# Optimistic and Efficient Concurrency Control for Asynchronous Collaborative Systems

**Haifeng Shen** **Yongyao Yan**

School of Computer Science, Engineering and Mathematics
Flinders University, Adelaide, Australia
Email: {haifeng.shen, yan0056}@flinders.edu.au

**Abstract**

Concurrency control is a key issue in distributed systems. A number of techniques have been devised to tackle the issue, but these techniques are generally unsuitable to be used in collaborative systems, which have the special requirements of consistency maintenance, responsiveness, and unconstrained interaction. OT (Operational Transformation) is an optimistic concurrency control technique originally invented for synchronous collaborative systems to meet these requirements. But existing transformation control algorithms are inefficient to be used in asynchronous systems. In this paper, we present an OT-based concurrency control solution for asynchronous collaborative systems, including an efficient contextualization-based transformation control algorithm underpinned by operation propagation and replaying protocols to achieve contextualization. The solution has been formally verified in terms of consistency maintenance and demonstrated by a variety of prototype collaborative applications.

*Keywords:* collaborative systems, concurrency control, consistency maintenance, operational transformation, contextualization.

## 1 Introduction

Interactive applications have been around for decades and are still among the most commonly-used ones in our work and daily lives. People use them to do various tasks such as coding, word processing, designing, modeling, and entertaining. Collaboration, which is naturally needed when multiple people are involved in the same or related tasks, remains cumbersome, tedious, and error-prone because most of these applications are primarily single-user-oriented.

Collaborative systems facilitate and coordinate collaboration among multiple users who are jointly fulfilling common tasks over computer networks, particularly the Internet. They can be brand new multi-user interactive applications built from scratch with collaboration in mind or augmented single-user interactive applications with additional collaborative functionality. Such a system typically consists of three characteristic components: multiple human users, interactive applications that provide the human-computer interfaces, and the high-latency Internet backbone that connects distributed computers.



Concurrency control is the key to managing the contention for distributed data resources and to ensure the orderly access to shared data. It is a major issue in most distributed systems and collaborative systems are of no exception. Over the past decades, a number of concurrency control techniques, such as floor control (Greenberg & Marwood 1994), locking (Knister & Prakash 1993), transactions (Bernstein et al. 1987), causal ordering (Raynal & Singhal 1996), and serialization (Karsenty & Beaudouin-Lafon 1993), have been devised for a variety of distributed systems. However, these techniques are generally unsuitable to be used in collaborative systems because these characteristic components have special concurrency control requirements (Ellis & Gibbs 1989, Sun et al. 1998).

First, human users, who interact with the interactive applications to manipulate the data sources via the human-computer interfaces, have the special requirements of fast response and unconstrained interaction, i.e., to interact with any data objects at any time freely and responsively. Second, as the communication latency of the Internet is high, collaborative systems usually achieve good responsiveness by means of replication to hide the latency - the shared data source is replicated across multiple collaborating sites and one only interacts with her/his own replica. However, replication creates the special requirement of consistency maintenance because inconsistencies may occur due to the concurrent activities from multiple users and the non-deterministic communication latency.

OT (Operational Transformation) (Ellis & Gibbs 1989, Sun & Ellis 1998) is an optimistic concurrency control technique originally invented for synchronous collaborative systems, where users collaborate at the same time by instantly propagating every operation generated at one site to others. It can achieve consistency maintenance without sacrificing good responsiveness and unconstrained interaction. OT consists of a set of transformation functions that transform one operation against another, and a transformation control algorithm that ensures the invocation of every transformation function satisfies required precondition. Over the past decade or so, quite a few transformation control algorithms have been devised for various synchronous collaborative systems.

For OT to be used for asynchronous collaborative systems, where users collaborate at different times by infrequently propagating batches of operations, an efficient transformation control algorithm is imperative because existing ones are catered for synchronous systems only and are inefficient to be used in asynchronous systems. In this paper, we present an OT-based concurrency control solution for asynchronous collaborative systems, including an efficient contextualization-based transformation control algorithm underpinned by operation propagation and re-

playing protocols to achieve contextualization. The solution has been formally verified in terms of consistency maintenance and demonstrated by a variety of prototype collaborative applications.

The rest of the paper is organized as follows. The next section introduces a consistency model and some background about OT. After that, the contextualization-based transformation control algorithm is presented and verified. The subsequent section presents the operation propagation and replaying protocols to achieve contextualization. Significance and applications of the work are discussed in the following section. Finally, the paper is concluded with a summary of major contributions and future work.

## 2 A Consistency Model and OT

The consistency model consisting of three consistency properties *Convergence*, *Causality preservation*, and *Intention preservation* (Sun et al. 1998), has been widely used to verify whether a concurrency control technique can maintain consistency in collaborative systems. The *convergence* property ensures the consistency of the final states of the shared data source at the end of a collaborative session. This property can be preserved by known techniques such as serialization. The *causality preservation* property ensures the consistency of the execution orders of causally related operations during a collaborative session. This property can be preserved by known techniques such as causal ordering. OT was particularly invented for *intention preservation* and extended for *Convergence* as well. Alternative solutions for *intention preservation* are exemplified by the *Mark and Retrace* technique (Gu et al. 2005).

Every interactive application provides a set of AOs (Application Operations) for a user or the API (Application Programming Interface) to manipulate data objects in the data source and these application-dependent AOs can be abstracted into the following three application-independent POs (Primitive Operations) (Sun et al. 2006):

- **Insert**(*addr*, *obj*) denotes an *Insert* operation that creates an object *obj* at address *addr* in the data source,
- **Delete**(*addr*) denotes a *Delete* operation that removes the object at address *obj* in the data source, and
- **Update**(*addr*, *key*, *val*) denotes an *Update* operation that changes the attribute *key*, to the value *val*, of the object at address *addr* in the data source.

To use OT for *intention preservation*, the *addr* parameter used to reference a PO's target object must follow a *data addressing model* that complies with *XOTDM* (eXtend Operational Transformation Data Model) consisting of a hierarchy of addressing groups, where each group consists of multiple independent linear addressing domains (Sun et al. 2006). The intention of an operation is the execution effect at the time when it is generated. To preserve an operation's intention, OT ensures the operation's target object is correctly addressed across all collaborating sites.

Consider a collaborative drawing session shown in Figure 1, where the data source consisting of three objects $Ob_0$(triangle), $Ob_1$(circle), and $Ob_2$(square) is replicated at two sites. For the sake of simplicity (without loss of generality), we assume the *data addressing model* consists of a single linearly addressed domain *Drawings*, which initially contains [$Ob_0$, $Ob_1$, $Ob_2$]. At *site 1*, operation $Op_1$ = **Delete**([(*Drawings*, 0)]) is executed to remove the object at address [(*Drawings*, 0)], which is $Ob_0$ (triangle). After that, *Drawings* contains [$Ob_1$, $Ob_2$]. At *site 2*, operation $Op_2$ = **Update**([(*Drawings*, 1)], *fill*, *b*) is executed concurrently to fill the object at address [(*Drawings*, 1)], which is $Ob_1$ (circle), with the *black* color. After that, *Drawings* remains [$Ob_0$, $Ob_1$, $Ob_2$].

If $Op_1$ and $Op_2$ were propagated and replayed as-is for consistency maintenance (as depicted by the dashed lines), inconsistency would occur in the sense that the two replicas of the shared data source become divergent and $Op_2$'s intention (filling the circle with the *black* color) is violated at *site 1*. The intention violation problem was essentially caused by the inconsistent addressing of the same object $Ob_1$(circle). At the time when $Op_2$ was generated at *site 2*, $Ob_1$(circle) was addressed by [(*Drawings*, 1)], which was also the case at *site 1* before the execution of $Op_1$. However, after the execution of $Op_1$ at *site 1*, $Ob_0$ was removed, and $Ob_1$ took up $Ob_0$'s position (index 0) in *Drawings*. Consequently, $Ob_1$ should be addressed by the new address [(*Drawings*, 0)]. The wrong object $Ob_2$(square) would be referenced if the old address [(*Drawings*, 1)] were still used.

