OPERATION TRANSFORMATION BASED CONCURRENCY CONTROL IN GROUP EDITORS

A Dissertation

by

RUI LI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2006

Major Subject: Computer Science

OPERATION TRANSFORMATION BASED CONCURRENCY CONTROL IN GROUP EDITORS

A Dissertation

by

RUI LI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Du Li
Committee Members,	Jianer Chen
	Jennifer L. Welch
	Ronald Zellner
Head of Department,	Valerie E. Taylor

August 2006

Major Subject: Computer Science

ABSTRACT

Operation Transformation Based Concurrency Control in

Group Editors . (August 2006)

Rui Li, B.E., Beijing University of Aeronautics & Astronautics;

M.E., Beijing University of Aeronautics & Astronautics;

M.E., Johns Hopkins University

Chair of Advisory Committee: Dr. Du Li

Collaborative editing systems (or group editors) allow a geographically dispersed group of human users to view and modify shared multimedia documents, such as research papers, design diagrams, web pages and source code together over a computer network. In addition to being useful tools, group editors are a classic research vehicle and model of interactive groupware applications, based on which a variety of social and technical issues have been investigated.

Consistency maintenance as a fundamental problem in group editors has attracted constant research attention. Operational transformation (OT) is an optimistic consistency maintenance method that supports unconstrained collaboration among human users. Although significant progress has been achieved over the past decade, there is still a large space for improvement on the theoretical part of OT. In this dissertation, we are concerned with three problems: (1) How to evaluate the correctness of OT-based consistency maintenance protocols; (2) How to design and prove correct OT-based protocols; (3) What are the consistency correctness conditions for group editing systems in general.

This dissertation addresses the above three problems and makes the following contributions: (1) propose a total order based framework including a new consistency model and the associated design methodology. This framework reduces the complexities of the OT design; (2) improve the total order based framework by introducing a natural order based framework. In contrast, this framework removes the requirement of defining a total order that is not necessary to the OT design; (3) establish a generic consistency model and propose the first set of practical design guidelines in OT based on this model.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. Du Li. Without his inspiration, guidance and encouragement, this dissertation would not be possible. He led me into the exciting field of groupware and distributed computing and taught me how to approach problems in a rigorous way. The methodology and philosophy that I learned in my research will definitely benefit my career for life.

I want to express my gratitude to the members of my advisory committee: Dr. Jianer Chen, Dr. Jennifer L. Welch and Dr. Ronald Zellner for their valuable comments and earnest help.

I also thank the fellow students in my research group: Yi Yang, and Jiajun Lu for their collaborations.

Finally, I would like to thank my wife, Lan Liang, for her understanding, support, and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Collaborative editing systems (or group editors) allow a geographically dispersed group of human users to view and modify shared multimedia documents such as research papers, design diagrams, web pages and source code together over a computer network. In addition to being useful tools, they are a classic research vehicle and model of interactive groupware applications, based on which a variety of social and technical issues have been investigated [1, 2, 3, 4, 5].

Among the technical issues, consistency maintenance has been attracting significant and continuous research attentions. Differently from in traditional distributed systems (e.g., distributed file systems and distributed database systems), however, consistency maintenance in collaborative editing systems must consider human factors. For example, it has been well understood that the following requirements must be satisfied: (1) high local responsiveness: the system must be as responsive as its familiar single-user tools [6, 7]; (2) high concurrency: the users must be able to concurrently and freely edit any part of the shared document at any time [8, 9]; (3) realtime awareness: the users must be able to see each other's modifications in a timely manner as a basis for resolving conflicts and coordinating local activities [6].

In pursuing consistency maintenance methods that address these requirements, operational transformation (OT) has been well established over the past decade due to their advantages over traditional methods such as locking and timestamping in interactive applications [10, 5, 11]. Conceptually, local editing operations are always executed immediately after they are generated. Remote operations are transformed

The journal model is *IEEE Transactions on Automatic Control.*

before execution to repair inconsistencies such that concurrent operations can be executed in any order at different sites. Hence local response time is not sensitive to nondeterministic communication latencies. The users are allowed to modify any part of the shared data without being constrained or serialized. OT-based protocols do not need to delay the propagation of operations and hence the users can be kept continuously aware of remote edits at best effort. As a result, OT-based group editors are a flexible collaboration medium on which the users are free to exercise different writing protocols while avoiding problems of imposing premature and rigid control [12, 13, 14, 15].

Despite the significant progress over the past decade, however, there is still a large space for improvement on the theoretical part of OT. In this dissertation, we are interested in three fundamental problems. First, due to the accepted consistency model of Sun et al. [5], three conditions must be verified, namely, convergence, causality and intention preservation (CCI), before claims can be made on the correctness of an OT algorithm. Unfortunately, a rigorous definition of intention preservation is still absent due to difficulties in modeling and formalizing user or operation intentions. Due to this problem, most existing OT algorithms have not been formally proved and counterexamples are often identified [16, 17, 11, 5, 18].

The second problem is the lack of a practicable methodology for guiding the design of OT algorithms. Most existing works [17, 5, 11, 18] theoretically require that transformation functions verify two well-known transformation properties established in [16]. However, these two properties have turned out extremely difficult to satisfy and prove even for transformation functions that only handle two primitive characterwise insert and delete operations [19]. It remains an open issue whether or not these properties can be verified or should be verified at all on new transformation functions [20, 21, 5], especially those with advanced operations, which are not considered in [16].

Thirdly, considering rich application semantics, researchers have proposed a wide variety of application-specific protocols, some of which are not OT-based [22, 23, 24]. Designing application-specific protocols necessarily involves system modeling. Unfortunately, all existing consistency models are tightly bound to specific application abstractions that are not compatible with each other. OT-based models are of little value for design of non-OT-based algorithms. Our goal is to establish a generic model by exploring the commonalities of a variety of group editing systems and develop general consistency conditions independent of specific protocols.

Addressing the above three problems, this dissertation is divided into three parts. First, we propose a consistency model based on a total order of characters and an associated design methodology. Secondly, we propose an alternative approach that is based on a partial order of characters. Finally, we abstract the consistency maintenance module as an I/O machine and establish a generic consistency model by studying related properties of the I/O machine.

The rest of the dissertation is organized as follows: Chapter II briefly introduces the background of operational transformation. Chapter III presents a total order based framework of designing OT-based algorithms. Chapter IV presents a natural order based framework of designing OT-based algorithms. Chapter V presents a generic consistency model for group editing systems and an in-depth analysis of the correctness conditions for developing OT protocols. Chapter VI surveys related work. Chapter VII summarizes contributions of this dissertation and points out possible future research directions.

CHAPTER II

BACKGROUND

A. Key Concepts and Notations

Definitions in this chapter mostly follow the established conventions in group editors [10, 16, 25, 11, 5]. For availability and responsiveness reasons, the shared document of a group editor is often replicated so that users can work locally. We model the shared data as a string of characters (objects) and define the following two primitive operations:

- $\mathrm{ins}(p, c)$: insert character c at position p.
- $\mathrm{del}(p, c)$: delete character c at position p.

Apparently the position parameter p of any operation o is relative to some *definition state* s, denoted as $s = \mathrm{dst}(o)$. We often annotate o with its *definition state* s like o^s. The definition state s is the *generation state* of o if o is generated in s, or the *execution state* of o if o is to be executed in s. The fact that the execution of operation o^s in state s yields state s' is denoted as $s' = \mathrm{exec}(s,o^s)$. Due to concurrency, the execution state of o may not always be the same as its generation state, e.g., when o is executed at remote sites.

Let the position of the first character in a string be zero. We assume that every appearance of any character has a different object id. In the rest of the present dissertation, a “character” refers to the object that carries the character, the ASCII code of which is possibly only one of its attributes. Note this assumption only serves analysis purposes and we do not really need object ids for characters in actual implementation. Notation $s[i]$ returns the character at position i in string s and $s[c]$ returns the position of character c in s.

Each site j maintains a state vector sv_j in the form of $< x_1, x_2, ..., x_N >$, where N is the number of sites. The i^{th} element x_i $(i = 1, 2, ..., N)$, denoted as $sv_j[i]$, is site j's knowledge of the number of operations executed at site i. All state vectors are initialized such that each element is zero. Each time a local operation o is generated at site j, it is executed immediately and $sv_j[j]$ is incremented by one. Then o is timestamped by sv_j and broadcast to remote sites. Each time site j executes a remote operation generated at site i, element $sv_j[i]$ is incremented by one.

Synchronization in group editors is achieved by exchanging messages that piggyback operations. For any operation o, attribute $o.t$ is its type (ins/del), $o.c$ is the effect character to be inserted or deleted, $o.p$ is its position parameter, $o.v$ is its vector timestamp, and $o.id$ is the id of the site that generates o. Note only $o.p$ is relative to dst(o), while other attributes will never be changed after o is generated.

The vector timestamps are used to determine the happened-before and concurrent relations between operations. Following the long established conventions in group editors [10][5].

Given two state vectors, v_1 and v_2, we say $v_1 = v_2$, iff $\forall i : v_1[i] = v_2[i]$. We say that $v_1 < v_2$, iff $v_1 \neq v_2$ and $\forall i : v_1[i] \leq v_2[i]$. We say operation o_1 happened before o_2, denoted by $o_1 \rightarrow o_2$, iff $o_1.v < o_2.v$. We say o_1 is concurrent with o_2, denoted by $o_1 \parallel o_2$, iff neither $o_1 \rightarrow o_2$ nor $o_2 \rightarrow o_1$.

B. Operational Transformation

The basic idea of OT is to execute any local operation as soon as it is generated for high local responsiveness. Remote operations are *transformed* against concurrent operations that have been executed locally before its execution. A history buffer HB is maintained at each site to keep track of all executed operations in their order of execution.

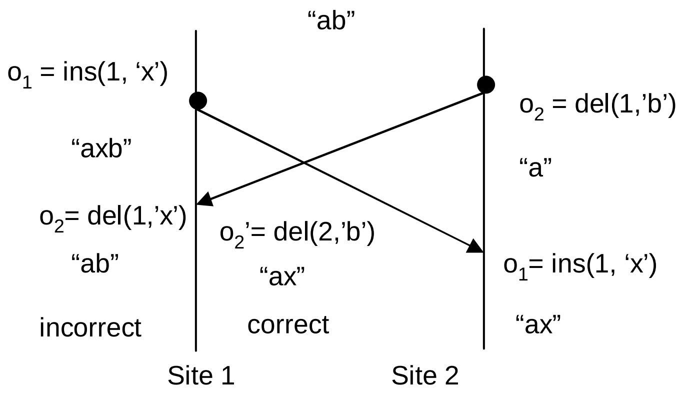

Fig. 1. OT transforms o_2 and then executes o_2'.

As a simple example, consider the scenario in Figure 1. Suppose two sites start from the same initial state s_1^0=s_2^0="ab". Site 1 performs o_1=ins(1,'x') to insert character 'x' before 'b', yielding s_1^1=exec(s_1^0, o_1)="axb", while site 2 concurrently performs o_2=del(1) to delete character 'b', yielding s_2^1=exec(s_2^0, o_2)="a". When o_2=del(1,'b') arrives at site 1, if it is executed as it is, then the wrong character 'x' will be deleted. This is because o_2 is generated in s_2^0 without the knowledge of o_1, but its execution state s_1^1 has been changed by the execution of o_1, which invalidates its position parameter. The intuition of OT [10] is to shift the position of o_2 to *incorporate* the effect of o_1 such that the result o_2' can be correctly executed in state s_1^1. This process is called *inclusion transformation* (IT) [5].

Because a character has been inserted by o_1 on the left of its intended position, o_2 should delete the character currently at position 2 instead of 1, i.e., $o_2' = \text{IT}(o_2, o_1)$ = del(2,'b'). The execution of o_2' in s_1^1 leads to the correct state s_1^2="ax", which is identical to the final state at site 2 after o_2 and o_1 are executed in a tandem. As a result, OT seems able to achieve convergence and preserve intentions of operations despite the different orders of execution at different sites.

Another type of transformation function is called *exclusion transformation* (ET) [5]. In the above example, given o_1 defined in s_1^0 and o_2' defined in $s_1^1 = \text{exec}(s_1^0, o_1)$, $\text{ET}(o_2', o_1)$ excludes the effect of o_1 from o_2' as if o_1 had not been executed in s_1^0. The result $o_2 = \text{ET}(o_2', o_1) = \text{del}(1, \text{‘b’})$ is exactly the execution form of o_2 as defined relative to s_1^0.

It has been generally accepted [11] that each OT algorithm consists of two parts: a set of *transformation functions* (such as IT and ET) that determine how one operation is transformed against another, and a *control procedure* that determines how an operation is transformed against a given operation sequence (e.g., the history buffer). The control procedure is also responsible for generating and propagating local operations as well as executing remote operations.

C. Convergence in Group Editors

Most existing consistency control methods, e.g., [26, 27, 28, 29], are motivated in traditional non-interactive applications in which user interface effects are not interesting. Hence convergence often suffices and can be achieved by enforcing a total order of writes. However, the example in Fig. 1 reveals that serialization may violate the operation intentions, e.g., when o_2 is executed after o_1 as it is.

The general assumption underlying interactive groupware applications such as group editors is that users are aware of the changes made by collaborators and are able to discover and resolve semantic conflicts (e.g., grammatic errors) in a timely manner [10, 6, 5]. Hence OT algorithms generally do not consider reads. Writes are often distinguished as two primitive operations, insert and delete, for creating new objects and removing existing objects. This distinction of writes makes it possible to achieve a high level of concurrency while reducing the chance of operation conflicts and overwriting.

The main purpose of Internet-based productivity applications such as group editors is to promote the productivity of human users as a group. It is accepted in this context that high local responsiveness and high concurrency are conducive to individual and group productivity [10, 5]. Traditional consistency control methods such as locking and serialization [27] generally sacrifice responsiveness and concurrency when they are pessimistic, and may cause the loss of interaction results when optimistic. Operational transformation seems a promising consistency control method that is able to achieve high responsiveness, high concurrency and convergence while preserving interaction results [5].

Preserving interaction results of all collaborators does not necessarily lead to a chaotic system. The results can be presented to users, e.g., in different colors and multiple versions [30], which is essentially a user interface problem and beyond the scope of this dissertation. However, the system should do as much work as it can to reduce cognitive overheads of users. That is, the data replicas must eventually converge (after all generated operations are executed at all sites) and the converged final state must be somehow constrained, e.g., by requiring that it preserve causality and operation intentions as in [5].

CHAPTER III

A TOTAL ORDER BASED FRAMEWORK

A. Motivation

Group editors are a classic model and research vehicle for distributed interactive groupware applications because they typically manipulate shared data in a coordinated manner [2, 10]. Operational transformation (OT) has been well accepted in group editors for achieving optimistic consistency control [10, 11]. OT allows local operations to execute in a nonblocking manner to achieve high local responsiveness and unconstrained collaboration. Remote operations are *transformed* before they are executed such that inconsistencies are repaired.

Despite the significant progress that has been achieved over the past 15 years, a notable fact in the history of OT is that the discovery and solution of various OT puzzles (i.e., correctness problems in previous OT algorithms) have been a main driver of research [14, 11, 31]. In our opinion, however, the existence of OT puzzles can be largely attributed to the lack of a suitable theoretical framework for guiding the design and verification of OT algorithms. More specifically, the well-established frameworks [16, 25, 5, 31] rely on conditions that are difficult to verify in practice and do not address how to develop correct OT algorithms.

In this chapter, we propose a novel OT framework to address this weaknesses of previous work. Based on a concept called “operation effects relation”, we define two criteria, causality preservation and operation effects relation preservation, for verifying the correctness of OT algorithms. Our framework comes with a practicable approach to developing and proving OT algorithms. In this approach, the sufficient conditions for transformation functions are first identified, and a particular transformation path is chosen to satisfy those conditions and thus the correctness of the whole algorithm.

A framework consists of a theory, which defines a consistency model of the system, and a technical approach, which guides the development of algorithms that fulfill the model. Two OT frameworks have been proposed in the literature:

The **first** OT framework is established in Ressel et al. [16]. In the theory part, it formalizes two consistency criteria, **c**ausality preservation and **c**onvergence (**CC**). In the approach, Ressel et al. [16] proves that any OT-based algorithm can achieve convergence in the presence of arbitrary transformation paths if its IT function can satisfy two transformation properties, TP1 and TP2, as defined below.

Definition 1 *Given three operations o_1, o_2 and o_3 that are defined in the same s, let $o_1' = IT(o_1, o_2)$ and $o_2' = IT(o_2, o_1)$,*

TP1: exec(s, $[o_1, o_2']$) = exec(s, $[o_2, o_1']$)

TP2: $IT(IT(o_3, o_1), o_2') = IT(IT(o_3, o_2), o_1')$

TP1 ensures that, if $o_1 \parallel o_2$, the effect of executing o_1 before o_2 is the same as executing o_2 before o_1. TP2 ensures that transforming any operation o_3 along different paths, $[o_1, o_2']$ and $[o_2, o_1']$, will yield the same result. These two properties ensure that arbitrary transformation paths can lead to a consistent final state. In other words, it is not necessary to enforce a global total order between operations in order to achieve convergence, because divergence can always be repaired with operational transformation.

The CC framework has influenced most of the OT algorithms developed since it was proposed in 1996, including adOPTed [16], GOT [5], GOTO [11, 31, 32], NICE [18], SOCT [25, 33], TIBOT [34], and SDT [19, 14], as they more or less struggle on the TP2 condition. Influential as it is, this framework is flawed in the following ways. First, the consistency model does not explicitly constrain convergence. Hence it is *theoretically* acceptable in the model that the system always converges in an empty state no matter what the users do. Second, in the technical approach part, TP1 and TP2 are only for constraining IT. As revealed later [25, 31, 11, 5], ET is conceptually an inverse of IT and indispensable for implementing group undo and building transformation paths. However, the CC framework does not constrain ET. Additionally, it does not provide guidelines on how to develop IT functions that satisfy TP2. In fact, TP2 had never been verified until in our recent work [19, 14, 35].

The **second** framework is established in Sun et al. [5] with three consistency criteria: **c**ausality preservation, **c**onvergence, and **i**ntention preservation (**CCI**). The intention preservation condition is the first attempt in the literature to explicitly constrain convergence in interactive groupware. Due to its intuitiveness, this model has been well-accepted in group editors, such as [36, 32, 25, 37, 33]. However, Sun et al. did not clearly formalize exactly what is intention-preserving and what is not in the original work [5] and its main follow-up work [31]. Hence it is difficult to evaluate the correctness of OT algorithms under the CCI framework.

Example 1 *A well-known controversial scenario follows: Suppose three sites start from the same initial state "abc". Concurrently, site 1 performs o_1=ins(2,'x') to insert 'x' between 'b' and 'c', site 2 performs o_2=ins(1,'y') to insert 'y' between 'a' and 'b', and site 3 performs o_3=del(1,'b') to delete 'b'. Intuitively the only correct final result must be "ayxc". However, result "axyc" is referred to as intention-preserving in [5].*

In the technical approach part of [31], the CCI framework inherits the TP1 and TP2 constraints on IT, while still not providing guidelines for developing such IT functions. It also defines several constraints that relate IT and ET, e.g., reversibility and transpose properties. However, ET is treated as a second-class citizen and no standalone constraints are defined. As an important contribution, Sun et al. [5, 31] define the precondition of IT(o_1, o_2) as est(o_1)=est(o_2) and the precondition of ET(o_1, o_2) as est(o_1) = exec(est(o_2),o_2). However, these two conditions are not really sufficient, as will be illustrated in Examples 3 and 4.

B. The Consistency Model

As revealed in Example 1, the main problem of the established CCI model [31, 5] is that it only considers the "intentions" or effects of single operations in their generation states while not capturing a global picture of the whole system. Specifically, if a global picture were present, we would know how to correctly determine the relation between their effect characters, 'x' and 'y', when transforming the two operations, o_1 and o_2.

We address this problem by introducing a new concept of *effects relation* $\prec$. Intuitively relation $\prec$ denotes the order between characters. In Example 1, we have 'b' $\prec$ 'x' $\prec$ 'c' when o_1 is generated, and 'a' $\prec$ 'y' $\prec$ 'b' when o_2 is generated. Then by transitivity we infer 'y' $\prec$ 'x', based on which we can always correctly transform o_1 and o_2. In this way we can model both the effects of single operations and the relation between effects of concurrent operations.

Our early result as presented in [14, 35] introduced but did not rigorously formalize the concept of effects relation. Additionally, it was developed under the influence of CCI and still strived to satisfy TP1 and TP2. Nevertheless, the TP2 proofs in [35] are very complicated although they only treat two characterwise primitives. It is still an open problem how to scale the proofs to more sophisticated operations such as string operations [5]. Related works in [34, 18, 5, 33] free TP2 by enforcing a unique transformation path. However, they are developed under the CCI framework and must verify the condition of intention preservation as required in CCI.

The purposes of this chapter are two-fold: First, we will present a rigorous definition of the effects relation. Second, we will explore a more practicable approach that no longer requires the verification of TP1 and TP2.

1. Effects Relation

Suppose there is an observer to the system (group editor) who observes the progress (every operation execution) at each site. A virtual global data structure Ω is maintained to indicate the effects relation $\prec$. Ω is initialized by the initial system state s^0 and incrementally updated by the execution of operations. Before we define rules for initializing and updating Ω, we study the order in which Ω is updated.

Here the same operation may be invoked to update Ω multiple times because it is executed once at every site in the system. Hence we have to examine these invocations at the event level. An allowed updating path is a sequence of operations (or invocations), $o_1, o_2, ..., o_n$, in which, for any two operations o_i and o_j, o_i precedes (or is invoked before) o_j, $1 \leq i < j \leq n$, iff one of the following conditions holds: (1) o_i is executed earlier than o_j at the same site, (2) o_i is the local execution and o_j is a remote execution of the same operation, or (3) there exists another execution o_k such that o_i precedes o_k and o_k precedes o_j.

We annotate Ω with a superscript to denote its incremental construction process. Ω^0 contains the initial effects relation: for any two characters $c_i, c_j \in s^0$, we add pair $< c_i, c_j >$ or $c_i \prec c_j$ into Ω^0 iff $s^0[c_i] < s^0[c_j]$. Given any updating path P $= [o_1, o_2, ..., o_n]$, the execution of o_i updates Ω^{i-1} to Ω^i, where $1 \leq i \leq n$. For convenience, we assume there are two invisible characters c_b and c_e in any state s such that $s[c_b]$ = -1 and $s[c_e] = |s|$ and for any visible character $c \in s$ we have $c_b \prec c \prec c_e$.

Assume the execution of any operation o in its generation state s is "correct". If o=ins(p, c), it inserts a new character c between two existing characters $s[p-1]$ and $s[p]$. Hence we add two pairs $<s[p-1], c>$ and $<c, s[p]>$ into Ω. On the other hand, if o=del(p, c), it is to delete an existing character $s[p]$. Its execution does not change the ordering between existing characters. Hence all pairs in Ω carry over.

Obviously Ω^0 defines a total order over all characters in s^0. We would like Ω be maintained as a total order over all characters that have appeared in the system at every update. To achieve so, we require (1) every execution of a deletion deletes the character it intended at its generation state, (2) only insertions at their generation states introduce new pairs while any remote execution of an insertion does not introduce pairs that contradict existing pairs in Ω. Therefore we only need to consider the local execution of insertions when we define the effects relation.

Definition 2 [Effects Relation] *Assume Ω^0 is initialized from the system initial state s^0 by adding $c_i \prec c_j$ into Ω^0 for any c_i and c_j such that $s^0[c_i] < s^0[c_j]$. Given an updating path P, assume Ω^{j-1} is a total order produced by the first $j-1$ executions in P. Now we consider how the j^{th} execution o_j extends Ω^{j-1} into Ω^j. Without loss of generality, assume o_j = ins(p_j, c_j) is an insertion to be executed in its generation state s. Suppose c_j is to be inserted between $\beta = s[p_j - 1]$ and $\alpha = s[p_j]$. Let Σ be the set of characters that appear in Ω^{j-1}. Let $B = \{\beta\} \cup \{c \in \Sigma | c \prec \beta\}$, $A = \{\alpha\} \cup \{c \in \Sigma | \alpha \prec c\}$, and $D = \Sigma - A - B = \{c \in \Sigma | \beta \prec c \prec \alpha\}$. For any character $c_i \in \Sigma$, we decide the relation between c_i and c_j by the following rules and add it to Ω^j:*

1. *if $c_i \in B$ then $c_i \prec c_j$;*
2. *if $c_i \in A$ then $c_j \prec c_i$;*
3. *if $c_i \in D$ and $c_i \in s^0$, we mandate $c_j \prec c_i$;*

4. *if $c_i \in D$ and $c_i \notin s^0$, assuming c_i is originally inserted by o_i and $o_i \rightarrow o_j$, we mandate $c_j \prec c_i$;*

5. *if $c_i \in D$ and $c_i \notin s^0$, assuming c_i is originally inserted by o_i and $o_i \parallel o_j$, we first (recursively) derive the relation between c_j and every character $c \in D$, where $c \in s^0$ or c is originally inserted by some o such that $o \rightarrow o_i$ or $o \rightarrow o_j$, and then consider the following rules:*

   (a) if there exists c in D such that either $c_i \prec c \prec c_j$ or $c_j \prec c \prec c_i$ is implied, then we add $c_i \prec c_j$ or $c_j \prec c_i$;

   (b) if no such relation can be inferred, we mandate $c_i \prec c_j$ if $o_i.id < o_j.id$, or $c_j \prec c_i$ if otherwise.

