Symbolic Model-Checking of Optimistic Replication Algorithms

Hanifa Boucheneb, Abdessamad Imine, Manal Najem

Symbolic Model-Checking of Optimistic Replication Algorithms

Hanifa Boucheneb[1], Abdessamad Imine[2], and Manal Najem[1]

[1] Laboratoire VeriForm, Department of Computer Engineering,
École Polytechnique de Montréal, P.O. Box 6079, Station Centre-ville, Montréal,
Québec,Canada, H3C 3A7
hanifa.boucheneb@polymtl.ca
[2] INRIA Grand-Est & Nancy-Université, France
imine@loria.fr

Abstract. The Operational Transformation (OT) approach, used in many collaborative editors, allows a group of users to concurrently update replicas of a shared object and exchange their updates in any order. The basic idea of this approach is to transform any received update operation before its execution on a replica of the object. This transformation aims to ensure the convergence of the different replicas of the object. However, designing transformation algorithms for achieving convergence is a critical and challenging issue. In this paper, we address the verification of OT algorithms with a symbolic model-checking technique. We show how to use the difference bound matrices to explore symbolically infinite state-spaces of such systems and provide symbolic counterexamples for the convergence property.

Key words: collaborative editors; operational transformation; difference bound matrices; symbolic model checking; convergence property.

1 Introduction

Motivations. Collaborative editing systems constitute a class of distributed systems where dispersed users interact by manipulating simultaneously some shared objects like texts, images, graphics, etc. One of the main challenges is the data consistency. To improve data availability, optimistic consistency control techniques are commonly used. The shared data is replicated so that the users update their local data replicas and exchange their updates between them. So, the updates are applied in different orders at different replicas of the object. This potentially leads to divergent (or different) replicas, an undesirable situation for collaborative editing systems. *Operational Transformation* (OT) is an optimistic technique which has been proposed to overcome the divergence problem [4]. This technique consists of an algorithm which transforms an update (previously executed by some other user) according to local concurrent updates in order to achieve convergence. It is used in many collaborative editors including Joint Emacs [8] (an Emacs collaborative editor), CoWord [13] (a collaborative version of Microsoft Word), CoPowerPoint [13] (a collaborative version of Microsoft PowerPoint) and, more recently, the Google Wave (a new google platform[3]).

It should be noted that the data consistency relies crucially on the correctness of an OT algorithm. According to [8], the consistency is ensured iff the transformation

[3] http://www.waveprotocol.org/whitepapers/operational-transform

function satisfies two properties $TP1$ and $TP2$ (explained in Section 2). Finding such a function and proving that it satisfies $TP1$ and $TP2$ is not an easy task. In addition, the proof by hand of these properties is often unmanageably complicated due to the fact that an OT algorithm has infinitely many states. Consequently, proving the correctness of OT algorithms should be assisted by automatic tools.

Related Work. Very little research has been done on automatically verifying the correctness of OT algorithms. To the best of our knowledge, [7] is the first work that addresses this problem. In this work, the authors have proposed a formal framework for modelling and verifying transformation functions with algebraic specifications. For checking the properties $TP1$ and $TP2$, they used an automatic theorem prover. However, this theorem proving approach has some shortcomings: (i) the model of the system is sound but not complete w.r.t. $TP1$ and $TP2$[4](i.e., it does not guarantee that the violation of property $TP1$ or $TP2$ is really feasible); (ii) there is no guidance to understand the counterexamples (when the properties are not verified); (iii) it requires some interaction (by injecting new lemmas) to complete the verification. In [3], the authors have used a model-checking technique to verify OT algorithms. This approach is not based on the verification of properties $TP1$ and $TP2$ but is instead based on the generation of the effective traces of the system. So, it allows to get a complete and informative scenario when a bug (a divergence of two copies of the shared object) is detected. Indeed, the output contains all necessary operations and the step-by-step execution that lead to the divergence situation. This approach guaranties that the detected divergence situations are really feasible. However, it needs to fix the shared object, the number of sites, the number of operations, the domains of parameters of operations and to execute explicitly the updates.

Contributions. We propose here a symbolic model-checking technique, based on difference bound matrices (DBMs) [1], to verify whether an OT algorithm satisfies properties $TP1$ and $TP2$. We show how to use DBMs, to handle symbolically the update operations of the collaborative editing systems and to verify symbolically the properties $TP1$ and $TP2$. The verification of these properties is performed automatically without carrying out different copies of the shared object and executing explicitly the updates. So, there is no need to fix the alphabet and the maximal length of the shared object. Thus, unlike [3], the symbolic model-checking proposed here enables us to get more abstraction and to build symbolic counterexamples. Moreover, for fixed numbers of sites and operations, it allows to prove whether or not an OT algorithm satisfies properties $TP1$ and $TP2$.

The paper starts with a presentation of the OT approach (Section 2). Section 3 is devoted to our symbolic model-checking. Conclusions are presented in Section 4.

2 Operational Transformation Approach

2.1 Background

OT is an optimistic replication technique which allows many sites to concurrently update the shared data and next to synchronize their divergent replicas in or-

[4] A model M of a system S is said to be sound w.r.t. a given property ϕ if M satisfies ϕ implies S satisfies ϕ. It is complete w.r.t. ϕ if S satisfies ϕ implies M satisfies ϕ.

der to obtain the same data. The updates of each site are executed on the local replica immediately without being blocked or delayed, and then are propagated to other sites to be executed again. The shared object is a finite sequence of elements from a data type $\mathcal{E}$ (alphabet). This data type is only a template and can be instantiated by many other types. For instance, an element may be regarded as a character, a paragraph, a page, a slide, an XML node, etc. It is assumed that the shared object can only be modified by the following primitive operations:

$$\mathcal{O} = \{Ins(p, e) | e \in \mathcal{E} \text{ and } p \in \mathbb{N}\} \cup \{Del(p) | p \in \mathbb{N}\} \cup \{Nop\}$$

where $Ins(p, e)$ inserts the element e at position p; $Del(p)$ deletes the element at position p, and Nop is the idle operation that has null effect on the object. Since the shared object is replicated, each site will own a local state l that is altered only by operations executed locally. The initial state of the shared object, denoted by l_0, is the same for all sites. Let $\mathcal{L}$ be the set of states. The function $Do : \mathcal{O} \times \mathcal{L} \rightarrow \mathcal{L}$, computes the state $Do(o, l)$ resulting from applying operation o to state l. We denote by $[o_1; o_2; \ldots; o_n]$ an operation sequence. Applying an operation sequence to a state l is defined as follows: (i) $Do([], l) = l$, where [] is the empty sequence and; (ii) $Do([o_1; o_2; \ldots; o_n], l) = Do(o_n, Do(\ldots, Do(o_2, Do(o_1, l))))$. Two operation sequences seq_1 and seq_2 are *equivalent*, denoted by $seq_1 \equiv seq_2$, iff $Do(seq_1, l) = Do(seq_2, l)$ for all states l.