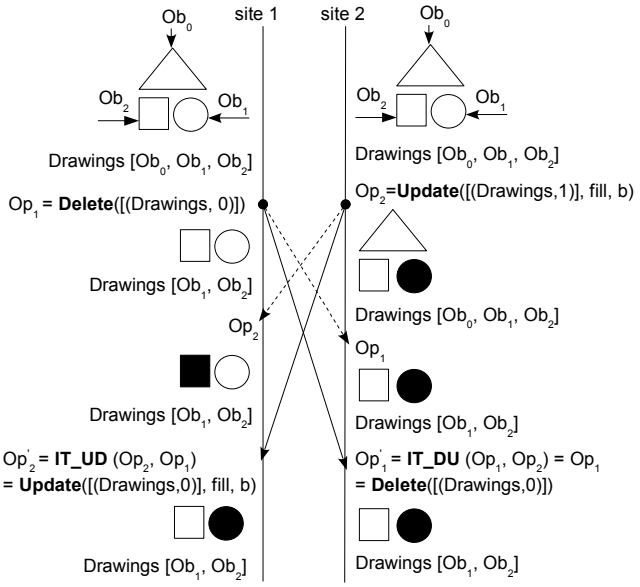


Figure 1: Intention preservation of $Op_2$ by OT

OT can solve the intention violation problem by compensating for the impact of concurrent operations on addressing an object's replicas. In this example, when $Op_2$ is propagated to *site 1* (as depicted by the solid line from *site 2* to *site 1*), it has to be transformed against the concurrent operation $Op_1$ to include its impact on addressing the referenced object $Ob_1$ by **IT_UD**($Op_2$, $Op_1$): $Op'_2$ = **Update**([(*Drawings*, 0)], *fill*, *b*) (**IT_UD**($Op_a$, $Op_b$) is a transformation function transforming an *Update* operation $Op_a$ against a *Delete* operation $Op_b$). The replaying of $Op'_2$ at *site 1* fills the circle with the *black* color, which has preserved $Op_2$'s intention.

Transformation function **IT**($Op_a$, $Op_b$): $Op'_a$ (Inclusion Transformation) transforms operations $Op_a$ against $Op_b$ in such a way that the impact of $Op_b$ is effectively included in the output operation $Op'_a$. Depending on $Op_a$'s and $Op_b$'s types, there are 9 (3×3) functions (**IT_UD**($Op_a$, $Op_b$) is one of them). Details of these functions can be found in (Sun et al. 1998, 2006).

It is noticed that the divergence problem has also been solved by OT in the above example. In fact, transformation functions are defined to take care of both *intention preservation* and *convergence* (there

are conditions and properties to verify them, which will be discussed in later sections).

## 3 OT-based Concurrency Control in Asynchronous Collaborative Systems

To transform operation $Op_a$ against operation $Op_b$ by the transformation function $\mathbf{IT}(Op_a, Op_b)$: $Op'_a$, $Op_a$ and $Op_b$ must meet a pre-condition.

**Definition 1** *(Operation Context)*

Given an operation $Op$, its context, denoted by $\Upsilon_{Op}$, is the state of the data source based on which $Op$ is defined.

**Definition 2** *(Operation Context Equivalent Relation)*

Given two operations $Op_a$ and $Op_b$, $Op_a$ is context-equivalent to $Op_b$, denoted by $Op_a \sqcup Op_b$, iff (if and only if) $\Upsilon_{Op_a} = \Upsilon_{Op_b}$.

**Definition 3** *(Operation Context Preceding Relation)*

Given two operations $Op_a$ and $Op_b$, $Op_a$ is context-preceding $Op_b$, denoted by $Op_a \mapsto Op_b$, iff $\Upsilon_{Op_b} = \Upsilon_{Op_a} \vdash Op_a$, where "$\vdash$" is the operation execution operator.

$\Upsilon_{Op_a}$ is the context on which $Op_a$ is defined. After the execution of $Op_a$ on $\Upsilon_{Op_a}$, the new context can be described by $\Upsilon_{Op_a} \vdash Op_a$. If operation $Op_b$ is defined on this new context, i.e., $\Upsilon_{Op_b} = \Upsilon_{Op_a} \vdash Op_a$, then $Op_a$ is context-preceding $Op_b$. In general, if the initial context is $\Phi$ and $n$ operations $Op_1, \cdots, Op_n$ have been executed in sequence, then the new context can be described by $\Phi \vdash Op_1 \vdash \cdots \vdash Op_n$. For $\forall i \in \{2, \cdots, n\}$, $Op_{i-1} \mapsto Op_i$ because $\Upsilon_{Op_1} = \Phi$ and $\Upsilon_{Op_i} = \Upsilon_{Op_{i-1}} \vdash Op_{i-1}$.

So the pre-condition of the transformation function $\mathbf{IT}(Op_a, Op_b)$: $Op'_a$ is $Op_a \sqcup Op_b$. This pre-condition is to ensure their *addr* parameters are directly comparable since they are defined on the same context. A few TCAs (Transformation Control Algorithms) such as *GOT* (Sun et al. 1998), *GOTO* (Sun & Ellis 1998), *adOPTed* (Ressel et al. 1996), *Jupiter* (Nichols et al. 1995), and *COT* (Sun & Sun 2006), have been devised to ensure the pre-condition of every transformation function invocation is satisfied in synchronous collaborative systems. The transformation function $\mathbf{IT}(Op_a, Op_b)$: $Op'_a$ also has a post-condition, which is $Op_b \mapsto Op'_a$ because $Op'_a$ has included the impact of $Op_b$ in such a way that the context of $Op'_a$ is defined on the one right after the execution of $Op_b$. Every transformation function must be defined to ensure this post-condition.

In addition, the following *OCRP* (Operation Context Relation Property) describes the correlation between "$\sqcup$" and "$\mapsto$" relations.

**Property 1** *Operation Context Relation Property (OCRP)*

Given three operations $Op_a$, $Op_b$, and $Op_c$, if $Op_a \mapsto Op_b$ and $Op_a \mapsto Op_c$, then $Op_b \sqcup Op_c$. **Proof:**

1. $\Upsilon_{Op_b} = \Upsilon_{Op_a} \vdash Op_a$ because $Op_a \mapsto Op_b$,
2. $\Upsilon_{Op_c} = \Upsilon_{Op_a} \vdash Op_a$ because $Op_a \mapsto Op_c$, and
3. $Op_b \sqcup Op_c$ because $\Upsilon_{Op_b} = \Upsilon_{Op_c}$.

### 3.1 Main Issues in Applying OT to Asynchronous Collaborative Systems

Existing TCAs are catered for synchronous collaborative systems in the sense that every operation generated at one site needs to be individually timestamped and instantly propagated to remote sites, and the replaying of every remote operation at any remote site needs to trigger a separate TCA invocation. However, in asynchronous systems, individual timestamping and instant propagation of every operation is either infeasible or unnecessary. On the one hand, it is infeasible in intermittent networking environments (e.g., wireless) because network connection is not persistently available. On the other hand, it is unnecessary because: 1) users are more sensitive to objects that are of their interests and therefore it makes better sense to batch coherent operations for propagation and replaying; 2) some micro-step operations never need to be propagated at all because they are either not interested by other users, or are overridden by or merged into later operations; 3) asynchronous collaboration does not need instant notification of each other's micro-step progress; and 4) in the Internet or wireless networking environments, where the network resource is scarce, it is wasteful to propagate every operation separately and instantly.

More importantly, it is rather inefficient to invoke a separate TCA for replaying every remote operation in asynchronous systems. Most existing TCAs used in synchronous systems have a quadratic time complexity or worse. If $n$ remote operations arrive at a site and every operation $Op$ needs to invoke a $\mathbf{TCA}(Op, C)$: $Op'$ ($C$ is the collection of operations that have been executed at that site and are concurrent with $Op$) to derive its execution form, then the TCA will be invoked $n$ times and the overall time complexity will be cubic or even worse! In synchronous systems, the quadratic execution time of TCA is acceptable as $C$ is normally small (because each operation is instantly garbage-collected after having been transformed against all concurrent operations at every collaborating site (Sun et al. 1998)). In contrast, $n$ and $C$ both tend to be big in asynchronous systems due to the infrequent propagation of operations, which makes the cubic time complexity unacceptable.

