# On Model-Checking Optimistic Replication Algorithms

Hanifa Boucheneb[1] and Abdessamad Imine[2]

[1] Laboratoire VeriForm, École Polytechnique de Montréal, Canada
hanifa.boucheneb@polymtl.ca
[2] INRIA Grand-Est & Nancy-Université, France
imine@loria.fr

**Abstract.** Collaborative editors consist of a group of users editing a shared document. The Operational Transformation (OT) approach is used for supporting optimistic replication in these editors. It allows the users to concurrently update the shared data and exchange their updates in any order since the convergence of all replicas, *i.e.* the fact that all users view the same data, is ensured in all cases. However, designing algorithms for achieving convergence with the OT approach is a critical and challenging issue. In this paper, we address the verification of OT algorithms with a model-checking technique. We formally define, using tool *UPPAAL*, the behavior and the convergence requirement of the collaborative editors, as well as the abstract behavior of the environment where these systems are supposed to operate. So, we show how to exploit some features of such systems and the tool *UPPAAL* to attenuate the severe state explosion problem. We have been able to show that if the number of users exceeds 2 then the convergence property is not satisfied for five OT algorithms. A counterexample is provided for every algorithm.

## 1 Introduction

Collaborative editors are a class of distributed systems, where two or more users (sites) may manipulate simultaneously some objects like texts, images, graphics, etc. In order to achieve an unconstrained group work, the shared objects are replicated at the local memory of each participating user. Every operation is executed locally first and then broadcast for execution at other sites. So, the operations are applied in different orders at different replicas of the object. This potentially leads to divergent (or different) replicas, an undesirable situation for replication-based collaborative editors. *Operational Transformation* (OT) is an approach which has been proposed to overcome the divergence problem [4]. This approach consists of an algorithm which transforms an operation (previously executed by some other site) according to local concurrent ones in order to achieve convergence. It has been used in many collaborative editors such as Joint Emacs [9] (an Emacs collaborative editor), CoWord [14] (a collaborative version of MicroSoft Word) and CoPowerPoint [14] (a collaborative version of MicroSoft PowerPoint).

As established in [12], an OT algorithm consists of two parts: (i) an *integration procedure* that is responsible for generating and propagating local operations as well as executing remote operations; (ii) a *transformation function* (called IT function) that

determines how an operation is transformed against another. This function depends on the semantics of the shared document. However, if an OT algorithm is not correct then the consistency of shared data is not ensured. Thus, it is critical to verify such an algorithm in order to avoid the loss of data when broadcasting operations. According to [9], only the transformation function of a shared data needs to fulfill two properties $TP1$ and $TP2$ (explained in Section 2) in order to ensure convergence. Finding such a function and proving that it satisfies $TP1$ and $TP2$ is not an easy task. This proof is often unmanageably complicated due to the fact that an OT algorithm has infinitely many states.

In this paper, we investigate the use of a model-checking technique [1] to verify whether an OT algorithm satisfies the convergence property or not. Model-checking is a very attractive and automatic verification technique of systems. It is applied by representing the behavior of a system as a finite *state transition system*, specifying properties of interest in a temporal logic and finally exploring the state transition system to determine whether they hold or not. The main interesting feature of this technique is the production of counterexamples in case of unsatisfied properties. Several Model-checkers have been proposed in the literature. The well known are *SPIN*[3], *UPPAAL*[4] and *NuSMV*[5]. Among these Model-checkers, we consider here the tool *UPPAAL*.

*UPPAAL* is a tool suite for validation and symbolic model-checking of real-time systems. It consists of a number of tools including a graphical editor for system descriptions, a graphical simulator, and a symbolic model-checker. This choice is motivated by the interesting features of *UPPAAL* tools [8], especially the powerful of its description model, its simulator and its symbolic model-checker. Indeed, its description model is a set of timed automata [1] extended with binary channels, broadcast channels, C-like types, variables and functions (functions can be used to abstract some complicated treatments). Its simulator is useful and convivial as it allows to get and replay, step by step, counterexamples obtained by its symbolic model-checker. Its model-checker[6], based on a forward on-the-fly method, allows to compute over 5 millions of states.

In this work, we deal with OT algorithms that have the same integration procedure but differ only by their transformation functions. To verify these algorithms, we formally describe, using *UPPAAL*, the behavior and the requirements of the replication-based collaborative editors, as well as the abstract behavior of the environment where these systems are supposed to operate. Two main models are studied and proposed for the verification of the convergence properties of OT algorithms: the *concrete model* and the *symbolic model*. The concrete model is very close to the system implementation in the sense that the selection and the effective execution of editing operations are performed during the construction of execution traces. However, this model runs up against a severe explosion of states (the number of signatures increases exponentially with the number of operations). We have not been able to verify some OT algorithms. The symbolic model aims to overcome the limitation of the concrete model by delaying the effective selection and execution of editing operations until the construction of

[3] http://spinroot.com
[4] http://www.uppaal.com
[5] http://nusmv.irst.itc.it
[6] The model-checker is used without the graphical interface

symbolic execution traces of all sites is completed. Using the symbolic model, we have been able to show that if the number of sites exceeds 2 then the convergence property is not satisfied for all OT algorithms considered here. A counterexample is provided for every algorithm.