The OT approach is based on two notions: concurrency and dependency of operations. Let o_1 and o_2 be operations generated at sites i and j, respectively. We say that o_2 *causally depends* on o_1, denoted $o_1 \rightarrow o_2$, iff: (i) $i = j$ and o_1 was generated before o_2; or, (ii) $i \neq j$ and the execution of o_1 at site j has happened before the generation of o_2. Operations o_1 and o_2 are said to be *concurrent*, denoted by $o_1 \parallel o_2$, iff neither $o_1 \rightarrow o_2$ nor $o_2 \rightarrow o_1$. As a long established convention in OT-based collaborative editors [4,11], the *timestamp vectors* are used to determine the causality and concurrency relations between operations. A timestamp vector is associated with each site and each generated operation. Every timestamp is a vector of integers with a number of entries equal to the number of sites. For a site j, each entry $V_j[i]$ returns the number of operations generated at site i that have been already executed on site j. When an operation o is generated at site i, a copy V_o of V_i is associated with o before its broadcast to other sites. $V_i[i]$ is then incremented by 1. Once o is received at site j, if the local vector V_j "dominates"[5] V_o, then o is ready to be executed on site j. In this case, $V_j[i]$ will be incremented by 1 after the execution of o. Otherwise, the o's execution is delayed.

Let V_{o_1} and V_{o_2} be timestamp vectors of o_1 and o_2, respectively. Using these timestamp vectors, the causality and concurrency relations are defined as follows:

(i) $o_1 \rightarrow o_2$ iff $V_{o_1}[i] < V_{o_2}[j]$; (ii) $o_1 \parallel o_2$ iff $V_{o_1}[i] \geq V_{o_2}[j]$ and $V_{o_2}[j] \geq V_{o_1}[j]$.

2.2 Operational Transformation approach

A crucial issue when designing shared objects with a replicated architecture and arbitrary messages communication between sites is the *consistency maintenance* (or *convergence*) of all replicas. To illustrate this problem, consider the group text editor scenario shown in Fig.1. There are two users (on two sites) working on a shared document represented by a sequence of characters. Initially, both copies hold the string " *efecte*". Site 1

[5] We say that V_1 dominates V_2 iff $\forall\, i$, $V_1[i] \geq V_2[i]$.

executes operation $o_1 = Ins(1,f)$ to insert the character f at position 1. Concurrently, site 2 performs $o_2 = Del(5)$ to delete the character *e* at position 5. When o_1 is received and executed on site 2, it produces the expected string "*effect*". But, when o_2 is received on site 1, it does not take into account that o_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 o_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 [4] was used to require that all sites execute o_1 and o_2 in the same order (*i.e.* a global order on concurrent operations) to obtain an identical result *effece*, this identical result is still inconsistent with the original intention of o_2.

To maintain convergence, the *Operational Transformation* (OT) approach has been proposed by [4]. When a site i gets an operation o that was previously executed by a site j on his replica of the shared object, the site i does not necessarily integrate o by executing it "as is" on his replica. It will rather execute a variant of o, denoted by o' (called a *transformation* of o) that *intuitively intends to achieve the same effect as* o. This transformation is based on an *Inclusive Transformation* (IT) function.

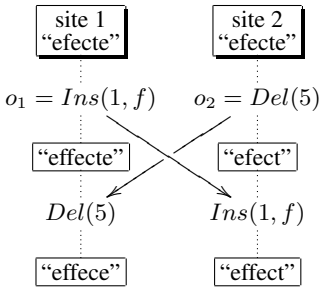

Fig. 1. Incorrect integration.

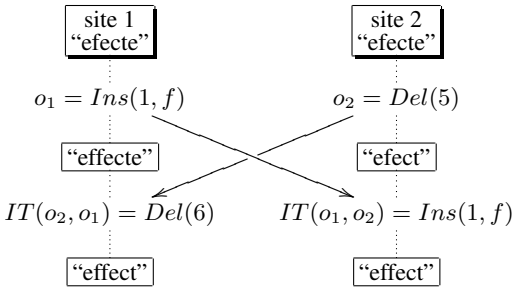

Fig. 2. Integration with transformation.

As an example, Fig.2 illustrates the effect of an IT function on the previous example. When o_2 is received on site 1, o_2 needs to be transformed according to o_1 as follows: $IT(Del(5), Ins(1,f)) = Del(6)$. The deletion position of o_2 is incremented because o_1 has inserted a character at position 1, which is before the character deleted by o_2. Next, o_2' is executed on site 1. In the same way, when o_1 is received on site 2, it is transformed as follows: $IT(Ins(1,f), Del(5)) = Ins(1,f)$; o_1 remains the same because f is inserted before the deletion position of o_2.

2.3 Inclusive transformation functions

We can find, in the literature, several IT functions: *Ellis*'s algorithm [4], *Ressel*'s algorithm [8], *Sun*'s algorithm [12], *Suleiman*'s algorithm [9] and *Imine*'s algorithm [6]. Due to the lack of space, we report, here, only the IT function proposed by Ellis and Gibbs [4]. In Ellis's IT function, the insert operation is extended with another parameter pr[6]. The priority pr is used to solve a conflict occurring when two concurrent insert

[6] This priority is calculated at the originating site. Two operations generated from different sites have always different priorities. Usually, the priority i is assigned to all operations generated at site i.

operations were originally intended to insert different characters at the same position. Note that concurrent editing operations have always different priorities. Fig.3 gives the four transformation cases for Ins and Del proposed by Ellis and Gibbs.

$$\text{IT}(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = \begin{cases} Ins(p_1, c_1, pr_1) & \text{if } (\mathrm{p_1} < \mathrm{p_2}) \vee (\mathrm{p_1} = \mathrm{p_2} \wedge \mathrm{c_1} \neq \mathrm{c_2} \wedge \mathrm{pr_1} < \mathrm{pr_2}) \\ Ins(p_1 + 1, c_1, pr_1) & \text{if } (\mathrm{p_1} > \mathrm{p_2}) \vee (\mathrm{p_1} = \mathrm{p_2} \wedge \mathrm{c_1} \neq \mathrm{c_2}) \wedge \mathrm{pr_1} > \mathrm{pr_2}) \\ Nop() & \text{if } \mathrm{p_1} = \mathrm{p_2} \wedge \mathrm{c_1} = \mathrm{c_2} \end{cases}$$

$$\text{IT}(Ins(p_1, c_1, pr_1), Del(p_2)) = \begin{cases} Ins(p_1, c_1, pr_1) & \text{if } \mathrm{p_1} < \mathrm{p_2} \\ Ins(p_1 - 1, c_1, pr_1) & \text{otherwise} \end{cases}$$

$$\text{IT}(Del(p_1), Ins(p_2, c_2, pr_2)) = \begin{cases} Del(p_1) & \text{if } \mathrm{p_1} < \mathrm{p_2} \\ Del(p_1 + 1) & \text{otherwise} \end{cases}$$

$$\text{IT}(Del(p_1), Del(p_2)) = \begin{cases} Del(p_1) & \text{if } \mathrm{p_1} < \mathrm{p_2} \\ Del(p_1 - 1) & \text{if } \mathrm{p_1} > \mathrm{p_2} \\ Nop() & \text{otherwise} \end{cases}$$

