An Algorithm for Selective Undo of Any Operation in Collaborative Applications *

Bin Shao†
School of Computer Science
Fudan University
Shanghai, China
binshao@microsoft.com

Du Li
Nokia Research Center
Palo Alto, CA, USA
lidu008@gmail.com

Ning Gu
School of Computer Science
Fudan University
Shanghai, China
ninggu@fudan.edu.cn

ABSTRACT

Selective undo allows users to undo any operation in the history and is considered a key feature in collaborative applications. Operational transformation (OT) is a powerful tool for implementing selective undo because it can be used to rearrange operations in a history in arbitrary orders. Despite the significant progress over the past two decades, however, there is still a space for improvements. Most existing works take time quadratic or even exponential in the size of the operation history H to undo an operation. Although this might be acceptable for real-time collaboration, it would be suboptimal in mobile and asynchronous collaborative applications in which a long history may accumulate. In addition, it is important to prove an algorithm with regard to the correctness criteria it assumes. This paper proposes a novel OT-based algorithm that provides integrated do and selective undo. The algorithm achieves time complexity of $O(|H|)$ in both do and undo by keeping the history in a special operation effects relation order. Its correctness is formally proved with regard to formalized, provable conditions that are extended from a recent theoretical framework.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces —*Collaborative computing*

General Terms

Algorithms, Theory

*The work is supported in part by National Natural Science Foundation of China under Grant No. 60736020, National Grand Fundamental Research 973 Program of China under Grant No. 2005CB321905, Shanghai Science & Technology Committee Key Fundamental Research Project under Grant No. 08JC1402700 and Shanghai Leading Academic Discipline Project under Grant No. B114.

†Bin Shao is currently with Microsoft Research Asia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GROUP'2010, November 7–10, 2010, Sanibel Island, Florida, USA.
Copyright 2010 ACM 978-1-4503-0387-3/10/11 ...$10.00.

1. INTRODUCTION

Undo is a key feature in interactive applications. Many familiar single-user applications, including text editors, word processors, design tools and even Web browsers, allow the user to undo operations in a chronological order. Undo is often used for user-level error recovery, e.g., to fix typos. It can also encourage users to explore unfamiliar capabilities in an application provided that the effects of erroneous operations can always be undone.

In collaborative applications, undo is significantly more challenging [9, 2, 14]. When the users are distributed and work in parallel, local and remote operations could be interleaved arbitrarily due to concurrency. A user request to undo could mean to undo a local operation or a remote operation. To undo a remote operation is even trickier because there can be multiple operations by several remote users and it is important to unambiguously specify which operation by which user. Otherwise, the undo action could be interpreted arbitrarily at different sites, leading to unpredictable consequences in the system. Therefore, in a distributed environment, it becomes necessary to support selective undo at the algorithm level, i.e., to undo the effects of any selected operations in the history.

Operational transformation (OT) [3, 15] is an optimistic consistency control technique that lies in the heart of many modern collaborative applications such as collaborative editing systems[1] and Google Wave[2]. Those applications have gone far beyond the pure textual group editors in which OT was originated [3]. In particular, it has been shown that OT is a powerful tool for implementing selective undo [9, 14] mainly because it is able to transform any pair of operations to commute. This property makes it possible to remove the effects of any operations and rearrange the history arbitrarily, e.g., to simulate any order of execution and explore numerous possible versions of the data under different permutations and combinations of operations.

Despite the great research progress in the topic of OT-based selective undo, e.g., [9, 10, 14, 4, 18, 17], two important issues deserve further investigation: First, how to make an undo solution efficient for real-time as well as non-realtime collaborative applications. For example, approaches in [9, 4] take time quadratic and that in [17] takes time exponential in the size of history H to undo an operation. There is a room for improvement when $|H|$ grows long, e.g., in mobile and asynchronous settings in which the collaborators often have to work in isolation before sync [12]. Secondly,

[1]http://www.ntu.edu.sg/home/czsun/
[2]http://www.waveprotocol.org/

how to formally prove the correctness of an undo solution with regard to certain conditions if they are assumed in the solution. For example, the so-called transformation properties TP1/TP2 [9, 11] are assumed in many OT works as conditions for achieving convergence, and most OT algorithms are developed under the framework of [16] which includes a condition called intention preservation.

In this paper we propose an alternative OT-based algorithm with integrated support for do and selective undo. This work makes the following two major contributions:

1. It can do as well as undo any operation in $O(|H|)$ time.
2. It is based on formalized conditions and its correctness formally proved with regard to all its conditions.

This work builds on a recent theoretical framework in this area, Admissibility-Based Transformation or ABT [6]. Originally, ABT is proposed for developing and proving OT-based do algorithms. It includes two formalized conditions, causality and admissibility preservation, which together imply convergence. Conceptually, admissibility means (1) that the invocation of any operation does not violate the relative position of objects that has been established by admissible operations invoked earlier, and (2) that every operation is admissible when it is generated.

This paper will extend the admissibility theory to undo and give a new algorithm (called ABT-Undo or ABTU) with integrated support of do and selective undo. The correctness of ABTU will be formally proved. Furthermore, the relative position (called the operation effects relation $\prec$) is utilized to keep operations in the history H in $\prec$ order so as to achieve $O(|H|)$ time of both do and undo. Note, however, that our work does not need an extra step to order the history.

The rest of this paper is organized as follows: Section 2 will first explain the basic ideas of OT. Section 3 surveys related previous works. Section 4 discusses the consistency model and Section 5 presents the ABTU algorithm. Section 6 proves and analyzes ABTU. Section 7 concludes.

2. BACKGROUND: OT AND UNDO

To explain the basic ideas of OT-based do and selective undo, consider the scenario as shown in Figure 1. Suppose that two users, Alice and Bob, are collaboratively editing a shared document with an initial string "b". The data is replicated at both sites so that the users can work in parallel. Following the conventions [3], we model the shared data as a linear string of characters (or atomic objects). Let the position of the first object be zero. Alice changes her data to "ab" by operation o_1=ins(0,a). Concurrently, Bob changes his data to "bc" by operation o_2=ins(1,c).

When Alice receives o_2, if o_2 were executed as-is, her document would become "acb", which would violate the original intention of o_2 to insert 'c' after 'b' [16]. The basic idea of OT [3] is to transform o_2 (i.e., to shift its position) into an appropriate form that can be correctly executed in current document state. Here we should transform o_2 to a new form o_2'=ins(2,c) to account for the effect of o_1 that is concurrent to o_2 and has been executed earlier. However, when Bob receives o_1, o_1 can be executed as-is because the execution of o_2 does not invalidate o_1's position.

Now the two sites converge in state "abc". Then, Alice issues o_3=del(0,a). After o_3 is executed, Bob issues o_4=del(0,b). The states at both sites become "c".

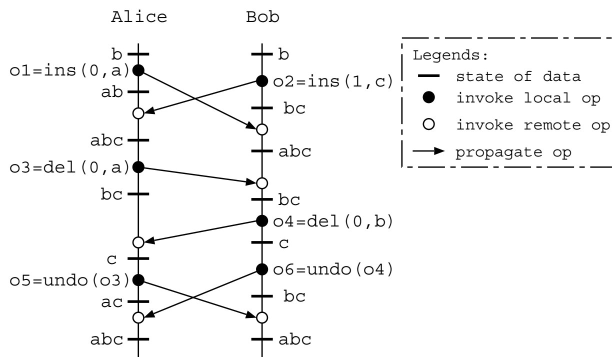

Figure 1: A scenario of OT-based do and undo: Operations can be invoked in any orders and any operations in the history can be selected for undo.