The paper starts with a presentation of the OT approach and one of the known OT algorithms proposed in the literature for synchronizing shared text documents (Section 2). Section 3 is devoted to the description of the symbolic model and its model-checking. Related work and conclusion are presented respectively in sections 4 and 5.

## 2 Operational Transformation Approach

### 2.1 Background

OT is an optimistic replication technique which allows many users (or sites) to concurrently update the shared data and next to synchronize their divergent replicas in order 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. Accordingly, every update is processed in four steps: (i) *generation* on one site; (ii) *broadcast* to other sites; (iii) *reception* on one site; (iv) *execution* on one site.

**The shared object.** We deal with a shared object that admits a linear structure. To represent this object we use the *list* abstract data type. A *list* is a finite sequence of elements from a data type $\mathcal{E}$. 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. Let $\mathcal{L}$ be the set of lists.

**The primitive operations.** It is assumed that a list state can only be modified by the following primitive operations: (i) $Ins(p,e)$ which inserts the element $e$ at position $p$; (ii) $Del(p)$ which deletes the element at position $p$. We assume that positions are given by natural numbers. The set of operations is defined as follows:

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

where $Nop$ is the idle operation that has null effect on the list state. Since the shared object is replicated, each site will own a local state $l$ that is altered only by local operations. The initial state, denoted by $l_0$, is the same for all sites. The function $Do : O \times \mathcal{L} \rightarrow \mathcal{L}$, computes the state $Do(o,l)$ resulting from applying operation $o$ to state $l$. We say that $o$ is *generated* on state $l$. We denote by $[o_1;o_2;\ldots;o_n]$ an operation sequence. Applying an operation sequence to a list $l$ 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 lists $l$.

**Definition 1.** ***(Causality Relation)*** *Let an operation $o_1$ be generated at site $i$ and an operation $o_2$ be generated at site $j$. 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 $o_2$.*

**Definition 2.** ***(Concurrency Relation)*** *Two 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 $o_2 \rightarrow o_1$.*

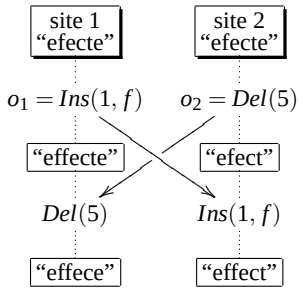


**Fig. 1.** Incorrect integration.

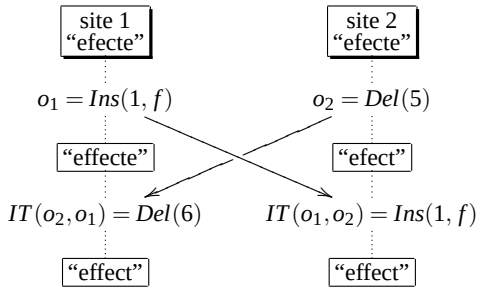


**Fig. 2.** Integration with transformation.

As a long established convention in OT-based collaborative editors [4,12], the *timestamp vectors* are used to determine the causality and concurrency relations between operations. Every timestamp is a vector $V$ of integers with a number of entries equal to the number of sites. For a site $j$, each entry $V[i]$ returns the number of operations generated at site $i$ that have been already executed on site $j$. Let $o_1$ and $o_2$ be two operations issued respectively at sites $s_{o_1}$ and $s_{o_2}$ and equipped with their respective timestamp vectors $V_{o_1}$ and $V_{o_2}$. The causality and concurrency relations are detected as follows: (i) $o_1 \rightarrow o_2$ iff $V_{o_1}[s_{o_1}] > V_{o_2}[s_{o_1}]$; (ii) $o_1 \parallel o_2$ iff $V_{o_1}[s_{o_1}] \leq V_{o_2}[s_{o_1}]$ and $V_{o_1}[s_{o_2}] \geq V_{o_2}[s_{o_2}]$.

### 2.2 Transformation principle

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.

*Example 1.* Consider the following group text editor scenario (see Fig.1): there are two users (on two sites) working on a shared document represented by a sequence of characters. These characters are addressed from 0 to the end of the document. Initially, both copies hold the string “*efecte*”. User 1 executes operation $o_1 = Ins(1, f)$ to insert the character $f$ at position 1. Concurrently, user 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 OT approach has been proposed by [4]. When User $X$ gets an operation $op$ that was previously executed by User $Y$ on his replica of the shared object User $X$ does not necessarily integrate $op$ by executing it “as is” on his replica. He will rather execute a variant of $op$, denoted by $op'$ (called a *transformation* of $op$) that *intuitively intends to achieve the same effect as* $op$. This approach is based on a transformation function $IT$, called *Inclusive Transformation*, that applies to couples of concurrent operations defined on the same state.