Fig. 3. IT function of Ellis et *al.*

Let $seq = [o_1; o_2; \ldots; o_n]$ be a sequence of operations. Transforming any editing operation o according to seq is denoted by $IT^*(o, seq)$ and is recursively defined by: $IT^*(o, []) = o$, where [] is the empty sequence, and $IT^*(o, [o_1; o_2; \ldots; o_n]) = IT^*(IT(o, o_1), [o_2; \ldots; o_n])$.

2.4 Integration procedures

Several integration procedures have been proposed in the groupware research area, such as dOPT [4], adOPTed [8], SOCT2,4 [10, 14] and GOTO [11]. Every site generates operations sequentially and stores these operations in a stack also called a *history* (or *execution trace*). When a site receives a remote operation o, the integration procedure executes the following steps:

1. From the local history seq, it determines the equivalent sequence seq' that is the concatenation of two sequences seq_h and seq_c where (i) seq_h contains all operations happened before o (according to the causality relation defined in Subsection 2.1), and (ii) seq_c consists of operations that are concurrent to o.
2. It calls the transformation component in order to get operation o' that is the transformation of o according to seq_c (*i.e.* $o' = IT^*(o, seq_c)$).
3. It executes o' on the current state and then adds o' to local history seq.

The integration procedure allows history of executed operations to be built on every site, provided that the causality relation is preserved. At stable state[7], history sites are not necessarily identical because the concurrent operations may be executed in different orders. Nevertheless, these histories must be equivalent in the sense that they must lead to the same final state.

2.5 Consistency criteria

An OT-based collaborative editor is *consistent* iff it satisfies the following properties:

1. *Causality preservation:* if $o_1 \rightarrow o_2$ then o_1 is executed before o_2 at all sites.

[7] A stable state is a state where all sites have executed the same set of operations but possibly in different orders.

2. *Convergence:* when all sites have performed the same set of updates, the copies of the shared document are identical.

To preserve the causal dependency between updates, timestamp vectors are used. In [8], the authors have established two properties $TP1$ and $TP2$ that are necessary and sufficient to ensure data convergence for *any number* of operations executed in *arbitrary order* on copies of the same object: For all o_0, o_1 and o_2 pairwise concurrent operations:

- $TP1$: $[o_0\,;IT(o_1,o_0)] \equiv [o_1\,;IT(o_0,o_1)]$.
- $TP2$: $IT^*(o_2,[o_0\,;IT(o_1,o_0)]) = IT^*(o_2,[o_1\,;IT(o_0,o_1)])$.

Property $TP1$ defines a *state identity* and ensures that if o_0 and o_1 are concurrent, the effect of executing o_0 before o_1 is the same as executing o_1 before o_0. Property $TP2$ ensures that transforming o_2 along equivalent and different operation sequences will give the same operation. Accordingly, by these properties, it is not necessary to enforce a global total order between concurrent operations because data divergence can always be repaired by operational transformation. However, finding an IT function that satisfies $TP1$ and $TP2$ is considered as a hard task, because this proof is often unmanageably complicated.

3 Modeling execution environment of the OT algorithms

We propose a symbolic model, based on the DBM data structure and communicating extended automata, to verify the consistency of OT algorithms. The DBM data structure is usually used to handle dense time domains in model-checkers of timed systems. It is used here to handle symbolically parameters (positions and symbols) of update operations (discrete domains) and verify properties $TP1$ and $TP2$ on traces (sequences of operations). Using DBM enables us to 1) abstract the shared object, 2) manipulate symbolically parameters of operations without fixing their sizes and 3) provide symbolic counterexamples for $TP1$ and $TP2$. First, we present the DBM data structure. Afterwards, our symbolic model of the OT execution environment is described. We show, at this level, how to use the DBM data structure to handle symbolically operations and verify properties $TP1$ and $TP2$.

3.1 Difference bound matrices

Let $X = \{x_1, ..., x_n\}$ be a finite and nonempty set of variables. An atomic constraint over X is a constraint of the form $x_i - x_j \prec c, x_i \prec c$, or $-x_j \prec c$, where $x_i, x_j \in X$, $\prec \in \{<, \leq\}$ and $c \in \mathbb{Z}$, $\mathbb{Z}$ being the set of integers. Constraints of the form $x_i - x_j \prec c$ are triangular constraints while the others are simple constraints. Constraints $x_i \prec x_j + c$, $x_i = x_j + c$, $x_i \geq x_j + c$, $x_i > x_j + c$, $x_i > c$, $-x_i > c$, $x_i \geq c$ and $-x_i \geq c$ are considered as abbreviations of atomic constraints.

In the context of this paper, X is a set of nonnegative integer variables (discrete variables), representing operation parameters (positions and lexical values of symbols). Therefore, atomic constraints $x_i - x_j < c, x_i < c$ and $-x_i < c$ are equivalent to $x_i - x_j \leq c - 1$, $x_i \leq c - 1$ and $-x_i \leq c - 1$, respectively. Moreover, we are interested in triangular constraints (i.e., all atomic constraints are supposed to be of the form $x_i - x_j \leq c$). In the rest of the paper, for simplicity, we will invariantly use atomic constraints or their abbreviations.

A difference bound matrix is used to represent a set of atomic constraints. Given a

set of atomic constraints A over the set of variables X. The DBM of A is the square matrix M of order $|X|$, where m_{ij} is the upper bound of the difference $x_i - x_j$ in A. By convention $m_{ii} = 0$, for every $x_i \in X$. In case, there is no constraint in A on $x_i - x_j$ ($i \neq j$), m_{ij} is set to ∞. For example, we report, in Table 1, the DBM M of the following set of atomic constraints:

$$A = \{x_2 - x_1 \leq 5, x_1 - x_2 \leq -1, x_3 - x_1 \leq 3, x_1 - x_3 \leq 0\}.$$

Though the same nonempty domain may be expressed by different sets of atomic constraints, their DBMs have a unique form called *canonical form*. The canonical form of a DBM is the representation with tightest bounds on all differences between variables. It can be computed, in $O(n^3)$, n being the number of variables in the DBM, using a shortest-path algorithm, like Floyd-Warshall's all-pairs shortest-path algorithm [1]. As an example, Table 1 shows the canonical form M' of the DBM M. Canonical forms make easier some operations over DBMs like the test of equivalence. Two sets of atomic constraints are equivalent iff the canonical forms of their DBMs are identical.