At this point, the history of Alice is $[o_1, o_2', o_3, o_4]$ and that of Bob is $[o_2, o_1, o_3, o_4]$. Next, suppose that Alice selectively undoes her last step o_3 by operation o_5 to restore object 'a', and concurrently Bob selectively undoes his last step o_4 by operation o_6 to restore object 'b'. The states of the two sites become "ac" and "bc", respectively. Note that the two undo operations, o_5 and o_6, are concurrent and defined on the same document state "c".

In general, we cannot disregard the inherent dependency between an undo and its corresponding do operation. Otherwise, e.g., in this scenario, if we naively interpreted o_5 and o_6 as to insert two *new* objects 'a' and 'b', respectively, it would be equally possible to get either result, "abc" or "bac", depending on the tie-breaking policies. However, if we knew that the *intentions* of the two undo operations are to restore objects originally in state "abc", the *correct* result would be "abc" without ambiguity.

Therefore, the problem of providing an integrated do/undo solution is manifold: (1) what the correctness criteria are, (2) how to design the algorithm and prove its correctness, and (3) how to scale the algorithm with the size of the history. We will address these three issues in ABTU.

3. RELATED WORK

The topic of undo and recovery has been well studied in databases with the main purpose to maintain the ACID properties of transactions. In traditional distributed systems, convergence is usually the key constraint to satisfy. In the literature of interactive and collaborative applications, many undo approaches are proposed that are not based on OT, e.g., [1, 2]. Those works usually consider read/write operations and an approach is correct as long as it converges, e.g., by finding a serializable schedule of operations.

OT-based works in general consider two primitives, insert and delete, in place of writes, to avoid overwriting of interaction results. Any pair of operations can be transformed to commute and operations can execute in arbitrary orders at different sites. As a lock-free, nonblocking optimistic technique, OT trades consistency for local responsiveness and extra constraints are imposed on convergence, such as intention preservation [16] and the effects relation [5, 6].

Traditionally, most OT works focus on supporting do, e.g., [3, 16, 15, 11, 13, 5, 6, 12]. Here we focus on OT-based selective undo. We first survey related previous works and then summarize the novel contributions in this work.

[Prakash and Knister 1994] DistEdit [9] is the first OT-based selective undo solution. To selectively undo operation o in the history H, it first transposes o to the end of H and generates its inverse $\overline{o}$. Then $\overline{o}$ is executed and appended to H as normal. An operation o cannot be undone if it cannot be transposed with some operation executed later. This happens when the two operations are considered as conflicting, i.e., their effect objects are in adjacent or overlapping positions. Because DistEdit needs to recursively remove the do-undo pairs in H that affect this conflict detection, its time complexity is $O(|H|^2)$.

[Ressel et al. 1999] adOPTed [10] only supports undo of local operations in chronological order. They interpret undo(o) as an inverse $\overline{o}$ generated in the state immediately after executing its original operation o, concurrent with all operations executed after o in the history. To undo the last local operation o in history H, it first generates its inverse $\overline{o}$ and then transposes $\overline{o}$ to the end of H. After that, $\overline{o}$ is executed in current state as normal and appended to H. Their work achieves convergence by assuming that the transformation functions satisfy two transformation properties (known as TP1 and TP2 [9, 11]). However, this is not really achieved, due to a counterexample given in [16].

[Ferrié et al. 2004] The approach in [4] supports selective undo while resembling [10] in its interpretation of undo. Their transformation functions are based on SOCT2 [13], in which TP2 is assumed but not really satisfied due to a counterexample given in [8]. Because they transpose H to extract concurrent operations when processing an operation, both do and undo take $O(|H|^2)$ time.

[Weiss et al. 2008] The selective undo approach of UNO [18] resembles that of [4]. Their transformation functions are based on TTF [7]. The time complexity of UNO is claimed as linear in the size of the shared document. The convergence of UNO also relies on TP1 and TP2. For verification they resort to theorem prover. Its correctness also relies on the intention preservation condition, which is specified similarly as in [5] yet not formally proved.

[Sun et al. 2009] Both GOTO [14] and COT [17] supports integrated do and selective undo. They also interpret undo as concurrent inverse as in [10] except that they couple o and $\overline{o}$ such that a do-undo pair behaves as an identity operation during transformation. They only mark the original operation o, without saving its inverse $\overline{o}$ in the history. In COT [17], they propose a new context relation theory to model the special relations between normal operations, inverse operations, and their transformed forms.

By their own analyses [17], the time complexity for undo in both GOTO and COT is exponential in the size of H. The time complexity for do in GOTO is exponential in the size of H; COT achieves a time complexity of $O(|H|)$ for do by a buffering scheme which saves up to N versions of every operation, where N is the number of sites.

Comparisons: Early works [9, 10] do not support full-featured selective undo. Works in [11, 4, 18, 14] assume TP1/TP2, which results in transformation functions that are often intricate and difficult to develop formal proofs. Moreover, [4, 18] and [14, 17] are developed under the framework of [16] with three conditions, namely, convergence, causality and intention preservation. To our best knowledge, COT [17] is the latest work in selective undo.

ABTU is based on formalized conditions and its correctness is formally proved. The two formal conditions, causality and admissibility preservation, together imply convergence [6]. Hence it no longer needs to assume TP1/TP2 for convergence. When processing undo(o), we treat $\overline{o}$ as one that happens after all operations in H yet contextually equivalent with all those in H that are ordered after o. As will be explained in Section 5.5, we can flexibly interpret the semantics of undo similarly as in either [9] or [10].

ABTU provides integrated support for do and selective undo, with the following unique properties: (1) operations in history H are kept in the order of their relative position, i.e., the effects relation $\prec$; (2) operations with the same effect objects are ordered by their happens-before relation, e.g., $\overline{o}$ immediately follows o in the history; (3) its space complexity is $O(|H|)$ and, due to the $\prec$ order of H, the time complexity of both do and undo is $O(|H|)$. Note that ABTU keeps all operations in one history whereas previous works [6, 12] keep insert/delete operations in separate histories.

4. THE CONSISTENCY MODEL

We will first explicate some general guidelines for providing integrated support of do and undo. Then we will formalize the correctness criteria.

4.1 General Principles

Due to concurrency, undo is more challenging in a collaborative application than in a single-user application. The following principles have been used implicitly or explicitly in previous works, e.g., [1, 9, 2, 14]. We also adopt them as guidelines in our work without making novelty claims:

(a) Undo is an operation that can also be undone.
(b) An operation can be undone exactly once.
(c) Given any operation o, if there are operations that depend on the effect of o, undo(o) should abort.
(d) The effects of undo operations should be consistent with normal (do) operations.

Now, we discuss why these principles are relevant.

(a) Like normal operations, an undo is just a request from the user to achieve certain effect, i.e., to remove the effect produced by some action requested by this or another user earlier. Not surprisingly, the undo action is also subject to a later undo request, i.e., to redo the original operation.

(b) The effect of any operation o cannot be undone more than once. In a collaborative environment, multiple users may concurrently request to undo the effect of the same operation. For example, after one user inserts a object, several other users request to undo the insertion. In this case, only one undo request should be honored. Otherwise, if the inverse (delete) operation were executed multiple times, some other objects could be deleted by mistake. This would violate the intention of undo, i.e., to restore the data to some state as if o had never been executed.

(c) Undo as a user operation is to restore the data to a previous state. Often another normal do operation may well serve the same purpose. For example, after o_1 inserts an object 'x', to remove this effect, we may choose to either undo o_1 by operation o_2 or just delete 'x' by operation o_3. The latter case can be considered as a type of *implicit undo*. Either operation, o_2 or o_3, depends on the effect of o_1. After either one is executed, any later request to undo o_1 should be ignored. This is compatible with principle (b).