To apply OT to distributed asynchronous collaborative systems, we need to devise an efficient TCA that does not rely on either timestamping or instant propagation of individual operations. In the following section, we present a new TCA that does not rely on timestamping or instant propagation of individual operations and can efficiently transform two contextualized lists of operations with a quadratic time complexity. Underpinned by an operation propagation protocol and an operation replaying protocol, the two contextualized lists can be of random length, making the algorithm viable for a wide spectrum of collaboration paradigms and particularly efficient for asynchronous collaboration, where the two lists tend to be long.

### 3.2 SLOT - A Contextualization-based TCA

**Definition 4** *(Contextualized List of Operations).*

Given a list of operations $L$, $L$ is said to be *contextualized*, iff $L[i] \mapsto L[i+1]$ ($0 \leq i < |L|$-1), where $L[i]$ is the $(i+1)^{th}$ operation in $L$ and $|L|$ is the length of $L$.

Operations in a contextualized list may come from the same site or from different sites. Take a list consisting of two operations $Op_a$ and $Op_b$ as an example. First, if $Op_a$ and $Op_b$ are consecutively generated at the same site, then $L = [Op_a, Op_b]$ is naturally contextualized because $Op_a \mapsto Op_b$. Second, if $Op_a$ and

$Op_b$ are concurrently generated at two different sites, and $Op_a \sqcup Op_b$, then $L = [Op_a, Op'_b]$ or $[Op_b, Op'_a]$, where $\mathbf{IT}(Op_b, Op_a)$: $Op'_b$ and $\mathbf{IT}(Op_a, Op_b)$: $Op'_a$, is contextualized because $Op_b \mapsto Op'_a$ and $Op_a \mapsto Op'_b$ (by the post-conditions of the two **IT** function invocations).

**Definition 5** *(List Context).*

Given a *contextualized* list $L$, its context, denoted by $\Psi_L$, is the context of the first operation in $L$, i.e., $\Psi_L = \Upsilon_{L[0]}$.

**Definition 6** *(List Context Equivalent Relation ).*

Given two *contextualized* lists $L_a$ and $L_b$, $L_a$ is context-equivalent to $L_b$, denoted by $L_a \amalg L_b$, iff $\Psi_{L_a} = \Psi_{L_b}$.

**Definition 7** *(List Context Preceding Relation).*

Given two *contextualized* lists $L_a$ and $L_b$, $L_a$ is context-preceding $L_b$, denoted by $L_a \dashrightarrow L_b$, iff $\Psi_{L_b} = \Psi_{L_a} \succ L_a$, where "$\succ$" is the list execution operator (operations in $L_a$ are executed in sequence on context $\Psi_{L_a}$).

If $L_a \dashrightarrow L_b$, then $\Upsilon_{L_b[0]} = \Upsilon_{L_a[0]} \vdash L_a[0] \vdash \cdots \vdash L_a[|L_a|\text{-}1]$ and $L_a[|L_a|\text{-}1] \mapsto L_b[0]$. Therefore, if $L$ is the catenation of $L_a$ and $L_b$, denoted by $L = L_a \succ L_b$, then $L$ is also contextualized. Furthermore, the following two properties describe the correlations between "$\amalg$" and "$\dashrightarrow$" relations.

**Property 2** *List Context Relation Property 1 (LCRP-1)*

Given three contextualized lists $L_a$, $L_b$, and $L_c$, if $L_a \dashrightarrow L_b$ and $L_a \dashrightarrow L_c$, then $L_b \amalg L_c$. **Proof:**

1. $\Psi_{L_b} = \Psi_{L_a} \succ L_a$ because $L_a \dashrightarrow L_b$,
2. $\Psi_{L_c} = \Psi_{L_a} \succ L_a$ because $L_a \dashrightarrow L_c$, and
3. $L_b \amalg L_c$ because $\Psi_{L_b} = \Psi_{L_c}$.

**Property 3** *List Context Relation Property 2 (LCRP-2)*

Given three contextualized lists $L_a$, $L_b$, and $L_c$, if $L_a \amalg L_b$ and $L_b \dashrightarrow L_c$, then $L_a \amalg (L_b \succ L_c)$. **Proof:**

1. $L_b \succ L_c$ is contextualized because $L_b \dashrightarrow L_c$,
2. $\Psi_{L_a} = \Psi_{L_b}$ because $L_a \amalg L_b$,
3. $\Psi_{L_b} = \Psi_{L_b \succ L_c} = \Upsilon_{L_b[0]}$, and
4. $L_a \amalg (L_b \succ L_c)$ because $\Psi_{L_a} = \Psi_{L_b \succ L_c}$.

The **SLOT** (Symmetric Linear Operation Transformation) control algorithm, which symmetrically transforms two context-equivalent lists $L_a$ and $L_b$, and returns transformed ones $L'_a$ and $L'_b$, is defined as follows.

**Algorithm 1** **SLOT**$(L_a, L_b)$: $(L'_a, L'_b)$

***Require:*** $L_a \amalg L_b$
***Ensure:*** $L_a \dashrightarrow L'_b$ *and* $L_b \dashrightarrow L'_a$
  $L'_a \leftarrow L_a$
  $L'_b \leftarrow L_b$
  ***for*** $(i = 0;\ i < |L'_a|;\ i{+}{+})$ ***do***
    ***for*** $(j = 0;\ j < |L'_b|;\ j{+}{+})$ ***do***
      $Op^j_{a,i} \leftarrow L'_a[i]$
      $Op^i_{b,j} \leftarrow L'_b[j]$
      $(Op^{j+1}_{a,i}, Op^{i+1}_{b,j}) \leftarrow \boldsymbol{SIT}(Op^j_{a,i}, Op^i_{b,j})$
      $L'_a[i] \leftarrow Op^{j+1}_{a,i}$
      $L'_b[j] \leftarrow Op^{i+1}_{b,j}$
    ***end for***
  ***end for***

  ***return*** $(L'_a, L'_a)$

The **SIT** (Symmetric Inclusion Transformation) function symmetrically transforms two context-equivalent operations. Its pre-condition and post-conditions are directly based on those of **IT** functions.

**Function 1** **SIT**$(Op_a, Op_b)$: $(Op'_a, Op'_b)$

***Require:*** $Op_a \sqcup Op_b$
***Ensure:*** $Op_a \mapsto Op'_b$ *and* $Op_b \mapsto Op'_a$
  $Op'_a \leftarrow \boldsymbol{IT}(Op_a, Op_b)$
  $Op'_b \leftarrow \boldsymbol{IT}(Op_b, Op_a)$

  ***return*** $(Op'_a, Op'_b)$

The **SLOT** control algorithm can be implemented as a function **SLOT**$(L_a, L_b)$: $(L'_a, L'_b)$, where original lists and transformed ones are kept separately, or as a procedure **SLOT**$(L_a, L_b)$, where the original lists are replaced by transformed ones.

If the pre-condition of a **SLOT** algorithm invocation is satisfied, the algorithm can ensure the pre-condition of every **SIT** function invocation is satisfied. If the post-condition of every IT function invocation is satisfied by the definitions of IT functions, the post-conditions of the **SLOT** algorithm invocation are automatically satisfied. The post-conditions of **SLOT**, $L_a \dashrightarrow L'_b$ and $L_b \dashrightarrow L'_a$, have two implications. One is that $L_a \succ L'_b$ and $L_b \succ L'_a$ are also contextualized. The other is that operations in the list $L'_b$ (or $L'_a$) are ready to be replayed in sequence after having taking into account the impact of $L_a$ (or $L_b$) by transformation.

Furthermore, to preserve *convergence*, IT functions must meet the following *TP-1* (Transformation Property 1).

**Definition 8** *(List Equivalent Relation).*

Given two lists $L_a$ and $L_b$, where $L_a \amalg L_b$, $L_a$ is equivalent to $L_b$, denoted by $L_a \equiv L_b$, iff $\Psi_{L_a} \succ L_a = \Psi_{L_b} \succ L_b$.

**Property 4** *Transformation Property 1 (TP-1)*

Given two operations $Op_a$ and $Op_b$, where $Op_a \sqcup Op_b$, if **SIT**$(Op_a, Op_b)$: $(Op'_a, Op'_b)$, then $[Op_a, Op'_b] \equiv [Op_b, Op'_a]$.