*Example 2.* In Fig.2, we illustrate the effect of $IT$ 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, $op'_2$ 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 Transformation function

We present here an IT function known in the literature for synchronizing linear objects [10] altered by insertion and deletion operations. In this work, the signature of insert operation is extended by two parameters *pre* and *post*. These parameters store the set of concurrent delete operations. The set *pre* contains operations that have deleted a character before the insertion position *p*. As for *post*, it contains operations that have removed a character after *p*. When an insert operation is generated the parameters *pre* and *post* are empty. They will be filled during transformation steps.

In Fig.3, we give the four transformation cases for *Ins* and *Del* proposed by Suleiman and *al* [10]. There is an interesting situation in the first case (*Ins* and *Ins*), called *conflict situation*, where two concurrent $Ins(p_1, c_1, pre_1, post_1)$ and $Ins(p_2, c_2, pre_2, post_2)$ have the same position (*i.e.* $p_1 = p_2$). To resolve this conflict, three cases are possible:

1. $(pre_1 \cap post_2) \neq \emptyset$: character $c_2$ is inserted before character $c_1$,
2. $(pre_1 \cap post_2) \neq \emptyset$: character $c_2$ is inserted after character $c_1$,
3. $(pre_1 \cap post_2) = (post_1 \cap pre_2) = \emptyset$: in this case function $code(c)$, which computes a total order on characters (*e.g.* lexicographic order), is used to choose among $c_1$ and $c_2$ the character to be added before the other. Like the site identifiers, $code(c)$ enables us to tie-break conflict situations [3].

Note that when two concurrent operations insert the same character (*e.g.* $code(c_1) = code(c_2)$) at the same position, the one is executed and the other one is ignored by returning the idle operation $Nop$. In other words, only one character is kept. The remaining cases of $IT$ are quite simple.

### 2.4 Transformation Properties

**Definition 3.** *Let seq be a sequence of operations. Transforming any editing operation o according to seq is denoted by $IT^*(o, seq)$ and is recursively defined as follows:*

$$IT^*(o, []) = o \text{ where } [] \text{ is the empty sequence;}$$
$$IT^*(o, [o_1; o_2; \ldots; o_n]) = IT^*(IT(o, o_1), [o_2; \ldots; o_n])$$

*We say that o has been concurrently generated according to all operations of seq.*

Using an IT function requires us to satisfy two properties [9]. For all $o$, $o_1$ and $o_2$ pairwise concurrent operations:

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

$$\mathrm{IT}(Ins(p_1,c_1,pre_1,post_1),Ins(p_2,c_2,pre_2,post_2)) =$$
$$\begin{cases} Ins(p_1,c_1,pre_1,post_1) & \text{if } \mathrm{p_1} < \mathrm{p_2} \\ Ins(p_1+1,c_1,pre_1,post_1) & \text{if } (\mathrm{p_1} > \mathrm{p_2}) \vee (\mathrm{p_1} = \mathrm{p_2} \wedge \mathrm{pre_1} \cap \mathrm{post_2} \neq \emptyset) \\ Ins(p_1,c_1,pre_1,post_1) & \text{if } \mathrm{p_1} = \mathrm{p_2} \wedge \mathrm{post_1} \cap \mathrm{pre_2} \neq \emptyset \\ Ins(p_1,c_1,pre_1,post_1) & \text{if } (\mathrm{pre_1} \cap \mathrm{post_2} = \emptyset \vee \mathrm{pre_1} \cap \mathrm{post_2} = \emptyset) \ \wedge \\ & \ p_1 = p_2 \wedge code(c_1) > code(c_2) \\ Ins(p_1+1,c_1,pre_1,post_1) & \text{if } (\mathrm{pre_1} \cap \mathrm{post_2} = \emptyset \vee \mathrm{post_1} \cap \mathrm{pre_2} = \emptyset) \ \wedge \\ & \ p_1 = p_2 \wedge code(c_1) < code(c_2) \\ Nop() & \text{otherwise} \end{cases}$$
$$\mathrm{IT}((Ins(p_1,c_1,pre_1,post_1),Del(p_2)) = \begin{cases} Ins(p_1,c_1,pre_1,post_1 \cup \{Del(p_2)\}) & \text{if } \mathrm{p_1} \leq \mathrm{p_2} \\ Ins(p_1-1,c_1,pre_1 \cup \{Del(p_2)\},post_1) & \text{otherwise} \end{cases}$$
$$\mathrm{IT}((Del(p_1),Ins(p_2,c_2,pre_2,post_2)) = \begin{cases} Del(p_1) & \text{if } \mathrm{p_1} < \mathrm{p_2} \\ Del(p_1+1) & \text{otherwise} \end{cases}$$
$$\mathrm{IT}(Del(p_1),Del(p_2)) = \begin{cases} Del(p_1) & \text{if } \mathrm{p_1} < \mathrm{p_2} \\ Del(p_1-1) & \text{if } \mathrm{p_1} > \mathrm{p_2} \\ Nop() & \text{otherwise} \end{cases}$$

**Fig. 3.** IT function of Suleiman and *al*.

Property $TP1$ defines a *state identity* and ensures that if $o_1$ and $o_2$ are concurrent, the effect of executing $o_1$ before $o_2$ is the same as executing $o_2$ before $o_1$. This property is necessary but not sufficient when the number of concurrent operations is greater than two. As for $TP2$, it ensures that transforming $o$ along equivalent and different operation sequences will give the same operation.