(d) For any do/undo operation o, undo(o) not only happens after o but also inherently depends on the effect of o.

Hence we cannot look at undo(o) naively as a new do operation to execute $\overline{o}$, the inverse of o. The effect of $\overline{o}$ must be somehow correlated to that of o. In other words, the effects of the undo operations must be *consistent* with those of their corresponding do operations. The question is how to formulate the notion of "consistency" in this context.

4.2 Correctness Criteria

We will formalize two consistency (or correctness) criteria, namely, causality and admissibility preservation, for formally proving algorithms that provide integrated support of do and undo. These two conditions are first proposed in the ABT framework [6] which only treats do operations. Here we extend them to treat both do and undo. Note, however, that for space reasons here we only explain the part for undo that is not presented in [6]. We say that an integrated do/undo solution is correct (or consistent) if these two conditions are always satisfied.

Following established notations [3, 15], for any two operations o_1 and o_2, their temporal relation is denoted as $o_1 \rightarrow o_2$ if o_1 happens before o_2; or it is denoted as $o_1 \parallel o_2$ if they are concurrent, i.e., neither $o_1 \rightarrow o_2$ nor $o_2 \rightarrow o_1$.

Definition 4.1. ***Causality Preservation**: Given any two (do/undo) operations, o_1 and o_2, if $o_1 \rightarrow o_2$, then o_1 is invoked before o_2 at any site in the system.*

The condition of causality preservation is compatible with the above principles (a)-(c). Since undo is treated as a normal operation by principle (a), it must observe the cause-effect relation. Further, the concurrent and dependency relation in principles (b) and (c) are easy to detect if the happens-before relation is observed in the system.

Definition 4.2. ***Admissibility Preservation**: The invocation of any (do/undo) operation does not violate the effects relation $\prec$ that has been established in the system.*

According to [5, 6], objects that ever appear in the same (linear) document state can be totally ordered by their relative positions. Consistently with this total order, operations can also be totally ordered by the relation of the objects they insert or delete. We use notation $\prec$ to denote this total order and the operation relation (called **effects relation**). The core notion of effects relation $\prec$ has been formally elaborated in [6]. Intuitively, the admissibility condition can be restated as follows: an invocation of any operation is admissible if it does not violate the effects relation established by other admissible invocations in the system; and any operation is admissible when it is generated. It is proved that these two conditions, causality and admissibility preservation, together imply convergence [6]. That is, after all operations are invoked at all sites, the data replicas will converge with all objects in the order of the effects relation $\prec$.

4.3 An Example

Here we use an example to explain the concepts here, as illustrated in the scenario in Figure 2. We focus on conceptually explaining what the "correct" results must be, without involving any specific do/undo algorithms.

Suppose that the three sites start from the same initial state "ab". The left site issues operations o_1 and o_2 to delete the objects 'a' and 'b', respectively. Concurrently, the middle site issues o_4 to insert 'x' after "ab". After these three

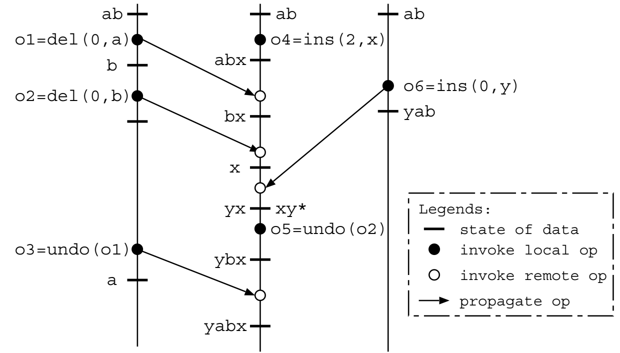

Figure 2: A scenario to illustrate the effects relation $\prec$ and admissibility.

operations (o_4, o_1 and o_2) are invoked, the state of the middle site must be "x". Consider that concurrently the right site issues o_6 to insert 'y' before "ab". When o_6 is invoked at the middle site, since its current state is "x", the result could be either "xy" or "yx", depending on the specific algorithm and its tie-breaking policies.

Using the notion of effects relation, the initial relation is $a \prec b$. When o_4 is generated, it becomes $a \prec b \prec x$. And when o_6 is generated, it is $y \prec a \prec b$. By transitivity, it must be $y \prec a \prec b \prec x$. Operations o_1 and o_2 are to delete objects that are already there and should not change the effects relation. Hence the state of the middle site must be "yx" without ambiguity after invoking o_4, o_1, o_2 and o_6. Furthermore, after undo operations o_5 and o_3 are invoked to restore the deleted objects 'b' and 'a', respectively, its state must be "yabx" without ambiguity according to the effects relation. Note that the effects relation is determined when operations are generated and hence inherent.

5. THE ABTU ALGORITHM

We will first overview the algorithm structure, introduce some basic notations and transformation functions, and then discuss involved functions. As will be shown, the entire algorithm is specified in four functions. Our approach maintains a history H at each site, operations in which are kept in the effects relation order. Undo's are treated similarly as do's and correlated with the effects of their corresponding do's.

5.1 Overview of ABTU

ABTU allows for selective undo of any operation in H. When an undo command is issued by a user (in context of current local history H), we denote the command as $undo(i)$ which means to undo $H[i]$, or the i-th operation in H.

On top of our selective undo algorithm, more intuitive interfaces can be provided for human users to pick the right operations to undo, e.g., as in [1]. For example, the history can be visualized to help users understand the implications of operations; a user is allowed to undo operations in chronological order, either the last local operation or the last operation executed locally; or the user can choose operations to undo by their positions in the document, as in Emacs. It is always possible to translate commands in those approaches into an $undo(i)$ command relative to our history H. In any approach, once the id of the target operation o is known, we

just need to compare the operation ids to locate the right $H[i]$, with overheads no more than $O(|H|)$.

In our algorithm, two threads run at every site, namely, Thread $\mathcal{L}$ and Thread $\mathcal{R}$, which processes local and remote operations, respectively. Any (do/undo) operation generated by a user is first executed at local site and recorded in the local history by Thread $\mathcal{L}$ before it is propagated to remote sites. When a remote operation is received, it is first appended to a receiving buffer RB. Thread $\mathcal{R}$ each time processes one (do/undo) operation o from RB that is causally-ready, i.e., all operations that happen before o have already been executed at current site. Every executed operation is added into H by the effects relation order. There is no extra step or cost to reorder H in our algorithm.

For the purposes of this paper, we assume that all operations are eventually received and processed at every site; and all sites start from the same version of the shared data.

5.2 Notations

Each site is assigned a unique id and maintains a state vector sv with N elements, where N is the number of sites in the system. Each element in sv is an integer that is initialized to zero. As will be shown, at each site, Threads $\mathcal{L}$ and $\mathcal{R}$ advance sv when executing operations, and Thread $\mathcal{L}$ timestamps new operations by sv.

We define two primitives, $ins(p, c)$ and $del(p, c)$, which insert and delete object c at position p, respectively. Each operation o is a nine-tuple *(id, t, p, c, v, dv, tv, ov, uv)* with the following properties: $o.id$ is the id of the site that generates o; $o.t$ is the operation type, either ins or del; $o.p$ is the position at which o is applied; $o.c$ is the object that o inserts or deletes (also called the effect object); the other five properties are **timestamps** explained as follows:

- $o.v$: the vector timestamp(s) of o;
- $o.dv$: timestamps of operations that depend on o;
- $o.tv$: timestamps of operations whose effect objects tie with (or are identical to) $o.c$;
- $o.ov$: if o is an undo, then $o.ov$ is the timestamp of the original operation it undoes, or otherwise $o.ov = \emptyset$;
- $o.uv$: if operation o is undone by some o', then $o.uv = o'.v$, or otherwise $o.uv = \emptyset$.

