), Ins(p_2, c_2, sid_2)) = true;
$sid_1 = sid_2$⇒ conc(Del(p_1, sid_1), Del(p_2, sid_2)) = false;
$sid_1 \neq sid_2$⇒ conc(Del(p_1, sid_1), Del(p_2, sid_2)) = true;

// Transforms
(p_1 < p_2) = true ⇒ T(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = Ins(p_1, c_1, sid_1);
(p_1 < p_2) = false, $p_1 = p_2$, (sid_1 < sid_2) = true ⇒ T(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = Ins(p_1, c_1, sid_1);
(p_1 < p_2) = false, $p_1 \neq p_2$⇒ T(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = Ins(p_1 + $s(0)$, c_1, sid_1);
(p_1 < p_2) = false, (sid_1 < sid_2) = false ⇒ T(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = Ins(p_1 + $s(0)$, c_1, sid_1);

T(Ins(p_1, c_1, sid_1), Del(p_2, sid_2)) = Ins(p_1, c_1, sid_1);

(p_1 < p_2) = true ⇒ T(Del(p_1, sid_1), Ins(p_2, c_2, sid_2)) = Del(p_1, sid_1);

(p_1 < p_2) = false ⇒ T(Del(p_1, sid_1), Ins(p_2, c_2, sid_2)) = Del(p_1 + $s(0)$, sid_1);

T(Del(p_1, sid_1), Del(p_2, sid_2)) = Del(p_1, sid_1);

// Reverse Transforms

(p_1 < p_2) = true ⇒ Trev(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = Ins(p_1, c_1, sid_1);
(p_1 < p_2) = false, p_1= p_2, (sid_1 < sid_2) = true ⇒ Trev(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = Ins(p_1, c_1, sid_1);
(p_1 < p_2) = false, $p_1 \neq p_2$⇒ Trev(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = Ins(p_1 − $s(0)$, c_1, sid_1);
(p_1 < p_2) = false, (sid_1 < sid_2) = false ⇒ Trev(Ins(p_1, c_1, sid_1), Ins(p_2, c_2, sid_2)) = Ins(p_1 − $s(0)$, c_1, sid_1);

Trev(Ins(p_1, c_1, sid_1), Del(p_2, sid_2)) = Ins(p_1, c_1, sid_1);

(p_1 < p_2) = true ⇒ Trev(Del(p_1, sid_1), Ins(p_2, c_2, sid_2)) = Del(p_1, sid_1);
(p_1 < p_2) = false ⇒ Trev(Del(p_1, sid_1), Ins(p_2, c_2, sid_2)) = Del(p_1 − $s(0)$, sid_1);

Trev(Del(p_1, sid_1), Del(p_2, sid_2)) = Del(p_1, sid_1);

// Fluent Functions

n = p⇒ car(n, do(Ins(p, c, sid), xSt)) = c;
$n \neq p$, (n > p) = true ⇒ car(n, do(Ins(p, c, sid), xSt)) = car(n − $s(0)$, xSt);
$n \neq p$, (n > p) = false ⇒ car(n, do(Ins(p, c, sid), xSt)) = car(n, xSt);
car(n, do(Del(p, sid), xSt)) = car(n, xSt);

n = p⇒ visible(n, do(Ins(p, c, sid), xSt)) = true;
$n \neq p$, (n > p) = true ⇒ visible(n, do(Ins(p, c, sid), xSt)) = visible(n − $s(0)$, xSt);
$n \neq p$, (n > p) = false ⇒ visible(n, do(Ins(p, c, sid), xSt)) = visible(n, xSt);
n = p⇒ visible(n, do(Del(p, sid), xSt)) = false;
$n \neq p$⇒ visible(n, do(Del(p, sid), xSt)) = visible(n, xSt);

length(do(Ins(p, c, sid), xSt)) = length(xSt) + $s(0)$;
length(do(Del(p, sid), xSt)) = length(xSt);

Properties:

system_is_sufficiently_complete;
system_is_strongly_sufficiently_complete;
system_is_ground_convergent;

Ind_priorities:

T poss ex;

Lemmas:

Strategy:

```
tautology_rule                  = delete(id, [tautology]);
negative_clash_rule             = delete(id, [negative_clash]);
subsumption_rule                = delete(id, [subsumption (L|C)]);
eliminate_redundant_literal_rule = simplify(id, [eliminate_redundant_literal]);
eliminate_trivial_literal_rule  = simplify(id, [eliminate_trivial_literal]);
positive_clash_rule             = simplify(id, [positive_clash]);
congruence_closure_rule         = simplify(id, [congruence_closure]);
negative_decomposition_rule = simplify(id, [negative_decomposition]);
auto_simplification_rule        = simplify(id, [auto_simplification]);
conditional_rewriting_rule2  = simplify(id, [conditional_rewriting(rewrite, R, *)]);
total_case_rewriting_rule       = simplify(id, [total_case_rewriting (simplify_strat, r, *)]);
induction1 = add_premise(generate,[id]);

stra = repeat (try (
            tautology_rule,
            negative_clash_rule,
            eliminate_redundant_literal_rule,
```

```
                    eliminate_trivial_literal_rule,
                    positive_clash_rule,
                    congruence_closure_rule,
                    negative_decomposition_rule,
                    auto_simplification_rule,
                    conditional_rewriting_rule2,
                    subsumption_rule,
                    print_goals,
                    total_case_rewriting_rule
        ));

        strategie = (induction1, stra, print_goals_with_history);
        rewrite   = (stra, print_goals_with_history);
        fullind   = (repeat(stra, induction1), print_goals_with_history);

start_with: fullind
```

conjectures:

// TP1 property
conc(op_1,op_2)=true, ex(do(op_1, xSt)) = true, ex(do(op_2, xSt)) = true ⇒
visible(c_i, do(T(op_2, op_1), do(op_1, xSt))) = visible(c_i, do(T(op_1, op_2), do(op_2, xSt)));
conc(op_1,op_2)=true, ex(do(op_1, xSt)) = true, ex(do(op_2, xSt)) = true ⇒
length(do(T(op_2, op_1), do(op_1, xSt))) = length(do(T(op_1, op_2), do(op_2, xSt)));
conc(op_1,op_2)=true, ex(do(op_1, xSt)) = true, ex(do(op_2, xSt)) = true ⇒
car(c_j, do(T(op_2, op_1), do(op_1, xSt))) = car(c_j, do(T(op_1, op_2), do(op_2, xSt)));

// TP2 property
conc(op_1,op_2)=true, conc(op_1,op_3)=true, conc(op_2,op_3)=true,
ex(do(op_1, xSt)) = true, ex(do(op_2, xSt)) = true, ex(do(op_3, xSt)) = true ⇒
T(T(op_3, op_1), T(op_2, op_1)) = T(T(op_3, op_2), T(op_1, op_2));

// Reversibility property
Trev(T(op_1,op_2),op_2) = op_1;

References

[1] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.

[2] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

[3] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[4] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[5] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving Correctness of Transformation Functions in Real-Time Groupware. In *Proceedings of the 8th European Conference on Computer-Supported Cooperative Work - ECSCW'03*, Helsinki, Finland, September 2003. ACM.

[6] A. Imine, P. Molli, G. Oster, and P. Urso. VOTE: Group editors analyzing tool. In *International Workshop on First-Order Theorem Proving - FTP'03*, Valencia, Spain, June 2003. Elsevier.

[7] A. Imine, M. Rusinovitch, G. Oster, and P. Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 2005.

[8] D. Li and R. Li. Ensuring content and intention consistency in real-time group editors. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems - ICDCS'04*, pages 748–755, Tokyo, Japan, 2004. IEEE Computer Society.

[9] D. Li and R. Li. Preserving operation effects relation in group editors. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work - CSCW 2004*, pages 457–466, New York, NY, USA, 2004. ACM Press.

[10] R. Li and D. Li. A landmark-based transformation approach to concurrency control in group editors. In *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work - GROUP 2005*, pages 284–293, New York, NY, USA, November 2005. ACM Press.

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

[12] F. Mattern. Virtual time and global states of distributed systems. In M. C. et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, October 1989. Elsevier Science Publishers B. V.

[13] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of the Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.

[14] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, pages 212–220, Sanibel Island, Florida, USA, 2003. ACM Press.

[15] G. Oster, P. Urso, P. Molli, and A. Imine. Edition collaborative sur reseau pair-a-pair à large échelle. In *Journées Francophones sur la Cohérence des Données en Univers Réparti - CDUR 2005*, Paris, France, Nov. 2005.

[16] G. Oster, P. Urso, P. Molli, and A. Imine. Real time group editors without operational transformation. Technical Report RR-5580, LORIA–INRIA Lorraine, May 2005.

[17] M. Ressel, D. Nitsche-Ruhland, and R. 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. ACM Press.

[18] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[19] S. Stratulat. A general framework to build contextual cover set induction provers. *Journal of Symbolic Computation*, 32(4):403–445, 2001.

[20] M. Suleiman, M. Cart, and J. 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, Phoenix, Arizona, United States, November 1997. ACM Press.

[21] M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 36–45, Orlando, Florida, USA, February 1998. IEEE Computer Society.

[22] C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(1):1–41, March 2002.

[23] C. Sun and C. 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, Seattle, Washington, United States, 1998. ACM Press.

[24] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. 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, 1998.

[25] N. Vidot, M. Cart, J. Ferrié, and M. 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. ACM Press.

Contents

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