Properties $TP1$ and $TP2$ are sufficient to ensure the convergence for *any number* of concurrent operations which can be executed in *arbitrary order* [9]. 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.

It should be noted that, using our model-checking technique, we detected subtle flaws in the IT function of Fig.3. These flaws lead to divergence situations (see Section 3).

### 2.5 Consistency criteria

A stable state in an OT-based collaborative editor is achieved when all generated operations have been performed at all sites. Thus, the following criteria must be ensured [4, 9, 12]:

**Definition 4.** ***(Consistency Model)*** *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.*
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. The concurrent operations are serialized by using IT function. As this technique enables

concurrent operations to be serialized in any order, the convergence depends on $TP1$ and $TP2$ that IT function must verify.

### 2.6 Operational Transformation algorithms

Every site is equipped by an OT algorithm that consists of two main components [4,9]: the *integration procedure* and the *transformation component*. The integration procedure is responsible for receiving, broadcasting and executing operations. It is rather *independent* of the type of the shared objects. Several integration procedures have been proposed in the groupware research area, such as dOPT [4], adOPTed [9], SOCT2,4 [11,15] and GOTO [12]. The transformation component is commonly an IT function which is responsible for merging two concurrent operations defined on the same state. This function is *specific* to the semantics of a shared object. 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 Definition 1), and; (ii) $seq_c$ consists of operations that are concurrent to $o$. For more details, see [3].
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.
4. It 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, 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. This equivalence is ensured iff the used IT function satisfies properties $TP1$ and $TP2$.

In this work, we deal with OT algorithms that have the same integration procedure but differ only by their transformation functions. Five IT functions have been considered (see [3]).

The rest of the paper is devoted to the specification and analysis of OT algorithms, by means of model-checker *UPPAAL*. We show how to exploit some features of OT algorithms and the specification language of *UPPAAL* to attenuate the state explosion problem of the execution environment of such algorithms.

## 3 Modelling OT algorithms with UPPAAL

### 3.1 UPPAAL's model

In *UPPAAL*, a system consists of a collection of processes which can communicate via some shared data and synchronize through binary or broadcast channels [8]. Each process is an automaton extended with finite sets of clocks, variables (bounded integers), guards and actions. In such automata, locations can be labelled by clock conditions and

edges are annotated with selections, guards, synchronization signals and updates. 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. An edge is enabled in a state if and only if the guard evaluates to true. The update expression of the edge is evaluated when the edge is fired. The side effect of this expression changes the state of the system. Edges labelled with complementary synchronization signals over a common channel must synchronize. Two or more processes synchronize through channels with a sender/receiver syntax [2]. For a binary channel, a sender can emit a signal through a given binary channel *Syn* ( *Syn*!), if there is another process (a receiver) ready to receive the signal (*Syn*?). Both sender and receiver synchronize on execution of complementary actions *Syn*! and *Syn*?. For a broadcast channel, a sender can emit a signal through a given broadcast channel *Syn* ( *Syn*!), even if there is no process ready to receive the signal (*Syn*?). When a sender emits such a signal via a broadcast channel, it is synchronized with all processes ready to receive the signal. The updates of synchronized edges are executed starting with the one of the sender followed by those of the receiver(s). The execution order of updates of receivers complies with their creation orders.

### 3.2 Modelling execution environment of OT algorithms

A 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 integration and transformation steps as explained in the previous section (see sub-section 2.6).

Two main models are proposed for the verification of the convergence properties of OT algorithms: the *concrete model* and the *symbolic model*. The main difference between these models concerns the effective execution of operation signatures. Indeed, in the concrete model, effective execution of editing operations is performed during the generation of traces (see Fig. 4) while, in the symbolic model, it is delayed until the construction of symbolic execution traces of all sites is completed (see Fig 5). In this paper, we focus on the symbolic model. For further details about the concrete model and the different variants of the concrete and symbolic models, we refer to [3].