In the above definition, rules (1) and (2) are straight-forward. Rules (3) and (4) are to mandate an order between c_j and those that have been deleted before o_j is generated in state s. Rule (5) conceptually breaks ties between two characters that are inserted concurrently between the same two characters. It appears complicated because the ties cannot be safely broken by simply comparing their site ids directly, as is illustrated by the following example.

Example 2 *Suppose three sites start from s^0 = "ab". By definition, Ω is initialized to include 'a' $\prec$ 'b'. Site 1 generates o_1 = ins(1,'x') to get "axb" and concurrently site 2 generates o_2 = ins(1,'y') to get "ayb". After executing o_1, site 3 generates o_3 = ins(1,'z') to get "azxb". We have $o_1 \parallel o_2$, $o_1 \rightarrow o_3$ and $o_2 \parallel o_3$. Suppose an updating path is [o_1, o_3, o_2]. The question is how to determine the order between y and z and x. If we directly compared their site ids, we would get 'y' $\prec$ 'z' and 'x' $\prec$ 'y', which violates the established order of 'z' $\prec$ 'x'. By rule (5), to determine the order between 'y' and 'z', we first determine the order between 'y' and 'x', which is 'x' $\prec$ 'y' by rule*

(5.b). Then by transitivity (rule 5.a), we infer 'z' $\prec$ 'x' $\prec$ 'y'. Hence the final result must be "azxyb".

Theorem 1 *The effects relation $\prec$ is a total order.*

Proof. We prove this theorem by induction. Consider a update sequence sq that contains n operations. Let Σ^0 be a character set composed of all characters in s^0.

Base: Consider $sq[0]$. Obviously, Σ^0 is of a total order. Let Σ^1=Σ^0+$sq[0].c$. It is obvious that after the execution of $sq[0]$, Σ^1 still constitutes a total order.

Induction: Assume that after execution of $sq[0, i-1]$, all characters are of a total order. Let Σ^i be a character set composed of all characters in Σ^0 and the effect characters of $sq[0, i-1]$. Then, Σ^i constitutes a total order. In other words, Σ^i can be denoted as a string, str. Also, let o denote $sq[i]$ and c denote $sq[i].c$. And, we use s to refer to a state where o is executed. According to Definition 2, Σ^i can be divided into five exclusive sets. To be convenient, we use Set_1,Set_2,Set_3,Set_4 and Set_5 to refer to character sets corresponding in case 1 to 5 in Definition 2, respectively. Further, they can be denoted by five strings, str_1, str_2, str_3, str_4 and str_5.

Now, we examine each possible case:

Case 1: All five sets are empty. Let Σ^{i+1} = Σ^i+c. Then, obviously, Σ^{i+1} only contains c. The assertion is true.

Case 2: Only Set_1 is not empty. It means that c is inserted after str. As a result, a new total order is generated.

Case 3: Only Set_2 is not empty. It means that c is inserted before str. Similarly, Σ^{i+1} is of a total order.

Case 4: Only Set_1 and Set_2 are not empty. It means that c is inserted between $s[o.p-1]$ and $s[o.p]$. Obviously, $s[o.p-1]$ is the last character in Set_1 in a total order and $s[o.p]$ is the first character in Set_2 in a total order. Then, Σ^{i+1} also constitutes a total order.

Case 5: Set_3 is not empty and both Set_4 and Set_5 are empty. Obviously, Set_3 is not contained by s. Then, the case implies that c is inserted before str_3 and str_1 (It is possible that str_1 is empty). Hence, Σ^{i+1} is still of a total order.

Case 6: At least one of Set_4 and Set_5 is not empty. Similar to case 5, these characters in Set_4 and Set_5 are not in s, while the logical relations between c and them cannot be derived by transitive property. Therefore, it is safe to adopt a policy to enforce an order such that after c is inserted, the resultant character set Σ_{i+1} is still of a total order. ■

Definition 3 [Landmark Characters] *For any two characters c_i and c_j that are originally inserted by o_i and o_j, respectively, character c is a landmark character between them iff (1) $c_i \prec c \prec c_j$ or $c_j \prec c \prec c_i$, and (2) $c \in s^0$ or the operation that originally inserted c happened before at least one of o_i and o_j. The set of the landmark characters between c_i and c_j is denoted by $C_{ld}(c_i, c_j)$.*

For example, in the scenario of Example 1, let c_1='x' and c_2='y'. By Definition 3, the set of landmark characters between them is $C_{ld}(c_1, c_2) = \{$'b'$\}$. In some updating paths, such as $[o_3, o_1, o_2]$, when o_2 is transformed with o_1, the landmark character 'b' is not present in the current state. This causes tie-breaking problems in previous work such as GOTO [11], which will be further explained in Example 3.

The concept of landmark character will be used in our correctness analysis. The reason we exclude operations that are concurrent with, or are generated after, both o_i and o_j is that, due to causality, the effects relation between c_i and c_j should not depend on the effects of those operations. By the following lemma, landmark characters will play an important role in determining the effects relation of operations.

Lemma 1 *Suppose that two characters c_i and c_j are originally inserted by o_i and*

o_j, respectively, and that o_i precedes o_j in an updating path and $c_i \notin gst(o_j)$. If $C_{ld}(c_i, c_j)=\emptyset$, then we only need rules (4) and (5.b) to determine the relation between c_i and c_j.

Proof. Consider an updating path where operations preceding o_j are only those happened before either o_i or o_j. Hence, if there exists a character $c \in \Sigma$ such that $< c_i, c_j >$ can be implied based on $< c_i, c >$ and $< c, c_j >$, thus c is a landmark character, namely $c \in C_{ld}(c_i, c_j)$ by the definition of $C_{ld}(c_i, c_j)$. Now, consider the condition of $C_{ld}(c_i, c_j)=\phi$. It is true that Σ does not contain any landmark character of c_i and c_j. This further implies that $< c_i, c_j >$ can not be inferred by a chain of relations through Ω^{j-1}. Therefore, due to the known conditions, it is easy to see that the assertion is true by definition 2. ∎

2. Correctness Criteria

A quiescent state in a group editor means all generated operations have been executed at all sites [10]. System consistency requires not only that all sites have the same set of objects (characters) but also that all these objects are ordered the same way. Based on the above notion of effects relation, we define the following correctness conditions.

Definition 4 [Correctness Criteria] *A group editor is correct if it always maintains the following two properties:*

1. Causality preservation: *For any two operations o_1 and o_2, if $o_1 \rightarrow o_2$, then o_1 is executed before o_2 at all sites.*
2. Effects relation preservation: *The effects relation is preserved at every operation execution.*

Causality preservation is a fundamental criterion in typical distributed systems as well as group editors [10, 16, 25, 5]. The effects relation preservation as a system specific measure constrains convergence in a group editor and its behavior. It is easy to show that a correct group editor converges in any quiescent state, because all sites have the same set of characters that are ordered in the same way.

Corollary 1 *Any correct group editor converges in quiescent states.*

In the following, we further study how to constrain all the possible states and operation executions in a group editor.

Definition 5 ***[Reachable States]*** *Any state s is reachable iff for any $c_1, c_2 \in s$, if $s[c_1] < s[c_2]$, then $c_1 \prec c_2$.*

Conceptually, a reachable state is such that the ordering of its characters is consistent with the effects relation.

Definition 6 ***[Admissible Operations]*** *Given an operation o that is defined in a reachable state s, i.e., s = est(o), we say o is admissible in s, iff one of the following conditions holds: (1) o.t=del and the execution of o in s only leads o.c to be deleted; (2) o.t=ins and s=gst(o); or (3) o.t=ins, s≠gst(o), and for any $c \in s$: $o.p \leq s[c]$ iff $o.c \prec c$, or $s[c] < o.p$ iff $c \prec o.c$.*

Intuitively, when executing a deletion, the system should not delete any character that is not intended by the original operation; when executing an insertion, the new character ordering should not violate the effects relation definition. As an example of admissible operations, consider the execution of o_1 and o_2 in Figure 1. At site 1, if o'_2 = del(1,'x') is executed, it is not admissible because it targets the wrong character. At site 2, if o'_1 = ins(0,'x') is executed instead, then 'x' is inserted at the wrong position, which violates the effects relation established by o_1 when it is generated.

Lemma 2 *If o is admissible in a reachable state s, then s' = exec(s, o) is reachable.*

Lemma 3 *Given an admissible operation o in a reachable state s, if $o = del(p, c)$, we have $o.c = s[p]$; or if $o = ins(p, c)$, we have $s[i] \prec c \prec s[j]$ for any i and j such that $-1 \leq i < p$ and $p \leq j \leq |s|$.*

Based on the concept of admissible operations, we may further simplify the verification of system correctness by the following conclusion: If every operation a system executes is admissible, the system preserves the effect relation.

C. Transformation Functions and Conditions

The concept of effects relation lays a theoretical foundation for analyzing system behavior. However, it relies on a global data structure that is expensive to maintain in practice. Here we explore an approach that only uses local information. More specifically, each site maintains a local history buffer HB for recording operations in their execution order, and a local table ER for recording the derived effects relations. The effects relation is derived only by information in HB and ER.

We use operation keys to uniquely identify operations. The key of any operation o, denoted by function $o.key$, is a pair of the id of its generation site $o.id$ and its sequence number at that site. Due to the established state vector maintenance protocol [10, 5], no two different operations share the same key value. Unlike $o.p$, the value of $o.key$ does not change. Since each operation has a unique effect character, $o.c$ can be safely replaced by $o.key$ in algorithm implementation. The relation ER is implemented as a hash table, which stores pairs of operation keys: we infer $o_x.c \prec o_y.c$ if pair $< o_x.key, o_y.key >$ is found in ER, or equivalently, $< o_x.c, o_y.c > \in ER$.

In this section, we define the IT/ET functions based on relation $\prec$ and study sufficient conditions under which relation $\prec$ can be correctly determined using only local information (as stored in ER and HB) and the basic operation parameters $o.p$,

$o.id$, and $o.t$.

1. Inclusion Transformation and Correctness

The purpose of $o'_1 = \mathrm{IT}(o_1,\, o_2)$, where $s = \mathrm{est}(o_1) = \mathrm{est}(o_2)$ and $s' = \mathrm{exec}(s, o_2)$, is to incorporate the effect of o_2 into o_1 such that the result o'_1 can be correctly executed in state s'. As defined in Figure 2, we first call a function in Figure 3 (get_er_IT) to determine the effects relation of o_1 and o_2, and then use this relation to decide how to compute o'_1.

Function 1 *IT(*o_1*,*o_2*):* o'_1

```
get_er_IT($o_1, o_2$);
   if($o_1.c = o_2.c$)
      **return** $o'_1 \leftarrow \phi$; // $\phi$ means identity operation
   else
      $o'_1 \leftarrow o_1$;
      if($o_2.c \prec o_1.c$)
         if($o_2.t$ = ins)
            $o'_1.p \leftarrow o_1.p$+1;
         else // $o_2.t$=del
            $o'_1.p \leftarrow o_1.p$-1;
         end if
      end if
      return $o'_1$;
end if
```

Fig. 2. IT(o_1, o_2): o'_1

If $o_1.c \prec o_2.c$, it means that $o_2.c$ is on the right side of $o_1.c$ and the execution of o_2 does not affect o_1. Hence o_1 is returned as it is. In the case of $o_2.c \prec o_1.c$, however, if o_2 inserted a character on the left of $o_1.c$, the position of o_1 should be increased by one; or if o_2 deleted a character on the left, the position of o_1 should be decreased by one.

Assuming that every insertion introduces a new character, it is impossible that

o_1 has the same effect character as another operation o_2. However, two concurrent deletions may intend to delete the same existing character. To handle this situation, we adopt a policy in which this character is only deleted once. The deletion received later is transformed into an **identity operation** ϕ, which will not be executed. This corresponds to lines 2-3 in Figure 2 and lines 8-9 in Figure 3.

Function 2 *get_er_IT(o_1, o_2): relation of $o_1.c$ and $o_2.c$*

```
1  return the relation if it is found in $ER$;
2  $er \leftarrow o_2.c \prec o_1.c$;
3  if($o_1.p < o_2.p$)
4     $er \leftarrow o_1.c \prec o_2.c$
5  else if($o_1.p = o_2.p$)
6     if($o_1.t = o_2.t = $ins $\wedge$ $o_1.id < o_2.id$)
7        $er \leftarrow o_1.c \prec o_2.c$;
8     else if($o_1.t = o_2.t = $del)
9        $er \leftarrow o_1.c = o_2.c$;
10    else if($o_1.t = $ins $\wedge$ $o_2.t = $del)
11       $er \leftarrow o_1.c \prec o_2.c$;
12    end if
13 end if
14 record $er$ in $ER$;
15 return $er$;
```

Fig. 3. get_er_IT(o_1, o_2): relation of $o_1.c$ and $o_2.c$

We define a function in Figure 3 for determining the effects relation between any two given operations. It returns the relation if it has already been recorded in ER. Otherwise it derives the relation by a set of rules and records the result in ER. In the following lemmas, we study by cases the sufficient conditions of get_er_IT(o_1, o_2), or more specifically, the rules in lines 2-13. They all assume that state $s = \text{est}(o_1) = \text{est}(o_2)$ is reachable and o_1, o_2 are admissible in s. We say get_er_IT(o_1, o_2) is correct iff the relation between $o_1.c$ and $o_2.c$ as determined by its rules is identical to the effects relation obtained by Definition 2.

It is well-accepted that the two operations in IT be defined relative to the same state, i.e., est(o_1)=est(o_2), for, otherwise, it does not make sense to compare their position parameters [25, 5]. However, condition est(o_1)=est(o_2) alone is not sufficient for correctly determining the relation between $o_1.c$ and $o_2.c$ in IT, as is shown in the following scenario.

Example 3 *In Example 1, consider how operations o_2 and o_1 are executed at site 3 by (the IT and control procedure of) GOTO [11]. Suppose o_2 is received first. We get $o_2' = IT(o_2, o_3) = ins(1, \text{‘y’})$ and the history buffer is $[o_3, o_2']$. When o_1 arrives, we first get $o_1' = IT(o_1, o_3) = ins(1, \text{‘x’})$ and then perform $o_1'' = IT(o_1', o_2')$. Because their positions tie, if we naively compare their site ids to break the tie, we get $o_1'' = ins(1, \text{‘x’})$ and the final state “axyc”, which is wrong due to our analysis in Example 1. In this scenario, the precondition est(o_1')=est(o_2') is satisfied but the system results in the wrong state that violates the correct effects relation, in which ‘y’ should precede ‘x’.*

Lemma 4 *get_er_IT(o_1, o_2) can correctly determine the effects relation between o_1 and o_2, if $o_1.t = o_2.t = ins$, $o_1 \parallel o_2$, and ($o_1.p \neq o_2.p$) or ($o_1.p = o_2.p$) $\wedge$ C_{ld}($o_1.c$, $o_2.c$)=$\emptyset$)).*

Proof. If $o_1.p \neq o_2.p$, then there must exist at least one character c between $o_1.p$ and $o_2.p$ in s. That is, $C_{ld}(o_1.c,\ o_2.c) \neq \emptyset$. Without loss of generality, by Lemma 3, suppose $o_1.c \prec c \prec o_2.c$, $o_1.p < o_2.p$ and $o_1.p \leq$ s[c] $<o_2.p$. By get_er_IT(), we derive $o_1.c \prec o_2.c$, which is consistent with Definition 2.

If $o_1.p$=$o_2.p$, there is no landmark character between $o_1.c$ and $o_2.c$ in state s. Suppose there is an execution path in which o_2 precedes o_1, and all operations preceding o_2 happened before o_1 or o_2 or both. Since $C_{ld}(o_1.c,\ o_2.c)$=$\emptyset$, from Definition 2 we can deduce the relation between $o_1.c$ and $o_2.c$ by comparing site ids. From get_er_IT(), if $o_1.p{=}o_2.p$ in state s, we break the tie by comparing site ids, the result of which is consistent with that by Definition 2. ■

Lemma 5 *get_er_IT(o_1, o_2) is correct if $o_1.t$ = $o_2.t$ = del.*

Proof. Since both o_1, o_2 are admissible deletions, $o_1.c$ and $o_2.c$ must be both present in s. Hence by Lemma 3 we can determine their relation by comparing $o_1.p{=}s[o_1.c]$ and $o_2.p{=}s[o_2.c]$. In particular, if $o_1.p{=}o_2.p$, they must be referencing the same character, i.e., $o_1.c{=}o_2.c$. ■

Lemma 6 *get_er_IT(o_1, o_2) is correct if $o_1.t$ = ins, $o_2.t$ = del, and $o_1.c \neq o_2.c$.*

Lemma 7 *get_er_IT(o_1, o_2) is correct if $o_1.t$ = del, $o_2.t$ = ins, and $o_1.c \neq o_2.c$.*

Corollary 2 *Given that o_1 and o_2 are admissible in a reachable state s = est(o_1) = est(o_2), then IT is correct or o_1' = IT(o_1, o_2) is admissible in state s' = exec(s, o_2), if the effects relation between $o_1.c$ and $o_2.c$ can be found in ER or can be correctly determined by the rules of get_er_IT(o_1, o_2).*

2. Exclusion Transformation and Correctness

The purpose of $o_2' = \mathrm{ET}(o_2, o_1)$, where $\mathrm{est}(o_2) = \mathrm{exec}(\mathrm{est}(o_1), o_1)$, is to exclude the effect of o_1 from o_2 such that $\mathrm{est}(o_2') = \mathrm{est}(o_1)$ and o_2' is admissible in $\mathrm{est}(o_1)$, as if o_1 had not been executed. As in Figure 4, we first call a function in Figure 5 to determine the effect relation between o_1 and o_2, and then use this relation to compute o_2'.

If $o_1.c$ and $o_2.c$ are the same, or o_2 is to delete the character inserted by o_1, it does not make sense if o_1 had not been executed before o_2 due to causality. In this case the system is halted. This corresponds to lines 13-14 in Figure 5 and lines 2-3 in Figure 4.

Function 3 *ET(o_2, o_1): o_2'*

```
get_er_ET($o_1, o_2$);
if($o_2.c = o_1.c$)
    halt;
else
    $o_2' \leftarrow o_2$;
    if($o_1.c \prec o_2.c$)
        if($o_1.t$ = ins)
            $o_2'.p \leftarrow o_2.p$-1;
        else // $o_1.t$=del
            $o_2'.p \leftarrow o_2.p$+1;
        end if
    end if
    return $o_2'$;
end if
```

Fig. 4. ET(o_2, o_1): o_2'

If $o_2.c \prec o_1.c$, or $o_1.c$ is on the right side of $o_2.c$, then excluding the effect of o_1 does not impact o_2. Hence o_2 is returned as it is. If $o_1.c \prec o_2.c$, however, the position parameter of o_2 should be decreased by one had o_1 not inserted $o_1.c$ on its left; or the position parameter of o_2 should be increased by one had o_1 not deleted on its left.

In Figure 5, we define rules for determining the relation of $o_2.c$ and $o_1.c$ in ET only by their position parameters. Previous work [11, 5] proposed that the precondition of ET be est(o_2) = exec(est(o_1), o_1). However, this condition alone is not sufficient for determining the relation of $o_1.c$ and $o_2.c$ in ET, as shown in the following scenario.

Example 4 *Suppose three sites start from the same state s^0 = "abc". Site 1 generates o_1 = del(1,'b') to yield "ac". Concurrently site 2 generates o_2 = ins(2,'x') to yield "abxc". At site 3, assume o_1 is executed first, yielding s_3^1 = exec(s^0, o_1) = "ac". After that, o_2 is received. We have o_2' = IT(o_2, o_1) = ins(1,'x') and s_3^2 = exec(s_3^1, o_2') = "axc". Then site 3 generates o_3 = ins(2,'y'), yielding s_3^3 = "axyc". By Definition 2 we know $o_1.c \prec o_2.c \prec o_3.c$, namely $o_1.c \prec o_3.c$. Now examine how the effects of o_2' and o_1*

Function 4 *get_er_ET(o_1, o_2): relation of $o_1.c$ and $o_2.c$*

```
1  return the relation if it is found in $ER$;
2  if($o_1.p<o_2.p$)
3     $er \leftarrow o_1.c \prec o_2.c$;
4  else if($o_1.p>o_2.p$)
5     $er \leftarrow o_2.c \prec o_1.c$;
6  else // $o_1.p=o_2.p$
7     if($o_1.t=o_2.t=$ins)
8        $er \leftarrow o_2.c \prec o_1.c$;
9     else if($o_1.t=o_2.t=$del)
10       $er \leftarrow o_1.c \prec o_2.c$;
11    else if($o_1.t=$del $\wedge$ $o_2.t=$ins)
12       $er \leftarrow o_2.c \prec o_1.c$;
13    else // $o_1.t=$ins $\wedge$ $o_2.t=$del
14       $er \leftarrow o_1.c = o_2.c$;
15    end if
16 end if
17 record $er$ in $ER$
18 **return** $er$;
```

Fig. 5. get_er_ET(o_1, o_2): relation of $o_1.c$ and $o_2.c$

are excluded from o_3 in turn. First process $o_3' = ET(o_3,\, o_2')$. According to get_er_ET(), we infer $o_2.c \prec o_3.c$ because $o_2'.p < o_3.p$. Hence $o_3' =$ ins(1,'y'). Next consider $o_3'' = ET(o_3',\, o_1)$. According to get_er_ET(), we have $o_3.c \prec o_1.c$ because their positions tie, which contradicts the correct relation $o_1.c \prec o_3.c$.

In the following lemmas we examine by cases the sufficient conditions of get_er_ET(o_1, o_2), or more specifically, the rules defined in lines 2-16. They all require the following conditions: est(o_2) = exec(est(o_1),o_1), *est*(o_1) and est(o_2) are reachable states, and o_1 and o_2 are admissible. We say get_er_ET(o_1, o_2) is correct iff the relation between $o_1.c$ and $o_2.c$ as determined by its rules is identical to the effects relation obtained by Definition 2.

Lemma 8 *get_er_ET(o_1, o_2) is correct if $o_1.t = o_2.t = ins$ and $o_1 \rightarrow o_2$.*

Proof. Since o_1 and o_2 are admissible and o_1 is executed before o_2, the character $o_1.c$ must be present in state s = exec(est(o_1), o_1). Due to Lemma 3, by comparing their positions $o_1.p = s[o_1.c]$ and $o_2.p$ we can correctly infer their effects relation. Specifically, if $o_2.p \leq o_1.p$ we get $o_2.c \prec o_1.c$, or $o_1.c \prec o_2.c$ if otherwise. ■

Lemma 9 *get_er_ET(o_1, o_2) is correct if $o_1.t$ = ins, $o_2.t$=del, and $o_1 \rightarrow o_2$.*

Proof. By the given conditions and Lemma 3, $o_1.c$ must be present in s = est(o_2) and $o_1.p = s[o_1.c]$. Since o_2 is a deletion and it is admissible in s, it must be that $o_2.p$ = $s[o_2.c]$. Hence we can correctly determine their effects relation by comparing their position parameters. In particular, if $o_1.p$=$o_2.p$, it must be that $o_1.c$=$o_2.c$. ■

Lemma 10 *get_er_ET(o_1, o_2) is correct if $o_1.t$ = $o_2.t$ = del, $o_1.c \neq o_2.c$, and $o_1 \rightarrow o_2$.*

Lemma 11 *get_er_ET(o_1, o_2) is correct if $o_1.t$ = del, $o_2.t$=ins, $o_1.c \neq o_2.c$, $o_1 \rightarrow o_2$, and (1) $o_1.p \neq o_2.p$ or (2) $o_1.p$=$o_2.p$ and $C_{ld}(o_1.c, o_2.c) = \phi$.*

Proof. Let s^1 = est(o_1) and s^2 = est(o_2). Since o_1 is a deletion admissible in s^1, by Lemma 3, it must be $o_1.p = s^1[o_1.c]$. If $o_1.p < o_2.p$, there must be at least one character between $o_1.p$ and $o_2.p$ in s^2. Hence we infer $o_1.c \prec o_2.c$. If $o_1.p > o_2.p$, similarly we infer $o_2.c \prec o_1.c$.