A set of atomic constraints may be inconsistent (i.e., its domain is empty). To verify the consistency of a set of atomic constraints, it suffices to apply a shortest-path algorithm and to stop the algorithm as soon as a negative cycle is detected. The presence of negative cycles means that the set of atomic constraints is inconsistent.

In the context of our work, we use, in addition to the test of equivalence, three other basic operations on DBMs: adding a constraint to a set of constraints, incrementing/decrementing a variable in a set of constraints. We establish, in the following, computation procedures for these operations which do not need any operation of canonization (computing canonical forms).

Let $X = \{x_1, ..., x_n\}$ be a finite and nonempty set of nonnegative integer variables, A a consistent set of triangular constraints over X, M the DBM, in canonical form, of A, x_i and x_j two distinct variables of X.

Incrementing by 1 a variable x_i in A is realized by replacing x_i with $x_i - 1$ ($old\ x_i = new\ x_i - 1$). Using the DBM M, in canonical form, of A, this incrementation consists of adding 1 to each element of the line x_i and subtracting 1 from each element of the column x_i. Intuitively, this corresponds to replacing each constraint $x_i - x_j \leq m_{ij}$ with $x_i - x_j \leq m_{ij} + 1$ and each constraint $x_j - x_i \leq m_{ji}$ with $x_j - x_i \leq m_{ji} - 1$. The resulting set of constraints and its DBM are denoted $A_{[x_i++]}$ and $M_{[i++]}$, respectively. The complexity of this operation is $O(n)$.

Similarly, to subtract 1 from a variable x_i in A, it suffices to replace x_i with $x_i + 1$ ($old\ x_i = new\ x_i + 1$). Using the DBM M, in canonical form, of A, this operation consists of subtracting 1 from each element of the line x_i and adding 1 to each element of the column x_i. The resulting set of constraints and its DBM are denoted $A_{[x_i--]}$ and $M_{[i--]}$, respectively. This operation is also of complexity $O(n)$. The following theorem establishes that $M_{[i++]}$ and $M_{[i--]}$ are in canonical form too. There is not need to compute their canonical forms.

Theorem 1. *(i) $A \cup \{x_i - x_j \leq c\}$ is consistent iff $m_{ji} + c \geq 0$. If $A \cup \{x_i - x_j \leq c\}$ is consistent, its DBM M', in canonical form, can be computed from M as follows: $M' = M$ if $m_{ij} \leq c$, and $(\forall k, l \in [1, n], m'_{kl} = Min(m_{kl}, m_{ki} + c + m_{jl}))$ otherwise. (ii) $M_{[i++]}$ and $M_{[i--]}$ are in canonical form.*

Proof. (i) A can be represented by a weighted and oriented graph where each constraint $x_l - x_k \leq d$ of A is represented by the edge (x_l, x_k, d). Since A is consistent, its

graph does not contain any negative cycle. Therefore, $A \cup \{x_i - x_j \leq c\}$ is consistent iff, in its graph, the shortest cycle going through edge (x_i, x_j, c) is nonnegative (i.e., $m_{ji} + c \geq 0$). If $A \cup \{x_i - x_j \leq c\}$ is consistent, then $\forall k, l \in [1, n], m'_{kl}$ is the weight of the shortest path connecting x_k to x_l, i.e., $m'_{kl} = Min(m_{kl}, m_{ki} + c + m_{jl})$. By assumption M is in canonical form. It follows that $m_{kl} \leq m_{ki} + m_{ij} + m_{jl}$ and then: $m_{ij} \leq c$ implies that $m'_{kl} = m_{kl}$.
(ii) M is in canonical form iff $\forall j, k, l \in [1, n], m_{jk} \leq m_{jl} + m_{lk}$.
We give the proof for $M_{[i++]}$. The proof for $M_{[i--]}$ is similar. By definition,

$$\forall j, k \in [1, n], \quad m_{[i++]_{jk}} = \begin{cases} m_{jk} & \text{if } j \neq i \wedge k \neq i \\ m_{jk} + 1 & \text{if } j = i \wedge k \neq i \\ m_{jk} - 1 & \text{if } j \neq i \wedge k = i \\ 0 & \text{if } j = i \wedge k = i \end{cases}$$

It follows that:
- If $j \neq i, k \neq i,$ and $l \neq i$ then: $m_{[i++]_{jk}} = m_{jk}$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} + m_{lk}$.
- If $j \neq i, k \neq i,$ and $l = i$ then: $m_{[i++]_{jk}} = m_{jk}$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} - 1 + m_{lk} + 1$.
- If $j = i, k \neq i,$ and $l \neq i$ then: $m_{[i++]_{jk}} = m_{jk} + 1$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} + 1 + m_{lk}$.
- If $j = i, k \neq i,$ and $l = i$ then: $m_{[i++]_{jk}} = m_{jk} + 1$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = 0 + m_{jk} + 1$.
- If $j \neq i, k = i,$ and $l \neq i$ then: $m_{[i++]_{jk}} = m_{jk} - 1$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} + m_{lk} - 1$.
- If $j \neq i, k = i,$ and $l = i$ then: $m_{[i++]_{jk}} = m_{jk} - 1$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} - 1 + 0$.
- If $j = i, k = i,$ and $l = i$ then: $m_{[i++]_{jk}} = 0$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = 0$.

By assumption $m_{jk} \leq m_{jl} + m_{lk}$. Then $m_{[i++]_{jk}} \leq m_{[i++]_{jl}} + m_{[i++]_{jl}}$. □

The complexity of the consistency test of $A \cup \{x_i - x_j \leq c\}$ is $O(1)$. The computation complexity of the canonical form of its DBM is reduced to $O(n^2)$.

For instance, consider the set of constraints A of the previous example and its DBM, in canonical form, M'. According to Theorem 1, $A \cup \{x_2 - x_3 \leq 0\}$ is consistent iff $m'_{32} + 0 \geq 0$ (i.e., $2 \geq 0$). We give in Table 1, DBMs M" of $A \cup \{x_2 - x_3 \leq 0\}$ and $M"_{2++}$.