**System definition.** A collaborative editor is modelled as a set of variables, functions, processes (one per user) and a broadcast channel. Note that the network is abstracted and not explicitly represented. This is possible by putting visible (in global variables) all operations generated by different sites and timestamp vectors of sites. In this way, there is no need to represent and manage queues of messages. Behaviors of sites are similar and represented by a type of process named *Site*. The only parameter of the process is the site identifier named *pid*. With *UPPAAL*, the definition of the system is given by the following declarations which mean that the system consists of *NbSites* sites of type *Site*:

*Sites(const pid_t pid) = Site(pid);*
*system Sites;*

**Input data and Variables.** Variables are of two kinds: those used to store input data and those used to manage the execution of operations. Note that almost all variables are defined as global to be accessible by any site (avoiding duplication of data in the representation of the system state). In addition, this eases the specification of the convergence property and allows to force the execution, in one step, some edges of different sites. The system model has the following inputs and variables:

1. The number of sites (*const int NbSites*); Each site has its own identifier, denoted *pid* for process identifier ($pid \in [0, NbSites-1]$).
2. The initial text to be shared by users and its alphabet. The text to be shared by users is supposed to be infinite but the attribute *Position* of operations is restricted to the window $[0, L-1]$ of the text. The length of the window is set in the constant $L$ (*const int L*).
3. The number of local operations of each site, given in array $Iter[NbSites]$ (*const int* $Iter[NbSites]$, $Iter[i]$ being the number of local operations of site *i*). We also use and set in constant named *MaxIter* the total number of operations (*const int* $MaxIter = \sum_{i \in [0, NbSites-1]} Iter[i]$);
4. The IT function (*const int algo*).
5. The timestamp vectors of different sites ($V[NbSites][NbSites]$).
6. Vector $Operations[MaxIter]$ to store the owner and the timestamp vector of each operation.
7. Vectors $Trace[NbSites][MaxIter]$ to save the symbolic execution traces of sites (the execution order of operations).
8. Boolean variable *Detected* to recuperate the truth value of the convergence property.
9. Vector $Signatures[MaxIter]$ to get back signatures $(operator, position, character)$ of operations which violate the convergence property.
10. $List[2][MaxIter]$ to save operation signatures as they are exactly executed in two sites (after integration steps).
11. The broadcast channel *Syn*

**Behavior of each site.** The process behavior of each site is depicted by the automaton shown in Fig.5. Each user executes *symbolically*, one by one, all operations (local and non local ones), on its own copy of the shared text. The symbolic execution of an operation (local or non local) is represented by the loop on location $l0$ which consists of 3 parts: the selection of a process identifier ($k : pid\_t$), the guard $guard(k)$ and the update $SymbolicExecution(k)$. The guard part verifies whether a site *pid* can execute an operation of site *k*. The update part is devoted to the symbolic execution of an operation of a site *k*. The execution order of operations must, however, respect the causality principle. The causality principle is ensured by the timestamp vectors of sites $V[NbSites][NbSites]$. For each pair of sites $(i, j)$, element $V[i][j]$ is the number of operations of site $j$ executed by site *i*. $V[i][i]$ is then the number of local operations executed in site *i*. Note that $V[i][j]$ is also the rank of the next operation of site $j$ to be executed by site *i*. Timestamp vectors are also used to determine whether operations are concurrent or dependent. Initially, entries of the timestamp vector of every site *i* are set to 0. Afterwards, when site *i* executes an operation of a site $j$ ($j \in [0, NbSites-1]$), it increments the entry of $j$ in its own timestamp vector (i.e., $V[i][j]++$).

**Symbolic execution of a local operation.** A local operation can be executed by a site *pid* if the number of local operations already executed by site *pid* does not yet reach its maximal number of local operations (i.e. $V[pid][pid] < Iter[pid]$). In this case, its timestamp vector is set to the timestamp vector of its site. Its owner and the timestamp vector are stored in array *Operations*. Its entry in *Operations* is stored in $Trace[pid]$.

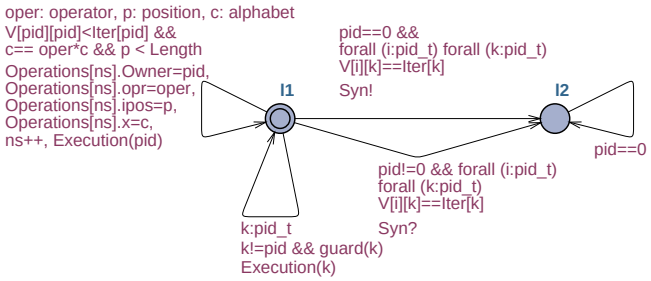


**Fig. 4.** The concrete model

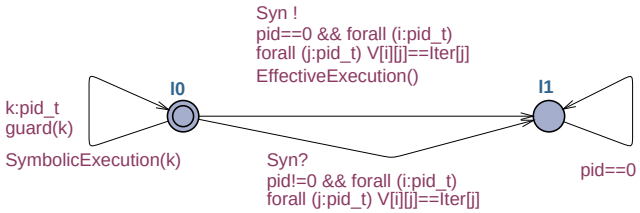


**Fig. 5.** The symbolic model

Its broadcast to other sites is simulated by incrementing the number of local operations executed ($V[pid][pid]++$).