In the case that $o_1.p$=$o_2.p$, by Lemma 3, it must be that o_2 inserts a character between the two characters $s^2[o_2.p\text{-}1] = s^1[o_2.p\text{-}1]$ and $s^2[o_2.p] = s^1[o_1.p]$. By Definition 3, $C_{ld}(o_1.c,\ o_2.c)$ considers all the characters that are present before either o_1 or o_2 is generated. If $C_{ld}(o_1.c,\ o_2.c) = \emptyset$, by Definition 2, we infer $o_2.c \prec o_1.c$. ■

Corollary 3 *Given that o_1 is admissible in a reachable state s and o_2 is admissible in $s' = exec(s, o_1)$, then ET is correct or $o_2' = ET(o_2, o_1)$ is admissible in s, if the effects relation between $o_1.c$ and $o_2.c$ can be found in ER or can be correctly determined by the rules of get_er_ET(o_1, o_2).*

D. Operation Integration

To execute (integrate) a remote operation o, we often have to transform o against a sequence of operations sq. We denote the process of inclusively transforming o against sq as ITSQ(o, sq) and that of exclusively transforming o against sq as ETSQ(o, sq). In this section we study how to execute remote operations and how to transform sequences.

1. Problem and Analysis

Conceptually, to integrate a remote operation o, we must first compute its admissible form o' relative to the current state at the local site and then execute it. This problem can be abstracted in the following way: *Given an operation o admissible in a given reachable state s, compute its admissible form o' in another reachable state s', where $s \neq s'$.*

In the abstract sense, computing o' includes two steps: (1) choose or build a proper path (or sequence) P from s to s', and (2) transform o against P. There may exist more than one path from s to s'. As shown in Figure 6, some paths are unidirectional such that we only need to transform o with some P. Some other paths may be bidirectional in that they first consider a path P_b from s to a significant intermediate state s'' and then one P_f from s'' to s'.

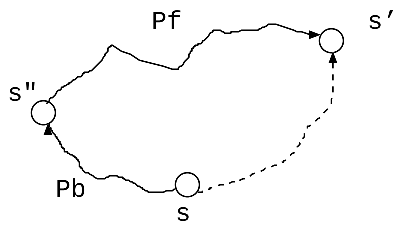

Fig. 6. Different transformation paths from state s to state s'.

Specifically, we maintain an operation log HB at every site to record operations in their order of execution. When a remote operation o is received, we integrate o iff all operations that happened before o have been executed locally. Suppose HB can be somehow transposed into sq_h+sq_c, where sq_h includes all operations that happened before o, sq_c includes all operations in HB that are concurrent with o. It must be that $s = \text{exec}(s^0, sq_h)$ and $s' = \text{exec}(s^0, HB)$.

Most previous OT algorithms use sq_c as the transformation path from s to s' and compute $o' = \text{ITSQ}(o, sq_c)$, such as adOPTed [16], GOTO [11, 31], SOCT2 [25], and SDT [19, 14]. However, the ordering between concurrent operations in sq_c is not constrained, except the causal order between them. It means that concurrent operations may be ordered arbitrarily at different sites. As a result, their IT functions must verify TP1 and TP2 to achieve convergence [16], which have turned out extremely difficult in practice [31, 5, 33]. Previous works in [34, 18, 5, 33] achieve convergence by enforcing a unique transformation path at all sites and avoid verification of TP2. However, the transformation paths they choose are effectively unconstrained in the same way. That is, arbitrary transformation path is still allowed as in previous work such as adOPTed, GOTO, SOCT2 and SDT.

In this work, we explore an alternative strategy which avoids arbitrary transformation paths. Instead, we build some special paths out of HB such that our IT and ET can always work correctly. The idea is to first find a backward path P_b from s to s'' and then a forward path P_f from s'' to s'. With these two paths, we first compute an admissible operation $o'' = \text{ETSQ}(o, P_b)$ relative to s'' and then compute the admissible $o' = \text{ITSQ}(o'', P_f)$ relative to s'.

However, our analyses in Section C reveal that $\text{IT}(o_1, o_2)$ and $\text{ET}(o_1, o_2)$ do not always work correctly. There are preconditions in addition to $\text{est}(o_1) = \text{est}(o_2)$ and $\text{est}(o_1) = \text{exec}(\text{est}(o_2), o_2)$, respectively. Intuitively, we somehow transpose sq_h into

$sq_{hi} + sq_{hd}$ such that sq_{hi} includes all insertions that happened before o and sq_{hd} includes all deletions that happened before o. Let P_b be sq_{hd} and P_f be $sq_{hd}+sq_c$. And we further somehow transpose P_f into $sq_i + sq_d$ such that sq_i only includes insertions and sq_d are all deletions. As a result, s'' is the state in which all characters deleted by sq_{hd} are recovered. Hence we can always correctly process ITSQ$(o'', sq_i + sq_d)$ because no landmark character, if any, between insertion o'' and any insertion in sq_i has been deleted, and IT between o'' and any deletion in sq_d is correct. Then the remaining problem is how to build such P_b and P_f and how to ensure the correctness of IT or ET at every step.

In the following, we discuss how to fulfill the above intuition. In Section 2 and 3 we first study some special sequences that ensure correctness of ITSQ and ETSQ, respectively. Then Sections 4 and 5 define some utility functions for transposing sequences and building special sequences. Section 6 shows how to integrate remote operations by constructing P_b and P_f.

2. IT-Safe Operation Sequences

Function 5 *ITSQ(o, sq): o′*

```
1  $o' \leftarrow o$;
2  for($i = 0$ to $|sq| - 1$){
3     $o' \leftarrow$ IT($o'$, $sq[i]$);
4     if($o' = \phi$)
5        **break**;
6     end if
7  }
8  **return** $o'$
```

Fig. 7. ITSQ(o, sq): o'

Function ITSQ(o, sq) in Figure 7 requires that o and sq be defined in the same state, or est(o) = est($sq[0]$). It inclusively transforms o against $sq[0], sq[1], ..., sq[|sq|-1]$ in turn. To ensure the correctness of every IT, we further require that sq is a special sequence, as defined below.

Definition 7 *A sequence sq is an IT-Safe Operation Sequence (ITSOS) iff for any two operations $sq[i]$ and $sq[j]$, if $sq[i]$.t=ins and $sq[j]$.t=del, then $i < j$ holds. If $|sq| < 2$, we also say that sq is (trivially) an ITSOS.*

An ITSOS sq can be rewritten as $sq_i + sq_d$, where sq_i includes all insertions in sq and sq_d all deletions in sq.

Lemma 12 *Given operation o and sequence sq, o' = ITSQ(o,sq) is admissible, if est(sq[0]) = est(o) and either (1) o.t=del, or (2) o.t=ins $\wedge$ sq is an ITSOS $\wedge$ for any insertion $sq[i]$: C_{ld}(sq[i].c, o.c) $\subseteq$ est(sq[i]).*

Proof. If $o.t$ = del, by Lemmas 5 and 7, IT(o, $sq[i]$) is correct for any operation $sq[i]$. Consider case $o.t$=ins. Because sq is an ITSOS, let $sq = sq_i + sq_d$ and ITSQ(o, sq) = ITSQ$(o, sq_i + sq_d)$. By the given condition, for any insertion $sq_i[j]$: C_{ld}($sq_i[j].c$, $o.c$) $\subseteq$est($sq_i[j]$), meaning that all characters in C_{ld}($sq_i[j].c$, $o.c$) must appear in est($sq_i[j]$). That is, $sq_i[j].p$= $o.p$ if C_{ld}($sq_i[j].c$, $o.c$) is $\emptyset$, or $sq_i[j].p \neq o.p$ if otherwise. Then by Lemma 4, when inclusively transforming o against any insertion in sq_i, no landmark character between them has been lost (or deleted). Hence IT($o, sq_i[j]$) is correct for any insertion $sq_i[j]$. By Lemma 6, IT($o, sq_d[k]$) is correct for any deletion $sq_d[k]$. ■

3. ET-Safe Operation Sequences

ETSQ(o, sq) requires that sq and o be contextually serialized, i.e., est(o) = exec(est($sq[0]$), sq). As shown in Figure 8, to compute ETSQ(o, sq), we exclusively transform o against $sq[n-1], sq[n-2], ..., sq[0]$ in turn. To ensure the correctness of every ET, we

Function 6 *ETSQ(o, sq): o'*

```
1  $o' \leftarrow o$
2  for( $i$=$|sq|$-1 to 0 step -1) {
3     $o' \leftarrow$ ET($o'$, $sq[i]$)
4     if($sq[i].c \prec o'.c$)
5        **break**;
6     end if
7  }
8  for( $j$=$i$-1 to 0 step -1) {
9     if($sq[j].t$ = ins)
10       $o'.p \leftarrow o'.p$-1;
11    else // $sq[j].t$=del
12       $o'.p \leftarrow o'.p$+1;
13    end if
14 }
15 **return** $o'$;
```

Fig. 8. ETSQ(o, sq): o'

further require that sq be a special sequence, as defined below. Due to properties of this sequence, we break the loop of lines 2-7 when some condition is satisfied because it is no longer necessary to call ET() on the remaining operations in sq. Instead, effects of those operations are excluded directly as in lines 8-14.

Definition 8 *A given sequence sq is an ET-Safe Operation Sequence (ETSOS) if for any i, j, where $0 \leq i < j \leq n-1$ and $|sq| = n$, either (1) $sq[i].c \prec sq[j].c$, or (2) $sq[i].c = sq[j].c$ and $sq[i].t$=ins and $sq[j].t$=del. If $|sq| < 2$, we also say that sq is (trivially) an ETSOS.*

Operations in an ETSOS sq are ordered by their effects relation, except that inverse operations are ordered by their causal order, i.e., a deletion o_d that deletes the character inserted by some o_i appears after o_i. As a simple example, given an initial state s^0 = "b" and three operations executed in a tandem: o_1=ins(0,'a'),

o_2=del(1,'b'), and o_3=ins(1,'c'). The effects relation is 'a' $\prec$ 'b' $\prec$ 'c' and $[o_1, o_2, o_3]$ is an ETSOS.

Lemma 13 *Given operation o and an ETSOS sq, if there exists some sq*$[i]$*,* $0 \leq i \leq |sq| - 1$*, such that sq*$[i]$*.c* $\prec$ *o.c, then sq*$[j]$*.c* $\prec$ *o.c for all j:* $0 \leq j \leq i-1$*.*

The correctness of Lemma 13 is straightforward by the definition of ETSOS. The result can simplify the implementation of ETSQ(o, sq). When o encounters some $sq[i]$ in ETSQ(), if $sq[i].c \prec o.c$, it means that the effects relation between o and any operation $sq[j]$, where $j < i$, is known, namely, $sq[j].c{\prec}o.c$. As a result, the remaining operations in sq can be correctly excluded from o without get_er_ET(). This property is used in ETSQ() to avoid calling ET(), as shown in lines 4-6 and 8-14. In addition, if o and some operation $sq[i]$ have the same effect character, namely, $o.c$= $sq[i].c$ or o deletes a character inserted by some $sq[i]$, it does not make sense to perform this and the remaining ET's due to causality. The system is halted in this case. Other algorithms that call ETSQ() will prevent this from happening.

Lemma 14 *Given operation o and sequence sq, where* $|sq| = n$ *and est(o) = exec(est(sq*$[0]$*), sq), let* C_{sq} *denote the set of characters that appear in est(sq*$[0]$*) and sq, that is,* $C_{sq} = \bigcup_{i=0}^{n-1}\{\ sq[i].c\ \} \cup$ *est(sq*$[0]$*). Then* o' *= ETSQ(o, sq) is admissible if (1) sq is an ETSOS, and (2)* C_{ld}*(o.c, sq*$[i]$*.c)* $\subseteq C_{sq}$ *for any sq*$[i] \rightarrow o$*.*

Proof. Due to Lemmas 8 through 11 and Corollary 3 in Section 2, the only case under which ET(o, $sq[i]$) can be unsafe for some $sq[i]$ is when $sq[i].p$=$o.p$, $sq[i] \rightarrow o$, $sq[i].t$=del, and $o.t$=ins. Hence we only need to prove the correctness of ET(o, $sq[i]$) under this unsafe condition. We prove this by induction.

Base: Consider ET(o,$sq[n$-1$]$). First, by definition of ETSOS, we know $sq[k].c \prec sq[n-1].c$ for $0 \leq k < n-1$. By condition $C_{ld}(o.c,\ sq[n-1].c) \subseteq C_{sq}$, for any $sq[k]$, where $0 \leq k < n-1$, it is impossible to have $sq[n-1].c \prec sq[k].c \prec o.c$. Secondly, under the above-identified unsafe condition of ET, $sq[n-1].t$=del, $o.t$=ins, $sq[n-1] \rightarrow o$, and $sq[n-1].p = o.p$, it is impossible to have any c in est(o) such that $sq[n-1].c \prec c \prec o.c$, for otherwise we would have got $sq[n-1].p < o.p$, which contradicts $sq[n-1].p = o.p$. Since est(o) = exec(est(sq[0]), sq) and $C_{sq} = \bigcup_{i=0}^{n-1}\{sq[i].c\} \cup \text{est}(sq[0])$, with the above two conclusions combined, we infer that there is no character c in C_{sq} such that $sq[n-1].c \prec c \prec o.c$. Therefore we conclude $C_{ld}(o.c,\ sq[n-1].c)$=$\emptyset$ or there exists at least one character c such that $o.c \prec c \prec sq[n-1].c$. Either case, by the definition of relation $\prec$, we get $o.c\prec\ sq[n-1].c$. Hence ET$(o, sq[n-1])$ is correct.

Induction: Assume that processing of ETSQ(o, $sq[i+1, n-1]$) is correct. Let o' be the execution form of o relative to est($sq[i+1]$). By Lemma 13 and Figure 8, at the time ET(o', $sq[i]$) is processed, o' must have been exclusively transformed with all operations from $sq[n-1]$ to $sq[i+1]$, inclusively. Hence, $o.c \prec sq[k].c$ must hold, where $i+1 \leq k \leq n-1$, because, otherwise, the ET process should have stopped for some k. Since $C_{ld}(o.c, sq[i].c) \subseteq C_{sq}$, similar to the base case, there must not exist c such that $sq[i].c \prec c \prec o.c$ for otherwise we would have got $sq[i].p < o.p$, which contradicts $sq[i].p = o.p$. Then by the definition of relation $\prec$, no matter whether or not there exists c such that $o.p \prec c \prec sq[i].c$, we get $o.c\prec\ sq[i].c$. Hence ET$(o, sq[i])$ is correct. ■

4. Transposing Sequences

Transpose(o_2, o_1) in Figure 9 is to transpose two contextually serialized operations o_2 and o_1 such that in the output o_1' is contextually serialized before o_2'. Since o_2 is contextually serialized before o_1, we first do o_1'=ET(o_1, o_2) such that o_1' and o_2 are defined in the same state, and then do o_2'=IT(o_2, o_1') such that o_2' is defined in state exec(est(o_1'),o_1'). Note if o_1 depends on o_2, i.e., $o_2.t$=ins, $o_1.t$=del, $o_2 \rightarrow o_1$, and

Function 7 *Transpose(o_2, o_1): $< o'_1, o'_2 >$*

```
// determined as in get_er_ET
if($o_1.c = o_2.c$)
    **return** $< o_1, o_2 >$;
else
    $o'_1 \leftarrow$ ET($o_1$, $o_2$);
    $o'_2 \leftarrow$ IT($o_2$, $o'_1$);
    **return** $< o'_1, o'_2 >$;
end if
```

Fig. 9. Transpose(o_2, o_1): $< o'_1, o'_2 >$

$o_1.p$=$o_2.p$, then we know $o_1.c$=$o_2.c$ and return $< o_1, o_2 >$.

Function 8 *TransposeOSq(sq, o): $< o', sq' >$*

```
$o' \leftarrow o$;
$sq' \leftarrow sq$;
for($i$ = |$sq$|-1 to 0 step -1) {
    $< o', sq'[i] > \leftarrow$ Transpose($sq'[i], o'$);
}
**return** $< o', sq' >$;
```

Fig. 10. TransposeOSq(sq, o): $< o', sq' >$

The precondition of TransposeOSq(sq, o) in Figure 10 is that sq is contextually serialized before o. The output is such that o' is contextually serialized before sq'. In general, all the transposition functions in this chapter produce effects-equivalent sequences. Specifically, $[o'_1, o'_2] \sim [o_2, o_1]$ in Transpose(o_2, o_1), $o' + sq' \sim sq + o$ in TransposeOSq(sq, o), $sq_h + sq_c \sim sq$ in TransposePreCon(o, sq), $sq_i + sq_d \sim sq$ in TransposeInsDel(sq) , and $sq' \sim sq$ in BuildETSOS(sq).

Functions TransposePreCon() in Figure 11 and TransposeInsDel() in Figure 12 have similar structures. TransposePreCon(o, sq) is adopted from [25, 11]. It transposes sq into $sq_h + sq_c$, such that sq_h, all operations that happened before o, are

Function 9 *TransposePreCon(o, sq): $< sq_h, sq_c >$*

```
$sq_h \leftarrow \emptyset$;
$sq_c \leftarrow \emptyset$;
for( $i$=0 to $|sq| - 1$) {
    if($sq[i] \parallel o$)
        $sq_c \leftarrow sq_c + sq[i]$
    else // $sq[i] \rightarrow o$
        $< o_i, sq_c > \leftarrow$ TransposeOSq($sq_c$, $sq[i]$);
        $sq_h \leftarrow sq_h + o_i$;
    end if
    }
return $< sq_h, sq_c >$;
```

Fig. 11. TransposePreCon(o, sq): $< sq_h, sq_c >$

Function 10 *TransposeInsDel(sq): $< sq_i, sq_d >$*

```
$sq_i \leftarrow \emptyset$;
$sq_d \leftarrow \emptyset$;
for($i$=0 to $|sq| - 1$) {
    if($sq[i].t$ = del)
        $sq_d \leftarrow sq_d$+sq[i];
    else // $sq[i].t$ = ins
        $< o_i, sq_d > \leftarrow$ TransposeOSq($sq_d$, $sq[i]$);
        $sq_i \leftarrow sq_i$+$o_i$;
    end if
}
return $< sq_i, sq_d >$;
```

Fig. 12. TransposeInsDel(sq): $< sq_i, sq_d >$

contextually serialized before sq_c, all those concurrent with o. It scans sq from left to right and appends every $sq[i]$ that is concurrent with o to sq_c. For every $sq[i]$ that happened before o, it first transposes $sq[i]$ and sq_c and then appends $sq[i]$ to sq_h. Similarly, TransposeInsDel(sq) transposes sq such that all insertions (sq_i) are contextually serialized before all deletions (sq_d).

5. Building ET-Safe Sequences

Function 11 *BuildETSOS(sq): sq′*

```
if($|sq| < 1$)
   return $sq' \leftarrow sq$;
end if
$sq' \leftarrow$ [ $sq[0]$ ];
for( $i$=1 to $|sq| - 1$) {
   $o \leftarrow sq[i]$;
   $flag \leftarrow$ false;
   for($j = |sq'|$-1 to 0 step -1) {
      if($flag$ = true)
         record $sq'[j].c \prec o.c$ into $ER$;
      else //effects relation determined as in get_er_ET
         if($sq'[j].c \prec o.c$)
            $sq' \leftarrow sq'[0, j] + o + sq'[j + 1, |sq'| - 1]$;
            $flag \leftarrow$ true;
         else
            $< o, sq'[j] > \leftarrow$ Transpose($sq'[j], o$);
         end if
      end if
   }
   if( $flag$ = false)
      $sq' \leftarrow o + sq'$
   end if
}
return $sq'$;
```

Fig. 13. BuildETSOS(sq): sq'

Given a sequence sq, function BuildETSOS(sq) in Figure 13 incrementally builds an ETSOS sq'. Initially sq' only includes $sq[0]$. Then each time a new element $sq[i]$ is added into sq'. It transposes $sq[i]$ with operations in sq' from right to left until some $sq'[j]$ is found (line 12) such that $sq'[j].c \prec o.c$, which is exactly the condition defined in Lemma 13. Under this case, o is inserted after $sq'[j]$ (line 13). Then the relation between $o.c$ and other operations in the rest of sq is known and directly recorded in the local effects relation ER (line 10). If the condition never appears, as in lines 20-22, then o is added to the head of sq'.

Lemma 15 *Given an admissible sequence sq defined in the initial system state s^0, which preserves the causal order of operations, sq' = BuildETSOS(sq) is correct, i.e., sq' is an ETSOS and every operation in sq' is admissible.*

Proof. Examine the process of building an ETSOS sq' from sq: each time we add a new operation $sq[i]$ into the partial result sq' that was built from $sq[0, i-1]$. Sequence sq' is scanned from right to left to find the right position to insert $sq[i]$. Note that each operation in sequence sq' actually represents a state transition. Hence, to insert $sq[i]$ before some position $sq'[j]$, we have to transpose $sq'[j, i-1]$ and $sq[i]$. That is, the correctness of BuildETSOS() depends on the correctness of Transpose().

In function Transpose(o_1, o_2), ET is always called before IT. If $o_1' = \mathrm{ET}(o_1, o_2)$ is correct, then the relation of $o_1.c$ and $o_2.c$ is known, which ensures the correctness of IT(o_2, o_1'). Hence we only need to prove the correctness of ETSQ($sq[i], sq'[j, i-1]$). This can be further reduced to proving that the conditions of Lemma 14 can be satisfied before calling BuildETSOS(). We prove this by induction.

Base: consider ET($sq[1]$, $sq[0]$). If $sq[1] \parallel sq[0]$, then ET($sq[1]$, $sq[0]$) is correct because the effects relation between concurrent operations has been recorded. Otherwise, it must be $sq[0] \rightarrow sq[1]$ by causality. Because gst($sq[0]$)=s^0 and there is no other operation, we have $C_{ld}(sq[0].c, sq[1].c) \subseteq C_{sq}$. Hence ET($sq[1]$, $sq[0]$) is correct.

Induction: Suppose sq' is the ETSOS correctly built from $sq[0, i-1]$. Now check how $sq[i]$ is added into sq'. Let $C_{sq'} = \bigcup_{j=0}^{i-1}(sq'[j].c) \cup s^0$. Since $sq'[0]$ is defined on s^0 and all operations that happened before $sq[i]$ are in sq', it is obvious that $C_{ld}(sq[i].c, sq'[j].c) \subseteq C_{sq'}$ for any $0 \leq j < |sq'|$ and $sq'[j] \rightarrow sq[i]$. By Lemma 14, ETSQ$(sq[i], sq')$ is correct. ■

6. The Integration Procedure

The top-level control algorithm executes local and remote operations. For best responsiveness, any local operation is executed and appended to HB once it is generated. Any remote operation o is queued until it is causally-ready, i.e., all operations that happened before o are in HB. Suppose the initial state is s^0, the current state is s', and o is generated in state s. Then function Integrate(o, HB) in Figure 14 is called to derive an o' such that o' is admissible in s'. If $o' = \phi$, it is discarded. Otherwise o' is executed and appended to HB. Apparently HB is a contextually serialized sequence.

In Integrate(o, HB) , we first call TransposePreCon(o, HB) to transpose HB into two sequences: sq_h, which includes all operations that happened before o, and sq_c, which includes all operations that are concurrent with o. Due to causality, we have $s = \text{gst}(o) = \text{exec}(s^0, sq_h)$.

If sq_c is empty, it means $s' = \text{gst}(o)$. Then o is returned as it is (lines 2-3) and will be executed in s' directly. If sq_c contains concurrent operations, we must incorporate the effects of those operations into o to get its execution form o' in s'. By the above analyses, we cannot perform ITSQ(o, sq_c) until the preconditions defined in Lemma 12 are satisfied. If $o.t$=del, ITSQ(o, sq_c) is guaranteed to be correct (lines 5-6).

However, if $o.t$=ins, we must first construct P_b and P_f, then compute $o'' =$ ETSQ(o, P_b), and finally $o' =$ ITSQ(o'', P_f). Sequence P_b is constructed in lines 8-9:

Function 12 *Integrate(o, HB): o′*

```
1  $< sq_h, sq_c > \leftarrow$ TransposePreCon($o$, $HB$);
2  if($sq_c = \emptyset$)
3     **return** $o' \leftarrow o$;
4  end if
5  if($o.t$=del)
6     **return** $o' \leftarrow$ ITSQ$(o, sq_c)$;
7  else // $o.t$=ins
8     $sq'_h \leftarrow$ BuildETSOS($sq_h$)
9     $< sq_{hi}, sq_{hd} > \leftarrow$ TransposeInsDel($sq'_h$);
10    $sq_{hdc} \leftarrow$ BuildETSOS($sq_{hd}$+$sq_c$);
11    $< sq_i, sq_d > \leftarrow$ TransposeInsDel($sq_{hdc}$);
12    $o'' \leftarrow$ ETSQ$(o, sq_{hd})$; // $P_b = sq_{hd}$
13    $o' \leftarrow$ ITSQ$(o'', sq_i + sq_d)$; // $P_f = sq_i + sq_d$
14 **return** $o'$;
15 end if
```

Fig. 14. Integrate(o, HB): o'

we first build an ETSOS sq'_h from sq_h, the sequence of all operations that happened before o, and then transpose sq'_h into two subsequences, sq_{hi} and sq_{hd}, such that sq_{hi} includes all insertions that happened before o and sq_{hd} includes all deletions that happened before o. Then sq_{hd} is just the P_b we need.

Next we construct P_f as in lines 10-11: we first build an ETSOS sq_{hdc} from $sq_{hd} + sq_c$, and then transpose sq_{hdc} into two sequences sq_i and sq_d such that sq_i includes all the insertions and sq_d all the deletions. Sequence P_f is just $sq_i + sq_d$, the concatenation of sq_i and sq_d.