Given two operations $Op_a$ and $Op_b$ concurrently generated at two collaborating sites and propagated to each other for consistency maintenance, *TP-1* is to ensure convergence in the sense that $\Upsilon_{Op_a} \vdash Op_a \vdash Op'_b = \Upsilon_{Op_b} \vdash Op_b \vdash Op'_a$. Although the two operations are executed in different orders at the two sites, IT functions must ensure an identical context after both operations have been executed at the two sites.

To use **SLOT** to control transformation in asynchronous collaborative systems, each collaborating site needs to maintain two linear buffers: *OB* (Outgoing Buffer) for storing unpropagated local operations and *IB* (Incoming Buffer) for storing unreplayed remote operations. The following example illustrates how the **SLOT** control algorithm preserves convergence and the intentions of operations by means of OT. As shown in Figure 2, consider a collaborative drawing session involving two sites, where the shared data source initially consists of three objects: $Ob_0$(a triangle), $Ob_1$(a circle), and $Ob_2$(a square), and the data addressing domain *Drawings* initially contains

$[Ob_0, Ob_1, Ob_2]$. The two buffers at the two sites, namely $OB_1$ and $IB_1$ ($OB$ and $IB$ at *site 1*), $OB_2$ and $IB_2$ ($OB$ and $IB$ at *site 2*), are all empty initially.

At *site 1*, operation $Op_1$ = **Delete**([(Drawings, 1)]) is executed to remove the object at address [(Drawings, 1)], which is $Ob_1$ (circle). After that, *Drawings* contains $[Ob_0, Ob_2]$, $OB_1$ contains $[Op_1]$, and $IB_1$ remains empty. Concurrently at *site 2*, operations $Op_2$ = **Delete**([(Drawings, 0)]) and $Op_3$ = **Update**([(Drawings, 1)], *fill*, *b*) are executed in sequence to first remove the object at address [(Drawings, 0)], which is $Ob_0$(rectangle) and then fill the object at address [(Drawings, 1)], which is $Ob_2$(square), with the *black* color. After that, *Drawings* contains $[Ob_1, Ob_2]$, $OB_2$ contains $[Op_2, Op_3]$, and $IB_2$ remains empty.

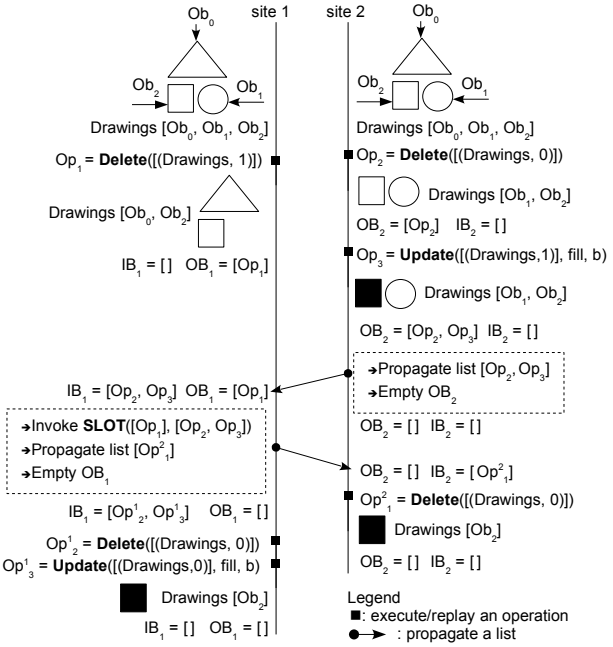


Figure 2: Consistency maintenance by SLOT

At *site 2*, because $IB_2$ is empty, list $[Op_2, Op_3]$ from $OB_2$ are propagated as-is to *site 1* and $OB_2$ is emptied. When the list arrives at *site 1*, it is appended to $IB_1$. At *site 1*, because $IB_1$ is not empty, to propagate list $[Op_1]$ from $OB_1$ to *site 2*, procedure **SLOT**($[Op_1]$, $[Op_2, Op_3]$) needs to be invoked as follows. The pre-condition of this procedure invocation has been satisfied, i.e., $[Op_1]$ II $[Op_2, Op_3]$, because $\Psi_{[Op_1]} = \Psi_{[Op_2,Op_3]} = Drawings[Ob_0, Ob_1, Ob_2]$.

1. **SIT**($Op_1$, $Op_2$): ($Op_1^1$, $Op_2^1$), where **IT_DD**($Op_1$, $Op_2$): $Op_1^1$ = **Delete**([(Drawings, 0)]) and **IT_DD**($Op_2$, $Op_1$): $Op_2^1$ = **Delete**([(Drawings, 0)]). The pre-condition is satisfied, i.e., $Op_1 \sqcup Op_2$ because $\Upsilon_{Op_1} = \Upsilon_{Op_2} = Drawings[Ob_0, Ob_1, Ob_2]$. The post-conditions are $Op_1 \mapsto Op_2^1$ and $Op_2 \mapsto Op_1^1$.
2. **SIT**($Op_1^1$, $Op_3$): ($Op_1^2$, $Op_3^1$), where **IT_DU**($Op_1^1$, $Op_3$): $Op_1^2$ = **Delete**([(Drawings, 0)]) and **IT_UD**($Op_3$, $Op_1^1$): $Op_3^1$ = **Update**([(Drawings, 0)], *fill*, *b*). Because $Op_2 \mapsto Op_1^1$ and $Op_2 \mapsto Op_3$, by Property 1: *OCRP*, the pre-condition is also satisfied, i.e., $Op_1^1 \sqcup Op_3$. The post-conditions are $Op_1^1 \mapsto Op_3^1$ and $Op_3 \mapsto Op_1^2$.

After the **SLOT** invocation, $OB_1 = [Op_1^2]$ and $IB_1 = [Op_2^1, Op_3^1]$. The post-conditions of the **SLOT** invocation are satisfied, i.e., $[Op_2, Op_3] \dashrightarrow [Op_1^2]$ (or $Op_2 \mapsto Op_3 \mapsto Op_1^2$) and $[Op_1] \dashrightarrow [Op_2^1, Op_3^1]$ (or $Op_1 \mapsto Op_2^1 \mapsto Op_3^1$), because:

1. $Op_2 \mapsto Op_3$ and $Op_3 \mapsto Op_1^2$ (*by the post-conditions of the second* **SIT** *invocation*), and
2. $Op_1 \mapsto Op_2^1$ (*by the post-conditions of the first* **SIT** *invocation*) and $Op_2^1 \mapsto Op_3^1$ (*by the following deduction:*)
   (a) $\Upsilon_{Op_3^1} = \Upsilon_{Op_1^1} \vdash Op_1^1$ because $Op_1^1 \mapsto Op_3^1$ (*by the post-conditions of the second* **SIT** *invocation*);
   (b) $\Upsilon_{Op_1^1} = \Upsilon_{Op_2} \vdash Op_2$ because $Op_2 \mapsto Op_1^1$ (*by the post-conditions of the first* **SIT** *invocation*);
   (c) $\Upsilon_{Op_2} \vdash Op_2 \vdash Op_1^1 = \Upsilon_{Op_1} \vdash Op_1 \vdash Op_2^1$ (*by TP-1 of the first* **SIT** *invocation*); and
   (d) $Op_2^1 \mapsto Op_3^1$ because $\Upsilon_{Op_3^1} = \Upsilon_{Op_1^1} \vdash Op_1^1$ (*by step a*) $= \Upsilon_{Op_2} \vdash Op_2 \vdash Op_1^1$ (*by step b*) $= \Upsilon_{Op_1} \vdash Op_1 \vdash Op_2^1$ (*by step c*).

Then the list $[Op_1^2]$ is propagated to *site 2* and $OB_1$ is emptied. When the list arrives at *site 1*, it is appended to $IB_2$. At *site 2*, because $[Op_2, Op_3] \dashrightarrow [Op_1^2]$ and $OB_2$ is empty, the remote operation $Op_1^2$ in $IB_2$ can be replayed as-is on the current context (i.e., the context right after the execution of local operations $Op_2$ and $Op_3$). When $Op_1^2$ = **Delete**([(Drawings, 0)]) is replayed, the object at address [(Drawings, 0)], which is $Ob_1$(circle), is removed. After that, $IB_2$ is emptied.