Table 1. Some examples of DBMs

M	x_1	x_2	x_3
x_1	0	-1	0
x_2	5	0	∞
x_3	3	∞	0

M'	x_1	x_2	x_3
x_1	0	-1	0
x_2	5	0	5
x_3	3	2	0

M"	x_1	x_2	x_3
x_1	0	-1	-1
x_2	3	0	0
x_3	3	2	0

$M'_{[2++]}$	x_1	x_2	x_3
x_1	0	-2	0
x_2	4	0	1
x_3	3	1	0

3.2 Our symbolic Model

Our model of OT-based collaborative editor is a network of communicating extended automata. A communicating extended automaton is an automaton extended with finite sets of variables, binary channels of communication, guards and actions. In such automata, edges are annotated with selections, guards, synchronization signals and blocks of actions. Selections bind non-deterministically a given identifier to a value in a given range (type). The other three labels of an edge are within the scope of this binding. A state is defined by the current location and current values of all variables. An edge is enabled in a state if and only if the guard evaluates to true. The block of actions of an edge is executed atomically when the edge is fired. The side effect of this block changes the state of the system. Edges labelled with complementary synchronization signals over a

common channel must synchronize. Two automata synchronize through channels with a sender/receiver syntax [2]. For a binary channel, a sender can emit a signal through a given channel Syn ($Syn!$), if there is another automaton ready to receive the signal ($Syn?$). Both sender and receiver synchronize on execution of complementary actions $Syn!$ and $Syn?$. The update of the sender is executed before the update of the receiver.

An OT-based collaborative editor is composed of two or more sites (users) which communicate via a network and use the principle of multiple copies, to share some object (a text). Initially, each user has a copy of the shared object. It can afterwards modify its copy by executing operations generated locally and those received from other users. When a site executes a local operation, it is broadcast to all other users. The execution of a non local operation consists of the integration and the transformation steps as explained in Sub-section 2.4. To avoid managing queues of messages, the network is abstracted by allowing access to all operations and all timestamp vectors (declared as global variables). We propose also to abstract away the shared object and managing symbolically, using DBMs, the update operations.

Our OT-based collaborative editor model consists of one automaton per site, named $Site$, and an automaton named $Integration$ devoted to the integration procedure and the verification of properties $TP1$ and $TP2$. Each automaton has only one parameter which is also an implicit parameter of all functions defined in the automaton. The parameter of automaton $Site$ is the site identifier named pid. The parameter of automaton $Integration$ is the property $TP1$ or $TP2$ to be verified. These automata communicate via shared variables and a binary channel named Syn.

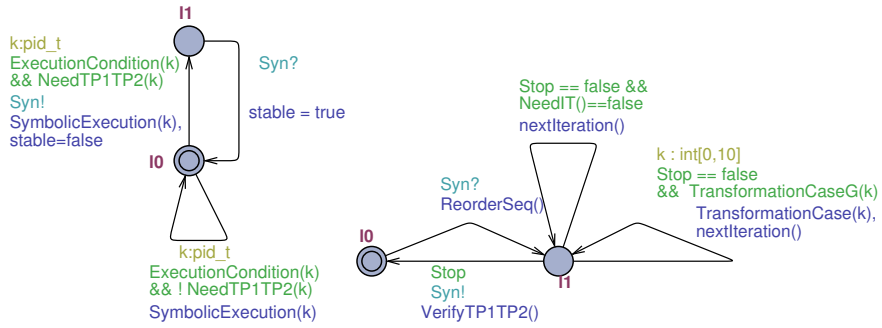

Fig. 4. Automata Site ($const\ pid\_t\ pid$) and Integration ($const\ Property\ prop$)

3.3 Automaton $Site$

This automaton, depicted in Fig.4 (left), is devoted to generate, using timestamp vectors of different sites ($V[NbSites][NbSites]$), all possible execution orders of operations (traces), respecting the causality principle. The loop on the initial location $l0$ specifies the symbolic execution of an operation leading to a situation where there is no need to verify properties $TP1$ and $TP2$. The transition from $l0$ to $l1$ corresponds to the symbolic execution of an operation that needs the verification of $TP1$ or $TP2$. The boolean function $ExecutionCondition(k)$ verifies whether a site pid can execute, according to the causality principle, an operation o of site k (k in pid_t). The function $SymbolicExecution(k)$ adds o to the symbolic trace of site pid, sets the timestamp vector of o to $V[pid]$, and then updates $V[pid]$ (i.e., $V[pid][pid]++$).

The integration of operations is performed by automaton $Integration$ when there is a need to verify $TP1$ or $TP2$ (i.e., function $NeedTP1TP2(k)$ returns $true$). For $TP1$, $NeedTP1TP2(k)$ returns $true$, if the execution, by site pid, of an operation o_1 of site k leads to a state where there is another site j s.t. traces of pid and j are $[seq_1; o_0; o_1]$ and $[seq_2; o_1; o_0]$, respectively, seq_1 and seq_2 are two equivalent operation sequences, o_0 and o_1 are two concurrent operations. This function returns $true$ for $TP2$, if the execution of an operation o_2 by site pid leads to a state where there is another site j s.t. traces of pid and j are $[seq_1; o_0; o_1; o_2]$ and $[seq_2; o_1; o_0; o_2]$, respectively, seq_1 and seq_2 are two equivalent sequences, o_0, o_1 and o_2 are pairwise concurrent operations.

3.4 Automaton $Integration$

This automaton, depicted in Fig.4 (right), is devoted to the verification of properties $TP1$ and $TP2$ on two sequences of operations. The verification starts when it receives a signal Syn from any site. It consists of applying the integration procedure to each sequence and then verifying that the resulting sequences satisfy the property $TP1$ or $TP2$. As explained in Section 2.4, the integration procedure of a non local operation o, in a sequence seq, consists of two steps: 1) computing a sequence seq' equivalent to seq, where operations dependent of o precede the concurrent ones (this is the role of function $ReorderSeq()$), and 2) transforming o against seq' (i.e. $IT^*(o, seq')$), realized by loops on location $l1$ and functions $TransformationCaseG(k)$, $TransformationCase(k)$ and $NextIteration()$. The loop containing $NeedIT() == false$ is executed if o does not need to be transformed against the current operation of seq'. Otherwise, the other loop is executed and o is transformed against the current operation of seq'. The verification of properties $TP1$ and $TP2$ on two sequences is performed by $VerifyTP1TP2()$, when the transformation process is completed for both sequences. The transformation and the verification are symbolic in the sense that operations are manipulated symbolically using DBMs.

3.5 Symbolic transformation

The transformation procedure is applied when there is a need to verify $TP1$ or $TP2$ on traces of two sites. To handle symbolically the update operations of these traces, we use a DBM over the positions of the original operations, their copies and also symbols of the original operations. Initially, there is no constraint on symbols of operations and the position of each original operation is identical to those of its copies. Note that, the transformation of an operation does not affect the symbols, but, in some IT functions, the transformation procedures depend on symbols. So, there is no need to represent, symbols of the copies of operations, since they are always equal to the original ones.