Theorem 2 *o' = Integrate(o, HB) is admissible in state s'.*

Proof. The assertion is easily verified if $o.t$=del. When $o.t$=ins, the correctness of Integrate() relies on fives steps: (1) TransposePreCon() in line 1, (2) the construction of P_b in lines 8-9 and P_f in lines 10-11, (3) ETSQ(o, P_b) in line 12, and (4) ITSQ(o'', P_f)

in line 13.

First, in TransposePreCon(o, HB), the cause-effect order between operations in HB are preserved. It can be shown that the Transpose algorithm is called only between concurrent operations, the reason of which follows: Due to the definition in Figure 11, all operations in sq_c are concurrent with o. Hence each time we call TransposeOSq$(sq_c, sq[i])$ or Transpose$(sq_c[j], sq[i])$ for some $sq_c[j]$, due to causality preservation, $sq[i]$ must have happened before o, or $sq[i] \rightarrow o$. In addition, since $sq[i]$ is stored after $sq_c[j]$ in HB due to Figure 11, we cannot have $sq[i] \rightarrow sq_c[j]$ due to causality. Hence we have either $sq_c[j] \rightarrow sq[i]$ or $sq_c[j] \parallel sq[i]$. The former case cannot happen because, if otherwise, we would have $sq_c[j] \rightarrow sq[i] \rightarrow o$ and $sq_c[j] \rightarrow o$ by transitivity, which contradicts $sq_c[j] \parallel o$. Therefore $sq_c[j] \parallel sq[i]$ must be true and the effects relation between them must have been recorded in the local memory ER, which ensures the correctness of Transpose of ET/IT at each step.

Second, by Lemma 15, sq'_h = BuildETSOS(sq_h) in line 8 is correct. As a result, operations in sq'_h are ordered by their effects relation and additionally, if an deletion o_d deletes the character inserted by some o_i, then o_i always appears before o_d. Consequently, in the algorithm of TransposeInsDel(sq'_h) in line 9, for any insertion $sq'_h[i]$ and any deletion $sq_{hd}[j]$ in Transpose$(sq_{hd}[j], sq'_h[i])$, the case of $sq[i].c{=}sq_d[j].c$ will never happen. Then because the effects relation recorded in BuildETSOS(sq_h), all ET/IT in TransposeInsDel(sq'_h) can be correctly processed.

Rewrite HB as $sq_h + sq_c = sq_{hi} + sq_{hd} + sq_c$. When building an ETSOS out of $sq_{hd} + sq_c$, no character has been lost (or deleted) in the definition state of $sq_{hd} + sq_c$ or state $s'' = \text{exec}(s^0, sq_{hi})$ due to the execution of sequence sq_{hi} in s^0. Hence the step of sq_{hdc} = BuildETSOS$(sq_{hd} + sq_c)$ in line 10 is correct, the proof of which resembles that of Lemma 15. Similarly, TransposeInsDel(sq_{hdc}) in line 11 is also correct.

Third, consider o'' = ETSQ$(o,\ P_b)$, where $P_b = sq_{hd}$. It can be shown from the algorithm of TransposeInsDel() that P_b preserves the effects relation based order between operations as a result of BuildETSOS(). Hence P_b itself is also an ETSOS, although all its operations are deletions. Let $C_{P_b} = \bigcup_{i=0}^{|P_b|-1}\{P_b[i].c\} \cup \text{est}(P_b[0])$. Because P_b only contains deletions, for any $P_b[i] \rightarrow o$, when processing $\text{ET}(o, P_b[i])$, condition $C_{ld}(o.c, P_b[i].c) \subseteq C_{P_b}$ must hold. By Lemma 14, o'' is admissible in state $s'' = \text{exec}(s^0, sq_{hi}) = \text{est}(P_b[0])$.

Fourth, consider $o' = \text{ITSQ}(o'',P_f)$, where $P_f = sq_i + sq_d$ is built from sequence $sq_{hd} + sq_c$. Conceptually o' must incorporate the effects of sq_c and additionally sq_{hd} because the effects of sq_{hd} have been excluded from o'' earlier. Sequence P_f is an ITSOS, because it is a concatenation of an insertion sequence sq_i and a deletion sequence sq_d, which were correctly computed earlier. Consequently, because sequences sq_{hi} and sq_i do not contain any deletions, the preconditions of Lemma 12 are satisfied. Therefore, $o' = \text{ITSQ}(o'', P_f)$ is admissible in the current state $s' = \text{exec}(s^0, HB) = \text{exec}(s'', P_f)$. ■

E. An Example

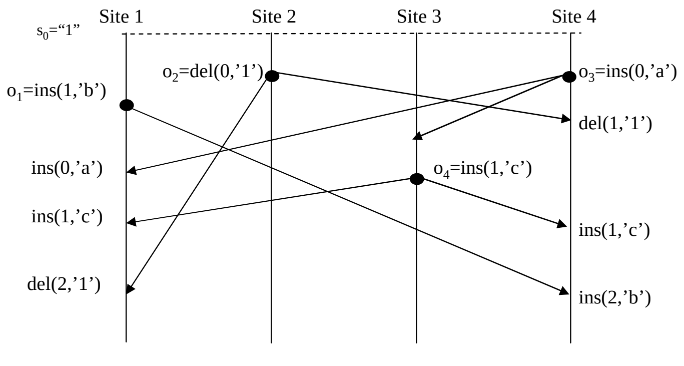

Fig. 15. A scenario in which four sites start from state "1" and converge in "acb".

As shown in Figure 15, four sites start from the same initial state s^0 = "1". Site 1 performs o_1 = ins(1,'b'). Concurrently, site 2 performs o_2 = del(0,'1') and site 4 performs o_3 = ins(0,'a'). After o_3 is executed, site 3 performs o_4=ins(1,'c'). We have $o_1 \parallel o_2$, $o_2 \parallel o_3$, $o_1 \parallel o_3$, $o_3 \rightarrow o_4$, $o_1 \parallel o_4$, and $o_2 \parallel o_4$. Suppose the execution order at site 4 is o_3, o_2, o_4, o_1. For space reasons, we only consider how o_1 and o_4 are integrated at site 4 to illustrate the working of our approach.

When o_4 arrives at site 4, $HB = [o_3, o_2']$, where o_2' = IT(o_2, o_3) = del(1,'1'), and the current state s_4^2 = exec(s^0, $[o_3, o_2']$) = "a". To compute o_4' relative to s_4^2, we first construct P_b, which includes all deletions that happened before o_4. Hence P_b is empty by definition. Then we construct P_f, which is initially $sq_{hd} + sq_c = sq_c = [o_2']$. Therefore, o_4'=ITSQ(o_4,$[o_2']$)=ins(1,'c'). After o_4' is executed, the state of site 4 becomes s_4^3 = "ac".

When o_1 arrives, $HB = [\ o_3, o_2', o_4'\ ]$. Similarly P_b is empty. Hence P_f is initially HB itself because all the operations are concurrent with o_1. We transpose P_f into $sq_i + sq_d$ such that $sq_i = [o_3, o_4]$ and $sq_d = [o_2'']$, where o_2'' = del(2,'1'). Then o_1' = ITSQ(o_1, $[o_3, o_4, o_2'']$) = ins(2,'b'). After o_1' is executed, the final state of site 4 becomes s_4^4 = "acb".

CHAPTER IV

A NATURAL ORDER BASED FRAMEWORK

A. Motivation

In the last chapter, we propose the total order based framework to systemically address the correctness problem of OT based protocols. However, this framework is not perfect. One main limitation comes from the requirement of defining a total order over objects that ever appear in the system. First, let us consider an example where a user only executes two operations o_1=del(0,'x') and o_2=ins(0,'y') in sequence relative to an initial state s_0="a". In this case, since 'x' and 'y' never have chance to appear in the same state, defining a logical order between 'x' and 'y' is not necessary.

The scenario illustrates that defining a total order over objects is not always a good design approach because it causes unnecessary design overheads. In addition, defining a total order is costly because designers need to carefully examine all cases that ever happen in the system.

Just these limitations motivate us to further explore a new design methodology. In this chapter, we aim to propose a natural order based framework. Compared to the total order based framework, defining a total order in this work is no longer needed. To achieve this goal, we first need to establish a new consistency model, and we will propose a set of new correctness criteria based on this model.

B. A Consistency Model

In this section we formalize a consistency model. Core to this model is a notion of operation effects relation $\prec$. Sec. 1 introduces a tool called effect relation graph for analyzing the behavior of group editors. Sec. 2 defines relation $\prec$ and related concepts. Sec. 3 defines correctness criteria of group editors.

1. Effect Relation Graph

The effect relation graph (or graph) is a global data structure. Note it is only for theoretical analysis purposes, not implemented in actual systems. The graph is constructed incrementally as operations are generated and executed at collaborating sites as if by an external observer of the system.

A graph G is a tuple $< V, E >$, where V is a set of nodes (or vertexes) and $E \subseteq V \times V$ is a set of directed edges. Every node corresponds to a character that ever appears in the system and every edge represents the relation between two characters. Each node $n \in V$ has three attributes: $n.char$ is the character (id) it corresponds to; $n.counter$ traces how the character is inserted or deleted; and $n.color$ indicates the status, *white* for normal and *red* for abnormal. An edge $< n_a, n_b > \in E$ means $n_a.char$ appears on the left of $n_b.char$ in some state of the group editor.

The algorithm in Figure 16 shows how G is maintained. Suppose V and E are initially empty and N is the number of sites in the system. We first use the initial state s_0 to initialize G as follows: If s_0 is not empty, for each character $c \in s_0$ add a new node n to V such that $n.char = c$, $n.counter =$ N, and $n.color =$ white. Then for any two nodes $n_b, n_a \in V$ add an edge $< n_b, n_a >$ to E, if $n_b.char = c_b$, $n_a.char = c_a$, and $s_0[c_b] + 1 = s_0[c_a]$.

We always invoke o once when it is generated and once when it is executed at every remote site. Every invocation updates G once. Hence G is updated N times in total by every o. The order of invocations must maintain their natural cause-effect relation: (1) given any operation o, its local invocation must be earlier than any remote invocation, and (2) given any o_1 and o_2, if $o_1 \rightarrow o_2$, invocation of o_1 must be earlier than invocation of o_2 at every site.

```
Initialize:
1  $\forall c \in s_0$ add a new node $n$ to $V$ such that
2      $n.char \leftarrow c$; $n.counter \leftarrow N$; $n.color \leftarrow$white;
3  $\forall n_b, n_a \in V$ if $s_0[n_b.char] + 1 = s_0[n_a.char]$
4      add edge $< n_b, n_a >$ to $E$;

Invoke an insertion $o$ in state $s$:
1   if    $\exists n \in V : n.char = o.c$
2           $n.counter \leftarrow n.counter + 1$;
3   else add a new node $n$ to $V$ such that
4           $n.char \leftarrow c$; $n.counter \leftarrow 1$; $n.color \leftarrow$white;
5   if $\exists n_b \in V : n_b.char = s[o.p - 1]$
6           add $< n_b, n >$ to $E$ if $< n_b, n > \notin E$;
7   if $\exists n_a \in V : n_a.char = s[o.p]$
8           add $< n, n_a >$ to $E$ if $< n, n_a > \notin E$;

Invoke a deletion $o$ in state $s$:
1   find $n \in V$ such that $n.char = o.c$;
2   if $s[o.p] \neq o.c$
3         $n.color \leftarrow$ red;
4   elseif $n.color \neq red$
5           $n.counter \leftarrow n.counter - 1$;
6           $n.color \leftarrow$ red if $n.counter = -1$;
```

Fig. 16. Maintaining a global graph G.

Every invocation of any o is always relative to some state s at some site. The local invocation of an insertion o always leads to the creation of a new node n in G with $n.char = o.c$, $n.counter = 1$, and $n.color =$ white. After that, every remote invocation of insertion o increments $n.counter$ by one. The invocation of o in s will cause a new character $o.c$ to be inserted between two characters $s[o.p-1]$ and $s[o.p]$. Hence if these two characters exist, we also add the two edges $< n_a, n >$ and $< n, n_b >$ into G, where $n_a.char = s[o.p - 1]$ and $n_b.char = s[o.p]$.

The invocation of any deletion o on G is trickier because $o.p$ may point to different characters in different execution states. Since $o.c$ is a constant once o is generated and we never delete nodes in G, it can be shown that we can always find the node $n \in V$ that contains character $o.c$. If $s[o.p]$ fails to point to the right character $o.c$, we turn the color of n to red. Otherwise we decrement $n.counter$ by one. If $n.counter$ has been decremented this way more than N times, we also turn the color of that node to red.

2. Definitions of Key Concepts

As an analysis tool rather than a specific concurrency control algorithm, the graph maintenance algorithm in Figure 16 itself does not calculate the execution form of any operation in its execution state. Instead, the operation parameters (esp. position) are determined by the group editor (specifically its concurrency control algorithm). The graph detects possible inconsistent behavior of the group editor.

Definition 9 *An effect relation graph G is consistent iff G is acyclic and without red nodes.*

We use G_f to denote the consistent graph resulted after all generated operations have been invoked at all sites. Due to the algorithm in Figure 16, no node is ever deleted from the graph. Hence every character that ever appears in the system has a corresponding node in G_f. Let C_t be the set of characters that ever appear.

Definition 10 *For any given two characters $c_1, c_2 \in C_t$, we say $c_1 \prec c_2$ iff there exists a path from node n_1 to n_2 in G_f, where $n_1.char = c_1$ and $n_2.char = c_2$.*

In general relation $\prec$ may only be a partial order. For example, if 'x' is deleted from state "axb" and then 'y' is inserted to yield "ayb", it does not order characters 'x' and 'y' directly unless 'x' is recovered to be in the same state as 'y'. However, for any two characters $c_1, c_2 \in C_t$, if they ever appear in the same state of a group editor, their relation must have been determined by the group editor, or more specifically, its concurrency control (do or undo) algorithm. There must be two nodes that contain c_1 and c_2, respectively, in the graph and at least one path exists between them.

Definition 11 *A state s corresponds to a graph G iff (1) for every $c \in s$, there is one and only one node $n \in G$ such that $n.char = c$, and (2) for every node n in G, there is one and only one character $c \in s$ such that $c = n.char$.*

Definition 12 *A given state s is reachable iff s corresponds to a subgraph of G_f and for any two characters $c_1, c_2 \in s$, if $s[c_1] < s[c_2]$ then $c_1 \prec c_2$ holds.*

Corollary 4 *Given any reachable state s, its corresponding graph is a subgraph of G_f and consistent.*

Definition 13 *Any operation o is admissible in state $s =$ dst(o) iff s is reachable (or its corresponding graph G is consistent) and its invocation yields a consistent graph.*

Axiom 1 *The initial state s_0 is reachable.*

Axiom 2 *The graph constructed from s_0 is consistent.*

Assumption 1 *Any operation o is admissible in its generation state s if s is reachable.*

Definition 14 *An operation sequence sq is an ordered list of operations such that $sq[i+1]$ is defined in the state resulted from executing $sq[i]$, or dst($sq[i+1]$) = exec(dst($sq[i]$), $sq[i]$), where $0 \leq i < |sq| - 1$. In particular, state dst($sq[0]$) is called the definition state of sq, or dst(sq) = dst($sq[0]$).*

Let $s = \text{dst}(sq)$. Then sq is executed in s as follows: first execute $sq[0]$ in s, then execute $sq[1]$ in state $\text{exec}(s, sq[0])$, and so forth. We extend the notion of operation execution such that $s' = exec(s, sq) = exec(exec(s, sq[0]), sq[1, n-1])$, where $n = |sq|$ and $sq[1, n-1]$ is the subsequence of sq ranging from its second to last operation.

Definition 15 *Any operation sequence sq is admissible if every operation in sq is admissible (in its definition state).*

Lemma 16 *Given an operation sequence sq, let $s = dst(sq)$ and $s' = exec(s, sq)$. If s is reachable and sq is admissible, then s' is also reachable.*

Definition 16 *Given two sequences sq_1 and sq_2 defined in the same state s, or $s = dst(sq_1) = dst(sq_2)$, we say that they are effects equivalent, denoted by $sq_1 \sim sq_2$, iff their execution in s yields the same final state.*

3. Correctness of Group Editors

Definition 17 *Assume all sites start from the same initial state s_0, a group editor is correct if the following two criteria are always satisfied:*

1. Causality preservation: *For any operations o_1 and o_2, if $o_1 \rightarrow o_2$, then o_1 is executed before o_2 at any site.*
2. Admissibility Preservation: *The invocation of every operation is admissible in its execution state.*

Given a reachable initial state, the graph remains consistent as admissible operations are invoked, because no invocation introduces a new cycle or red node. As a result, in the quiescent state, in which all generated operations have been executed at all sites, the final graph G_f is consistent and relation $\prec$ is well-defined. That is, the invocation of every operation *eventually* preserves the effect relation $\prec$.

This model can be used to check the correctness of existing group editors and concurrency control algorithms, even if they do not explicitly have the notion of effects relation. Due to the causality-preserving process of graph construction, the order between characters is essentially established by the initial state and invocations of local operations in their generation states. A group editor is correct if the execution of any operation (in group do or undo) does not contradict the character order established earlier by itself, i.e., not causing inconsistencies in the graph by introducing cycles and red nodes. The existence of a cycle between any two nodes (say n_1 and n_2) in the graph means that $n_1.char$ precedes $n_2.char$ in some state while $n_2.char$ precedes $n_1.char$ in some other state. The existence of a red node n in the graph means that the character $n.char$ has been wrongly deleted. Either case often leads to divergence of states at different sites or the violation of the established character order even if the final states converge.

In Definition 17, we do not include an explicit convergence condition as in previous work [10, 16, 25, 5]. This is because convergence is implied by the two given conditions. Assuming that each site maintains a history of locally executed operations, the following theorem asserts that all replicas of the shared data converge in the quiescent state.

Theorem 3 *In a correct group editor, after all generated operations are executed at all sites, any two histories in the system are effects equivalent.*

Therefore, the key problem for any concurrency control algorithm in a group editor is to ensure that every remote operation is admissible in its execution state. Here we address this problem in the context of operational transformation. Section C first examines the sufficient conditions under which two operations commute while ensuring admissibility. Then Section D gives an OT-based concurrency control algorithm based on these sufficient conditions.

C. Transformation Functions and Conditions

Two operations o_1 and o_2 commute if the two different orders of their execution in the same state s yield the same state s'. That is, $s' = \mathrm{exec}(s, [o_1, o_2]) = \mathrm{exec}(s, [o_2, o_1])$, or simply $[o_1, o_2] \sim [o_2, o_1]$.

However, two operations may not commute if they are executed in different orders as they are. For example, let s=“abc” be the state in which o_1 = ins(1, ‘x’) and o_2 = ins(2, ‘y’) are defined. o_1 is to insert ‘x’ between ‘a’ and ‘b’ and o_2 is to insert ‘y’ between ‘b’ and ‘c’. If o_1 is executed first in s, we get $s_1 = \mathrm{exec}(s, o_1)$ = “axbc”. If we execute o_2 in state s_1 as it is, we get $s_{12} = \mathrm{exec}(s_1, o_2)$ = “axybc”. Similarly in the other execution order we get $s_{21} = \mathrm{exec}(s, [o_2, o_1])$ = “axbyc”. Obviously these two resulted states are not the same, or $[o_1, o_2] \not\sim [o_2, o_1]$.

The basic idea of operational transformation (OT) [10] is to transform operations such that almost any pair of operations commute. The problem is to determine the transformed versions of o_1 and o_2 (o_1' and o_2', respectively) such that $[o_1, o_2'] \sim [o_2, o_1']$. In the above example, o_1' = ins(1,‘x’), o_2' = ins(3,‘y’), and
exec($s, [o_1, o_2']$)=exec($s, [o_2, o_1']$)=“axbyc”.

We define three basic commute operators in Section 1. However, counterexamples show that these operators may not work correctly in some boundary cases [14]. Hence we study their sufficient conditions in Sections 4– 6.

1. Commute Operators

For any two operations o_1 and o_2, we say o_2 **depends on** o_1, iff o_2 deletes the character inserted by o_1, or more formally, $o_1.t = ins$, $o_2.t = del$, $o_1.c = o_2.c$ and $o_1 \to o_2$. Similar to [5], we say any two operations o_1 and o_2 are **contextually equivalent**, denoted as $o_1 \sqcup o_2$, iff $\mathrm{dst}(o_1) = \mathrm{dst}(o_2)$; they are **contextually serialized**, denoted as $o_1 \mapsto o_2$, iff $\mathrm{dst}(o_2) = \mathrm{exec}(s, o_1)$, where $s = \mathrm{dst}(o_1)$.

An operation o is a transformed version of o' if they are different invocations of the same operation in different states. Note only the position parameter of an operation is state-dependent. We define three binary commute operators, inclusion transformation (IT), exclusion transformation (ET), and SWAP, as follows.

Function 13 *IT(o_1, o_2): o_1'*

```
1  if $o_2.p < o_1.p$
2     if $o_2.t = \mathrm{ins}$
3        $o_1.p \leftarrow o_1.p + 1$;
4     else if $o_2.t = \mathrm{del}$
5        $o_1.p \leftarrow o_1.p - 1$;
6  else if $o_2.p = o_1.p$
7     if $o_1.t = \mathrm{del} \wedge o_2.t = \mathrm{ins}$
8        $o_1.p \leftarrow o_1.p + 1$;
9     else if $o_1.t = o_2.t = \mathrm{ins} \wedge o_1.id > o_2.id$
10       $o_1.p \leftarrow o_1.p + 1$;
11    else if $o_1.t = o_2.t = \mathrm{del}$
12       $o_1 \leftarrow \phi$;
13 return $o_1' \leftarrow o_1$;
```

Fig. 17. Including the effect of o_2 into o_1

The purpose of $o_1' = \mathrm{IT}(o_1, o_2)$, where $o_1 \sqcup o_2$, is to include the effect of o_2 into o_1 such that $o_2 \mapsto o_1'$. Figure 17 defines the basic rules of $\mathrm{IT}(o_1, o_2)$. Let $s = \mathrm{dst}(o_1)$ $= \mathrm{dst}(o_2)$ and $s' = \mathrm{exec}(s, o_2) = \mathrm{dst}(o_1')$. If $o_1.p > o_2.p$ and $o_2.t = \mathrm{ins}$, meaning that o_2 inserts a character on the left of the target position of o_1, we increment $o_1.p$ by one because the original position of o_1 has been shifted by the execution of o_2. If o_2 deletes a character on the left of the target position of o_1, or $o_1.p > o_2.p$ and $\mathrm{t}(o_2) =$ del, we decrement $o_1.p$ by one. If o_1 deletes at the same position as o_2 inserts (lines 7-8), we increment $o_1.p$ by one because the original character $o_1.c$ in s has been shifted by the insertion of $o_2.c$. For the two cases in lines 9 and 11, we define the following two policies:

1. If o_1 and o_2 insert at the same position in s, we compare their site ids to order $o_1.c$ and $o_2.c$ such that one with a smaller site id precedes the other (lines 9-10). Hence we increment $o_1.p$ if $o_2.id < o_1.id$.

2. If o_1 and o_2 attempt to delete the same character in s, denoted as $o_1.c \equiv o_2.c$, the one to be executed later is transformed into an identity operation ϕ so that the same object will not be deleted twice (lines 11-12).

Function 14 *ET(o_1, o_2): o_1'*

```
1   if $o_2.p < o_1.p$
2      if $o_2.t$ = ins
3         $o_1.p \leftarrow o_1.p - 1$;
4      else //if $o_2.t$ = del
5         $o_1.p \leftarrow o_1.p + 1$;
6   else if $o_2.p = o_1.p$
7      if $o_1.t = o_2.t$ = del
8         $o_1.p \leftarrow o_1.p + 1$;
9      else if $o_1.t$ = del $\wedge$ $o_2.t$ = ins
10        return exception;
11  return $o_1' \leftarrow o_1$;
```

Fig. 18. Excluding the effect of o_2 from o_1

The purpose of $o_1' = \mathrm{ET}(o_1, o_2)$, where $o_2 \mapsto o_1$, is to exclude the effect of o_2 from o_1 such that $o_1' \sqcup o_2$. Figure 18 defines the basic rules of ET. If $o_2.p < o_1.p$, meaning that o_2 inserted (or deleted) a character on the left of $o_1.c$, we decrement (or increment) $o_1.p$ by one to exclude the effect of o_2 from o_1. If o_1 and o_2 are both deletes and $o_1.p = o_2.p$, it means o_2 deleted the character that immediately preceded $o_1.c$, because they delete in a tandem in state s, and hence we increment $o_1.p$ by one to exclude the effect of o_2. Otherwise if o_1 deletes the character inserted by o_2, meaning $o_1.c \equiv o_2.c$, it does not make sense logically to exclude the effect of o_2 from o_1 and we raise an exception.