We use these timestamps to denote the correlation between operations. For example, if o' undoes operation o, then $o.uv = o'.v$ and $o'.ov = o.v$; meanwhile, o' is stored right after o, their effect objects tie, and $o'.tv = o.v$; if o' deletes the object o inserts, $o'.dv = o.v$. If multiple operations delete the same object or they undo the same operation, they will be combined into one operation o in H and $o.v$ is a set of timestamps of those operations.

Besides the two primitives $ins(p, c)$ and $del(p, c)$, we define a special *do-nothing operation* or *identity operation*, denoted as ϕ. The document state remains as-is after executing the operation ϕ on any document state.

Based on our notations, we adapt the established happens-before relation $\rightarrow$ and concurrent relation $\|$ as follows. Given any two operations o_1 and o_2, if $\exists vt_1 \in o_1.v$ and $\exists vt_2 \in o_2.v$ such that $vt_1 < vt_2$, [3] then $o_1 \rightarrow o_2$. If neither $o_1 \rightarrow o_2$ nor $o_2 \rightarrow o_1$, then $o_1 \| o_2$.

Due to [16], $o.p$ is always defined relative to some state called the definition state of o, denoted as $dst(o)$. Let $dst(o) = s$. The state obtained by applying o to state s is denoted as $s \circ o$. Given any two operations o_1 and o_2, if $dst(o_1) = dst(o_2)$, we say that o_1 and o_2 are **contextually equivalent**, denoted as $o_1 \sqcup o_2$; if $dst(o_1) = s$ and $dst(o_2) = s \circ o_1$, we say that o_1 and o_2 are **contextually serialized**, denoted as $o_1 \mapsto o_2$. If two contextually serialized operations are ordered by their *effects relation*, then we say that the two operations are in **effects relation order**.

A list of operations, $[o_1, o_2, \ldots, o_n]$, is a **sequence** if $o_1 \mapsto o_2 \mapsto \ldots \mapsto o_n$. Notation $s \circ L$ denotes the state obtained by applying sequence L to state s, where L is defined in state s or $dst(L)$=s. Given any two sequences L_1 and L_2, where $dst(L_1)$=$dst(L_2)$=s or $L_1 \sqcup L_2$, if $s \circ L_1$=$s \circ L_2$, then L_1 and L_2 are **effects-equivalent**, denoted as $L_1 \cong L_2$.

We use $\overline{o}$ to denote the inverse of operation o. We interpret the semantics of $\overline{o}$ as $o \mapsto \overline{o}$, $\overline{o}.t \neq o.t$ and $\overline{o}.p = o.p$. In the case of $o = ins(p, c)$, then $\overline{o} = del(p, c)$; on the other hand, if $o = del(p, c)$, then $\overline{o} = ins(p, c)$.

5.3 Basic Transformation Functions

In the literature (e.g., [15, 6]), the following three transformation functions, IT, ET and $SWAP$ are often defined.

Inclusion Transformation (IT): Given two operations o_1 and o_2 that satisfy $o_1 \sqcup o_2$, $IT(o_1, o_2)$ transforms o_1 into o_1' such that $o_2 \mapsto o_1'$, incorporating the effect of o_2 into o_1. For example, given two operations $o_1 = del(3, x)$, $o_2 = ins(1, y)$, if o_1 and o_2 are defined in the same document state, then $o_1' = IT(o_1, o_2) = del(4, x)$.

Exclusion Transformation (ET): Given two operation o_1 and o_2 that satisfy $o_1 \mapsto o_2$, $ET(o_1, o_2)$ transforms o_2 into o_2' such that $o_1 \sqcup o_2'$, excluding the effect of o_1 from o_2. For example, after serially executing two operations $o_1 = del(1, x)$ and $o_2 = ins(3, y)$, $o_2' = ET(o_1, o_2) = ins(4, y)$.

Swap Transformatioin ($SWAP$): Given two operations o_1 and o_2 that satisfy $o_1 \mapsto o_2$, $SWAP(o_1, o_2)$ transforms o_1 and o_2 into o_1' and o_2' such that $o_2' \mapsto o_1'$. SWAP exchanges the execution order of two operations while preserving their effects. For example, given two serially executed operations $o_1 = del(1, x)$ and $o_2 = ins(3, y)$, then $o_1' = del(1, x)$ while $o_2' = ins(4, y)$ after $SWAP(o_1, o_2)$.

Conceptually, SWAP(o_1, o_2) is equivalent to first doing $o_2' = \text{ET}(o_1, o_2)$ and then $o_1' = \text{IT}(o_1, o_2')$. As will be shown, we mainly use SWAP and the usage is implicit, i.e., we do not call the SWAP function explicitly. We can do so because we keep operations in the effects relation order. As in the above example for SWAP, for instance, if $o_1 \mapsto o_2$ and $o_1 \prec o_2$, after exchanging their execution order by SWAP(o_1, o_2), o_1 will remain as-is. That is, executing o_2' first does not affect the effect of o_1. Generalizing this observation, we establish the following properties of SWAP without proofs.

P1 Given two operations o_1 and o_2, if $o_1 \mapsto o_2$ and $o_1 \prec o_2$, then o_1 remains as-is after $SWAP(o_1, o_2)$.

P2 Given two operations o_1 and o_2, if $o_1 \mapsto o_2$ and $o_2 \prec o_1$, then o_2 remains as-is after $SWAP(o_1, o_2)$.

P3 Given a non-empty sequence sq whose operations are in effects relation order, for any $i : 0 \leq i < n = |sq|$, $sq[i]$ remains as-is after swapping $sq[i]$ with $sq[i+1] \cdots sq[n-1]$ successively.

P4 Given any operation o and a non-empty sequence sq whose operations are in effects relation order, $|sq| = n$, if $sq[n-1] \prec o$, then sq remains as-is after swapping o with the operations in sq from right to left.

[3] $vt_1 < vt_2$ iff $\forall i : vt_1[i] \leq vt_2[i]$ and $\exists j : vt_1[j] < vt_2[j]$

5.4 Operators for Ordering History

In spirit of the effects relation $\prec$, we introduce three operators $<_e$, $>_e$ and $=_e$ for determining the operation effects relation, which will simplify presentation and proofs of the algorithm. When all operations are ordered in the same history, there are cases in which operations are effects-identical and must be handled differently from previous works [6, 12].

Operator $<_e$: Given any two operations o_1 and o_2, where $o_1 \sqcup o_2$, we say $o_1 <_e o_2$ if one of the following three conditions holds: (1) $o_1.p < o_2.p$; (2) $o_1.p = o_2.p$, $o_1.t = ins$, $o_2.t = del$; or (3) $o_1.p = o_2.p$, $o_1.t = o_2.t = ins$, $o_1.id < o_2.id$. In the case of $o_1 \mapsto o_2$, we say $o_1 <_e o_2$ if one of the following two conditions holds: (1) $o_1.p < o_2.p$; or (2) $o_1.p = o_2.p$, $o_1.t = o_2.t = del$.

Operator $>_e$ is symmetric to operator $>_e$. Given any two operations o_1 and o_2, where $o_1 \sqcup o_2$, we say $o_1 >_e o_2$ if one of the three conditions holds: (1) $o_1.p > o_2.p$; (2) $o_1.p = o_2.p$, $o_1.t = del$, $o_2.t = ins$; or (3) $o_1.p = o_2.p$, $o_1.t = o_2.t = ins$, $o_1.id > o_2.id$. In the case of $o_1 \mapsto o_2$, we say $o_1 >_e o_2$ if one of the two conditions holds: (1) $o_1.p > o_2.p$; or (2) $o_1.p = o_2.p$, $o_1.t = o_2.t = ins$.