Let us explain, by means of an example, how to handle and transform symbolically an operation against another operation. Suppose that we need to verify $TP1$ on sequences $seq_0 = [o_0; o_1]$ of site 0 and $seq_1 = [o_1; o_0]$ of site 1, where operations o_0 and o_1 are concurrent and generated at sites 0 and 1, respectively. The automaton $Integration$ starts by calling function $ReorderSeq()$ to reorder sequences seq_0 and seq_1 as explained in Section 2.4 and create the initial DBM of the set of constraints $A = \{p_0 = p'_0 = p''_0, p_1 = p'_1 = p''_1\}$, where $p_i, p'_i, p''_i, i \in \{0, 1\}$ represent positions of operations o_i and its copies o'_i and o''_i, respectively. Afterwards, two transformations are performed sequentially by the automaton (loops on location $l1$): $IT(o'_1, o'_0)$ for Seq_0

and $IT(o''_0, o''_1)$ for Seq_1.

For $IT(o'_1, o'_0)$, the process offers different possibilities of transformation, through the selection block on k, which are explored exhaustively. Each value of k corresponds to a case of transformation. For instance, for Ellis's IT function, the eleven cases of transformation are shown in Fig.5. These cases are trivially derived from Ellis's IT algorithm given in Fig. 3. Note that initially, the kinds of operations are not fixed. They will be fixed when a case of transformation is selected. For example, $k = 10$ corresponds to the case where o'_0 is a delete operation, o'_1 is an insert operation and $p'_1 = p'_0$. The function $TransformationCaseG(10)$ returns true iff the set of constraints $A \cup \{p'_0 = p'_1\}$ is consistent, o'_0 is either a delete operation or not fixed yet[8] (Nfx), and o'_1 is either an insert operation or not fixed yet. In our case $TransformationCaseG(10)$ returns true and the function $TransformationCase(10)$ adds the constraint $p'_0 = p'_1$ to A (i.e., $A = \{p_0 = p'_0 = p''_0 = p_1 = p'_1 = p''_1\}$), decrements p'_1 in the resulting A (i.e., $A = \{p_0 = p'_0 = p''_0 = p_1 = p''_1, p_1 - p'_1 = 1\}$), sets o_0, o'_0, o''_0 to delete operations, and o_1, o'_1, o''_1 to insert operations.

For $IT(o''_0, o''_1)$, the automaton $Integration$ offers only one possibility of transformation corresponding to the case fixed by the previous transformation, i.e., o''_0 is a delete operation, o''_1 is an insert operation and $p''_0 = p''_1$. In this case, the function $TransformationCase$ adds the constraint $p''_0 = p''_1$ to A (i.e., $A = \{p_0 = p'_0 = p''_0 = p_1 = p''_1, p_1 - p'_1 = 1\}$) and increments p''_0 in the resulting A, (i.e., $A = \{p_0 = p'_0 = p_1 = p''_1, p_1 - p'_1 = 1, p_0 - p''_0 = -1\}$) (see Fig.6 and Fig.7).

3.6 Verification of $TP1$ and $TP2$

Property $TP1$ ensures that the execution of two operations o_0 and o_1 in different orders, on two identical copies of a text, has the same effect. To compare the effects of sequences $[o'_0; o'_1]$ and $[o''_0; o''_1]$ on two identical copies of a text, it suffices to determine the final positions, in each copy, of all parts affected, when both operations are executed. The affected parts must be the same in both copies. Since the operations are handled symbolically (represented by a set of atomic constraints A (i.e., DBMs)), the effect of these symbolic operations must be also represented by a set of constraints. Property $TP1$ is not satisfied if the number of idle operations differs from one sequence to the other. It is also not satisfied if there is an idle operation in each sequence and the remaining ones are of different types, their positions are different or their symbols are different.

For the other cases, let us first explain how to verify $TP1$ on our previous example (see Fig.7). Since, in $A = \{p_0 = p'_0 = p_1 = p''_1 = p'_1 + 1 = p''_0 - 1\}$, constraint $p'_1 < p'_0$ is always satisfied, it follows that after executing the sequence $[Del(p'_0); Ins(p'_1, c_1)]$, the positions of the deleted and the inserted elements are $p'_0 + 1$ and p'_1, respectively. In A, constraint $p''_1 > p''_0$ is also always satisfied. Therefore, after executing the sequence $[Ins(p''_1, c_1); Del(p''_0)]$, the inserted and the deleted elements are at positions p''_1 and p''_0, respectively. Property $TP1$ is satisfied iff each valuation of the domain of A, satisfies the both constraints: $p'_0 + 1 = p''_0$ and $p'_1 = p''_1$, i.e., $A = A \cup \{p'_0 + 1 = p''_0, p'_1 = p''_1\}$. Since, in A, we have $p'_1 \neq p''_1$, it follows that $TP1$ is not satisfied for Ellis's IT algorithm

[8] Nfx means that the operation type is not fixed yet and then can be set to Del or Ins.