Function 15 *SWAP(o_1, o_2): $< o'_1, o'_2 >$*

```
if $o_1.p > o_2.p$
   if $o_2.t$ = ins
      $o_1.p \leftarrow o_1.p - 1$;
   else //$o_2.t$ = del
      $o_1.p \leftarrow o_1.p + 1$;
else if $o_1.p = o_2.p$
   if $o_1.t = o_2.t$ = del
      $o_1.p \leftarrow o_1.p + 1$;
   else if $o_1.t$ = del $\wedge$ $o_2.t$ = ins
       return exception;
   else if $o_1.t = o_2.t$ = ins
      $o_2.p \leftarrow o_2.p + 1$;
   else //$o_1.t$ = ins $\wedge$ $o_2.t$ = del
      $o_2.p \leftarrow o_2.p + 1$;
else //$o_1.p < o_2.p$
   if $o_1.t$ = ins
      $o_2.p \leftarrow o_2.p + 1$;
   else //$o_1.t$ = del
      $o_2.p \leftarrow o_2.p$ - 1;
return $< o_1, o_2 >$;
```

Fig. 19. SWAP $[o_2, o_1]$ into $[o'_1, o'_2]$

The third commute operator SWAP(o_1, o_2) is to transpose two operations o_2, o_1, where $o_2 \mapsto o_1$, into o'_1, o'_2, such that $o'_1 \mapsto o'_2$ and $[o_2, o_1] \sim [o'_1, o'_2]$. Conceptually this amounts to first processing $o'_1 = \text{ET}(o_1, o_2)$ to get $o'_1 \sqcup o_2$ and then $o'_2 = \text{IT}(o_2, o'_1)$ to get $o'_1 \mapsto o'_2$. As will be shown later, IT and ET have different sufficient conditions. This way the correctness of SWAP will depend on both IT and ET. Observe, however, the relation between $o_1.c$ and $o_2.c$ is known after ET is processed. Hence we can merge the rules of ET and IT to define SWAP, as shown in Figure 19, and then the correctness condition of SWAP is equivalent to that of ET.

2. Problems and Analysis

Our definitions of IT and ET are similar to most definitions in the literature, including [10, 16, 5, 33], in that they all only use the basic parameters of an operation o such as $o.p$, $o.id$, and $o.tp$. These definitions are intuitive and straight-forward. However, IT and ET as defined are not always able to work correctly.

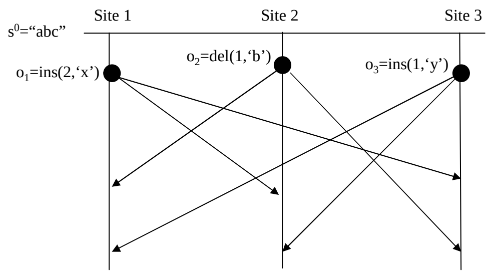

Fig. 20. A well-known scenario that suggests problems in previous commutativity conditions.

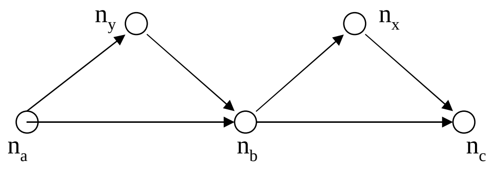

Fig. 21. G' after the local invocations of o_1, o_2, o_3.

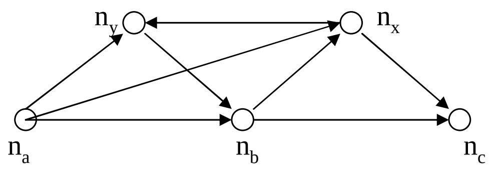

Fig. 22. G'' is cyclic (inconsistent) after invoking o_1', o_3'' at site 2.

The following example illustrates a well-known scenario that first appeared in [25, 5]. Suppose three sites start from the same initial state s^0 = "abc". Three concurrent operations o_1 = ins(2, 'x'), o_2 = del(1, 'b') and o_3 = ins(1, 'y') are generated, as shown in Figure 20.

Let G be the effect relation graph constructed from s^0. Hence G only contains three nodes n_a, n_b and n_c, corresponding to 'a', 'b' and 'c', respectively. The local invocations of o_1, o_2, o_3 yield G' as shown in Figure 21, where n_x and n_y correspond to 'x' and 'y', respectively.

Here we only consider site 2. The reader can similarly try the invocations of operations at other sites. After o_2 is generated and invoked locally, the state of site 2 is s_2^1 = "ac". Then o_1 and o_3 are received and invoked in a tandem. First we get o_1' = IT(o_1, o_2) = ins(1,'x'), and s_2^2 = exec(s_2^1, o_1') = "axc". Invoking o_1' at site 2 adds edge $< n_a, n_x >$ to G'. Next we have o_3' = IT(o_3, o_2) = ins(1,'y') and o_3'' = IT(o_3', o_1') = ins(2,'y'). The execution of o_3' in state s_2^2 yields s_2^3 = "axyc". Invoking o_3'' at site 2 adds edge $< n_x, n_y >$ to G', which yields a cycle as shown in Figure 22. Obviously o_2 and o_1' are admissible but o_3'' is not.

It is easy to verify that the uses of IT in the above scenario satisfy the precondition, contextual equivalence, as is defined in [11, 5]. Hence this well-accepted precondition is not sufficient. Similar problems exist with the well-accepted precondition, contextual serialization, of ET [11, 5], as shown in [19, 14]. In Section 4 and 5, we will identify a set of sufficient conditions of IT and ET, respectively.

3. Deciding the Effect Relation

To determine the relation between any two given objects, if we know whether or not there exists at least one object between them, the problem becomes more straightforward.

Definition 18 [Landmark Object] *For any three objects $x_1, x_2, x_3 \in C_t$, we say that x_3 is a landmark object between x_1 and x_2, iff either $x_1 \prec x_3 \prec x_2$ or $x_2 \prec x_3 \prec x_1$.*

Let C_t be the set of objects that ever appear in a group editing session. We denote the set of landmark objects between x_1 and x_2 as $C_{ld}(x_1, x_2) = \{x_3 \in C_t | x_1 \prec x_3 \prec x_2 \vee x_2 \prec x_3 \prec x_1\}$. It is abbreviated as C_{ld} when there is no confusion in the context. Since it is the existence of C_{ld} that matters in determining the relation of x_1 and x_2, we do not really need to compute the whole set of C_{ld}.

To be specific, for any two insertions o_1 and o_2, we decompose $C_{ld}(o_1.c, o_2.c)$ into two complementary object sets, C_1 and C_2, where C_1 consists of the effect objects of operations that happened before neither o_1 nor o_2, and C_2 includes the effect objects of operations that happened before either o_1 or o_2. More formally, $C_1 = \{o.c \in C_t | o.v \not< max(o_1.v, o_2.v)\}$, and $C_2 = C_t$ - C_1 - $\{o_1.c, o_2.c\}$.

Lemma 17 *For any two insertions o_1 and o_2, objects in $C_1 = \{o.c \in C_t | o.v \not< max(o_1.v, o_2.v)\}$ is not necessary for determining the relation between $o_1.c$ and $o_2.c$.*

Proof. Let $o.c$ be any object in C_1. By definition we have $o \not\to o_1$ and $o \not\to o_2$. Not violating causality, the execution of o, o_1 and o_2 at any site i could be in one of the following three orders: (1) o is executed before both o_1 and o_2, (2) o is executed after both o_1 and o_2, or (3) o is executed between o_1 and o_2. Without loss of generality, assume that o_1 is executed before o_2. In the execution path of case (2), when o_2 is executed, site i does not even know the existence of o only by its local information. Hence the correct execution of o_2 should not have relied on o. That is, the relation between $o_1.c$ and $o_2.c$ can be determined without knowledge of o and thus C_1 is not necessary in this case. In cases (1) and (3), due to causality preservation, the effect relation of o_1 and o_2 can be correctly determined only by C_2. Hence the information carried in o is redundant, meaning C_1 is also not necessary for determining the relation of o_1 and o_2 in these two cases. ■

Due to Lemma 17, in the remainder of this chapter, we always use landmark objects that appear in C_2 for $C_{ld}(o_1.c, o_2.c)$. This effectively narrows down the scope of landmark object set and eases proofs.

4. Conditions of IT

Theorem 4 *Given $o_1' = IT(o_1, o_2)$, where $o_1 \sqcup o_2$ and they are both defined in s, then o_1' is admissible in $s' = exec(s, o_2)$, if o_1 and o_2 are admissible in s and one of the following conditions holds:*

1. $o_1.p \neq o_2.p$
2. $(o_1.p = o_2.p) \wedge ((o_1.t = ins \wedge o_2.t = del) \vee (o_1.t = del \wedge o_2.t = ins) \vee (o_1.t = o_2.t = del))$
3. $(o_1.p = o_2.p) \wedge (o_1.t = o_2.t = ins) \wedge (o_1 \parallel o_2) \wedge C_{ld}(o_1.c, o_2.c) = \emptyset$

This theorem is proved by the following three lemmas.

Lemma 18 *o_1' is admissible in s' if condition (1) holds.*

Proof. Without loss of generality, assume $o_2.p < o_1.p$. We need to prove four cases: (1) $o_1.t = o_2.t = ins$, (2) $o_1.t = del \wedge o_2.t = ins$, (3) $o_1.t = ins \wedge o_2.t = del$, and (4) $o_1.t = o_2.t = del$. For space reasons, here we only prove (1). Proofs of other cases are similar.

Let G is the consistent graph corresponding to s. Due to $o_2.p < o_1.p$, we have $o_2.p - 1 < o_2.p \leq o_1.p - 1 < o_1.p$. Consider the four characters $c_1 = s[o_2.p - 1]$, $c_2 = s[o_2.p]$, $c_3 = s[o_1.p - 1]$, and $c_4 = s[o_1.p]$. Their relation is either $c_1 \prec c_2 \prec c_3 \prec c_4$ or $c_1 \prec c_2 \equiv c_3 \prec c_4$.

Since o_2 is admissible in s, the invocation of o_2 on G produces a consistent graph G' and s' is also reachable. It introduces a new node containing $o_2.c$ and two new edges if they are not in G yet: one from c_1 to $o_2.c$ and the other from $o_2.c$ to c_2. This does not change the order between c_1, c_2, c_3, and c_4. Relative to s', however, because of the insertion of $o_2.c$, we have $c_1 = s'[o_2.p - 1]$, $o_2.c = s'[o_2.p]$, $c_2 = s'[o_2.p + 1]$, $c_3 = s'[o_1.p]$, and $c_4 = s'[o_1.p + 1]$. Their relation is either $c_1 \prec o_2.c \prec c_2 \prec c_3 \prec c_4$ or $c_1 \prec o_2.c \prec c_2 \equiv c_3 \prec c_4$.

By the definition of IT in Fig. 17, we have $o'_1.p = o_1.p + 1$. Hence $c_3 = s'[o'_1.p - 1]$, and $c_4 = s'[o'_1.p]$. The invocation of o'_1 on G' produces graph G'', which introduces a new node containing $o_1.c$ and two new edges (if the node and edges are not in G yet): one from c_3 to $o_1.c$ and the other from $o_1.c$ to c_4. Their relation is either $c_1 \prec o_2.c \prec c_2 \prec c_3 \prec o_1.c \prec c_4$ or $c_1 \prec o_2.c \prec c_2 \equiv c_3 \prec o_1.c \prec c_4$. No new cycle is introduced by o'_1. Hence G'' is also consistent, which means that o'_1 is admissible in s'. ■

Lemma 19 *o'_1 is admissible in s' if condition (2) holds.*

Lemma 20 *o'_1 is admissible in s' if condition (3) holds.*

Proof. Since the assertion must hold in spite of specific execution order of concurrent operations, we consider a simple case in which the effects of all operations concurrent with both o_1 and o_2 are not in s.

Let G be the consistent graph corresponding to state s, $c_a = s[o_2.p - 1]$, and $c_b = s[o_2.p]$. The invocation of o_2 on G yields a consistent graph G', which introduces a new node containing $o_2.c$ and two new edges (if they are not in G yet): one from c_a to $o_2.c$ and the other from $o_2.c$ to c_b. Due to $o_1.p = o_2.p$, the invocation of o_1 in G would similarly yield a consistent graph with a new node containing $o_1.c$ and two new edges: one from c_a to $o_1.c$ and the other from $o_1.c$ to c_b. Now invoke o_1 on G' and add node $o_1.c$ and edges $< c_a, o_1.c >$ and $< o_1.c, c_b >$ first without considering the relation between $o_1.c$ and $o_2.c$. Let the resulting graph be G''.

Due to the previous analysis, we know that the landmark characters in the graph are not always present in the current execution state s. However, if we can somehow find a transformation path such that state s includes all possible landmark characters, namely, $C_{ld}(o_1.c, o_2.c) \subseteq s$, deciding the relation between $o_1.c$ and $o_2.c$ becomes straightforward. Specifically, if $C_{ld}(o_1.c, o_2.c) \neq \emptyset$, there must be at least one landmark character between $o_1.c$ and $o_2.c$ in s and G. This implies $o_1.p \neq o_2.p$, which is the case of condition (1). On the other hand, if $C_{ld}(o_1.c, o_2.c) = \emptyset$, the node of $o_1.c$ and the node of $o_2.c$ are unorderable in G''. Then only adding an edge between node $o_1.c$ and node $o_2.c$ introduce no new cycle in G''. According to the IT function in Fig. 17, if $o_1.id < o_2.id$, an edge $< o_1.c, o_2.c >$ is added; or otherwise $< o_2.c, o_1.c >$ is added. ■

5. Conditions of ET and SWAP

Theorem 5 *Given o_1' = ET(o_1, o_2), where $o_2 \mapsto o_1$ and o_2 is defined in s, then o_1' is admissible in s, if o_2 is admissible in s, o_1 is admissible in s' = exec(s, o_2), and one of the following conditions holds:*

1. $o_1.p \neq o_2.p$
2. $(o_1.p{=}o_2.p) \wedge (o_1.t{=}o_2.t{=}ins)$
3. $(o_1.p{=}o_2.p) \wedge (o_1.t{=}o_2.t{=}del)$
4. $(o_1.p{=}o_2.p) \wedge (o_1.t{=}ins) \wedge (o_2.t{=}del) \wedge (o_2 \rightarrow o_1)$

Lemma 21 *o_1' is admissible in s if condition (1) holds.*

Lemma 22 *o_1' is admissible in s if condition (2) holds.*

Proof. By the ET rules in Fig. 18, $o_1'.p = o_1.p$ in state s. Let G be the consistent graph corresponding to s. The invocation of o_1' on G yields graph G', in which a new node containing $o_1.c$ and two new edges are introduced (if not in G yet): one from $s[o_1.p - 1]$ to $o_1.c$ and the other from $o_1.c$ to $s[o_1.p]$. This does not generate a cycle given G is acyclic. Hence G'' is consistent and o'_1 is admissible in s. ∎

Lemma 23 *o'_1 is admissible in s if condition (3) holds.*

Lemma 24 *o'_1 is admissible in s if condition (4) holds.*

Proof. By the ET rules in Fig. 18, $o'_1.p = o_1.p$ in state s. Let G be the consistent graph when o_1 has only been invoked once locally. Then the two nodes in G containing $o_1.c$ and $o_2.c$, respectively, are not ordered in G. Since G is acyclic, addition of either $< o_1.c, o_2.c >$ or $< o_2.c, o_1.c >$ in G does not introduce any new cycle. When o'_1 is any remote invocation, if we take a consistent policy here to add the edge between $o_1.c$ and $o_2.c$, no cycle will result. Therefore o'_1 is admissible. ∎

6. Conditions of Reordering Sequences

Definition 19 **[Causality-Preserving Sequence]** *An operation sequence sq is causality-preserving iff for any two operations $sq[i]$ and $sq[j]$, where $0 \leq i, j < |sq|$, if $sq[i] \rightarrow sq[j]$, then $i < j$ must hold.*

In a causality-preserving sequence sq, for any $o \in sq$, all operations in sq that happened before o must precede o.

Definition 20 **[Dependency-Preserving Sequence]** *An operation sequence sq is dependency-preserving iff for any two operations $sq[i]$ and $sq[j]$, where $0 \leq i, j < |sq|$, if $sq[j]$ depends on $sq[i]$, then $i < j$ must hold.*

In a dependency-preserving sequence sq, any operation that depends on any o must follow o in sq. The cause-effect relation is only preserved between pairs that have dependencies. Hence a causality-preserving sequence must also be dependency preserving, but not vice versa.

Definition 21 [**ID Sequence**] *Given a sequence sq, it is an ID sequence, iff* $\forall i, j : 0 \leq i, j < |sq|$, $i < j$ *if either (1)* $sq[i].t = ins \wedge sq[j].t = del$*, or (2)* $sq[i].t = sq[j].t \wedge sq[i] \rightarrow sq[j]$*.*

Notation $sq_1 \bullet sq_2$ returns the concatenation of two sequences and $sq \bullet o$ appends an operation o to the end of sequence sq. By definition, an ID sequence sq is the concatenation of an insertion subsequence sq_i and a deletion subsequence sq_d, or $sq = sq_i \bullet sq_d$. Note that both sq_i and sq_d are causality-preserving; sequence sq is dependency-preserving but not necessarily causality-preserving.

Definition 22 [**HC and IHC Sequences**] *Given an operation o and a sequence sq, we say that sq is an HC sequence with regard to o, iff* $\forall i, j : 0 \leq i, j < |sq|$, $i < j$ *if either (1)* $sq[i] \rightarrow o \wedge sq[j] \parallel o$*, or (2)* $sq[i] \rightarrow sq[j]$*. If all operations in an HC sequence sq are insertions, we say that sq is an IHC sequence.*

By definition, an HC sequence sq w.r.t. o is a concatenation of two subsequences $sq = sq_h \bullet sq_c$, where sq_h includes all operations in sq that happened before o and sq_c includes all operations concurrent with o. Note sq, sq_h and sq_c are all causality-preserving sequences.

Theorem 6 *Given an admissible sequence* $sq \bullet o$*, where sq is an ID sequence and o is a local invocation, there must exist an ID sequence* sq' *such that* $(sq \bullet o) \sim sq'$*.*

Proof. The fact that o is a local invocation implies all operation in sq happened before o. If o is a deletion, we get sq' by appending o to sq, or $sq' = sq \bullet o$. Since sq is an ID sequence, we rewrite $sq = sq_i \bullet sq_d$, where sq_i is an insertion sequence and sq_d

is a deletion sequence. If o is an insertion, we first swap o with every operation in sq_d from right to left, yielding sq'_d and o' as the result, and then get $sq' = sq_i \bullet o' \bullet sq'_d$. By Theorem 5, this is correct. ■

Theorem 7 *Given an admissible operation o and an admissible, causality-preserving sequence sq, where all operations in sq are insertions, there must exist an IHC sequence sq' with regard to o such that $sq \sim sq'$.*

Function 16 convert2IHC(o, sq): sq'

```
$sq_h \leftarrow \emptyset$; $sq_c \leftarrow \emptyset$;      // empty sets
for($j = 0$; $j < |sq|$; $j++$)
    if $sq[j] \parallel o$
        $sq_c \leftarrow sq_c \bullet sq[j]$;
    else //if $sq[j] \rightarrow o$
        for($k = |sq_c| - 1$; $k \geq 0$; $k--$)
            $<sq[j], sq_c[k]> \leftarrow$ SWAP($sq[j]$, $sq_c[k]$);
        $sq_h \leftarrow sq_h \bullet sq[j]$;
return $sq' \leftarrow sq_h \bullet sq_c$;
```

Fig. 23. To convert a causality-preserving insertion sequence into an IHC sequence

Proof. Fig. 23 gives an algorithm for finding the required sq'. The algorithm scans the input sequence say sq from left to right and appends every operation concurrent with o to sequence sq_c. Every operation that happened before o is swapped with sq_c and the result is appended to sequence sq_h. Obviously if the input sq is causality preserving, then sq_h and sq_c are each causality preserving. Since sq is an insertion sequence, SWAP is only used between insertions in the algorithm. By Theorem 5, this is correct. ■

D. Integration Algorithm

The conditions in Theorems 4, 5, 6 and 7 are sufficient because the assertions always hold once they are satisfied. This section gives algorithms for executing local and remote operations. The main idea is to ensure every operation is admissible in its execution state, by satisfying these sufficient conditions while avoiding conditions that are not included.

We consider a group editor of N sites that start from the same initial state s_0. For simplicity, we assume a reliable network environment without network partition or node failure. Each site maintains a queue RQ to store remote operations received from other sites. To ensure causality preservation, a remote operation is invoked only when it is causally ready [10, 5]: Given a remote operation o generated at site i, o is causally ready at site j, iff (1) $o.v[i] = sv_j[i] + 1$ and (2) for $\forall k \neq i : o.v[k] \leq sv_j[k]$.

Each site maintains an operation log H, which is an ID sequence. H is often rewritten as $H = H_i \bullet H_d$, where H_i includes all the insertions and H_d all the deletions.

Figure 24 gives the control algorithm at site i. Every time a local operation o is generated, we always execute it directly in its generation state $s = \text{exec}(s_0, H)$. All operations in H happened before o and o is admissible in s. After o is executed, we call function updateHL(o, H) to compute its admissible form o' in state $s' = \text{exec}(s_0, H_i)$, the state just after only all insertions (that happened before o) have been invoked in s_0. Then we propagate o' to remote sites.

We scan RQ from left to right for the first causally-ready remote operation o. Then o is removed from RQ and function updateHR(o, H) is called to compute its admissible form o' in current state s=exec(s_0, H). Then if o' is not ϕ, we execute o' in s. In fact, o' is ϕ only when two sites concurrently attempt to delete the same character. In this case, to avoid the appearance of red nodes in the (conceptual)

```
Initialize:
1   $RQ \leftarrow \emptyset$; $H \leftarrow \emptyset$;
2   $sv \leftarrow < 0, 0, ..., 0 >$

Invoke (generate) a local operation o:
1   execute $o$;
3   $sv[i] \leftarrow sv[i] + 1$;
4   $o.v \leftarrow sv$;
5   $< o', H > \leftarrow$ updateHL($o$, $H$);
6   broadcast $o'$ to other sites;

Receive o from network:
1   $RQ \leftarrow RQ \bullet o$;

Invoke a remote operation:
1   if $\exists RQ[k]$ that is causally-ready
2       $o \leftarrow RQ[k]$; remove $RQ[k]$ from $RQ$;
3       $< o', H > \leftarrow$ updateHR($o$, $H$);
4       if( $o' \neq \phi$ ) execute $o'$;
5       $sv[j] \leftarrow sv[j]$+1 where $j = o.id$;
```

Fig. 24. The control algorithm at site i

effect relation graph, the deletion to be invoked later is transformed into an identity operation ϕ and discarded.

Figure 25 defines function updateHL(o, H), which serves two purposes: (1) to compute the admissible form of o in state $s' = \text{exec}(s_0, H_i)$, and (2) to add o into H. We know $H = H_i \bullet H_d$ is an ID sequence and o is generated in state $s = \text{exec}(s_0, H)$. Hence its admissible form o' relative to s' can be computed by swapping o from right to left with every operation in H_d. Then if $o.t = del$, we append the original form o to H; or if $o.t = ins$, we append o' to H_i. This is actually the algorithm used in the proof of Lemma 6.

Figure 26 shows function updateHR(o, H), which serves two purposes: (1) to compute the admissible form of o in current state $s = \text{exec}(s_0, H)$, and (2) to add o into H. When a remote operation o is received, according to function updateHL(),

Function 17 updateHL(o, H): $< o', H' >$

```
1  $o' \leftarrow o$; $L \leftarrow H$;
2  for$(i = |H_d| - 1; i \geq 0; i - -)$
3      $< o', H_d[i] > \leftarrow$ SWAP($o'$, $H_d[i]$);
4  if $o.t$ = del
5      $H \leftarrow L \bullet o$;
6  else // $o.t$ = ins
7      $H \leftarrow H_i \bullet o' \bullet H_d$;
8  return $< o', H >$;
```

Fig. 25. To add a local operation o into H and make o' admissible in $s' = exec(s_0, H_i)$

Function 18 ITSQ(o, sq): o'

```
1  for($i$=0; $i < |sq|$; $i + +$)
2      $o \leftarrow$ IT$(o, sq[i])$;
3      if( $o = \phi$ ) break;
4  return $o' \leftarrow o$;
```

Function 19 updateHR(o, H): $< o', H' >$

```
1  $H_i \leftarrow$ convert2IHC$(o, H_i)$;      // $H_i = H_{ih} \bullet H_{ic}$
2  $o'' \leftarrow$ ITSQ($o$, $H_{ic}$);
3  $o' \leftarrow$ ITSQ($o''$, $H_d$);      // $H = H_{ih} \bullet H_{ic} \bullet H_d$
4  if $o' \neq \phi$ and $o.t$ = del
5      $H \leftarrow H \bullet o'$;
6  else if $o.t$ = ins
7      $o_x \leftarrow o''$;      // to protect $o''$ for line 12
8      for$(k = 0; k < |H_d|; k + +)$
9          $o_y \leftarrow o_x$;      // to protect $o_x$ for line 11
10         $o_x \leftarrow$ IT$(o_x, H_d[k])$;
11         $H_d[k] \leftarrow$ IT$(H_d[k], o_y)$;
12     $H \leftarrow H_i \bullet o'' \bullet H_d$;
13 return $< o', H >$;
```

Fig. 26. To add a remote operation o into H and make o' admissible in $s = \text{exec}(s_0, H)$.

it must have excluded the effects of all deletions that happened before o and have included the effects of all insertions that happened before o. However, o must have not included the effects of any operations in H_d and may not have included the effects of all operations in H_i, because H_i may contain insertions concurrent with o.