Operator $<_e$ determines the precedence relation between the effects of two operations. When $o_1.p < o_2.p$, either $o_1 \sqcup o_2$ or $o_1 \mapsto o_2$, $o_1.c$ must precede $o_2.c$ (or $o_1.c \prec o_2.c$) in any state after o_1 and o_2 are invoked.

When $o_1 \sqcup o_2$, $o_1.t = ins$, $o_2.t = del$ and $o_1.p = o_2.p$, $o_1.c \prec o_2.c$ must hold because $o_2.c$ is an existing object in $dst(o_1)$. For example, let the common definition state be "x", $o_1 = ins(0, y)$, $o_2 = del(0, x)$ and $o_1 \sqcup o_2$. Obviously, after executing o_1, the state is "yx", the effect object 'y' of o_1 precedes that of o_2 ('x').

When $o_1 \sqcup o_2$, $o_1.p = o_2.p$ and both o_1 and o_2 are insertions, they insert new objects that are not present in the definition state. In this case, we impose an order between them by some priority policy. In ABTU, if $o_1.id < o_2.id$, then $o_1.c$ will precede $o_2.c$ after both are invoked.

When $o_1 \mapsto o_2$, $o_1.t = o_2.t = del$ and $o_1.p = o_2.p$, the two operations delete at the same position in a row and hence $o_1.c \prec o_2.c$ must hold. For example, consider state "xy", in which a user first performs $o_1 = del(0, x)$ and then $o_2 = del(0, y)$. The relation $o_1.c \prec o_2.c$ exists even before the two operations are invoked.

Operator $=_e$: Given any two operations o_1 and o_2, we say $o_1 =_e o_2$ if neither $o_1 <_e o_2$ nor $o_2 <_e o_1$. In practice, $o_1 =_e o_2$ if $o_1.p = o_2.p$ and one of the two conditions holds: (1) $o_1 \sqcup o_2$, $o_1.t = o_2.t = del$; or (2) $o_1 \mapsto o_2$, $o_1.t \neq o_2.t$.

Condition (1) means that o_1 and o_2 (concurrently) delete the same object in a common definition state. They are effects-identical and neither $o_1 <_e o_2$ nor $o_2 <_e o_1$.

The condition of $o_1 \mapsto o_2$ and $o_1.t \neq o_2.t$ covers three cases: (1) $o_1.t = ins$, $o_2.t = del$ and both are normal operations. In this case, o_2 deletes the object inserted by o_1. (2) $o_1.t = del$, $o_2.t = ins$ and both are normal operations. In this case, o_2 inserts a object at the same position where o_1 deletes. Unless $o_1.c$ and $o_2.c$ appear in some state simultaneously, their relation is neither $o_1 <_e o_2$ nor $o_2 <_e o_1$. (3) o_2 undoes o_1 or $o_2 = \overline{o_1}$. If $o_1.t = ins$, it is similar to case (1); or if $o_1.t = del$, it is similar to case (2).

Discussions: When any two operations are contextually equivalent or serialized, the above definitions cover all possible cases. Otherwise, we design the algorithm such that any two operations can be transposed to be contextually equivalent or serialized. As a result, for any two operations in the system, one of the relations, $<_e$, $>_e$ or $=_e$, must hold. Therefore, the union of relations, $<_e \cup >_e \cup =_e$, can totally order the set of operations in the system. For any two operations $L[i]$ and $L[j]$ in sequence L, where $i < j$, if $L[i] \leq_e L[j]$ always holds, then we say that L is in ER (Effects Relation) order. This is how we order history H.

5.5 Processing of Local Do/Undo Operations

Algorithm 1 Thread $\mathcal{L}$: process local operation o at site k

```
1: if o is a normal operation then
2:     o′ ← o
3:     sv[k] ← sv[k] + 1
4:     o′.v ← sv
5: else //o = undo(i)
6:     if H[i].uv ≠ ∅ or H[i].dv ≠ ∅ then
7:         exit   //undo request invalid
8:     else
9:         o′ ← overline(H[i])
10:        sv[k] ← sv[k] + 1
11:        o′.v ← sv
12:        o′.ov ← H[i].v
13:        H[i].uv ← o′.v
14:    end if
15: end if
16: execute o′
17: o′′ ← integrateL(o′)
18: propagate o′′
```

In ABTU, any local operation o is either a normal (do) operation or an undo operation. In either case, Thread $\mathcal{L}$ is activated to process the submitted o, as specified in Algorithm 1 in which site k is the local site.

If o is a normal operation, it is directly executed on the local data replica. The element in sv for site k is advanced to indicate that site k has executed a new operation. Then o is timestamped by sv. After being integrated into the local history H, o is propagated to remote sites. Procedure $integrateL(o)$ adds o into H by the effects relation order.

On the other hand, if the submitted o is an undo operation $undo(i)$, we first check its original operation $H[i]$ to see whether it has been undone or is depended on by other operations (line 6). If so, Thread $\mathcal{L}$ will abort this undo request. Otherwise, we generate the inverse of $H[i]$, advance sv, and timestamp the inverse o'. To keep track of the do-undo relationship, $o'.ov$ is set to the timestamp of the original operation (line 12) and the uv property of the original operation $H[i]$ is set to $o'.v$ (line 13). After that, o' is processed as a normal operation, i.e., it is executed locally, integrated into H, and propagated to remote sites.

Note that the inverse o' in our work can be interpreted either as in [9] or as in [10], which are followed in other works as analyzed in Section 3. This flexibility in our work comes from that our history is in the effects relation order.

Using the strategy in [9], we first transpose $H[i]$ to the end of H and then generate the inverse in current state. Because H is ordered by the effects relation, by Property P3, after the transposition, $H[i]$ remains as-is and the resulting inverse as defined in current state is just $\overline{H[i]}$.

Using the strategy in [10], we first generate the inverse $o' = \overline{H[i]}$ that is contextually serialized after $H[i]$. Then o' is considered as concurrent (or more accurately contextually equivalent) with all operations that follow $H[i]$ in H.

We could make o' defined in current state by transforming o' with those concurrent operations. Similarly, however, because H is in effects relation order, those operations do not affect o' during the transformation and consequently o' remains as-is after the transformation. Hence the resulting inverse as defined in current state is also just $\overline{H[i]}$.

5.6 Processing of Remote Do/Undo Operations

Algorithm 2 Thread $\mathcal{R}$: remote operation o from site r

```
1:  if o.ov ≠ ∅ then            // o is undo
2:      find H[i] such that H[i].v ∩ o.ov ≠ ∅
3:      if (H[i].uv ≠ ∅) then
4:          find H[j] such that H[j].v ∩ H[i].uv ≠ ∅
5:          H[j].v ← (H[j].v) ∪ (o.v)
6:          sv[r] ← sv[r] + 1
7:          return
8:      else
9:          H[i].uv ← o.v
10:         o.ov ← o.ov ∪ H[i].v
11:     end if
12: end if
13: o' ← integrateR(o)
14: sv[r] ← sv[r] + 1
15: if o' ≠ φ then execute o'
```

When the system is not busy processing local operations, Thread $\mathcal{R}$ processes one causally-ready (do/undo) remote operation o at a time. If $o.ov \neq \emptyset$, then o is an undo operation. Otherwise, it is a normal (do) operation.

If o is a normal operation (lines 13-15), it will be processed by function *integrateR*(o): First, it transforms o to incorporate the effects of concurrent operations and obtain a resulting version of o that can be safely executed in current state. Second, it adds o to H by the effects relation. After the transformation, if the resulting o' is not an identity operation ϕ, it will be executed in current state.