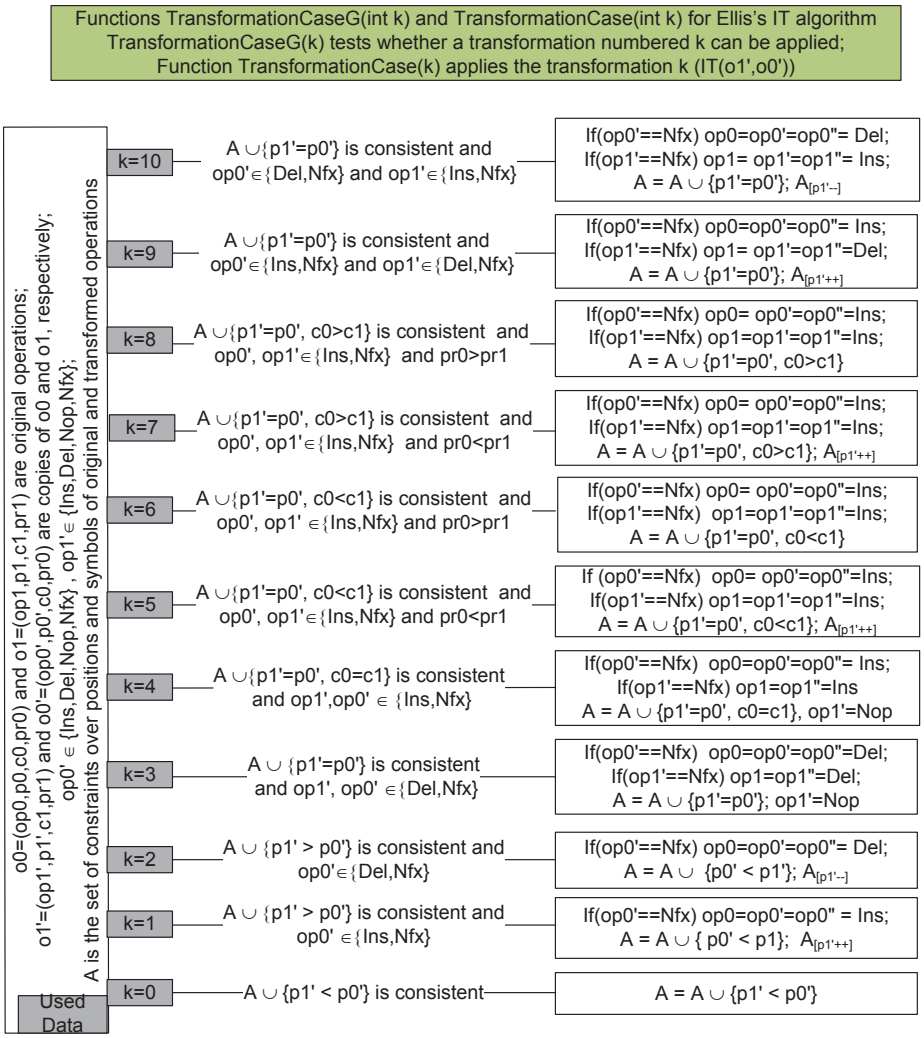

Fig. 5. Symbolic IT algorithm of Ellis

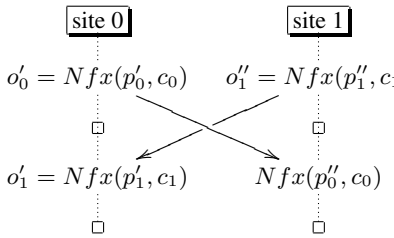

Fig. 6. Before integration: $A = \{p_0 = p'_0 = p''_0, p_1 = p'_1 = p''_1\}$

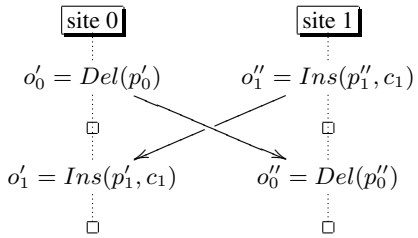

Fig. 7. After Integration for $k = 10$: $A = \{p_0 = p'_0 = p_1 = p''_1 = p'_1 + 1 = p''_0 - 1\}$

and then the previous example is a symbolic counterexample for $TP1$. Each nonnegative valuation of positions p_0 and p_1 that satisfies constraints of A but do not satisfy constraints of $A \cup \{p'_0 = p''_0, p'_1 = p''_1\}$ corresponds to a concrete counterexample. As an example, for $p_0 = p_1 = 2$, we obtain the counterexample where sequences executed by sites 0 and 1 are $[Del(2); Ins(1, c_1)]$ and $[Ins(2, c_1); Del(3)]$, respectively. These sequences lead to a divergence. For instance, if the initial text is $abcde$, the previous sequences lead to two different copies: ac_1bde and abc_1de.

To verify $TP1$, function $VerifyTP1TP2$ partitions the domain of A in three or four partitions as shown in Table 2 and then determines, for each consistent partition, the positions of the inserted/deleted elements when both sequences are executed. We give, in Table 2, the different partitions of A and the associated conditions that guarantee to get the same effect on both copies of the shared text, for different sequences of two non idle operations. As an example, suppose two sequences $seq' = [Ins(p'_0, c_0); Ins(p'_1, c_1)]$ and $seq'' = [Ins(p''_1, c_1); Ins(p''_0, c_0)]$. Let us compare the effects of these sequences. The positions of the inserted elements, when seq' is executed, depend on the relationships between p'_0 and p'_1: $p'_0 + 1$ and p'_1, if $p'_0 \geq p'_1$; and p'_0 and p'_1 otherwise. Similarly, the positions of the inserted elements when seq'' is executed are: $p''_1 + 1$ and p'_0, if $p''_1 \geq p''_0$; and p''_1 and p''_0 otherwise. Therefore, to compare the effects of both sequences, four cases are considered (see Fig.8). Each case has its own condition of convergence. For example, for $p'_0 \geq p'_1 \wedge p''_1 \geq p''_0$, the condition of convergence is $A \cup \{p'_0 \geq p'_1, p''_1 \geq p''_0\} = A \cup \{p'_0 \geq p'_1, p''_1 \geq p''_0, p'_0 = p''_1, p'_1 = p''_0, c_0 = c_1\}$.

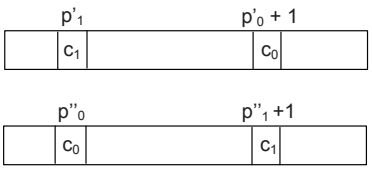

Fig. 8. Effect of seq' and seq'' in case $p'_0 \geq p'_1 \wedge p''_1 \geq p''_0$

The verification of property $TP2$ is much more simpler than $TP1$. Let $[seq_0; o_0; o_1; o_2]$ and $[seq_1; o_1; o_0; o_2]$ be a pair of equivalent sequences, such that o_0, o_1 and o_2 are pairwise concurrent operations. Verifying $TP2$ consists of testing that o_2 is transformed in the same manner against sequences $[seq_0; o_0; o_1]$ and $[seq_1; o_1; o_0]$.

For both properties, function $VerifyTP1TP2$ sets a boolean variable named $Detected$ to $true$ as soon as the violation of property $TP1$ or $TP2$ is detected. This variable is initially set to $false$.

Our approach is sound and complete w.r.t. to the comvergence property as it generates only feasible traces (in respect with the causality principle) and the integration procedure is performed exactly as in OT-based collaborative editors. Properties $TP1$ and $TP2$ are verified on feasible traces in conformity with their definitions.

We have used the on-the-fly model-checker UPPAAL[9] to test the symbolic model proposed here. The Computation tree logic (CTL) formula [5] $AG\ not\ Detected$ allows us to verify whether or not property $TP1$ or $TP2$ is satisfied. We have considered several IT functions: $Ellis$ [4], $Ressel$ [8], Sun [12], $Suleiman$ [9] and $Imine$ [6]. To test different IT functions, it suffices to rewrite accordingly functions

[9] www.uppaal.com

$TransformationCaseG(k)$ and $TransformationCase(k)$. We give, in Table 3, the results obtained for both properties $TP1$ and $TP2$ in the case of four operations o_0, o_1, o_2, o and three sites. All operations are pairwise concurrent, except that o_2 is causally dependent of o. The property $TP2$ is not satisfied for all considered IT functions. A symbolic counterexample is provided for each unsatisfied property. We report also the number of explored/computed abstract states and the time, in second, of the verification, under UPPAAL, of CTL formula $AG\ not\ Detected$.