Back to *site 1*, because $[Op_1] \dashrightarrow [Op_2^1, Op_3^1]$ and $OB_1$ is empty, remote operations $Op_2^1$ and $Op_3^1$ in $IB_1$ can be replayed in sequence as-is on the current context (i.e., the context right after the execution of local operation $Op_1$). When $Op_2^1$ = **Delete**([(Drawings, 0)]) is replayed, the object at address [(Drawings, 0)], which is $Ob_0$(rectangle), is removed. Subsequently, when $Op_3^1$ = **Update**([(Drawings, 0)], *fill*, *b*) is replayed, the object at address [(Drawings, 0)], which is $Ob_2$(square), is filled with the *black* color. After that, $IB_1$ is emptied. It is clear that the intentions of all operations are preserved and the two sites are convergent at the end of the collaborative session.

### 3.3 Verification of the SLOT TCA

The above example has shown that the **SLOT** control algorithm can preserve convergence and the intentions of operations. To systematically prove its correctness, we need to verify it against the *intention preservation* and *convergence* consistency properties. However, due to space limitation, we only formally verify the *intention preservation* property. The *causality preservation* property has nothing to do with **SLOT** and will be discussed in a later section.

Transformation function **IT**($Op_a$, $Op_b$): $Op'_a$ can preserve the intention of $Op_a$ by including the impact of $Op_b$ into $Op'_a$, provided that the pre-condition $Op_a \sqcup Op_b$ is satisfied. Therefore, to prove **SLOT** control algorithm can achieve *intention preservation*, we only need to prove it can ensure the pre-condition of every **SIT** function invocation is satisfied, as described by Theorem 1.

**Theorem 1** *Given two lists $L_a$ and $L_b$, where $L_a$* II *$L_b$,* **SLOT***$(L_a, L_b)$ ensures the pre-condition of every* **SIT** *function invocation is satisfied.*

**Proof:** Assume that $L_a = [Op_{a,0}, \cdots, Op_{a,m-1}]$ ($m = |L_a|$) and $L_b = [Op_{b,0}, \cdots, Op_{b,n-1}]$ ($n = |L_b|$). Furthermore, $Op^k_{a,i}$ ($0 \leq i < m$ and $0 \leq k \leq n$) denotes the transformed $Op_{a,i}$ that has been transformed against the first $k$ operations in $L_b$, and $Op^l_{b,j}$ ($0 \leq j < n$ and $0 \leq l \leq m$) denotes the transformed $Op_{b,j}$ that has been transformed against the first $l$ operations in $L_a$. In particular, $Op^0_{a,i} = Op_{a,i}$ ($0 \leq i < m$) and $Op^0_{b,j} = Op_{b,j}$ ($0 \leq j < n$).

**SLOT**($L_a$, $L_b$) needs to perform $m \times n$ **SIT** function invocations, namely, **SIT**($Op^j_{a,i}$, $Op^i_{b,j}$): ($Op^{j+1}_{a,i}$, $Op^{i+1}_{b,j}$), where $i = 0,\cdots,m\text{-}1$ and $j = 0,\cdots,n\text{-}1$. We need to prove that for $\forall i \in \{0, \cdots, m\text{-}1\}$ and $\forall j \in \{0, \cdots, n\text{-}1\}$, $Op^j_{a,i} \sqcup Op^i_{b,j}$ holds.

For $m = 1$, $L_a = [Op_{a,0}]$, and **SLOT**($L_a$, $L_b$) needs to perform $n$ **SIT** function invocations, namely **SIT**($Op^j_{a,0}$, $Op_{b,j}$): ($Op^{j+1}_{a,0}$, $Op^1_{b,j}$), where $j = 0, \cdots, n\text{-}1$. The theorem holds, i.e., for $\forall j \in \{0, \cdots, n\text{-}1\}$, $Op^j_{a,0} \sqcup Op_{b,j}$ holds, because:

- for $n = 1$, $L_b = [Op_{b,0}]$, $Op_{a,0} \sqcup Op_{b,0}$ obviously holds because $L_a$ II $L_b$,
- for $n = N$, hypothesize that for $\forall j \in \{0, \cdots, N\text{-}1\}$, $Op^j_{a,0} \sqcup Op_{b,j}$ holds($\boxtimes$),
- for $n = N\text{+}1$, one more **SIT**($Op^N_{a,0}$, $Op_{b,N}$): ($Op^{N+1}_{a,0}$, $Op^1_{b,N}$) invocation is required, and by Property 1: *OCRP*, $Op^N_{a,0} \sqcup Op_{b,N}$ also holds because:
  1. $Op_{b,N-1} \mapsto Op_{b,N}$ as $L_b$ is contextualized; and
  2. $Op_{b,N-1} \mapsto Op^N_{a,0}$ as a result of the post-conditions of **SIT**($Op^{N-1}_{a,0}$, $Op_{b,N-1}$): ($Op^N_{a,0}$, $Op^1_{b,N-1}$) invocation, which is legitimate since $Op^{N-1}_{a,0} \sqcup Op_{b,N-1}$ (*by the induction hypothesis*$^{\boxtimes}$), and
- further based on the induction hypothesis$^{\boxtimes}$, for $\forall j \in \{0, \cdots, N\}$, $Op^j_{a,0} \sqcup Op_{b,j}$ holds, and therefore by the induction argument, for any $n$ and for $\forall j \in \{0, \cdots, n\text{-}1\}$, $Op^j_{a,0} \sqcup Op_{b,j}$ holds.

For $m = M$, hypothesize that the theorem holds, i.e., for $\forall i \in \{0, \cdots, M\text{-}1\}$ and $\forall j \in \{0, \cdots, n\text{-}1\}$, $Op^j_{a,i} \sqcup Op^i_{b,j}$ holds($\dagger$).

For $m = M\text{+}1$, $n$ more **SIT** invocations are required, namely, **SIT**($Op^j_{a,M}$, $Op^M_{b,j}$): ($Op^{j+1}_{a,M}$, $Op^{M+1}_{b,j}$), where $j = 0, \cdots, n\text{-}1$. For $\forall j \in \{0, \cdots, n\text{-}1\}$, $Op^j_{a,M} \sqcup Op^M_{b,j}$ holds because:

- for $n = 1$, $L_b = [Op_{b,0}]$, and by Property 1: *OCRP*, $Op_{a,M} \sqcup Op^M_{b,0}$ holds because:
  1. $Op_{a,M-1} \mapsto Op_{a,M}$ as $L_a$ is contextualized; and
  2. $Op_{a,M-1} \mapsto Op^M_{b,0}$ as a result of the post-conditions of **SIT**($Op_{a,M-1}$, $Op^{M-1}_{b,0}$): ($Op^1_{a,M-1}$, $Op^M_{b,0}$) invocation, which is legitimate since $Op_{a,M-1} \sqcup Op^{M-1}_{b,0}$ (*by the induction hypothesis*$^{\dagger}$),
- for $n = N$, hypothesize that for $\forall j \in \{0, \cdots, N\text{-}1\}$, $Op^j_{a,M} \sqcup Op^M_{b,j}$ holds($*$), and
- for $n = N\text{+}1$, one more **SIT**($Op^N_{a,M}$, $Op^M_{b,N}$): ($Op^{N+1}_{a,M}$, $Op^{M+1}_{b,N}$) is required, and $Op^N_{a,M} \sqcup Op^M_{b,N}$ also holds because:
  1. $Op^M_{b,N-1} \mapsto Op^N_{a,M}$ as a result of the post-conditions of **SIT**($Op^{N-1}_{a,M}$, $Op^M_{b,N-1}$): ($Op^N_{a,M}$, $Op^{M+1}_{b,N-1}$), which is legitimate since $Op^{N-1}_{a,M} \sqcup Op^M_{b,N-1}$ (*by the induction hypothesis*$^{*}$);
  2. $Op^{N-1}_{a,M-1} \mapsto Op^M_{b,N-1}$ and $Op^{M-1}_{b,N-1} \mapsto Op^N_{a,M-1}$ as the results of the post-conditions of **SIT**($Op^{N-1}_{a,M-1}$, $Op^{M-1}_{b,N-1}$): ($Op^N_{a,M-1}$, $Op^M_{b,N-1}$), which is legitimate since $Op^{N-1}_{a,M-1} \sqcup Op^{M-1}_{b,N-1}$ (*by the induction hypothesis*$^{\dagger}$);
  3. $Op^N_{a,M-1} \mapsto Op^M_{b,N}$ as a result of the post-conditions of **SIT**($Op^N_{a,M-1}$, $Op^{M-1}_{b,N}$): ($Op^N_{a,M-1}$, $Op^M_{b,N-1}$), which is legitimate since $Op^N_{a,M-1} \sqcup Op^{M-1}_{b,N}$ (*by the induction hypothesis*$^{\dagger}$);
  4. $\Upsilon_{Op^{M-1}_{b,N-1}} \vdash Op^{M-1}_{b,N-1} \vdash Op^N_{a,M-1} = \Upsilon_{Op^{N-1}_{a,M-1}} \vdash Op^{N-1}_{a,M-1} \vdash Op^M_{b,N-1}$ (*by TP-1 of* **SIT**(*$Op^{N-1}_{a,M-1}$, $Op^{M-1}_{b,N-1}$): ($Op^N_{a,M-1}$, $Op^M_{b,N-1}$)*);
  5. $\Upsilon_{Op^M_{b,N}} = \Upsilon_{Op^N_{a,M}}$ because $\Upsilon_{Op^M_{b,N}} = \Upsilon_{Op^{M-1}_{b,N-1}} \vdash Op^{M-1}_{b,N-1} \vdash Op^N_{a,M-1}$ (*by step 2 and 3, and the definition of* "$\mapsto$") $= \Upsilon_{Op^{N-1}_{a,M-1}} \vdash Op^{N-1}_{a,M-1} \vdash Op^M_{b,N-1}$ (*by step 4*) $= \Upsilon_{Op^N_{a,M}}$ (*by step 1 and 2, and the definition of* "$\mapsto$"); and
  6. further based on the induction hypothesis$^{*}$, for $\forall j \in \{0, \cdots, N\}$, $Op^j_{a,M} \sqcup Op^M_{b,j}$ holds, and therefore by the induction argument, for any $n$ and for $\forall j \in \{0, \cdots, n\text{-}1\}$, $Op^j_{a,M} \sqcup Op^M_{b,j}$ holds.

Further based on the induction hypothesis$^{\dagger}$, for $m = M\text{+}1$, the theorem also holds, i.e., for $\forall i \in \{0, \cdots, M\}$ and $\forall j \in \{0, \cdots, n\text{-}1\}$, $Op^j_{a,i} \sqcup Op^i_{b,j}$ holds. By the induction argument, the theorem holds, i.e., for any $m$ and $n$, and for $\forall i \in \{0, \cdots, m\text{-}1\}$ and $\forall j \in \{0, \cdots, n\text{-}1\}$, $Op^j_{a,i} \sqcup Op^i_{b,j}$ holds. The theorem has hereby been proved.

## 4 Protocols for Achieving Contextualization

If the pre-condition of every **SLOT** invocation is satisfied, *intention preservation* and *convergence* can be achieved by OT (e.g., by Theorem 1). The question is how to ensure the pre-condition of every **SLOT** invocation, i.e., the two lists to be transformed must be contextualized and context-equivalent.

To facilitate asynchronous collaboration, each collaborating site maintains an *OB* and an *IB* to separate local and remote operations, where *OB* stores unpropagated local operations and *IB* stores unreplayed remote operations. On the one hand, operations generated at a local site are executed instantly to gain good responsiveness and then appended to *OB*, waiting to be propagated via an operation propagation

protocol. On the other hand, operations propagated from remote sites are appended to *IB*, waiting to be replayed locally via an operation replaying protocol. The two protocols are essential to ensure *OB* and *IB* at any site are contextualized and context-equivalent.

### 4.1 CCOP: A Coordinated Operation Propagation Protocol

Most synchronous systems do not use coordinated operation propagation protocols. In these systems, any site can freely propagate new operations upon generation without coordinating propagation requests from other sites. For example *GOT*, *GOTO*, *adOPTed*, and *COT* rely on timestamping individual operations with state vectors to capture causal/concurrent relationships among operations. Some systems adopt coordinated propagation protocols for the purpose of reducing the complexity of TCAs or relaxing some constraints on transformation functions. *Jupiter* fall into this category.

We adopt a coordinated operation propagation protocol to achieve contextualization and list context equivalence. Enforced by a coordination rule, a site can only initiate a new propagation (of a list of operations) when there are no outstanding propagations, in other words, when all preceding propagations have been received by all designated sites. The coordination rule can be warranted by employing a sequencer or a token manager, or by running a centralized or distributed synchronization protocol.

At site $s$, to propagate all operations from $OB_s$, the **CCOP** (Coordinated Contextualized Operation Propagation) protocol is executed as follows.

**Protocol 1 CCOP**

1. Start the protocol by following the coordination rule.
2. Skip this step if $IB_s$ is empty. Otherwise, transform the list from $OB_s$ with the one from $IB_s$ by **SLOT**($OB_s$, $IB_s$).
3. Propagate the list from $OB_s$ to designated sites and empty $OB_s$.
4. Append the propagated list to $IB_t$ when the propagation arrives at a designated site $t$.
5. End the protocol when the propagation has arrived at all designated sites.

To ensure good local responsiveness, new local operations are allowed to be generated at any time, even during the execution of the **CCOP** protocol. Operations generated in the meantime are first stored in a temporary buffer and then moved to $OB_s$ after the protocol is ended. The protocol may take a while to end, especially when the number of collaborating sites is big or the propagated list is big, but the protocol execution time has nothing to do with the local responsiveness (it only affects when local operations are to be replayed at remote sites).

It is worth pointing out that coordinated operation propagation may become a performance bottleneck in synchronous systems with a large number of collaborating sites. However, it is generally not a problem in asynchronous systems because: 1) the frequency of propagations is normally not high; 2) consequently, cases where multiple sites request for propagating operations at the same time are not common; and 3) even some cases do exist, postponement of some propagations would not affect the system performance as synchronous notification of each other's progress is not expected. Nevertheless, for the **SLOT** control algorithm to be used in synchronous systems, the propagation protocol can be optimized in various ways. For example, The **SCOP** protocol used by the *NICE* collaborative editing system adopts a notification server to synchronize concurrent propagations before broadcasting them to destination sites (Shen & Sun 2002).

### 4.2 ACOR: An Adaptive Operation Replaying Protocol

At any collaborating site, when local operations are to be propagated from *OB*, the **CCOP** protocol ensures contextualization of *OB* and *IB* and context equivalence between them . Similarly, when remote operations are to be replayed from *IB*, an operation replaying protocol is required to ensure contextualization of *OB* and *IB* and context equivalence between them.

Because local and remote operations modify the same shared data source, execution of local operations and replaying of remote operations must be mutually exclusive. Furthermore, in case of contention between local and remote operations, local operations must be given the priority to ensure a good local responsiveness. Therefore, remote operations are ideally to be replayed when no local operations are being executed. However, it is infeasible to predict when a user is going to issue operations. In synchronous systems, remote operations arrive at a site one-by-one and each remote operation is replayed when no local operation is being executed at that site. If the user at that site issues an operation in the meantime when a remote operation is being replayed, execution of the local operation will be delayed until replaying of the remote operation is over. Postponement of the local operation execution by one remote operation replaying time would not affect local responsiveness much as replaying one remote operation usually takes very little time.

However, in asynchronous systems, remote operations arrive at a site batch-by-batch and each batch could consist of as many as hundreds of operations. If all operations in *IB* at that site were replayed as a continual stream, local operations would suffer starvation, resulting in poor local responsiveness. To tackle this issue, after replaying a remote operation from *IB*, the operation replaying protocol should give local operations a chance to execute (if any) before replaying the next one.

At site $t$, to replay all operations from $IB_t$, the **ACOR** (Adaptive Contextualized Operation Replaying) protocol is executed as follows.