If o is an undo operation, we first scan H to find the original operation $H[i]$ such that $H[i].v \cap o.ov \neq \emptyset$. Then, we check whether $H[i]$ has been undone by other concurrent operations. If $H[i]$ has already been undone, then we find its undo operation $H[j]$ such that $H[j].v \cap H[i].uv \neq \emptyset$, and combine timestamp $o.v$ into the timestamp $H[j].v$. In this case, o will not be really executed because the original operation can only be undone once. However, the local sv is still advanced to acknowledge the fact that the effect of o has already been reflected in the data replica of this site. On the other hand, if $H[i]$ has not been undone yet, we set $H[i].uv$ to $o.v$ and combine $H[i].v$ into $o.ov$. Then o is transformed and integrated into H similarly as a normal operation.

5.7 Procedure to Integrate Local Operations

Algorithm 3 integrateL(o) is called by Thread $\mathcal{L}$ to add a new local operation o to H by the effects relation order. Note that when a local operation is generated in current state we have $H \mapsto o$. There are two cases to consider:

First, if o is a normal operation, then $o.ov = \emptyset$ must hold. Since $H \mapsto o$, we swap o with the operations in H from right to left until we find an operation $H[i]$ such that $H[i] \leq_e o$. In this process, for every $H[i]$, if $H[i] >_e o$, we incorporate the effect of o into $H[i]$ (lines 5-7). If $H[i] <_e o$ or $H[i] =_e o$, we must insert o right after $H[i]$. We break with current k carrying the target position.

Algorithm 3 *integrateL*(o): o'

```
1:  o' ← o;   k ← |H|
2:  offset ← (o'.t = ins)?1 : −1
3:  if o'.ov = ∅ then        // o is a normal do operation
4:      for (i ← |H| − 1; i ≥ 0; i−−) do
5:          if H[i] >_e o' then
6:              k ← i
7:              H[i].p ← H[i].p + offset
8:          else if H[i] <_e o' then
9:              break
10:         else // H[i] =_e o'
11:             o'.tv ← H[i].v
12:             if o'.t = del then H[i].dv ← o'.v
13:             break
14:         end if
15:     end for
16: else          // o is undo
17:     find H[i] such that H[i].v ∩ o'.ov ≠ ∅
18:     o'.tv ← H[i].v
19:     k ← i + 1
20:     for (j ← k; j < |H|; j++) do
21:         H[j].p ← H[j].p + offset
22:     end for
23: end if
24: insert o' into H at position k
25: return o'
```

If $H[i] =_e o$, the positions of $H[i]$ and o tie, we set $o.tv$ to $H[i].v$ to flag this tie. Moreover, if $o.t = del$, it must be that o deletes the object inserted by $H[i]$. That is, o depends on $H[i]$. Hence we set $H[i].dv$ to $o.v$ to indicate this dependency such that $H[i]$ will not be undone.

Second, if o is an undo operation, we only need to find the target operation $H[i]$ such that $H[i].v \cap o.ov \neq \emptyset$. Then we insert o right after $H[i]$ at target position $k = i+1$ and, since these two operations have the same effect objects, we set $o.tv$ to $H[i].v$ to flag this tie. Furthermore, to accommodate the effect of o, we shift the positions of all the operations in H after the target position k (lines 19-22).

5.8 Procedure to Integrate Remote Operations

Algorithm 4 integrateR(o) is called by Thread $\mathcal{R}$ with dual purposes: the first is to transform o such that its resulting form o' can be executed in current state; the second is to add o at an appropriate position k in H such that the resulting H remains in the effects relation order. Consider two cases:

(1) Case $o.tv = \emptyset$: There is no operation that happens before o and whose effect ties with that of o.

Since o is causally-ready, all operations that happen before o must have been included in H. Conceptually, as in [15, 13], we need to transpose H into two subsequences, sq_h and sq_c, that are contextually serialized, where all operations in sq_h happen before o and all operations in sq_c are concurrent with o. Then $sq_h \mapsto o$ and $o \sqcup sq_c$ must hold. To serve the first purpose, o must be transformed with sq_c. To serve the second purpose of adding o into H by the effects relation order, we also have to examine operations in sq_h.

The fact that H is in the effects relation order could simplify the algorithm. As analyzed in [12], because H is in the effects relation order, by Property P4, the above transposition of H does not shift any operation in sq_c at all. Here we achieve the two purposes without transposing H.

Algorithm 4 *integrateR*(o) : o'

```
1: o' ← o;   k ← |H|
2: if o'.tv = ∅ then
3:     for (i ← 0; i < |H|; i++) do
4:         if H[i] ∥ o' then
5:             offset ← (H[i].t = ins)?1 : −1
6:             if H[i].tv ≠ ∅ or H[i] <_e o' then
7:                 o'.p ← o'.p + offset
8:             else if H[i] >_e o' then
9:                 k ← i; break
10:            else // H[i] =_e o'
11:                o' ← φ
12:                H[i].v ← H[i].v ∪ o'.v
13:                break
14:            end if
15:        else if H[i] >_e o' then      // H[i] → o'
16:            k ← i
17:            break
18:        end if
19:    end for
20: else // o'.tv ≠ ∅, covering the undo case
21:    find H[i] such that H[i].v ∩ o'.tv ≠ ∅
22:    o'.p = H[i].p;   k ← i + 1
23:    while H[k].tv ∩ H[i].v ≠ ∅ do   //H[k] =_e H[i]
24:        if H[k] <_e o' then
25:            o'.p ← o'.p + offset
26:        else if H[k] >_e o' then
27:            break
28:        else // H[k] =_e o'
29:            o' ← φ
30:            H[k].v ← H[k].v ∪ o'.v
31:            break
32:        end if
33:        k ← k + 1
34:    end while
35: end if
36: if o' ≠ φ then
37:    offset ← (o.t = ins)?1 : −1
38:    for (j ← k; j < |H|; j++) do
39:        H[j].p ← H[j].p + offset
40:    end for
41:    insert o' into H at position k
42: end if
43: return o'
```

To explain the ideas, consider a history in ER order, $H = [h_1, c_1, c_2, h_2, h_3, c_3]$, where $h_i \to o$ and $c_j \parallel o$. Suppose that the initial state is s_0 and that o should be inserted between h_2 and h_3. After transposition, the history becomes $H' = [h_1,\ h'_2,\ h'_3,\ c_1,\ c_2,\ c_3]$ and $H' \cong H$. After transforming o with $[c_1,\ c_2,\ c_3]$, let the result be o'. Then $\mathrm{dst}(o') = s_0 \circ [h_1, h'_2, h'_3, c_1, c_2, c_3]$. If now we transpose H' back to H, we have $\mathrm{dst}(o') = s_0 \circ\ [h_1,\ c_1,\ c_2,\ h_2,\ h_3,\ c_3]$ due to $H' \cong H$. That is, we can take $H \mapsto o'$ after transforming o with all concurrent operations. Also because H is ordered, by Property P3, transposing any h_i to the end of H does not shift h_i. However, after the transposition, the relation $h_i \mapsto o$ holds. In other words, we can not only transform o with any c_i but also compare any h_i with o in place!

Back to the algorithm, first consider concurrent operations. For each $H[i] \parallel o$, if $H[i] <_e o$, we incorporate the effect of $H[i]$ into o (line 7). Note the condition $H[i].tv \neq \emptyset$ (line 6), which means that there exists some operation $H[j]$, where $j < i$, that has been examined (i.e., $H[j] <_e o'$) and whose effect ties with $H[i]$. In this case, $H[j] <_e o'$ and $H[j] =_e H[i]$ and hence $H[i] <_e o'$ holds implicitly.