Therefore we first call function convert2IHC(o, H_i) to transpose H_i into an IHC sequence $H_{ih} \bullet H_{ic}$ with regard to o. Due to causality preservation and function updateHL(), o must be admissible in state exec(s_0, H_{ih}). In line 2 we transform o such that the result o'' is admissible in state $s' =$ exec(s_0, H_i), by calling $o'' =$ ITSQ(o, H_{ic}) to include the effects of H_{ic} into o. Then in line 3 we perform $o' =$ ITSQ(o'', H_d) such that o' is admissible in current state s.

After that, we add o into H, as follows: If o is a deletion, we append o' to H directly if it is not ϕ. However, if o is an insertion, we add o'' between H_i and H_d (line 12) and thus have to include the effect of o'' into every operation in H_d (lines 7-11). To achieve the latter, it is not enough to just do IT in line 11. In the loop of lines 8-11, initially we have $o'' \sqcup H_d[0]$. For every k, since $o_x \sqcup H_d[k]$, we include the effect of $H_d[k]$ into o_x and, at the same time, the effect of o_x into $H_d[k]$ such that the resulted o_x satisfies $H_d[k+1] \sqcup o_x$ and next time we still inclusively transform two context equivalent operations $H_d[k+1]$ and o_x.

1. Analysis and Proof

The time to invoke a local operation is dominated by the execution of updateHL(), which is obviously $O(|H_d|)$. The time to invoke a remote operation is dominated by function updateHR(). Inside function updateHR(), lines 2 and 3 (8-11) take time $O(|H_{ic}|)$ and $O(|H_d|)$, respectively. By Figure 23, the worst-case and expected execution time of convert2IHC(o, H_i) is $O(|H_i|^2)$. Since $H_i = H_{ih} \bullet H_{ic}$, the time to invoke a remote operation is $O(|H_i|^2 + |H_d|)$.

Lemma 25 *Given an admissible operation o and an admissible ID history $H = H_1 \bullet H_2$, where (1) o and H_2 are contextually equivalent, (2) H_1 is an insertion sequence, and (3) all operations in H that happened before o are in H_1, then $o' =$ ITSQ(o, H_2) is admissible.*

Proof. Let $|H_2| = n$. The assertion trivially holds when $n = 0$. We prove that o' is admissible by induction on $n > 0$.

Base: $n = 1$. Let $o' = \mathrm{IT}(o,\ H_2[0])$. We only consider the case in which $o.p = H_2[0].p$ and $o.t = H_2[0].t =$ ins because other cases are easy to prove. Because all operations in H_1 are insertions, no character has been deleted in dst(o). Characters in $C_{ld}(o.c, H_2[0].c)$ must be all in dst(o) if they exist at all. By Lemma 20, o' is admissible.

Induction: $n = k \rightarrow k + 1$. Assume that $o^k =$ ITSQ(o, $H_2[0, k-1]$), where $k > 0$, is admissible. Let $o' = \mathrm{IT}(o^k, H_2[k])$. Similarly here we only consider the case of $o.t = H_2[k].t =$ins and $o.p = H_2[k].p$. Because $H_1 \bullet H_2$ is an ID sequence and $sq[k].t$ = ins, then $H_1 \bullet H_2[0, k-1]$ must be an insertion sequence. Hence $C_{ld}(o^k.c, H_2[k].c)$ must be all in state dst(o^k) if they exist at all. By Lemma 20, the result o' is admissible. ■

Lemma 26 *The control algorithm ensures that the invocation of any operation is admissible.*

Proof. We prove this assertion by induction on n invocations at any site i. When $n = 1$, it is trivially true. Assume the first k invocations are admissible at site i . Consider the $(k+1)^{th}$ invocation o in the following two cases.

First, o is a local operation. By Assumption 1, o is admissible in current state. To ensure the correctness of performing forthcoming remote operations, we maintain H as an ID sequence and call function updateHL(o, H) to reorder $H \bullet o$ into another

ID sequence, the correctness of which is ensured by Lemma 6. Therefore both the updated H and propagated o' are admissible.

Second, a remote operation o generated at site j is invoked at site i, where $i \neq j$. Function updateHR(o, H) is called to make it admissible in the current state s. In updateHR(), first H_i is converted into an IHC sequence $H_{ih} \bullet H_{ic}$ with regard to o. Lemma 7 ensures the correctness of this step. Hence we have $H = (H_{ih} \bullet H_{ic}) \bullet H_d = H_{ih} \bullet (H_{ic} \bullet H_d)$, The next step of updateHR() is to compute $o' =$ ITSQ(o, $H_{ic} \bullet H_d$). Due to causality preservation and function updateHL(), o is admissible in state exec(s_0, H_{ih}). Hence o and H_{ic} are contextually equivalent. By Lemma 25, o' is admissible in s. ■

2. Examples

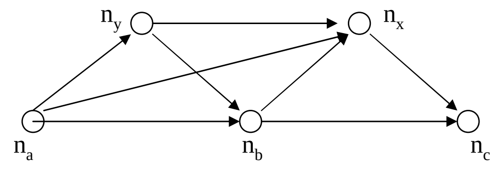

Fig. 27. A consistent graph corresponding to the example shown in Figure 20.

We use the example in Fig. 20 to illustrate how our approach works.

Site 1: When o_2 arrives, we compute $o_2^{s_1^1} = \text{IT}(o_2,\ o_1) =$ del(1, ‘b’). After o_3 arrives, $H_1 = [o_1, o_2^{s_1^1}]$. We get $o_3^{s_1^2} =$ ins(1, ‘y’) and update H_1 to $[o_1, o_3^{s_1^1}, o_2^{s_1^2}]$, where $o_3^{s_1^1} =$ ins(3,‘y’) and $o_2^{s_1^2} =$ del(2,‘b’). The execution of $o_3^{s_1^2}$ leads to state $s_1^3 =$ “ayxc”.

Site 2: When o_1 arrives, $H_2 = [o_2]$. By updateHR(), we get $o_1^{s_2^1} =$ ins(1, ‘x’) and $o_2^{s_2^{1'}} =$ del(1, ‘b’). As a result, $H_2 = [o_1, o_2^{s_2^{1'}}]$. When o_3 arrives, we get $o_3^{s_2^2} =$ ITSQ(o_3, H_2) = ins(1, ‘y’). The resulting state $s_2^3 =$“ayxc”.

Site 3: Similarly we get $o_1^{s_3^1} =$ ins(3, ‘x’) and $o_2^{s_3^2} =$ del(2, ‘b’). The final state is $s_3^3 =$ “ayxc”.

Obviously, after the three operations are executed at all three sites, the system converges. To check whether the system is consistent, we need to check the corresponding effect relation graph $G =< V, E >$. Initially, V contains three nodes n_a, n_b and n_c, and $V = \{(n_a,n_b), (n_b,n_c)\}$.

At first consider the three invocations at site 1: o_1 leads to a new node n_x and two new edges, (n_b,n_x) and (n_x,n_c); $o_2^{s_1^1}$ does not change the vertexes and edges of G; and $o_3^{s_1^2}$ leads to a new node n_y and two new edges (n_a,n_y), (n_y,n_x). After these invocations at site 1, G is still consistent. At site 2, the local invocation of o_2 does not change G. The invocations of $o_1^{s_2^1}$ and $o_3^{s_2^2}$ generate one new edge (n_a,n_x). Similarly, the three invocations at site 3 add one new edge, (n_y,n_b). After these operations have been executed at all sites, G as shown in Fig. 27 is still consistent.

CHAPTER V

A GENERIC CONSISTENCY MODEL

A. Motivation

In the last two chapters, we have established two consistency models, the CE model and the CA model. However, both of them are bound to a specific OT-based protocol with assumptions of characterwise do operation and linear string underlying data structure. As a result, two corresponding frameworks that are developed based on them are of little value for design of protocols that handle undo operations or stringwise operations. In this chapter, we start to establish a generic model independent of specific details of OT-based protocols and process an indepth discussion for the correctness conditions of protocols based on the generic model.

We require that a generic consistency model should observe the following principles:

- *Completeness:* The model must completely describe the system behavior, including its operations, states, and state transitions.
- *Disambiguity:* The model must unambiguously specify correctness conditions as what system behavior is legal and what is not.
- *Generality:* The model must be formalized independently of specific details (e.g., data structures) and characteristics (e.g., synchrony and optimism) of protocols that implement the model.
- *Practicability:* To be useful in practice, the model must provide feasible guidelines as how to design consistency control protocols that fulfill the model.

Unfortunately, no consistency model proposed for interactive groupware follows all the above principles to our knowledge. Those established in traditional distributed systems [27] in general fail to address human factors and are long found unsuitable for groupware [10, 6, 5]. Among those proposed in the groupware field, Sun et al. [5, 31] includes conditions that are not well-formalized for verification purposes; Ressel et al. [16] formalizes conditions but does not address how to satisfy them; The CE and the CA models are not general because they are tightly coupled with specific protocols. As a consequence, discovering and fixing bugs in consistency control protocols have been a main driver of research [16, 17, 5, 11, 14]. This calls for a more general consistency model that can serve as a guideline for the design and evaluation of a variety of consistency control protocols for interactive groupware applications.

B. A Generic Consistency Model

In this section we formalize a consistency model based on the observable behavior of an abstract group editor. We first examine its local behavior and then extend to its global behavior. In the next section we elaborate guidelines for designing consistency control protocols based on this model in the context of OT-based group editors.

1. Abstraction: I/O Machine

Assume that a group editor has a number of cooperating sites that are connected by a computer network and coordinate themselves by message passing. We also assume that all sites start from the same initial state and user-initiated operations are the driver of state transitions in the system. It is not interesting in this chapter whether the system is synchronous or asynchronous, consistency control is pessimistic or optimistic, and how the shared data is replicated.

As shown in Figure 28, the consistency control module is abstracted as a black-

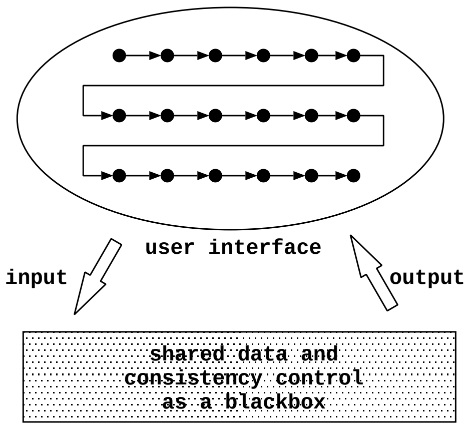

Fig. 28. The consistency control module as an I/O machine that translates user inputs into observable outputs that change view states at each site.

box that translates input operations into observable output operations. Inside the blackbox are protocol-specific data structures and algorithms. The module takes input operations and executes them somehow on the internal data structures. The execution produces observable effects on the user interface, which are abstracted as output operations. Conceptually the input is to tell the system what to do and the output is the user interface effects that reflect what system actually does.

As in Figure 28, we are interested in visual objects (filled circles) on the user interface and their relationship (arrows). Objects here can be anything interesting that appear on the user interface, texts and graphics, in a granularity of interest. On a typical 2D, 2.5D or 3D interface, we define the relationship between objects as their spatial order, e.g., left to right, top to bottom, and front to rear. At any moment, it is obvious that no object can appear on the left of another object and on its right at the same time. Hence the relationship is acyclic and can be defined as a linear total order. To simplify discussions, we use a string-like structure to represent visual objects and their relationship.

Note that the object relationship in our abstraction does not carry application semantics. For example, consider a diagram editor which displays two nodes connected by two links which form a semantic cycle. Mapping this case to our model, these two links are also visual objects. Hence the resulting graph has four objects and their relationship can still be defined in a total order.

We use notation $\mathcal{E}$ to denote the group editing system in question. Each site runs a copy of the consistency control module. It is the consistency control modules' responsibility to keep the local views at all sites globally "consistent". For this purpose, we need to first define view states and state transitions in the system.

Assume that each visual object that ever appears in the system has a unique id. We denote the set of objects by notation $\mathcal{X}(\mathcal{E})$. A (view) state includes all objects and their relationship that appear at one site at one moment. Notation $\mathcal{S}(\mathcal{E})$ denotes all states in the system at all times. We use string s to denote a state and $s[i]$ refers to the object at the abstract position i, where $0 \leq i < |s|$ and $|s|$ denotes the number of objects in s. For convenience, we use $s[x]$ to denote the position of object x in a state s.

State transitions are triggered eventually by user input but more directly by output of the consistency control module. To distinguish, we use the term "operation" for input from the user interface and the term "invocation" for output to the user interface. An operation o usually triggers multiple invocations due to the existence of multiple sites in the system. Notations $\mathcal{I}(\mathcal{E})$ and $\mathcal{O}(\mathcal{E})$ denote the sets of operations and invocations, respectively. Notation (s, e, s') denotes a state transition, which means that the execution of an invocation e in a state s yields a new state s'. We define the following primitive invocations:

- ins(p, x): insertion of a new object x at position p,
- del(p, x): deletion of object x at position p, and

- identity invocation ϕ that does not modify a state, i.e., (s, ϕ, s) for any state s.

Typical user operations in interactive applications, such as undo, copy-paste, move, and update, are eventually translated into these three user interface primitives. In particular, if an object is "moved" from one position to another, we consider that it is deleted first at the old position and then a new object (with the same attributes, if any) is inserted at the new position. Additionally, if an attribute (e.g., color) of an object is modified, the object is considered to be deleted first and then another object with the new attribute is inserted at the same position.

It is worth noting that, although all objects in any state is totally ordered, the relationship may not be defined between any pair of objects in $\mathcal{X}(\mathcal{E})$. For example, the relationship between a deleted object and a newly inserted object is not defined, if they never appear in the same state.

Also note that, to have a meaningful discussion of consistency control among cooperating sites, here we assume that all view states use the same type of relationship at all times. As an example, consider the case of a multiuser file browser. When the same set of file icons are sorted in different orders, it is possible that file A appears before file B in one view but after B in another view. However, in this case, the relationships in the two views should be labeled by their sorting criteria, e.g., file name or size. They are not considered as the same type of relationship.

2. Developing a Global Picture

To study the "global" behavior of the system, we introduce an analysis tool called "behavior graph". The idea is to first construct a global graph from the initial system state s_0 and then incrementally apply every invocation on the graph. Note, however, the behavior graph is only for theoretical analysis of consistency conditions and does not have to be implemented in actual systems.

A behavior graph G is a tuple $< V, E >$, where V is a set of nodes (or vertexes) and $E \subseteq V \times V$ is a set of directed edges. Every node corresponds to an object in $\mathcal{X}(\mathcal{E})$ and every edge represents the relationship between two objects. Each node $n \in V$ has three attributes: $n.x$ is the object id it corresponds to; n.counter traces how the object is inserted or deleted; and n.color indicates the status, *black* for normal and *red* for abnormal. An edge $< n_b, n_a > \in E$ means $n_b.x$ immediately precedes $n_a.x$ in some state of the system.

```
(a) Initialize:
(a.1)   $\forall x \in s_0$ add a new node $n$ to $V$ with
(a.2)       $n$.x=$x$, $n$.counter=N, and $n$.color=black;
(a.3)   $\forall n_b, n_a \in V$ if $s_0[n_b$.x$] + 1 = s_o[n_a$.x$]$
(a.4)       add edge $< n_b, n_a >$ to $E$;

(b) Invoke an insertion $e$ in state $s$:
(b.1)   if  $\exists n \in V : n$.x $= e$.x //remote invocation
(b.2)       $n$.counter $\leftarrow n$.counter + 1;
(b.3)   else add a new node $n$ to $V$ with //local invocation
(b.4)       $n$.x=$e$.x, $n$.counter=1, and $n$.color=black;
(b.5)   if $\exists n_b \in V$ such that $n_b$.x = $s[e$.p-1$]$
(b.6)       add edge $< n_b, n >$ to $E$;
(b.7)   if $\exists n_a \in V$ such that $n_a$.x = $s[e$.p$]$
(b.8)       add edge $< n, n_a >$ to $E$;

(c) Invoke a deletion $e$ in state $s$:
(c.1)   find $n \in V$ such that $n$.x=$e$.x;
(c.2)   if $s[e$.p$] \neq n$.x
(c.3)       $n$.color $\leftarrow$ red;
(c.4)   elseif $n$.color $\neq$ red
(c.5)       $n$.counter $\leftarrow n$.counter - 1;
(c.6)       $n$.color $\leftarrow$ red if $n$.counter $< 0$;
```

Fig. 29. Maintaining a global behavior graph G.

Figure 29 shows how G is maintained. Suppose V and E are initially empty and

N is the number of sites in the system. We first use the initial state s_0 to initialize G as follows: If s_0 is not empty, for each object $x \in s_0$ create a new node n such that n.x $= x$, n.counter = N, and n.color = black. Then for any two nodes $n_b, n_a \in V$ add an edge $< n_b, n_a >$ to E if n_b.x precedes n_a.x in state s_0.

Every invocation $e \in \mathcal{O}(\mathcal{E})$ is relative to some execution state $s \in \mathcal{S}(\mathcal{E})$. Suppose that any non-identity invocation e has two attributes: e.x is the object to be inserted or deleted and e.p is the position relative to its execution state. To invoke an insertion e in s, we first find e.x in the graph. If it is found in node n, we increment n.counter by one; otherwise we create a new node n with n.x = e.x, n.counter = 1, and n.color = black. If there exists n_b in G such that n_b.x is at the position immediately after which e.x is to be inserted in s, we add an edge $< n_b, n >$. If there exists n_a in G such that n_a.x is at the position immediately before which e.x is to be inserted in s, we add an edge $< n, n_a >$.

Once a node is created in G, we never delete the node. When invoking a deletion e, we can always find the node $n \in V$ that contains object e.x. However, e.p may point to different objects in different execution states. If $s[e.\text{p}]$ fails to point to the same object as e.x, we turn n.color to red. Otherwise we decrement n.counter by one. If n.counter has been decremented this way more than N times, or n.counter < 0, we turn the color of node n to red.

a. Normal Behavior Graph

It is worth emphasizing that, as an analysis tool rather than a specific concurrency control protocol, the graph maintenance algorithm in Figure 29 itself does not calculate the invocation form of any operation. Instead, the invocation parameters are determined by the group editor or more specifically its concurrency control protocol. The graph serves to detect possible inconsistent behavior of the system, as will be shown in the remainder of this section. Here we first define the concept of "normal" behavior graph.

Definition 23 *A behavior graph G is normal iff G is acyclic and does not have red nodes.*

b. Execution Traces

An execution trace is a sequence of invocations, starting from the initial system state s_0, in the order that they are observed by (or applied on) the behavior graph. Notation $\mathcal{T}(\mathcal{E})$ denotes the set of all possible traces in the system. In Subsection 4, we will study what traces are "correct" and how invocations are ordered in correct traces.

By the algorithm in Figure 29, from any given trace β, we can construct a behavior graph G, which is called the corresponding graph of β. For any trace β and invocation $e \in \beta$, function prefix(β,e) returns the prefixing subsequence of β from the first to the invocation right before e. If e is the first invocation in β, prefix(β, e) returns an empty trace. The predecessor behavior graph of e, denoted by function pred_bg(β, e) is the corresponding graph of prefix(β, e). The successor behavior graph of e, denoted by succ_bg(β, e), is the graph obtained by invoking e on pred_bg(β, e). We say that e is an expansion of pred_bg(β, e).

c. Similar Behavior Graphs

Definition 24 *Given any two nodes, n_1 and n_2, in a normal behavior graph G, the order between them, denoted by order$_G(n_1, n_2)$, is (1) $n_1 \rightsquigarrow n_2$, if there exists a path from n_1 to n_2; (2) $n_2 \rightsquigarrow n_1$, if there exists a path from n_2 to n_1; or (3) unorderable, if there is no path between n_1 and n_2.*

The relation $\rightsquigarrow$ defined above is a partial order over the set of nodes in the behavior graph because not every pair of nodes are orderable. Due to the one-to-one mapping between nodes in the graph and the set of objects $\mathcal{X}(\mathcal{E})$, relation $\rightsquigarrow$ also applies to set $\mathcal{X}(\mathcal{E})$. For example, in a text editor, if 'x' is deleted from state "axb" and then 'y' is inserted to yield "ayb", characters 'x' and 'y' are not ordered unless 'x' appears in the same state as 'y' somehow, e.g., by undo. In general, for any two objects $x_1, x_2 \in \mathcal{X}(\mathcal{E})$, if they ever appear in the same state, their relation must have been determined by the system, or more specifically, its consistency control protocol. In that case, there must be two nodes that contain x_1 and x_2 in the graph, respectively, and at least one path exists between them.

Definition 25 *Any two normal graphs G_1 and G_2 are similar, denoted by $G_1 \sim G_2$, iff (1) G_1 and G_2 have the same set of nodes, and (2) any two nodes n_1 and n_2 are ordered the same way, i.e., $order_{G_1}(n_1, n_2) = order_{G_2}(n_1, n_2)$.*

It is obvious that a normal graph is similar to itself and relation $\sim$ is transitive, as stated below.

Corollary 5 *Given any three normal behavior graphs G_1, G_2 and G_3, that have the same set of nodes (objects), if $G_1 \sim G_2$ and $G_2 \sim G_3$, we have $G_1 \sim G_3$.*

The equivalent relation below further requires that nodes containing the same object ids have the same counter values.

Definition 26 *Any two normal graphs G_1 and G_2 are equivalent, denoted by $G_1 \equiv G_2$, iff (1) $G_1 \sim G_2$ and (2) any two nodes in G_1 and G_2 that contain the same object id also have the same counter value.*

3. An Example of Behavior Graph

The example in Figure 30 shows how a behavior graph G is constructed. In the following we use x, y and b to represent visual objects of interest, which are not

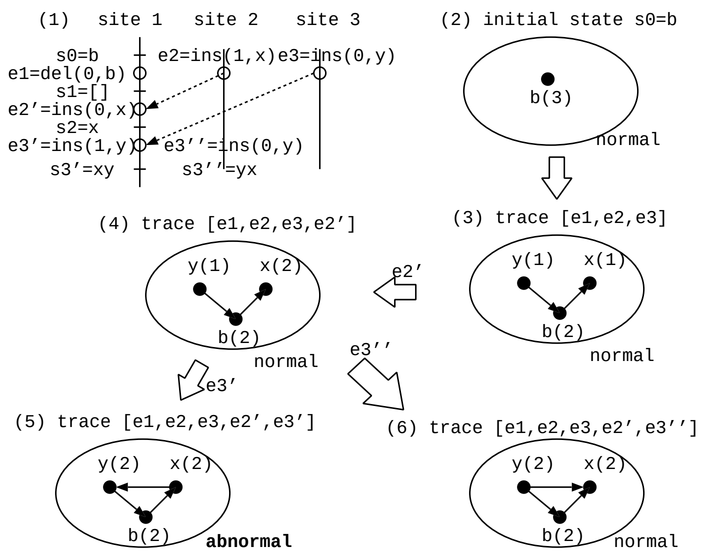

Fig. 30. Normal and abnormal behavior graphs.

necessarily characters. To simplify notations, we replace nodes with objects that they contain in G. Suppose three sites start from the same initial view state s_0 with only one object b. By the graph algorithm, G is initialized as in Figure 30(2), which has only one black node b with counter value 3.

Suppose three sites respectively generate three operations o_1, o_2, and o_3 independently of each other. o_1 by site 1 deletes object b in s_0, o_2 by site 2 inserts a new object x after b in s_0, and o_3 by site 3 inserts a new object y before b in s_0. Those operations are first invoked locally at their generation sites as e_1 = del(0,b), e_2 = ins(1,x) and e_3 = ins(0,y), respectively. Without loss of generality, assume the observed trace is $\beta = [e_1, e_2, e_3]$. The graph corresponding to β is shown in Figure 30(3), in which the counter of b is 2 due to the invocation of e_1. When e_2 is invoked locally in state s_0, it creates a new black node containing x with counter value 1 and adds an edge from b to x because x is inserted after b. Similarly, the invocation of e_3 adds a new node y with counter 1 and a new edge $< y, b >$.

Now consider how o_2 and o_3 are invoked at site 1. After invoking e_1 in state s_0, the view of site 1 becomes an empty state s_1 without any object, i.e., $s_1 = s_0 + e_1 =$ []. Suppose the concurrency control protocol somehow translates o_2 into invocation $e'_2 = \text{ins}(0,x)$. By applying e'_2 in s_1 we get $s_2 = s_1 + e'_2 = x$. Correspondingly G is updated as shown in Figure 30(4), in which the counter of node x becomes 2. Note that no new edge is added this time because s_2 is empty.