**Protocol 2 ACOR**

```
1:  s ← 0;
2:  if (|OB_t| > s) then
3:     e ← |OB_t| − 1;
4:     SLOT(IB_t, OB_t[s, e]);
5:     s ← e + 1;
6:  end if
7:  while (|IB_t| > 0) do
8:     give_way();
9:     acquire_lock();
10:    if (|OB_t| > s) then
11:       e ← |OB_t| − 1;
12:       SLOT(IB_t, OB_t[s, e]);
13:       s ← e + 1;
14:    end if
15:    Op ← IB_t[0];
16:    replay_operation(Op);
17:    remove_operation(IB_t, 0);
18:    release_lock();
19: end while
```

When site $t$ is in the process of executing **ACOR** to replay remote operations from $IB_t$, it can still receive remote operations, which are first temporarily buffered and then moved to $IB_t$ after the protocol is ended. This implies that $IB_t$ will not grow during the protocol execution, and newly arrived remote operations will be replayed in the subsequent **ACOR** protocol execution. In contrast, $OB_t$ may grow and the protocol must ensure operations in $IB_t$ are transformed with all operations in $OB_t$ no matter when these operations are generated and appended to $OB_t$ (lines 2-6 or 10-14).

To minimize the impact on local responsiveness, each remote operation should "give way" to local operations (line 8). The *give_way*() procedure halts the protocol execution until the data source is not being modified by any local operation. Checking the state of the data source is done by a thread, which alternates between checking and sleeping (if the data source is unavailable) until the data source becomes available. The sleeping time is adaptive in the sense that it is dynamically calculated according to the type of the local operation being executed and the frequency in which recent local operations have been generated.

When a remote operation is ready to be replayed, the protocol will first exclusively lock the data source to prevent it from being modified by local operations (line 9). During the locking period, raw local events such as keystrokes may be buffered for the generation of new local operations. Then operations in $IB_t$ must be transformed (line 10-14) with new local operations (generated in the meantime when the remote operation was giving way). Finally, after the remote operation is replayed (line 16), it is removed from $IB_t$ (line 17) and the lock on the data source is released (line 18). After that, the next operation in $IB_t$ will look for its chance to be replayed and the protocol proceeds until $IB_t$ becomes empty.

### 4.3 Verification of the CCOP and ACOR Protocols

The example in Figure 2 has actually used the **CCOP** protocol to propagate local operations and the **ACOR** protocol to replay remote operations. It has shown that *intention preservation* and *convergence* have been achieved by using the two protocols in a collaborative session involving two sites. To systematically prove the two protocols together can maintain consistency in a distributed collaborative session involving arbitrary number of collaborating sites (two or more), we need to verify them against the three consistency properties.

For *causality preservation*, we need to prove that given any two operations $Op_a$ and $Op_b$, if $Op_a$ is causally before $Op_b$, then $Op_a$ must be executed before $Op_b$ at all sites. For *intention preservation*, we need to prove that all **SLOT** invocations in **CCOP** and **ACOR** have satisfied their pre-conditions. For *convergence*, we need to prove that the context of the shared data source is identical across all sites after all operations have been executed at all sites. However, due to space limitation, we only formally verify the *causality preservation*, as described by Theorem 2.

**Theorem 2** *Given any two operations $Op_a$ and $Op_b$, if $Op_a$ is causally before $Op_b$, then $Op_a$ must be executed before $Op_b$ at all collaborating sites.*

**Proof:** There are two possibilities. One is that $Op_a$ is generated before $Op_b$ at the same site, e.g., site $i$. If $Op_b$ is generated before the propagation of $Op_a$, $Op_a$ will be positioned before $Op_b$ in $OB_i$. There are two cases in propagating $Op_a$ and $Op_b$. First, if $Op_a$ and $Op_b$ are propagated in the same list (i.e., in one **CCOP** protocol execution), for any remote site $t$, $Op_a$ and $Op_b$ will arrive at site $t$ and be appended to $IB_t$ at the same time, where $Op_a$ is still positioned before $Op_b$. Then the execution of **ACOR** protocol at site $t$ ensures $Op_a$ is replayed before $Op_b$. Second, if $Op_a$ and $Op_b$ are propagated in two lists (i.e., in two **CCOP** protocol executions), the execution of two **CCOP** protocols in sequence ensures the propagation containing operation $Op_a$ arrives at any remote site before the one containing operation $Op_b$ does. At any remote site, $Op_a$ is always replayed before $Op_b$ no matter whether they are replayed in one **ACOR** protocol execution or in two **ACOR** protocol executions in sequence. If $Op_b$ is generated after the propagation of $Op_a$, it will be the same as the second case.

Another possibility is that $Op_a$ and $Op_b$ are generated at two sites, e.g., site $i$ and $j$ respectively, and $Op_b$ is generated after the execution of $Op_a$ at site $j$. This implies that the propagation containing operation $Op_a$ must have arrived at site $j$ before $Op_b$ is generated. According to the **CCOP** protocol, the propagation containing operation $Op_b$ cannot be initiated before the one containing operation $Op_a$ has arrived at all sites. This ensures the propagation containing operation $Op_a$ must arrive at any remote site before the one containing operation $Op_b$ does. Furthermore, at any remote site, $Op_a$ is always replayed before $Op_b$ no matter whether they are replayed in one **ACOR** execution or in two **ACOR** executions in sequence. The theorem has hereby been proved.

## 5 Significance and Applications

First of all, most TCAs designed for synchronous systems, such as *dOPT*, *GOT*, *GOTO*, *adOPTed*, and *COT*, require every operation to be timestamped with a state vector in order to capture their causal/concurrent relationships. The **SLOT** control algorithm, specifically designed for asynchronous systems, does not require operations to be timestamped because their causal/concurrent relationships are captured by separating local and remote operations in two buffers (*OB* and *IB*) and by enforcing operation propagation and replaying protocols (**CCOP** and **ACOR**) to achieve contextualization and list context equivalence.

Second, most TCAs designed for synchronous systems avoid using ET (Exclusion Transformation) functions (Sun et al. 1998) because it is difficult to design ET functions that satisfy the *Reversibility Property* - $ET(IT(Op_a, Op_b), Op_b) = Op_a$ (Sun & Sun 2006, **?**). Function $\mathbf{ET}(Op_a, Op_b)$ transforms $Op_a$ against $Op_b$ in such a way that the impact of $Op_b$ is effectively excluded in the output operation $Op'_a$. ET functions are required by algorithms, such as *GOT*, *GOTO*, and *SOCT3*, which are based on a single linear history buffer to store all executed local and remote (transformed) operations. The **SLOT** control algorithm has also avoided using ET functions.

Third, in addition to Property 4: *TP-1*, which is essential for preserving convergence, IT functions need to satisfy *Transformation Property 2 (TP-2)*, which requires IT functions to be defined in such a way that transformation of an operation against a set of mutually concurrent operations is irrelevant of the order in which the set of operations are mutually transformed (Sun & Ellis 1998).

**Property 5** *Transformation Property 2 (TP-2)*

Given three concurrent operations $Op_a$, $Op_b$, and $Op_c$, if $Op'_a = \mathbf{IT}(Op_a, Op_b)$ and $Op'_b = \mathbf{IT}(Op_b, Op_a)$, then $\mathbf{IT}(\mathbf{IT}(Op_c, Op_a), Op'_b) = \mathbf{IT}(\mathbf{IT}(Op_c, Op_b), Op'_a)$.

While it is relatively easy to satisfy *TP-1*, it is rather difficult to design IT functions and verify them against *TP-2* that is also essential for preserving convergence. The violation of *TP-2* is mainly caused by uncoordinated operation propagation, where the same set of concurrent operations may arrive at different collaborating sites in different orders and hence may be transformed in different orders. TCAs subject to *TP-2* include *dOPT*, *GOTO*, and *adOPTed*.

As it is non-trivial to satisfy *TP-2*, some TCAs choose to break its pre-condition either by turning to coordinated operation propagation, which forces the same set of concurrent operations to arrive at all collaborating sites in the same order or by re-ordering the set of concurrent operations according to a total order so that they are transformed in the same order at all sites. *Jupitor* belongs to the first category, while *GOT* and *COT* belong to the second category. Some work takes a different approach that solves the *TP-2* problem instead of avoiding it. For example, SDT solved the problem by fixing problems in IT functions (Li & Li 2004) and a new OT framework was proposed to tackle *TP-2* and other OT-related problems (Li & Li 2007). **SLOT** belongs to the first category because **CCOP** ensures the same set of operation lists arrive at different collaborating sites in the same order. Furthermore, because **SLOT** is only invoked between the lists in $OB$ and $IB$ at the same site, it is never possible for a list in $OB$ to be transformed with lists in multiple $IB$s and in different orders.