Meanwhile, to serve the second purpose of keeping H in order, similarly to Algorithm 3, we scan H until some $H[i]$ such that $H[i] \geq_e o$. When $o' =_e H[i]$, it must be that o' and $H[i]$ delete the same object; we transform o into an identity operation ϕ and combine $H[i].v$ with $o.v$.

Now consider operations that happen before o. We scan H until operation $H[i] \to o$ such that $H[i] >_e o'$ (lines 15-18), and then add o' at position i. This suffices because we do not need to transform o with operations that happen before it, which means those with effects preceding o are simply skipped; and those with identical effects are handled in the case of $o.tv \neq \emptyset$ (line 20).

(2) Case $o'.tv \neq \emptyset$: There must exist some $H[i]$ such that $H[i]{\to}o$ and $H[i]{=_e}o$, which covers undo operations.

To keep the effects relation order, o is added right after $H[i]$. If we knew that there are no operations that are concurrent with o and whose effects objects tie with that of $H[i]$, we could just set $o.p$ to $H[i].p$ and add o at position $i+1$, as in line 22. Otherwise, we need to transform o with those operations (lines 23-34). Similarly to the steps in lines 3-19, we scan H until some $H[k]$ such that $H[k] \geq_e o'$.

When $H[k]{=_e}H[i]$, if $o'.t{=}ins$, it must be that o' and $H[k]$ concurrently insert at the same position where $H[i]$ deletes, we order $H[k]$ and o' by their site ids. If $o'.t{=}del$, it must be that $H[k]$ and o' concurrently delete the same effect object; o' is combined into $H[k]$ and an identity operation ϕ is returned, which is not added to H nor executed.

6. ANALYSES AND PROOFS

We first show how well-known undo puzzles are solved in ABTU, which also serve as examples. Then we will present formal proofs with regard to conditions formalized in Section 4. After that, we will analyze the complexities of ABTU.

6.1 Undo Puzzles

We consider four representative undo puzzles given in [14]:

Case 1: Given state "a", first execute $o_1 = del(0,a)$ and $o_2 = ins(0,b)$, where $o_1 \to o_2$. Then undo o_1.

After executing o_1 and o_2, the history is $H = [o_1,\ o_2] = [del(0,a),\ ins(0,b)]$, where $o_1 =_e o_2$. The inverse of o_1 is $\overline{o_1} = ins(0,a)$. After executing $\overline{o_1}$, the state is "ab". The history becomes $[del(0,a),\ ins(0,a),\ ins(1,b)]$.

Case 2: Given state "a", invoke two concurrent operations $o_1 = del(0,a)$ and $o_2 = ins(0,b)$. Then undo o_1.

Since $o_1 \parallel o_2$ and $o_1 \sqcup o_2$, $o_2 <_e o_1$ holds and the effect object of o_2 should precede that of o_1 in the final result. After executing o_1 and o_2, the history is $H = [o_2, o'_1] = [ins(0,b), del(1,a)]$. Undo o'_1 is to execute its inverse $ins(1,a)$, resulting in state "ba".

Case 3: Given state "ab", first execute $o_1 = del(0,a)$ and $o_2 = del(0,b)$, where $o_1 \to o_2$. Then undo o_1 and o_2 concurrently, i.e., undo$(o_1) \parallel$ undo(o_2).

The history is $[del(0,a), del(0,b)]$ after executing o_1 and o_2. There are two cases depending on the undo order.

1) Undo(o_1) is executed before undo(o_2): Executing the inverse of o_1 yields state "a" and history $[del(0,a),\ ins(0,a),\ del(1,b)]$. Now undo o_2 by executing its inverse operation $ins(1,b)$. The state becomes "ab" and the history $[del(0,a),\ ins(0,a),\ del(1,b),\ ins(1,b)]$.

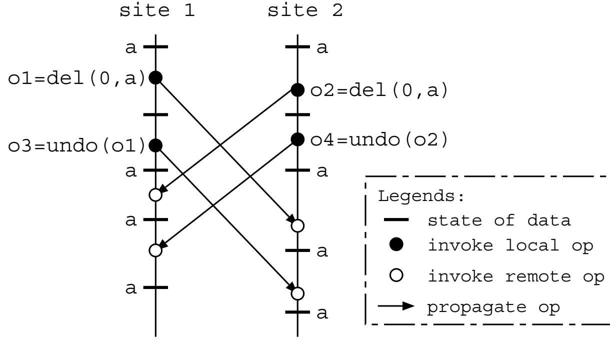

Figure 3: An undo puzzle scenario for Case 4.

2) Undo(o_2) is executed before undo(o_1): After executing the inverse of o_2, $ins(0, b)$, we get state "b" and history [$del(0, a)$, $del(0, b)$, $ins(0, b)$]. After executing the inverse of o_1, $ins(0, a)$, we get state "ab" and history [$del(0, a)$, $ins(0, a)$, $del(1, b)$, $ins(1, b)$].

Case 4: Given state "a", first invoke two concurrent operations $o_1 = del(0, a)$ and $o_2 = del(0, a)$ with the same effect. Then undo o_1 and/or o_2.

A scenario of this case is illustrated in Figure 3. Take site 1 for example: After executing o_1 and o_3, the state becomes "a" and history $H = [o_1 : del(0, a), o_3 : ins(0, a)]$. After receiving o_2, it is transformed into a ϕ operation. The state remains "a" and H becomes $[o_1/o_2 : del(0, a), o_3 : ins(0, a)]$. When o_4 arrives, we find that its original operation o_2 has already been undone. The state remains unchanged and H=$[o_1/o_2 : del(0, a), o_3/o_4 : ins(0, a)]$. The execution process at site 2 is similar and omitted. Therefore, undo o_1 and/or o_2 yields a unique result "a" in our approach.

Note that we interpret undo effects not exactly the same as in [14, 10]. In case (4), they take o_1 and o_2 as two *different* operations, and the character 'a' will be restored only if o_1 and o_2 are both undone. However, o_1 and o_2 are interpreted as the *same* operation in our correctness model and combined during the transformation. Hence 'a' will be restored as long as o_1 and/or o_2 are undone.

6.2 Correctness Proofs

THEOREM 6.1. *ABTU satisfies Causality Preservation.*

PROOF. In ABTU, vector timestamps are used such that any remote operation o can be invoked by Thread $\mathcal{R}$ only if it is causally ready, that is, all operations that happen before o have been invoked. □

LEMMA 6.2. *Given a history H that is in ER order and any local operation o that is generated in the context of H, suppose the history becomes H' after Thread $\mathcal{L}$ invokes o. Then $H' \cong H \cdot o$ and H' is in ER order.*

PROOF. First consider the case in which o is a normal operation. Algorithm 3 adds o into H by the effects relation order, say at position k. By the algorithm, o is always inserted such that $\forall i < k : H[i] \leq_e o$ and $\forall j \geq k : H[j] >_e o$. Hence, the resulting history H' is in ER order.

Meanwhile, o is transposed with every operation in H from right to left until the target position k. By the algorithm, $o <_e H[j]$ holds for every $H[j]$ that is transposed with o. By Property P2, the position of o remains as-is while the position of $H[j]$ is shifted. Because $H \mapsto o$, each transposition step is effects-equivalent. Hence, after the integration, $H' \cong H \cdot o$.

Next consider the case in which o is to undo operation $H[i]$. Algorithm 3 directly adds o right after $H[i]$ and $H[i] =_e o$. On one hand, if $H[i].t = ins$, $H[i + 1] >_e o$ must hold. Otherwise, it must be $H[i + 1] =_e H[i]$ and, by definition of $=_e$, $H[i + 1].t = del$ and $H[i + 1].p = H[i].p$, which means that $H[i]$ has already been undone explicitly or implicitly. Then o would have aborted.