**Symbolic execution of non local operations.** A site *pid* can execute an operation of another site *k* if there is an operation of *k* executed by *k* but not yet executed by *pid* (i.e.: $V[pid][k] < V[k][k]$) and its timestamp vector is less or equal to the timestamp vector of site *pid* (i.e.: $\forall j \in [0, NbSites-1], V[pid][j] >= Operations[num].V[j]$, *num* being the identifier of the operation). Recall that, the transformation and effective execution of operations (Insert and Delete) are not performed at this level. They are realized when the construction of all traces is completed.

**Effective execution of operations.** When all sites complete the construction of their respective traces, they are forced to perform synchronously, via the broadcast channel *Syn*, their respective edges connecting locations $l0$ and $l1$ (synchronization on termination). The update part of edge connecting locations $l0$ and $l1$ of site 0 is devoted to testing all signatures possibilities of operations and then verifying the convergence property. The test of all these possibilities is encapsulated in a C-function, called $Effective Execution$ which is stopped as soon as the violation of the convergence property is detected. This property is violated if there exist two sites which have completed the same set of operations but their texts are not identical. In this case, signatures of operations and exact traces of both sites which violate the convergence property are returned in vectors *Signatures* and *List*, and the variable *Detected* is set to *true*. The integration steps (see sub-section 2.6) are treated at this level (i.e., in this function).

### 3.3 Verification of the convergence property

The convergence property states that whenever two sites complete the execution of the same set of operations, their resulting texts must be identical. A stable state of the system is a situation where all sent operations are received and executed (there is no operation in transit). A site $i$ is in a stable state if all operations sent to site $i$ are received and executed by $i$ (i.e. $forall(k : pid\_t)\ V[i][k] == V[k][k]$). The convergence property can be rewritten using the notion of stable state as follows: "Whenever two sites $i$ and $j$ are in stable state, they have identical texts". For the concrete model [3], we use the negation of this property specified by the following *UPPAAL*'s $CTL$ formula $\phi_1$:

$$E\Diamond\ (exists(i : pid\_t)\ exists(j : pid\_t)$$

$$i! = j\ \&\&\ forall(k : pid\_t)\ V[i][k] == V[k][k]\ \&\&\ V[j][k] == V[k][k])$$

$$\&\&\ exists(l : int[0, L-1])\ text[i][l]! = text[j][l]$$

This formula means that there is an execution path leading to some situation where two sites $i$ and $j$ are in stable states and their copies of text $text[i]$ and $text[j]$ are different. For the symbolic model, the verification of the convergence propriety is based on a variable named *Detected*. This variable is set to *true* when the convergence propriety is violated. Therefore, the convergence propriety is violated iff UPPAAL's $CTL$ formula $\phi_1' : E\Diamond\ Detected$ is satisfied.

We have tested five IT functions known in the literature for synchronizing linear objects. Each IT function produces a new instance of OT algorithm, where only the transformation function changes. These OT algorithms are denoted respectively: $Ellis$ [4], *Ressel* [9], *Sun* [13], *Suleiman* [10] and *Imine* [6]. Two models are used : concrete and symbolic models.

| Alg. | Prop. | Val. | Expl./Comp./Time | Val. | Expl./Comp./Time |
|---|---|---|---|---|---|
| *Ellis* 3 3 | $\phi_1/\phi_1'$ | true | 825112/1838500/121.35 | true | 1625/1739/0.14 |
| *Ellis* 3 3 | $\phi_2$ | ? | ? | true | 1837/1837/0.68 |
| *Ressel* 3 3 | $\phi_1/\phi_1'$ | true | 833558/1851350/122.76 | true | 1637/1751/0.25 |
| *Ressel* 3 3 | $\phi_2$ | ? | ? | true | 1837/1837/1.63 |
| *Sun* 3 3 | $\phi_1/\phi_1'$ | true | 836564/1897392/122.33 | true | 1625/1739/0.14 |
| *Sun* 3 3 | $\phi_2$ | ? | ? | true | 1837/1837/0.38 |
| *Suleiman* 3 3 | $\phi_1/\phi_1'$ | false | 3733688/3733688/365.06 | false | 1837/1837/0.83 |
| *Suleiman* 3 3 | $\phi_2$ | ? | ? | true | 1837/1837/2.22 |
| *Suleiman* 3 4 | $\phi_1/\phi_1'$ | ? | ? | true | 18450/19380/2.45 |
| *Imine* 3 3 | $\phi_1/\phi_1'$ | false | 3733688/3733688/361.16 | false | 1837/1837/0.81 |
| *Imine* 3 3 | $\phi_2$ | ? | ? | true | 1837/1837/2.18 |
| *Imine* 3 4 | $\phi_1/\phi_1'$ | ? | ? | true | 18401/19331/2.45 |