Lying on the operation propagation and replaying protocols to achieve contextualization and list context equivalence, the **SLOT** control algorithm can be used in a wide range of distributed collaborative systems. It is particularly more efficient than other algorithms when used in asynchronous systems. This OT-based concurrency control solution, including the **SLOT** algorithm, and **CCOP** and **ACOR** protocols have been implemented in a few prototype collaborative systems, e.g. the *CoEclipse* system that supports collaborative programming.

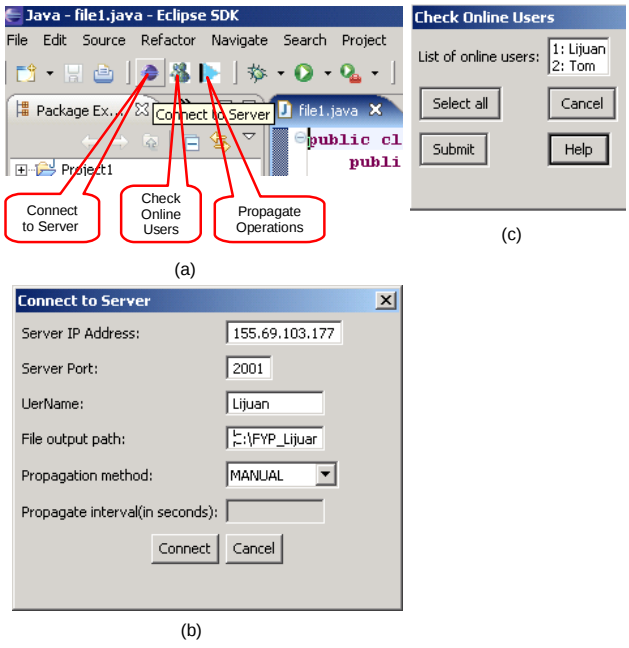


Figure 3: CoEclipse

*Eclipse* is well-known for the Java IDE (Integrated Development Environment). We developed *CoEclipse* to support collaborative programming in Java - a group of programmers can code the same Java project or source code file asynchronously or synchronously. As shown in Figure 3(a), *CoEclipse* appears as three plug-in buttons on the *Eclipse* user interface, which are *Connect to Server*, *Check Online Users*, and *Propagate Operations* from left to right.

To propagate operations made on the shared source code to other programmers online, one needs to first connect to the collaboration server by pushing the *Connect to Server* button. The server is to manage the list of online programmers and to facilitate running the operation propagation protocol. Figure 3(b) shows the dialog box before the connection is made. If the propagation method is set to *MANUAL*, each propagation has to be initiated by pushing the *Propagate Operations* button and the collaboration mode is asynchronous. If the propagation method is set to *AUTOMATIC* with a short interval, the collaboration mode is synchronous instead. One has the flexibility of choosing online programmers to whom her/his operations will be propagated by pushing the *Check Online Users* button. As shown in Figure 3(c), one may choose to propagate to all online programmers or only some of them.

## 6 Conclusions and Future Work

A collaborative system differs from other distributed systems by three characteristic components: multiple human users, interactive applications, and the high-latency Internet. Concurrency control is the key to managing the contention for distributed data resources and to ensure the orderly access to shared data. It is a major issue in most distributed systems and collaborative systems are of no exception. However, concurrency control techniques devised for distributed systems are generally unsuitable for collaborative systems because these characteristic components have special concurrency control requirements.

OT is a concurrency control technique originally invented for synchronous collaborative systems to meet those special requirements. However, to apply OT to asynchronous systems, a major technical challenge is to devise an efficient transformation control algorithm because existing algorithms are catered for synchronous systems only and are inefficient to be used in asynchronous systems.

Our major contribution in this paper is a novel, efficient, contextualization-based transformation control algorithm **SLOT** underpinned by an operation propagation protocol **CCOP** and an operation replaying protocol **ACOR** to achieve contextualization and list context equivalence. This algorithm is particularly more efficient than existing ones when used in asynchronous systems, and can also be used to support a wide spectrum of collaboration paradigms because it is as efficient when used in synchronous systems. Furthermore, the algorithm has most of the merits that some of existing algorithms have achieved, such as no timestamping of operations, no ET functions, and free of *TP-2*.

The correctness of the algorithm and protocols in terms of consistency maintenance has been formally verified. The usefulness of the algorithm and protocols has been demonstrated by a variety of prototype collaborative applications. We are currently investigating other collaborative techniques such as collaborative undo and asymmetric collaboration, by extending the solution presented in this paper.


## Acknowledgment

The authors wish to thank Ms Lijuan Geng for the initial development of the *CoEclipse* prototype at Nanyang Technological University in Singapore.

## References


Bernstein, P., Goodman, N. & Hadzilacos, V. (1987), *Concurrency Control and Recorvery in Database Systems*, Addision-Welsley.

Ellis, C. & Gibbs, S. (1989), Concurrency control in groupware systems, *in* 'Proceedings of ACM SIGMOD Conference on Management of Data', pp. 399–407.

Greenberg, S. & Marwood, D. (1994), Real time groupware as a distributed system: concurrency control and its effect on the interface, *in* 'Proceedings of ACM Conference on Computer Supported Cooperative Work', ACM Press, pp. 207–217.

Gu, N., Yang, J. & Zhang, Q. (2005), Consistency maintenance based on the mark and retrace technique in groupware systems, *in* 'Proceedings of the ACM Conference on Supporting Group Work', ACM Press, pp. 264–273.

Karsenty, A. & Beaudouin-Lafon, M. (1993), An algorithm for distributed groupware applications, *in* 'Proceedings of 13th Conference on Distributed Groupware Computing Systems', pp. 195–202.

Knister, M. & Prakash, A. (1993), 'Issues in the design of a toolkit for supporting mulitple group editors', *Journal of Usenix Association* **6**(2), 135–166.

Li, D. & Li, R. (2004), Ensuring content and intention consistency in real-time group editors, *in* 'Proceedings of the IEEE International Conference on Distributed Computing Systems', pp. 748–755.

Li, R. & Li, D. (2007), 'A new operational transformation framework for real-time group editors', *IEEE Transactions on Parallel and Distributed Systems* **18**(3), 307 – 319.

Nichols, D. A., Curtis, P., Dixon, M. & Lamping, J. (1995), High-latency, low-bandwidth windowing in the jupiter collaboration system, *in* 'Proceedings of the ACM Symposium on User Interface Software and Technology', Distributed User Interfaces, pp. 111–120.

Raynal, M. & Singhal, M. (1996), 'Logical time: capturing causality in distributed systems', *IEEE Computer Magazine* **29**(2), 49–56.

Ressel, M., Nitsche-Ruhland, D. & Gunzenbauser, R. (1996), An integrating, transformation-oriented approach to concurrency control and undo in group editors, *in* 'Proceedings of ACM Conference on Computer Supported Cooperative Work', ACM Press, pp. 288–297.

Shen, H. & Sun, C. (2002), Flexible Notification for Collaborative Systems, *in* 'Proceedings of ACM Conference on Computer-Supported Cooperative Work', ACM Press, pp. 77–86.

Sun, C. & Ellis, C. (1998), Operational transformation in real-time group editors: Issues, algorithms, and achievements, *in* 'Proceedings of ACM Conference on Computer Supported Cooperative Work', pp. 59–68.

Sun, C., Jia, X., Zhang, Y., Yang, Y. & Chen, D. (1998), 'Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems', *ACM Transactions on Computer-Human Interaction* **5**(1), 63 – 108.

Sun, C., Xia, S., Sun, D., Chen, D., Shen, H. & Cai, W. (2006), 'Transparent adaptation of single-user applications for multi-user real-time collaboration', *ACM Transactions on Computer-Human Interaction* **13**(4), 531–582.

Sun, D. & Sun, C. (2006), Operation context and context-based operational transformation, *in* 'Proceedings of ACM Conference on Computer Supported Cooperative Work', pp. 279–288.