As shown in Figure 30(5), if o_3 is invoked in state s_2 as $e'_3 = \text{ins}(1,y)$, we get state $s'_3 = s_2 + e'_3 = xy$. Accordingly the counter of node y is incremented to 2 and a new edge is added from x to y. It is obvious that this leads to a cyle and hence the resulting graph and is abnormal.

On the other hand, if o_3 is invoked in state s_2 as $e''_3 = \text{ins}(0,y)$, we get another state $s''_3 = s_2 + e''_3 = yx$. As shown in Figure 30(6), the counter of node y becomes 2 and a new edge $< y, x >$ is added. The resulting graph is still normal.

4. Deriving Consistency Conditions

The behavior graph constructed as above develops a global picture of the system that is observed by an external referee. Apparently some constraints (or properties) must be imposed on the invocations. For example, as an established convention in groupware [10, 5], the invocations must not violate their natural cause-effect order. It is also important that the effect of every operation must be applied at every site once and only once. Additionally, the relationships observed at different sites must agree with each other. Notation $\mathcal{T}(\mathcal{P})$ denotes the set of (legal) traces in the system that satisfy a given set of properties $\mathcal{P}$. In the following, we define three properties: causal ordering, fairness, and safety.

Definition 27 ***[happened-before relation*** $\rightarrow$***:]*** *Given any two invocations e_1 and e_2, we say that e_1 happened before e_2, denoted as $e_1 \rightarrow e_2$, iff one of the following* *holds: (1) e_1 is a local invocation of an input operation o and e_2 is a remote invocation of o at another site; (2) e_2 is an invocation of an input operation o that is generated after the invocation of e_1 at some site; and (3) there exists a third invocation e_3 such that $e_1 \rightarrow e_3$ and $e_3 \rightarrow e_2$.*

Note that condition (1) is an assumption that simplifies following discussions. In early group editors [10, 16], it is possible that a remote invocation is executed before a local invocation of the same operation. However, allowing this would unnecessarily complicate discussions. Also note that local invocation preceding remote invocations does not necessarily imply optimistic consistency control: Even if it is pessimistic, the effect can still be applied at the local site before remote sites when the operation is confirmed by the consistency control module.

Definition 28 ***[Causal Ordering Property:]*** *Given any trace $\beta \in \mathcal{T}(\mathcal{P})$ and two invocations $e_1, e_2 \in \beta$, if $e_1 \rightarrow e_2$, then e_1 must be invoked before e_2 in β.*

By the causal ordering property, any local invocation of an insertion e must be applied on the behavior graph earlier than any remote invocation of e. Hence in Figure 29 the former creates a new node in G while the latter does not. Additionally, the insertion of any object must be invoked earlier than any deletion that deletes the same object. Hence in Figure 29 the invocation of a deletion is always able to find the node in the graph that contains the object to be deleted.

Definition 29 ***[Fairness Property:]*** *Given a system with N sites and trace $\beta \in \mathcal{T}(\mathcal{P})$, if $e \in \beta$ is an invocation of an operation $o \in \mathcal{I}(\mathcal{E})$, there must exist a trace $\delta \in \mathcal{T}(\mathcal{P})$ that contains N invocations of o.*

By the fairness property, for any operation o generated in the system, it will be eventually invoked N times. Note that, in pessimistic consistency control, it is

Table I. Notations of States and State Transitions

Notation	Description
$\mathcal{E}$	The abstract group editing system.
$\mathcal{X}(\mathcal{E})$	The set of objects in the system.
$\mathcal{S}(\mathcal{E})$	The set of states of the system.
$\mathcal{I}(\mathcal{E})$	The set of input operations in the system.
$\mathcal{O}(\mathcal{E})$	The set of output operations in the system.
$\mathcal{T}(\mathcal{E})$	The set of traces in the system.
$\mathcal{P}$	The set of properties of the system.
$\mathcal{T}(\mathcal{P})$	The set of admissible traces.
$\mathcal{O}(\mathcal{P})$	The set of admissible invocations.
$\mathcal{S}(\mathcal{P})$	The set of reachable states.

possible to translate an input operation into N identity invocations (ϕ) if it is rejected by the protocol. Every invocation e updates G once, meaning that G is updated N times in total by all invocations of every input operation $o \in \mathcal{I}(\mathcal{E})$. Hence ideally after all operations are invoked at all sites, the counter of every node in the graph must be either zero or N. Therefore, as in Figure 29, if a node has been deleted more than N times, it is turned red to signal that something is wrong.

While the fairness property says that some "good" thing eventually happens, the following safety property states that some "bad" thing never happens in a "consistent" system.

Definition 30 ***[Safety Property:]*** *Given any trace $\beta \in \mathcal{T}(\mathcal{P})$ and any input $o \in$*

$\mathcal{I}(\mathcal{E})$, the effects of all invocations of o in β must be consistent with each other.

The behavior graph effectively visualizes the view states and their transitions at all sites together in one picture. The underlying assumption is that the initial behavior graph is normal. If the system is consistent, the final graph must also be normal, i.e., it has neither cycles nor red nodes. A cycle between two nodes n_1 and n_2 means that object n_1.x precedes object n_2.x in one view state while n_2.x precedes n_1.x in another view state. A red node n means that object n.x has been incorrectly deleted. Neither case is desirable. Conversely, we say that, if the behavior graph of a system is not normal, the system is not consistent. Therefore, the safety property can be rephrased in the following theorem.

Theorem 8 *If a consistency control protocol is correct, every invocation when applied on the behavior graph must yield no cycle nor red node.*

Note that the above properties only constrain that the ordering of invocations in (legal) traces preserve the happened-before relation $\rightarrow$. Given two invocations e_1 and e_2, if neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$, we say that they are concurrent, denoted by $e_1 \parallel e_2$. The ordering between concurrent invocations could be arbitrary in any legal trace since there is no constraint. That is, multiple possible legal traces exist for the same set of invocations.

Definition 31 *Given two traces $\beta_1, \beta_2 \in \mathcal{T}(\mathcal{P})$ that are different orderings of invocations of the same set of operations, we say that β_1 and β_2 are indistinguishable if their corresponding behavior graphs are similar.*

Conceptually, this definition emphasizes the effects of traces rather than their orderings. In other words, if two traces produce the same effects on the user interface, they are indistinguishable from the users' perspective.

For any invocation e and trace β, where $e \in \beta$, if both pred_bg(β, e) and succ_bg(β, e) are normal, we say that e is a safe expansion of pred_bg(β, e). However, an invocation safe for one trace may not be safe for another.

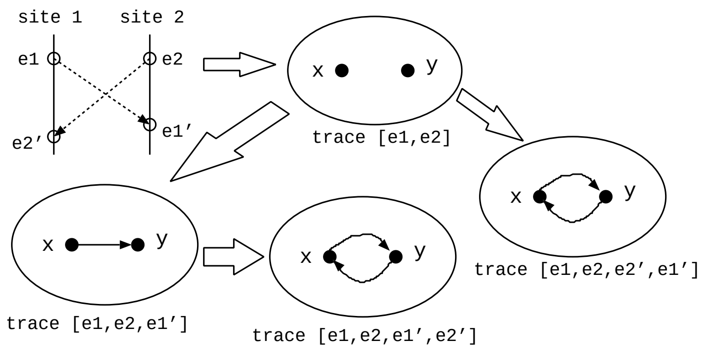

Fig. 31. Different orderings of concurrent invocations yield multiple possible traces.

For example, as in Figure 31, consider that two sites concurrently generate two operations, attempting to create two objects, x and y, in an empty initial state, respectively. Let the base trace $\beta = [e_1, e_2]$ and its corresponding graph is shown in Figure 31. Suppose o_1 is translated into two invocations e_1 and e'_1, where $e_1 = e'_1 =$ ins(0,x), and o_2 is translated into two invocations e_2 and e'_2, where $e_2 = e'_2 =$ ins(0,y). Let trace $\beta_1 = [e_1, e_2, e'_1]$ and trace $\beta_2 = [e_1, e_2, e'_2, e'_1]$. Obviously e'_1 is a safe expansion of pred_bg(β_1, e'_1), while e'_1 is not a safe expansion of pred_bg(β_2, e'_1).

This example illustrates that finding a safe expansion for one trace is not sufficient. A correct concurrency control protocol must be able to ensure that every invocation be a safe expansion for any trace containing it. Such an invocation is called an *admissible invocation.* If every invocation in trace β is admissible, we say that β is an admissible trace. Additionally, a *reachable state* is defined as the view state at one site that is the result of applying an admissible invocation on another reachable state, assuming the initial system state is a reachable state itself.

Theorem 9 *The safety property requires that every invocation in a consistent group* *editor be admissible.*

Based on the above discussions, a group editor $\mathcal{E}$ is consistent if it satisfies the set of properties $\mathcal{P}$, namely, causal ordering, fairness, and safety. In such a system, all traces are admissible, all invocations are admissible, and all states are reachable. That is, as shown in Table I, $\mathcal{T}(\mathcal{E}) = \mathcal{T}(\mathcal{P})$, $\mathcal{O}(\mathcal{E}) = \mathcal{O}(\mathcal{P})$, and $\mathcal{S}(\mathcal{E}) = \mathcal{S}(\mathcal{P})$.

C. An Embodiment of Guidelines

1. Operational Transformation

OT is an optimistic consistency control protocol which allows users to edit any part of the shared data at any time. Local operations are executed immediately once they are generated. Operations are broadcast to remote sites for synchronization. Remote operations are executed in the current state at a site after they are transformed with all concurrent operations that have been executed at that site. The algorithm must ensure consistency across all data replicas after all operations have been executed at all sites, possibly in different orders of execution.

To illustrate, consider the scenario shown in Figure 31. Given an empty initial state and two concurrent operations o_1 and o_2, the problem is to transform them into the "right" invocations at each site. At site 1, when o_2 is received, its view state has only one object x due to the local invocation e_1 of operation o_1. If o_2 is invoked as it is, i.e., its invocation $e_2' = e_2 = \text{ins}(0, y)$, the view state becomes yx, meaning that object y precedes object x. At site 2, when operation o_1 is received, if it is invoked in the current state y as $e_1' = e_1 = \text{ins}(0, x)$, the view state becomes xy, meaning that object y precedes object x. Hence the final view states at two sites diverge. By constructing a global behavior graph, a cycle is produced between objects x and y after all the four invocations are applied.

The basic idea of OT is to transform operations at each site such that their remote invocations possibly take different parameters as their local invocations. For example, in Figure 31, if the OT protocol somehow translates o_1 and o_2 into the following four invocations instead: $e_1 = \text{ins}(0, x)$, $e_2 = \text{ins}(0, y)$, $e'_1 = \text{ins}(0, x)$, and $e'_2 = \text{ins}(1, y)$, then it is easy to show that the resulting behavior graph is normal. That is, two sites converge in the same view state xy.

The challenge is how to transform remote operation into an admissible invocation without maintaining an actual global behavior graph. In a typical OT algorithm, every site i maintains a history buffer HB_i to record all executed local and remote operations in their order of execution. An OT algorithm usually consists of the following two layers: a number of transformation functions that decides how to transform any two given operations, and a control procedure decides how to compute the invocation of any given (remote) operation against the local history buffer.

To develop an intuition of the problem, we consider a simple scenario in which a group editor with n sites starts from an empty initial state. Suppose that each site generates an operation which independently creates a new object. To simplify discussions, here we do not distinguish between operations and their invocations. An operation or invocation is uniformly represented by v_i^j, denoting the invocation at site j of an operation generated at site i. Then an local invocation is denoted as v_i^i.

Let G be the behavior graph corresponding to the base trace $\beta = [v_1^1, ..., v_i^i, ..., v_n^n]$, which only contains n local invocations. The ordering of these local invocations in β is not important in this scenario. As a result, G contains n nodes and there is no edge between any two nodes in G.

At any site k, after executing all n invocations, its history buffer HB_k contains n operations. Since the local invocation v_k^k is executed first, it is obvious that the other $(n-1)$ invocations are from remote operations. That is, $HB_k[0]$ is a local invocation, while the subsequence $HB_k[1, n-1]$ are remote invocations. Now consider a trace β_k which invokes operations in $HB_k[1, n-1]$ after operations in the base trace β. Let G_k be the corresponding behavior graph of β_k. Effectively operations in $HB_k[1, n-1]$ expands G to G_k. We have the following theorem.

Theorem 10 *For any site i and site j in a group editing system, achieving consistency requires that G_i and G_j are normal and $G_i \sim G_j$.*

Every time an operation from site i is invoked at site j, it is transformed with sequence $HB_j[0, k-1]$ which contains the k operations that have been executed at site j. This sequence is called a transformation path. In a simple case in which all these k operations are concurrent with v_i^j, there are $k!$ possible transformation paths with different ordering of the k operations. An OT algorithm must be able to compute the right invocation v_i^j regardless of transformation paths and, more generally, the ordering of concurrent operations in the local history buffer. In the following, we analyze several existing approaches with regard to our consistency model.

2. Approach A: TP1 and TP2

The "mainstream" OT algorithms [14, 17, 11] roughly work as follows: For each remote operation v, the control procedure transposes the history buffer into two subsequences L_h and L_c such that L_h contains all operations that happened before v and L_c contains all operations concurrent with v. Then v is (inclusively) transformed with every operation in L_c to incorporate their effects. Finally the resulting operation v' is invoked in current state. Because for any v its concurrent operations could be executed in any order at different sites, it is well established that the (inclusion) transformation function T must satisfy the following two transformation properties (TP) [21, 16, 17, 11]:

TP1: for any two concurrent operations v_1 and v_2 that are defined in the same state s, let $v_1' = \mathrm{T}(v_1,v_2)$ and $v_2' = \mathrm{T}(v_2,v_1)$, then (1) v_1' (or v_2') preserves the effect of v_1 (or v_2) in s, and (2) two sequences $[v_1,v_2']$ and $[v_2,v_1']$ when applied in s yield the same state s'.

TP2: for any three concurrent operations v_1, v_2, and v_3 defined in the same state s, transforming v_3 with v_1 and v_2 in different invocation orders yields the same result. That is, $\mathrm{T}(\mathrm{T}(v_3, v_2),\mathrm{T}(v_1,v_2)) = \mathrm{T}(\mathrm{T}(v_3, v_1),\mathrm{T}(v_2,v_1))$.

These two properties inductively ensure that transforming any remote operation with concurrent operations along arbitrary paths achieve the same invocation result [16]. In the following we analyze what satisfying TP1 and TP2 means in the context of our new consistency model.

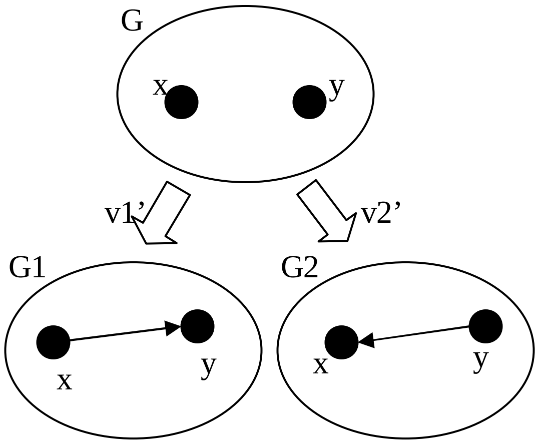

Fig. 32. TP1 ensures that, for any two concurrent operations, G_1 and G_2 are normal and $G_1 \sim G_2$.

a. Interpreting TP1

The first condition in TP1 is rephrased as "intention preservation" in [17, 5]: any remote invocation of operation o must preserve the original intention (or effect) of o

in its generation state. The key issue here is how to define the original intention of an operation and the condition of intention preservation. Based on our consistency model, the intention preservation condition can be specified as that v_1' and v_2' are admissible or preserve the effects of their local invocations, respectively.

Without loss of generality, suppose that there is a trace $\beta = [v_1, v_2]$, where v_1 and v_2 are two local invocations and $v_1 \parallel v_2$, and G is the normal graph corresponding to β. Assume that only v_1' and v_2' (invocations of v_1 and v_2, respectively) are executed to expand G. There are two traces that have the same prefix β: Let $\beta_1 = \beta \bullet v_1'$ and $\beta_2 = \beta \bullet v_2'$. Let G_1 and G_2 be two behavior graphs corresponding to succ_bg(β_1,v_1') and succ_bg(β_2,v_2'), respectively.

According to our model, the first condition of TP1 can be interpreted as that v_1' and v_2' are safe expansions of G. That is, both G_1 and G_2 be normal.

The next problem is whether G_1 is similar to G_2, because this ensures consistency between two sites. To illustrate this problem, consider a case where both v_1 and v_2 are insertions and there is no edge between the two nodes they created in G. For example, let s_0 be empty, v_1 = ins(0,x) and v_2=ins(0,y). Suppose v_1'=v_1 and v_2'=v_2 and their executions result in two view states $s' = s_0 + [v_2, v_1']$ and $s''= s_0 + [v_1, v_2']$, respectively. As shown in Figure 32, G_1 and G_2 are both normal because $order_G(x, y)$ is not defined. However, after G is expanded, we have $order_{G_1}(x, y) \neq order_{G_2}(x, y)$. Hence we need the second condition in TP1, s'=s'', to ensure that $G_1 \sim G_2$ or, essentially, $order_{G_1}(x, y) = order_{G_2}(x, y)$.

b. Interpreting TP2

The first question is why we still need TP2 if TP1 is satisfied. As mentioned above, TP1 only ensures that, given any *two* concurrent operations, the expansion is always safe regardless of the ordering of invocations. However, it cannot ensure consistency between the number of operations goes above two. TP2 extends to three concurrent operations.

As an example, consider a scenario shown in Figure 33. Suppose three sites from an initial empty state and they concurrently create three objects x, y, and z, respectively. Let

TP1 ensures tha

Without loss of g

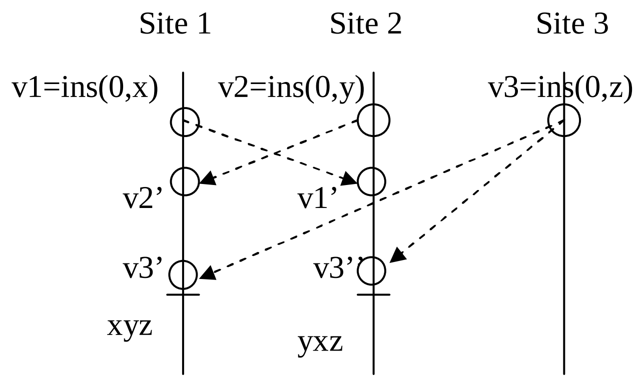

Fig. 33. A scenario where TP1 is not sufficient to ensure consistency between different sites.

Consider two traces $\beta_1 = [v_1, v_2, v_3, v_2', v_3']$ and $\beta_2 = [v_1, v_2, v_3, v_1', v_3'']$. Based on TP1, graphs G_1 and G_2 corresponding to prefix(β_1, v_3') and prefix(β_2, v_3''), respectively, are normal and $G_1 \sim G_2$. Here we refer them as G because G_1 is actually equivalent to G_2. In graph G, there are three objects x, y, and z with only one edge $< x, y >$.

As shown in Figure 34, G is expanded by v_3' and v_3'', which results in G_3 and G_4, respectively. TP1 ensures that G_3 and G_4 are normal. In other words, TP1 ensures that the order between x, y, and z is acyclic but says nothing about the uniqueness of the order. This implies that there may exist more than one possible acyclic order between these three nodes. Without loss of generality, assume that after the invocation of v_3' and v_3'', respectively, the view state of site 1 is xyz and that of

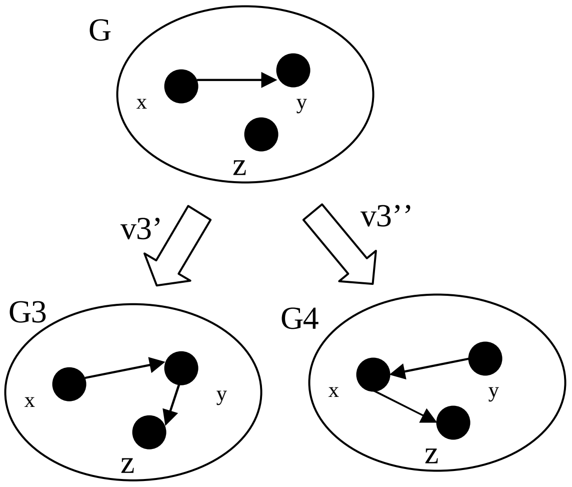

Fig. 34. TP2 ensures that, for any three concurrent operations, G_3 and G_4 are normal and $G_3 \sim G_4$.

site 2 is yzx. Fig. 34 shows the corresponding behavior graphs.

By definition, there must exist a trace $\delta = [v_1, v_2, v_3, v_2', v_1'', v_1', v_2'', v_3', v_3'']$. The corresponding graph of δ visualizes the inconsistent orders in G_3 and G_4 together. It is obviously not normal because these exists a cycle between x, y and z. This example illustrates that TP1 cannot ensure a normal graph when the number of operations is three or above.

By observation, the problem in the above scenario is caused by the fact that $v_3' \neq v_3''$. If TP2 is satisfied, then we have $v_3 = v_3''$ and $G_3 \sim G4$. However, the above scenario is special in that there exists an edge between x and y before the invocation of v_3' and v_3''. How about the more general cases in which there is not such an edge?

To generalize, consider a system in which three "active" sites concurrently create three objects as above, and there are adequate number of "passive" sites that do not generate operations but invoke all generated operations in every possible order. Let 1, 2 and 3 represent the three concurrent operations from sites 1, 2 and 3, respectively. Hence we need 3! or 6 sites to accommodate the following six different execution orders: $1 \rightarrow 2 \rightarrow 3$, $2 \rightarrow 1 \rightarrow 3$, $1 \rightarrow 3 \rightarrow 2$, $3 \rightarrow 1 \rightarrow 2$, $2 \rightarrow 3 \rightarrow 1$, $3 \rightarrow 2 \rightarrow 1$.

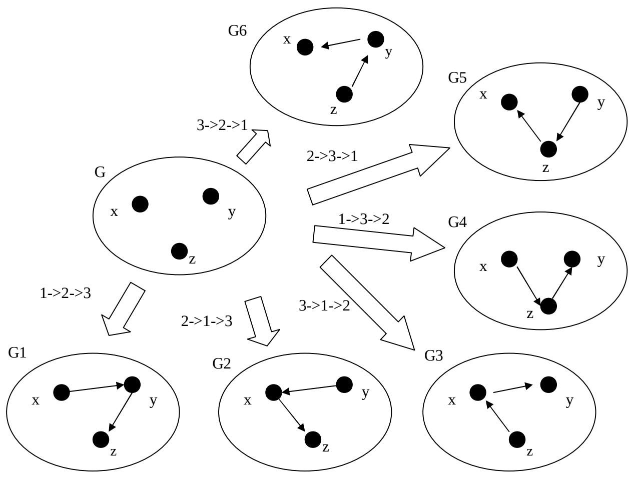

Fig. 35. G has six possible site expansions.

Let G be the graph corresponding to a base trace $\beta = [v_1,v_2,v_2]$. A "site expansion" refers to expanding G with invocations applied in the execution order at one site, i.e., an execution (transformation) path. Then there are six possible site expansions: $\beta_1{=}\beta \bullet [v_2^1,\ v_3^1]$, $\beta_2{=}\beta \bullet [v_1^2,\ v_3^2]$, $\beta_3{=}\beta \bullet [v_1^3,\ v_2^3]$, $\beta_4{=}\beta \bullet [v_1^4,\ v_3^4,\ v_2^4]$, $\beta_5{=}\beta \bullet [v_2^5,\ v_3^5,\ v_1^5]$, and $\beta_6{=}\beta \bullet [v_3^6,\ v_2^6,\ v_1^6]$. They correspond to graphs $G_1 - G_6$, as shown in Figure 35. TP1 and TP2 ensure that all these graphs are normal and similar, as stated in the following lemma.

Lemma 27 *In the scenario as shown in Figure 35, any two site expansions to G are similar.*

Proof. : First consider G_1 and G_2. Their first two invocations are the same operations executed in different orders and the third operations are the same. By TP2, $G_1 \sim G_2$ must hold. Similarly, we have $G_3 \sim G_4$ and $G_5 \sim G_6$. Now we prove $G_1 \sim G_4$. G_1 is obtained by execution order $[v_1^1,\ v_2^1,\ v_3^1]$ and G_4 is by $[v_1^4,v_3^4,v_2^4]$. This means that two concurrent operations v_2 and v_3 are invoked in the same state in different orders. TP1 ensures that $G_1 \sim G_4$. By corollary 5, we get $G_1 \sim G_2 \sim G_3 \sim G4$. Similarly, we can prove $G_2 \sim G5$. Again, by Corollary 5, the assertion holds. ■

This lemma implies that, for any three operations that are concurrently generated in the same state, no matter in which order they are executed at a site, the resultant state is the same. Similarly we can prove that the assertion holds for n concurrent operations. Therefore, TP1 and TP2 are sufficient conditions to ensure our proposed consistency model.