On the other hand, if $H[i].t = del$, $H[i+1] >_e o$ also must hold. Otherwise, if $H[i + 1] =_e H[i]$, then $H[i + 1].t = ins$ must hold. If $H[i + 1]$ were to undo $H[i]$, then o would have aborted. If $H[i + 1]$ is just a normal insert operation, $H \mapsto o$ and, by Property P3, $H[i + 1]$ remains as-is if it is transposed to the end of H. Then, the three conditions $H[i + 1] \mapsto o$, $H[i + 1].p = o.p$ and $H[i + 1].t = o.t = ins$ imply $H[i + 1] >_e o$ by definition of $>_e$.

Therefore, when an undo operation o is integrated at position k, $\forall i < k : H[i] \leq_e o$ and $\forall j \geq k : H[j] >_e o$. Furthermore, Algorithm 3 shifts the positions of those operations after o in H to incorporate the effect of o. Hence, after the integration, $H' \cong H \cdot o$. □

LEMMA 6.3. *Given a history H that is in ER order and a remote operation o, suppose that the history is H' after Thread $\mathcal{R}$ integrates o into H and transforms o into o'. Then $H' \cong H \cdot o'$ and H' is in ER order.*

PROOF. Consider the following three cases:

1) $o.tv = \emptyset$ and $\exists H[i] : H[i] =_e o$. It must be that o and $H[i]$ concurrently delete the same object, and o will be transformed into an identity operation ϕ by Algorithm 4. In this case, o is combined into $H[i]$. The resulting H' remains in ER order and $H' \cong H \cdot o'$ because $o' = \phi$.

2) $o.tv = \emptyset$ and $\forall H[i]$: either $H[i] <_e o$ or $H[i] >_e o$. In this case, o will be inserted at some position k such that $\forall i < k : H[i] <_e o$ and $\forall j \geq k : H[j] >_e o$. Hence the resulting H' is in ER order. Furthermore, all effects of the concurrent operations whose effect objects precede that of o have been included into o' during the integration (line 7 of Algorithm 4). The operations after the insert position k have included the effect of o after the integration (lines 37-40 of Algorithm 4). Hence, $H' \cong H \cdot o'$.

3) If $o.tv \neq \emptyset$, there must be one $H[i]$ that happens before o and $H[i] =_e o$. Consider the following two cases:

(3.a) If $H[i + 1] \neq_e H[i]$, it must be $H[i+1] >_e o$. In this case, o will between $H[i]$ and $H[i + 1]$ with $o'.p = H[i].p$. Hence, $\forall k \leq i : H[k] \leq_e o$ and $\forall j > i : H[j] >_e o$. Moreover, the effect of o will be included into the operations that follow o in H after the integration. Therefore, H' is in ER order and $H' \cong H \cdot o'$.

(3.b) There exists some operation whose effect object ties with that of $H[i]$. On one hand, if $o.t = del$, o will be transformed into ϕ and, similarly to case 1), after the integration, H' is in ER order and $H' \cong H \cdot o'$. On the other hand, if $o.t = ins$, o will be integrated by some tie-breaking policy. And the effect of o will be included into the operations that follow o in H. Hence, after the integration, H' is in ER order and $H' \cong H \cdot o'$. □

THEOREM 6.4. *The invocation of any operation in ABTU satisfies Admissibility Preservation.*

PROOF. According to Lemmas 6.2 and 6.3, given any history H that is in ER order and any operation o, the integration of o into H does not violate the ER order that is already established between operations in H. That is, the invocation of any o is admissibility preserving. □

After all generated operations are invoked in the system, the histories at all sites will consist of the same set of operations that are in the same ER order. Note that this does not contradict the fact that operations are allowed to be invoked in arbitrary orders at different sites, as long as the causality preservation condition is observed. Applying these histories to the same initial state produces the same final state. Without loss of generality, assume that the initial state is empty and all objects are produced by user operations. Then all objects in the final state must be in the same ER order.

COROLLARY 6.5. *After all generated operations are invoked at all sites, all the data replicas converge in the same final state in which objects are in the ER order.*

6.3 Complexities

Since we do not need to save extra information other than the history H, the space complexity of ABTU is $O(|H|)$. The processing of any local operation in Thread $\mathcal{L}$ (and Algorithm 3) takes time $O(|H|)$ because it only needs to scan the history once. The processing of any remote operation in Thread $\mathcal{R}$ (and Algorithm 4) takes time $O(|H|)$ because it can be done by scanning H only once.

Note that the current algorithm presentation is for conceptual clarity and our actual implementation is more efficient. For example, in line 4 of Algorithm 2, it seemingly requires to scan H. However, $H[j]$ must be following $H[i]$ because $H[j] =_e H[i]$. Hence it only needs to scan from $H[i+1]$. That is, Algorithm 2 only needs to scan H once.

7. CONCLUSION

This paper presents a novel approach to OT-based selective undo for distributed collaborative applications. The proposed algorithm ABTU achieves $O(|H|)$ space and time complexities for both do and undo under a unified framework, which drastically improves the latest result [17] that is exponential-time in undo. Moreover, the correctness of ABTU is formally proved with regard to two formalized conditions, causality and admissibility preservation. Due to simplicity of the effects relation based approach, both the algorithm and the proofs are simple.

In the journal version of this paper, we will prove ABTU with regard to transformation properties such as TP1/TP2 [9, 11] as well as IP1/IP2/IP3 [9, 14]. In future research, we will focus on studying user interface and usability issues of selective undo in context of specific applications. Our ongoing work is further optimizing the presented algorithm, which is fully replicated, for the Web (client/server) architecture to support Google Wave like Web 2.0 services.

8. REFERENCES

[1] Thomas Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM ToCHI*, 1(3):269–294, 1994.

[2] Rajiv Choudhary and Prasun Dewan. A general multi-user undo/redo model. In *ECSCW*, pages 231–246, 1995.

[3] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD*, pages 399–407, 1989.

[4] Jean Ferrié, Nicolas Vidot, and Michèle Cart. Concurrent undo operations in collaborative environments using operational transformation. In *CoopIS*, pages 155–173, 2004.

[5] Du Li and Rui Li. Preserving operation effects relation in group editors. In *ACM CSCW*, pages 457–466, 2004.

[6] Du Li and Rui Li. An admissibility-based operational transformation framework for collaborative editing systems. *J. CSCW*, 19(1):1–43, 2010.

[7] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *CollaborateCom*, pages 10–20, November 2006.

[8] Géald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Proving Correctness of Transformation Functions in Collaborative Editing Systems. Research Report RR-5795, LORIA – INRIA Lorraine, December 2005.

[9] Atul Prakash and Michael J. Knister. A framework for undoing actions in collaborative systems. *ACM ToCHI*, 1(4):295–330, 1994.

[10] Matthias Ressel and Rul Gunzenhäuser. Reducing the problems of group undo. In *ACM GROUP*, pages 131–139, 1999.

[11] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *ACM CSCW*, pages 288–297, 1996.

[12] Bin Shao, Du Li, and Ning Gu. A sequence transformation algorithm for supporting cooperative work on mobile devices. In *ACM CSCW*, Savannah, GA, USA, 2010.

[13] Maher Suleiman, Michèle Cart, and Jean Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *ACM GROUP*, pages 435–445, 1997.

[14] Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM ToCHI*, 9(4):309–361, 2002.

[15] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *ACM CSCW*, pages 59–68, 1998.

[16] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM ToCHI*, 5(1):63–108, 1998.

[17] David Sun and Chengzheng Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, 2009.

[18] Stéphane Weiss, Pascal Urso, and Pascal Molli. An undo framework for p2p collaborative editing. In *CollaborateCom*, pages 529–544, November 2008.