**Table 1.** Model-checking the concrete and the symbolic models

We report in Table 1 the results obtained, for two properties: absence of deadlocks ($\phi_2 :\ A[]\ notdeadlock$) and the violation of the convergence property ($\phi_1$ or $\phi_1'$) defined above, in case of 3 sites ($NbSites = 3$), 3 or 4 operations ($MaxIter = 3$ or $MaxIter = 4$), and a window of the observed text of length $L = 2 * MaxIter$. A state $q$ of a model is in

deadlock iff there is no edge enabled in $q$ nor in states reachable from $q$ by time progression. Property $\phi_2$ is always satisfied and allows us to compute the size of the entire state space. Note that all tests are performed using the version 4.0.6 of UPPAAL 2k on a 3 Gigahertz Pentium-4 with 1GB of RAM. We give, in column 4, for each algorithm and each property, the number of explored states, the number of computed states and the execution time (CPU time in seconds). A question mark indicates a situation where the verification was aborted due to a lack of memory. We report in Table 2, the counterexamples obtained for the convergence property and the symbolic model (each operation $oi$, for $i = 1,3$, is generated by $Sitei$, $o11$ and $o12$ are generated in this order by $Site1$). Note that counterexamples obtained for the concrete and the symbolic models may be different. These results show that the symbolic model allows a significant gain in both time and space comparatively to the concrete model. With the symbolic model, we have been able to prove that the convergence property is not satisfied for five OT algorithms and to provide counterexamples.

| *Alg.* | Operations | Traces |
|---|---|---|
| Ellis | o1: Ins(1,0), o2: Ins(1,1), o3: Ins(1,0) | Site1: o1; o2; o3 Site3: o3; o2; o1 |
| Ressel | o1: Ins(2,0), o2: Ins(1,1), o3: Del(1) | Site1: o1; o2; o3 Site3: o3; o2; o1 |
| Sun | o1: Ins(1,0), o2: Ins(2,0), o3: Ins(2,1) | Site1: o1; o2; o3 Site3: o3; o1; o2 |
| Suleiman | o11: Del(1), o12: Ins(1,0),<br>o2: Ins(1,0), o3: Ins(2,0) | Site2 : o2; o3; o11; o12<br>Site3 : o3; o2; o11; o12 |
| Imine | o11: Ins(1,0), o12: Ins(2,0),<br>o2: Ins(2,0), o3: Del(1) | Site1 : o11; o12; o2; o3<br>Site3 : o3; o2; o11; o12 |

**Table 2.** Counterexamples obtained for the tested IT functions

For instance, in Fig.6, we report a divergence scenario for OT algorithm based on transformation function proposed by Suleiman and *al* [10] (see Fig.3), where $o_0$, $o_2$ and $o_3$ are pairwise concurrent and $o_0 \rightarrow o_1$.

**State space reduction.** To reduce the size of the state space to be explored, we propose some reductions (see [3] for more details) which preserve the convergence property. The first reduction consists of synchronization of the execution of non local operations in sites which have finished the execution of their local operations. This synchronization preserves the convergence property since when a site completes the execution of all local operations, it does not send any information to other sites and the execution of non local operations affects only the state of the site. With this synchronization, intermediate states resulting from different interleavings of these operations are not accessible. This reduction has been implemented in the variant models of the concrete and the symbolic models [3]. The second reduction forces to stop the construction of concrete/symbolic traces as soon as two any sites have completed the construction of their own traces. As sites have symmetrical behaviors, this reduction does not alter the convergence property. In the concrete and the symbolic models, edges connecting location $l0$ to $l1$ and the broadcast channel $Syn$, implement this reduction.

Another factor which contributes to the state explosion problem is the timestamp

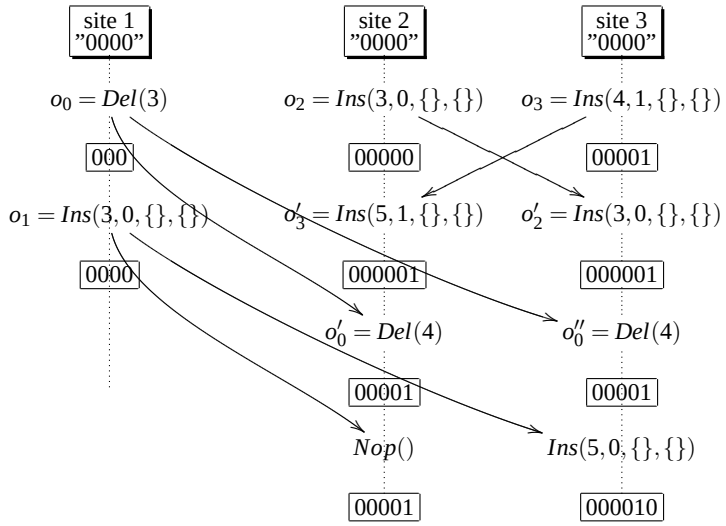


**Fig. 6.** Complete divergence scenario for Suleiman's algorithm.

vectors of different sites and operations. These vectors are used to ensure the causality principle. To attenuate this state explosion problem, we offer the possibility to replace the timestamp vectors by a relation of dependence over operations. This model allows to test whether an OT algorithm works or not under some relation of dependence (see [3] for more details).