$[Ins(p'_0, c_0); Ins(p'_1, c_1)] \mid\mid [Ins(p''_1, c_1); Ins(p''_0, c_0)]$	
Partitions of A	Convergence condition
$A_1 = A \cup \{p'_1 \leq p'_0, p''_1 < p''_0\}$	$A_1 = A_1 \cup \{p'_0 + 1 = p''_0, p'_1 = p''_1\}$
$A_2 = A \cup \{p'_1 \leq p'_0, p''_1 \geq p''_0\}$	$A_2 = A_2 \cup \{p'_0 = p''_1, p'_1 = p''_0, c_0 = c_1\}$
$A_3 = A \cup \{p'_1 > p'_0, p''_1 < p''_0\}$	$A_3 = A_3 \cup \{p'_0 = p''_1, p'_1 = p''_0, c_0 = c_1\}$
$A_4 = A \cup \{p'_1 > p'_0, p''_1 \geq p''_0\}$	$A_4 = A_4 \cup \{p'_0 = p''_0, p'_1 + 1 = p''_1\}$

$[Del(p'_0); Del(p'_1)] \mid\mid [Del(p''_1); Del(p''_0)]$	
Partitions of A	Convergence condition
$A_1 = A \cup \{p'_1 < p'_0, p''_1 \leq p''_0\}$	$A_1 = A_1 \cup \{p'_0 = p''_0 + 1, p'_1 = p''_1\}$
$A_2 = A \cup \{p'_1 < p'_0, p''_1 > p''_0\}$	$A_2 = A_2 \cup \{p'_0 = p''_1, p'_1 = p''_0\}$
$A_3 = A \cup \{p'_1 \geq p'_0, p''_1 \leq p''_0\}$	$A_3 = A_3 \cup \{p'_0 = p''_1, p'_1 = p''_0\}$
$A_4 = A \cup \{p'_1 \geq p'_0, p''_1 > p''_0\}$	$A_4 = A_4 \cup \{p'_0 = p''_0, p'_1 = p''_1 + 1\}$

$[Del(p'_0); Ins(p'_1, c_1)] \mid\mid [Ins(p''_1, c_1); Del(p''_0)]$	
Partitions of A	Convergence condition
$A_1 = A \cup \{p'_1 < p'_0\}$	$A_1 = A_1 \cup \{p'_0 + 1 = p''_0, p'_1 = p''_1\}$
$A_2 = A \cup \{p'_1 \geq p'_0, p''_1 \leq p''_0\}$	$A_2 = A_2 \cup \{p'_0 = p''_1, p'_1 + 1 = p''_0, p'_0 = p'_1\}$
$A_3 = A \cup \{p'_1 \geq p'_0, p''_1 > p''_0\}$	$A_3 = A_3 \cup \{p'_0 = p''_0, p'_1 = p''_1 - 1\}$

Table 2. Symbolic verification of TP1

4 Conclusion

We have proposed here a symbolic model-checking technique to verify that an OT algorithm used, in replication-based collaborative editors ensures convergence of replicas. In our technique, the shared objects are abstracted and their update operations are handled symbolically using difference bound matrices. Unlike in [3], there is no need to fix neither the shared object nor sizes of parameters of its update operations. Unlike in [7], our approach allows us to provide symbolic feasible counterexamples for the convergence property. Indeed, in [7], the verification of convergence is not based on only feasible traces. Consequently, it is sound but not complete. Our approach is sound and complete. However, its termination needs to fix the numbers of sites and operations. We plan to determine, if they exist, the smallest values for m and n s.t. an OT algorithm ensures convergence for m operations and n sites implies that the OT algorithm ensures convergence for any arbitrary numbers of operations and sites.

References

1. G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *Theoretical Computer Science*, 8(3), 2006.
2. B. Bérard, P. Bouyer, and A. Petit. Analysing the pgm protocol with UPPAAL. *International Journal of Production Research*, 42(14):2773–2791, 2004.
3. H. Boucheneb and A. Imine. On model-checking optimistic replication algorithms. In *FMOODS/FORTE*, pages 73–89, 2009.

IT function	Verification of TP1 and TP2
Ellis $TP1$	$false, o_0 \in \{Ins(p_0, c_0), Del(p_0)\}, o_1 = Ins(p_1, c_1), p_1 < p_0$
Sizes / Time	234 / 157 / 0.452
$TP2$	$false, o_0 = Del(p_0), o_1 = Ins(p_0 - 1, c_1), o_2 = Ins(p_0, c_1)$
Sizes / Time	667 / 561 / 1.029
Ressel $TP1$	$true$
Sizes / Time	788 / 788 / 0.811
$TP2$	$false, o_0 = Del(p_0), o_1 = Ins(p_0 + 1, c_1), o_2 = Ins(p_0, c_2)$
Sizes / Time	477 / 413 / 0.686
Sun $TP1$	$false, o_0 \in \{Ins(p_0, c_0), Del(p_0)\}, o_1 = Ins(p_1, c_1), p_1 < p_0$
Sizes / Time	225 / 156 / 0.405
$TP2$	$false, o_0 = Ins(p_0, c_0), o_1 = Del(p_0 - 1), o_2 = Ins(p_0 - 1, c_2)$
Sizes / Time	477 / 413 / 0.717
Suleiman $TP1$	$true$
Sizes / Time	1023 / 1023 / 1.060
$TP2$	$false, o = Del(p_0 - 1), o_0 = Ins(p_0, c_0), o_1 = Ins(p_0 - 1, c_1),$ $o_2 = Ins(p_0 - 1, c_2), c_1 < c_2 < c_0$
Sizes / Time	10961 / 9913 / 2.124
Imine $TP1$	$true$
Sizes / Time	963 / 963 / 1.130
$TP2$	$false, o = Del(p), o_0 = Del(p_0), o_1 = Ins(p_0, c_1),$ $o_2 = Ins(p_0, c_2), p + 1 \leq p_0, c_2 < c_1$
Sizes / Time	9730 / 8808 / 2.130

Table 3. Verification of $TP1$ and $TP2$ for five IT functions proposed in the literature

4. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.
5. E. A. Emerson. *Temporal and modal logic*. In Handbook of Theoretical Computer Science (vol. B): Formal Methods and Semantics, Chapter 16, 1990.
6. A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *ECSCW'03*, Helsinki, Finland, 14.-18. September 2003.
7. A. Imine, M. Rusinowitch, G. Oster, and P. Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 351(2):167–183, 2006.
8. M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *ACM CSCW'96*, pages 288–297, Boston, USA, November 1996.
9. M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *ACM GROUP'97*, pages 435–445, November 1997.
10. M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *IEEE ICDE'98*, pages 36–45, 1998.
11. C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *ACM CSCW'98*, pages 59–68, 1998.
12. 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 Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.
13. C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13(4):531–582, 2006.
14. N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *ACM CSCW'00*, Philadelphia, USA, December 2000.