3. Approach B: Predefined Total Order

Influential and significant as TP1 and TP2 are, they suffer from the following limitations in hindsight. TP1 and TP2 as formalized in [16] implicitly imply that the signature of transformation functions be in the form of v_i'=T(v_i,v_j), in which T is an inclusion transformation (IT) function [11] that transforms v_i with v_j to incorporate its effect into v_i'. It is also implied that v_i and v_j are characterwise insertions and deletions, that they are concurrent with each other, and that they are defined in the same state. However, after their proposal in [16], there have been stringwise operations [5] and non-IT transformation functions [21, 38, 17, 5]. It remains an open issue whether or not TP1 and TP2 apply to these new transformation functions.

Conceptually, all transformation functions can be divided into two steps: first figure out the logical relation between involved objects, and then, based on the logical relation, implement the desired transformation. Once the logical relation is known, the subsequent transformation is trivial.

Observe that any non-ϕ operation is associated with at least one object. Those objects are called its effect objects. Here we propose the generalized principle one (GP1) for designing a set of "generalized" transformation functions $\mathcal{F}(\mathcal{E})$. $GP1$ assumes that, in a system $\mathcal{E}$, a logical relation can be somehow defined over the set of objects $\mathcal{X}(\mathcal{E})$.

Definition 32 [**GP1:**] *For any transformation function $T \in \mathcal{F}(\mathcal{E})$, its results preserve the predefined logical relation between the effect objects of involved operations.*

Compared to TP1 and TP2, GP1 has the following two major advantages. First, it generalizes the concept of transformation function. It no longer constrains the purpose and signature of T as do TP1 and TP2. Hence it naturally covers all transformation functions that have appeared in the literature [21, 17, 5] as well as stringwise operations [5]. Second, to our best knowledge, GP1 is the first in the literature that gives heuristics for designing transformation functions. The following shows that GP1 is more general than TP1 and TP2.

In fact, For transformation functions specified in [16], if GP1 is satisfied, TP1 and TP2 are also satisfied. To explain, first, TP1 ensures that a transformation function always preserve the effect of each operation. If a proper logical relation of all objects is predefined, GP1 ensures that the transformation function can correctly determine the logical relation between its effect objects. Hence the transformation can be correctly performed. Secondly, TP2 is to remove the ambiguities of determining the ordering of two effect objects. However, due to GP1, such ambiguities are easily solved due to their predefined relation.

GP1 has been successfully embodied in design of the LBT protocol [39, 40]. LBT supports two binary transformation functions, inclusion transformation (IT) and exclusion transformation (ET), that transform characterwise insert and delete operations. It first defines a total order over all characters that ever appear in the system, after a careful cases coverage analysis. Second, it designs the transformation functions such that operation position, type, and site id are used to determine the predefined total order, with an extra local data structure that records the logical relations computed from previous transformations. However, analyses show that such transformation functions still fail in some cases. The solution in LBT is to identify such exception cases (conditions) and build special transformation paths to avoid them. As a result, LBT correctly preserves the predefined total order when invoking any operation.

4. Approach C: Natural Partial Order

GP1 says that all objects that will ever appear in a system and their relationship have to be determined as a total order at design time This requirement is costly and not natural because the total order is not a necessary condition for any group editor, especially if not all the objects will appear in the same state. To address this problem, we require instead that any emergent logical relation must not violate relations that have been established earlier in the same system. The resulting relation is a partial order because it does not impose an order between all pairs of objects.

In OT protocols, computing an invocation of any operation boils down to transformation functions, which must first determine the logical relation between involved objects. Hence it is essentially the transformation functions that introduce new relations. For example, consider that transformation function T is called to transform v and it first needs to determine the relation between two objects x and y. There must exist a trace β that only consists of all invocations that happened before v. Let G be the normal behavior graph corresponding to β. If x and y are in G at all, $order_G(x, y)$ must be either defined ($x \rightsquigarrow y$ or $y \rightsquigarrow x$) or undefined. If it is defined, T just uses the existing relation, or, otherwise, T simply introduces one by some policy. We propose the following generalized principle two (GP2):

Definition 33 [GP2:] *For any two objects $x, y \in \mathcal{X}(\mathcal{E})$, their relation determined in any transformation function $T \in \mathcal{F}(\mathcal{E})$ must be equal to the existing relation between them if there is any when T is processed; their relations determined in any two functions $T_1, T_2 \in \mathcal{F}(\mathcal{E})$ must be equal.*

Similarly to GP1, for transformation functions specified in [16], if GP2 is satisfied, TP1 and TP2 are also satisfied.

To explain, if any objects involved in a transformation already have an existing order, it is obvious that TP1 and TP2 can be satisfied. If an order does not exist yet, since GP1 requires that their relations determined by different transformation functions are equal, TP1 is satisfied. Additionally, since transformation functions always return the same relation between objects no matter what forms the operations take, transforming the same operation along different paths always yield the same result. Hence TP2 is also implied.

GP2 has been successfully embodied in the design of the ABT protocol [38]. It supports characterwise insertions and deletions as well as three transformation functions (IT, ET and SWAP). For more details, refer to chapter IV.

Li and Li [14, 35] (CSM) is the first to our knowledge that explicitly recognizes the importance of logical relationship between objects in group editors. Based on the concept of operation effects relation, it proposes to preserve single operation effects (S) and multiple operation effects relation (M), which replace convergence and intention preservation in the CCI model. The SDT algorithm [35] is the first that is formally proved to satisfy TP1 and TP2. However, CSM is tightly bound to textual group editors with only characterwise operations and does not rigorously define the logical relation (total order) of objects it is based on.

The CSM model is improved in [39, 40] (CR) by giving a strict definition of the effects relation. The S and M conditions are combined and rephrased as effects relation preservation (R). CR is essentially the same as CSM with more rigorous definitions. However, CR still fails to fulfill the generality requirement because of its predefined total order of all characters. For example, there is no natural order between a deleted character and newly inserted characters. Imposing an artificial order between them effectively involves application or protocol specific semantics and policies.

The CA model [38] proposes two consistency criteria, causality preservation (C) and admissibility preservation (A). It no longer requires to impose a predefined total order. However, it is still tightly bound with a specific protocol and its underlying data structures and operations. Hence it is not considered as a general consistency model.

CHAPTER VI

RELATED WORK

A. Design Framework

Commutativity-based concurrency control in other collaborative computing domains (e.g., [41, 26, 42, 28]) also exploits operation semantics to attain more concurrency. However, they do not assume fine-grained awareness and sharing as in groupware. Operation parameters are typically not modified in commute operators and convergence of data replicas is often the goal to pursue in concurrency control.

It is well-understood that convergence must be further constrained in groupware [10, 14, 16, 25, 5]. Sun et al. [5] propose that a group editor is correct if it achieves convergence, causality and intention preservation. The intention preservation condition constrains that the execution of any operation at remote sites achieve the same effects as it is generated. This condition is very intuitive and has been well accepted in the groupware field. However, no guideline has been provided in Sun et al. [31, 5] as how to develop OT algorithms that are able to achieve intention preservation. Moreover, “intention preservation” as a correctness criterion is under-formalized for verification purposes. Consequently no previous work, including [25, 31, 5, 33], has been formally proved to be “correct” with regard to intention preservation to our best knowledge.

Ressel et al. [16] propose two well-known conditions of IT such that any two causality-preserving histories can achieve effects equivalence (convergence). However, no guideline has been provided as how to develop IT functions that satisfy the conditions. In fact, these conditions have turned out extremely difficult to verify in practice [16, 25, 11, 5, 33]. While [16, 25, 11] assume these conditions to achieve convergence, our recent work SDT [14] is the first that has been formally proved to verify these two conditions. However, the proofs in [35] are extremely complicated even with only two characterwise primitive operations (ins/del) and scale poorly to the number and complexity of primitive operations. Alternative approaches [34, 18, 5, 33] achieve convergence by avoiding conditions of [16]. However, they still have to verify intention preservation before claims can be made on correctness.

While conditions in [16] fail to consider ET or SWAP, Knister and Prakash [21] define conditions of SWAP such that histories can be reordered for the purposes of group undo. However, their conditions only allow for cases in which the position parameters of two operations are not tied. Sun et al. [5] define preconditions of IT and ET as context equivalence and context serialization, respectively. However, counterexamples show that these two conditions are not sufficient for preserving the effects relation [14]. Our conditions in Theorems 4 and 5 are (mathematically) sufficient because, once they are satisfied, IT and ET/SWAP are guaranteed to be correct, respectively.

B. Consistency Model

The first significant consistency model in groupware is established in Ressel et al. [16]. It defines two conditions, causality preservation and convergence (CC). This model to some extent satisfies the completeness and disambiguity requirements. However, its convergence condition is not sufficient to address the needs of groupware applications that require not only the same set of objects but also certain logical relationship between them [5, 14]. Secondly, in this model, the system behavior description (although somehow complete) is tightly coupled with a specific protocol called adOPTed. Thirdly, although it proposes that transformation functions must satisfy TP1 and

TP2, it does not provide guidelines as how to design them.

Sun et al. [5] (CCI) extends the CC model by introducing a new condition called intention preservation. In principle, intention preservation is a generic criterion independent of specific application and protocol semantics. Unfortunately, it violates the disambiguity principle because a rigorous specification of intention preservation has been absent. As a result, this leads to difficulties in verifying protocols (e.g., [17, 5, 11, 31]) that are based on this model. In addition, the CCI model does not completely describe the system behavior as does this work.

A few approaches [18, 5, 33] were explored to free the TP2 condition that turned out extremely difficult to satisfy in practice. However, they are based on the CCI model and thus suffer from problems verifying intention preservation. Additionally, these early works are theoretically incomplete in that they fail to compensate the losses of freeing TP2. As a result, counterexamples identified in [14, 35] show that, although achieving convergence, they are not able to preserve the logical relationship between objects in some cases.

Li and Li [14, 35] (CSM) is the first to our knowledge that explicitly recognizes the importance of logical relationship between objects in group editors. Based on the concept of operation effects relation, it proposes to preserve single operation effects (S) and multiple operation effects relation (M), which replace convergence and intention preservation in the CCI model. The SDT algorithm [35] is the first that is formally proved to satisfy TP1 and TP2. However, CSM is tightly bound to textual group editors with only characterwise operations and does not rigorously define the logical relation (total order) of objects it is based on.

The CSM model is improved in chapter III (CR) by giving a strict definition of the effects relation. The S and M conditions are combined and rephrased as effects relation preservation (R). CR is essentially the same as CSM with more rigorous definitions. However, CR still fails to fulfill the generality requirement because of its predefined total order of all characters. For example, there is no natural order between a deleted character and newly inserted characters. Imposing an artificial order between them effectively involves application or protocol specific semantics and policies.

The CA model in chapter IV proposes two consistency criteria, causality preservation (C) and admissibility preservation (A). It no longer requires to impose a predefined total order. However, it is still tightly bound with a specific protocol and its underlying data structures and operations. Hence it is not considered as a general consistency model.

CHAPTER VII

CONCLUSIONS

A. Contributions

1. A Total Order Based Framework

This framework includes two correctness criteria, **c**ausality preservation and operation effects **r**elation preservation, (CR), for interpreting and developing OT algorithms. In the theory part, it formalizes a new constraint, effects relation preservation, for constraining convergence in interactive groupware applications. Compared to the intention preservation constraint in the state-of-the-art framework CCI [5], it is well-formalized and subject to correctness proofs. The effects relation is naturally defined when operations are generated, except some "artificial" tie-breaking policies for handling boundary cases.

In the technical approach part, CR reveals an alternative way of developing OT algorithms, which follows three main steps: First, it defines an application-specific effects relation between objects for constraining the execution of operations. Secondly, it formulates a set of transformation functions and provides sufficient conditions for them to satisfy the defined effects relation. Thirdly, it requires to find a special transformation path that must be always available when computing an admissible operation and be able to ensure the conditions of the transformation functions.

To evaluate the proposed CR framework, we exemplify a novel OT algorithm (called landmark-based transformation or LBT), which is testimony to the above design guidelines. Specifically, the transformation functions in LBT only uses basic information such as operation type, position parameter, and site id. The concepts of IT/ET safe operation sequences reveal practicable heuristics for constructing the special transformation paths. As demonstrated by the design of the LBT algorithm, the CR framework simplifies the design of transformation functions, because it no longer demands that transformation functions (IT) work in all possible cases. That is, it theoretically frees the difficult TP2 condition that is established in [16] and well-accepted in the literature [25, 11, 31].

2. A Natural Order Based Framework

This framework contributes a novel consistency model and a novel OT approach. Theoretically, we formalize an alternative correctness criterion, called admissibility preservation, based on a graph-based analysis tool. It only requires that an OT algorithm integrate every operation in an admissible manner, i.e., without violating the character order established earlier by the algorithm itself. To some extent, this could be interpreted as a means to formalize operation intentions. Compared to the total order based framework described in chapter III, this work no longer requires a predefined total order of characters, which in turn greatly simplifies design and proof of OT algorithms.

Practically, it establishes a principled methodology for developing and proving OT algorithms. In this approach, it first identifies sufficient conditions for basic IT/ET functions and then builds special transformation paths to ensure the correctness of IT/ET. As a result, it no longer requires IT/ET to work correctly on arbitrary transformation paths, which greatly eases the design of IT/ET. Compared to previous works, our IT/ET do not need extra operation parameters and the algorithm does not save extra information for ensuring correctness. As a result, the algorithm is lucid, without hidden details and costs, simple to present, completely proved, without lurking correctness puzzles, and more efficient. It has been well-accepted that correctness and efficiency are the basis for OT-based group editors to be useful and usable [16].

3. A Generic Consistency Model

First, this work formalizes a general data consistency model for a range of interactive groupware applications that can be abstracted as group editors. The model satisfies the four identified requirements, i.e., completeness, disambiguity, generality, and practicability. In particular, it is based on observable user interface effects and independent of specific protocol details. Hence it can be used to guide the design and verification of a range of consistency control protocols. It is the first consistency model in groupware that achieves so to our best knowledge. Second, it elaborates three distinct approaches to designing a well-established family of OT-based protocols. The proposed guidelines have been well-evaluated by proved design practices.

B. Future Work

In future work, we plan to further explore undo and performance issues in group editors.

1. Undo Issue

No matter in multi-user and single-user applications, undo is an important approach of error recovery. Conventionally, chronological undo is widely adopted in most applications. However, users cannot selectively undo operations. Many researchers [31] [43] [44] [45] have made great contributions in developing selective undo mechanisms in group editors. However, ensuring the correctness of selective undo is still an open issue. Another issue related to selective undo is how to design a user-friendly user interface.

Our future research focuses in this area include: (1) solving the correctness problem of the selective undo in the environment of group editors, (2) developing a more user-friendly undo mechanisms called regional undo, which was pioneered in our early work [46].

2. Performance Issues

Existing OT-based approaches are normally designed for real-time fine-granularity sharing, specifically character-level collaboration. They can work well over high bandwidth network, such as LAN. However, ensuring the performance of OT-based approaches in more complicated network environments, such as wide area network, is still an open issue. In such an environment, coarse-granularity asynchronous collaboration mode seems more likely to be adopted. Compression of operation sequence is a promising technique in solving this problem [47]. However, some issues associated with this technique have not been solved yet, including: (1) how to ensure the correctness of operation sequence compression algorithm, and (2) how to develop an suitable OT-based control algorithm based on the technique.

REFERENCES

[1] R. M. Baecker, D. Nastos, I. R. Posner, and K. L. Mawby, "The user-centered iterative design of collaborative writing software," in *Proc. of the InterCHI'93 Conf. on Human Factors in Computing Systems*, Amsterdam, The Netherlands, Apr. 1993, pp. 309–405.

[2] P. Dewan, R. Choudhary, and H. Shen, "An editing-based characterization of the design space of collaborative applications," *Journal of Organizational Computing*, vol. 4, no. 3, pp. 219–240, 1994.

[3] C. A. Ellis, S. J. Gibbs, and G. Rein, "Groupware: Some issues and experiences," *Commun. ACM*, vol. 34, no. 1, pp. 39–58, 1991.

[4] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris, "Computer support for distributed collaborative writing: Defining parameters of interaction," in *Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, Oct. 1994, pp. 145–152.

[5] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 63–108, 1998.

[6] S. Greenberg and D. Marwood, "Real time groupware as a distributed system: Concurrency control and its effect on the interface," in *Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, Oct. 1994, pp. 207–217.

[7] J. Grudin, "Computer supported cooperative work: History and focus," *Journal of IEEE Computer*, vol. 27, no. 5, pp. 19–26, 1994.

[8] J. Begole, M. B. Rosson, and C. A. Shaffer, "Flexible collaboration transparency: Supporting worker independence in replicated application-sharing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 6, no. 2, pp. 95–132, 1999.

[9] C. M. Hymes and G. M. Olson, "Unblocking brainstorming through the use of a simple group editor," in *Proc. of the 1992 ACM Conf. on Computer Supported Cooperative Work*, Toronto, Ontario, Canada, Nov. 1992, pp. 99–106.

[10] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proc. of the 1989 ACM SIGMOD International Conf. on Management of Data*, Portland, OR, June 1989, pp. 399–407.

[11] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements," in *Proc. of the 1998 ACM Conf. on Computer Supported Cooperative Work*, Seattle, WA, Nov. 1998, pp. 59–68.

[12] R. Bentley and P. Dourish, "Medium versus mechanism: Supporting collaboration through customisation," in *Proc. of the European Conf. on Computer Supported Cooperative Work (ECSCW'95)*, Stockholm, Sweden, Sept. 1995, pp. 131–146.

[13] J. Grudin, "Why CSCW applications fail: Problems in the design and evaluation of organization of organizational interfaces," in *Proc. of the 1988 ACM Conf. on Computer Supported Cooperative Work*, Portland, OR, Sept. 1988, pp. 85–93.

[14] D. Li and R. Li, "Preserving operation effects relation in group editors," in *Proc. of the 2004 ACM Conf. on Computer Supported Cooperative Work*, Chicago, IL, Nov. 2004, pp. 457–466.

[15] S. Noel and J. Robert, "Empirical study on collaborative writing: What do

co-authors do, use, and like," *Computer Supported Cooperative Work*, vol. 13, no. 1, pp. 63–89, 2004.

[16] M. Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser, "An integrating, transformation-oriented approach to concurrency control and undo in group editors," in *Proc. of the ACM CSCW'96 Conf. on Computer Supported Cooperative Work*, Boston, MA, Nov. 1996, pp. 288–297.

[17] M. Suleiman, M. Cart, and J. Ferrié, "Serialization of concurrent operations in a distributed collaborative environment," in *ACM GROUP'97 Proc.*, Phoenix, AZ, Nov. 1997, pp. 435–445.

[18] H. Shen and C. Sun, "Flexible notification for collaborative systems," in *Proc. of the 2002 ACM Conf. on Computer Supported Cooperative Work*, New Orleans, LA, Nov. 2002, pp. 77–86.

[19] D. Li and R. Li, "Ensuring content and intention consistency in real-time group editors," in *Proc. of the 24th IEEE International Conf. on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, Mar. 2004, pp. 748–755.

[20] P. Molli, H. Skaf-Molli, G. Oster, and A. Imine, "Using the transformational approach to build a safe and generic data synchronizer," in *ACM Conf. on Supporting Group Work (GROUP'03)*, Sanibel Island, FL, Nov. 2003, pp. 212–220.

[21] A. Prakash and M. J. Knister, "A framework for undoing actions in collaborative systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 4, pp. 295–330, 1994.

[22] M. Ionescu and I. Marsic, "Tree-based concurrency control in distributed groupware," *CSCW: The Journal of Collaborative Computing*, vol. 12, no. 3, pp.

329–350, 2003.

[23] C. D. Correa and I. Marsic, “An optimization approach to group coupling in heterogeneous collaborative systems,” in *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work, GROUP 2005*, Sanibel Island, FL, Nov. 2005, pp. 274–283.

[24] N. Gu, J. Yang, and Q. Zhang, “Consistency maintenance based on the mark & retrace technique in groupware systems,” in *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work, GROUP 2005*, Sanibel Island, FL, Nov. 2005, pp. 264–273.

[25] M. Suleiman, M. Cart, and J. Ferrié, “Concurrent operations in a distributed and mobile collaborative environment,” in *Proc. of the IEEE ICDE'98 International Conf. on Data Engineering*, Feb. 1998, pp. 36–45.

[26] B. R. Badrinath and Krithi Ramamritham, “Semantics-based concurrency control: Beyond commutativity,” *ACM Transactions on Database Systems*, vol. 17, no. 1, pp. 163–199, Mar. 1992.

[27] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, New York: Addison-Wesley, 1987.

[28] W. E. Weihl, “Commutativity-based concurrency control for abstract data types,” *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1488–1504, 1988.

[29] H. Yu and A. Vahdat, “Design and evaluation of a conit-based continuous consistency model for replicated services,” *ACM Transactions on Computer Systems*, vol. 20, no. 3, pp. 239–282, 2002.

[30] D. Sun, S. Xia, C. Sun, and D. Chen, "Operational transformation for collaborative word processing," in *Proc. of the 2004 ACM Conf. on Computer Supported Cooperative Work*, Chicago, IL, Nov. 2004, pp. 437–446.

[31] C. Sun, "Undo as concurrent inverse in group editors," *ACM Trans. Comput.-Hum. Interact.*, vol. 9, no. 4, pp. 309–361, 2002.

[32] A. H. Davis, C. Sun, and J. Lu, "Generalizing operational transformation to the standard general markup language," in *Proc. of the 2002 ACM Conf. on Computer Supported Cooperative Work*, New Orleans, LA, Nov. 2002, pp. 58–67.

[33] N. Vidot, M. Cart, J. Ferrie, and M. Suleiman, "Copies convergence in a distributed realtime collaborative environment," in *Proc. of ACM CSCW'00 Conf. on Computer Supported Cooperative Work*, Philadelphia, PA, Dec. 2000, pp. 171–180.

[34] R. Li, D. Li, and C. Sun, "A time interval based consistency control algorithm for interactive groupware applications," in *IEEE International Conf. on Parallel and Distributed Systems (ICPADS)*, Newport Beach, CA, July 2004, pp. 429–436.

[35] D. Li and R. Li, "An approach to ensuring consistency in peer-to-peer real-time group editors," *Computer Supported Cooperative Work*, To appear.

[36] P. Bellini, P. Nesi, and M. B. Spinu, "Cooperative visual manipulation of music notation," *ACM Transactions on Computer-Human Interaction*, vol. 9, no. 3, pp. 194–237, 2002.

[37] C. Sun and D. Chen, "Consistency maintenance in real-time collaborative graphics editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 9, no. 1, pp.

1–41, 2002.

[38] R. Li and D. Li, "Commutativity-based concurrency control in groupware," Tech. Rep., Computer Science, Texas A&M University, July 2005, http://cocasoft.csdl.tamu.edu/∼lidu/papers/lbto.pdf.

[39] R. Li and D. Li, "A landmark-based transformation approach to concurrency control in group editors," in *Proc. of the ACM GROUP'05 Conf. on Supporting Group Work*, Sanibel Island, FL, Nov. 2005, pp. 284–293.

[40] R. Li and D. Li, "A new operational transformation framework for real-time group editors," *IEEE Transactions on Parallel and Distributed Systems*, To appear.

[41] A. E. Abbadi D. Agrawal and A. K. Singh, "Consistency and orderability: semantics-based correctness criteria for databases," *ACM Transactions on Database Systems*, vol. 18, no. 3, pp. 460–486, 1993.

[42] M. J. Fischer and A. Michael, "Sacrificing serializability to attain high availability of data in an unreliable network," in *Proc. of ACM Symposium on Principles of Database Systems*, Los Angeles, CA, Mar. 1982, pp. 70–75.

[43] T. Berlage, "A selective undo mechanism for graphical user interfaces based on command objects.," *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 3, pp. 269–294, 1994.

[44] A. Prakash and H. S. Shim, "Distview: Support for building efficient collaborative applications using replicated objects," in *Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, Oct. 1994, pp. 153–164.

[45] G. D. Abowd and A. J. Dix, "Giving undo attention.," *Interactive with Computers*, vol. 4, no. 3, pp. 317–342, 1992.

[46] Rui Li and Du Li, "A regional undo mechanism for text editing," in *The 5th International Workshop on Collaborative Editing Systems (IWCES-5), jointly with the ECSCW'03 Conf.*, Helsinki, Finland, Sept. 2003.

[47] H. Shen and C. Sun, "A log compression algorithm for operation-based version control systems," in *Proc. of IEEE International Computer Software and Application Conf.*, Oxford, England, Aug. 2002, pp. 867–872.

VITA

Rui Li was born in Sheng Yang, Liao Ning Province, China. He received his B.E. and M.E. degrees in Aircraft Design and Applied Mechanics from Beijing University of Aeronautics & Astronautics in 1994 and 1997, and his M.E. degree in Computer Science from Johns Hopkins University in 2001. He began pursuing a Ph.D. degree in Computer Science at Texas A&M University in 2002. Since then, he has worked as a graduate teaching assistant and research assistant for Dr. Du Li in the Department of Computer Science, Texas A&M University. His permanent address is: Department of Computer Science, Texas A&M University, 301 Harvey R. Bright Bldg, College Station, TX 77843-3112.