## 4 Related Work

To our best knowledge, there exists only one work on analyzing OT algorithms [7]. In this work, the authors proposed a formal framework for modelling and verifying IT functions with algebraic specifications. For checking the properties $TP1$ and $TP2$, they used a theorem prover based on advanced automated deduction techniques. For all IT functions considered here, they showed that: (i) $TP1$ is only satisfied for Suleiman's and Imine's IT functions; (ii) $TP2$ is always violated.

For example, consider the IT function proposed by Suleiman et *al.* [10] (see Fig.3). A theorem prover-based verification revealed a $TP2$ violation in this function [5], as illustrated in Fig.7. As this is related to $TP2$ property, there are three concurrent operations (for all positions $p$ and all characters $x$ and $y$ such that $Code(x) < Code(y)$):
$o_1 = Ins(p,x,\{\},\{\})$, $o_2 = Ins(p,x,\{\},\{Del(p)\})$ and $o_3 = Ins(p,y,\{Del(p)\},\{\})$ with the transformations $o'_3 = IT(o_3,o_2)$, $o'_2 = IT(o_2,o_3)$, $o'_1 = IT^*(o_1,[o_2;o'_3])$ and $o''_1 = IT^*(o_1,[o_3;o'_2])$.

However, the theorem prover's output gives no information about whether this $TP2$ violation is reachable or not. Indeed, we do not know how to obtain $o_2$ and $o_3$ (their $pre_1$ and $post_2$ parameters are not empty respectively) as they are necessarily the results of transformation against other operations that are not given by the theorem prover. Using our model-checking-based technique, we can get a complete and informative scenario

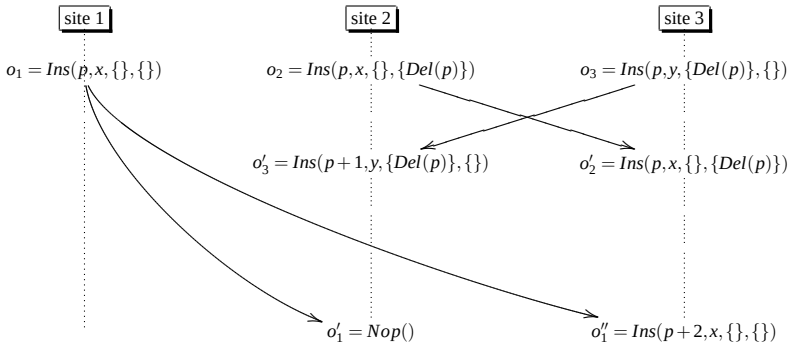


**Fig. 7.** $TP2$ violation for Suleiman's algorithm.

when a bug is detected. Indeed, the output contains all necessary operations and the step-by-step execution that lead to divergence situation. Thus, by model-checking verification, the existence of the $TP2$ violation depicted in Fig.7 is proved (or certified) by the complete scenario given in Table 2.

As they are the basis cases of the convergence property, $TP1$ and $TP2$ are sufficient to ensure the data convergence for any number of concurrent operations which can be performed in any order. Thus, a theorem prover-based approach remains better for proving that some IT function satisfies $TP1$ and $TP2$. But it is partially automatable and, in the most cases, less informative when divergence bugs are detected. A model-checking-based approach is fully automatable for finding divergence scenarios. Nevertheless, it is more limited as the convergence property can be exhaustively evaluated on only a specific finite state space.

## 5 Conclusion

We proposed here a model-checking technique, based on formalisms used in tool UPPAAL, to model the behavior of replication-based collaborative editors. To cope with the severe state explosion problem of such systems, we exploited their features and those of tool UPPAAL to establish and apply some abstractions and reductions to the model. The verification has been performed with the model-checking module of UPPAAL. An interesting and useful feature of this module is to provide, in case of failure of the tested property, a trace of an execution for which the property is not satisfied. We used this feature to give counterexamples for five OT algorithms, based on different transformation functions proposed in the literature to ensure the convergence property. Using our model-checking technique we found an upper bound for ensuring the data convergence in such systems. Indeed, when the number of sites exceeds 2 the convergence property is not achieved for all OT algorithms considered here. We think that our work is a forward step towards an efficient framework for formally developing shared objects based on the OT approach.

However, the serious drawback of the model-checking is the state explosion. So, in future work, we plan to investigate the following directions: (i) It is interesting to find, under which conditions, the model-checking verification problem can be reduced to a finite-state problem. (ii) Combining theorem-prover and model-checking approaches in order to attenuate the severe state explosion problem.

## References


1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
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. Experiments in model-checking optimistic replication algorithms. Research Report 6510, INRIA, April 2008.
4. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.
5. A. Imine. *Conception formelle d'algorithmes de réplication optimiste. Vers l'édition Collaborative dans les réseaux Pair-à-Pair*. Phd thesis, University of Henri Poincaré, Nancy, France, December 2006.
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. K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Journal of Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
9. 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.
10. 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.
11. M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *IEEE ICDE'98*, pages 36–45, 1998.
12. C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *ACM CSCW'98*, pages 59–68, 1998.
13. 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.
14. 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.
15. 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.