Operational Transformation for Collaborative Word Processing

David Sun, Steven Xia, Chengzheng Sun, and David Chen
School of Computing and Information Technology
Griffith University
Brisbane, Qld 4111, Australia
{D.Sun,Q.Xia,C.Sun,D.Chen}@griffith.edu.au

ABSTRACT

Operational Transformation (OT) is a technique originally invented for supporting consistency maintenance in collaborative text editors. Word processors have much richer data types and more comprehensive operations than plain text editors. Among others, the capability of updating attributes of any types of object is an essential feature of all word processors. In this paper, we report an extension of OT for supporting a generic *Update* operation, in addition to *Insert* and *Delete* operations, for collaborative word processing. We focus on technical issues and solutions involved in transforming *Updates* for both consistency maintenance and group undo. A novel technique, called Multi-Version Single-Display (MVSD), has been devised to resolve conflict between concurrent *Updates*, and integrated into the framework of OT. This work has been motivated by and conducted in the CoWord project, which aims to convert MS Word into a real-time collaborative word processor without changing its source code. This OT extension is relevant not only to word processors but also to a range of interactive applications that can be modelled as editors.

Categories and Subject Descriptors

H.4.1 [**Information Systems Applications**]: Office Automation—*Word processing*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*Collaborative computing, Synchronous interaction*

General Terms

Algorithms, Design, Theory

Keywords

Operational transformation, multi-versioning, group undo

1. INTRODUCTION

Operational Transformation (OT) is a technique for consistency maintenance and group undo in collaborative systems [5, 15, 16, 17, 14, 20, 21, 18, 22]. It was originally devised to support multiple users to *insert* and *delete* characters in text documents concurrently [5]. Due to its unique capability of achieving system consistency without imposing restrictions on users, OT has become the choice of consistency maintenance and group undo technique for many collaborative text editors, including Grove [5], DistEdit [15], Jupiter [13], Joint Emacs [17], REDUCE [21, 18], Flexible JAMM [1], ICT [10], and TreeOPT [6].

Word processors are among the most widely used computer applications, but none of the existing word processors is able to support real-time collaboration. Compared with plain text editors, word processors have much richer data types (text, graphics, images, etc.), more complex document structures, and more comprehensive editing operations. In addition to supporting object creation and deletion, word processors also support users to *update* attributes of existing objects. Word processors would be little more than automated typewriters if they support only inserting and deleting texts. Their status as one of the most pervasive computer applications can be largely attributed to their capability for supporting users to *customize* or *update* the "look-and-feel" of their letters, memos, forms, or reports. Word processor users typically spend long hours formatting the document during and after the composition process. For collaborative word processors to adequately support group work, it is essential to have the collaborative updating capability.

In this paper, we report an extension of OT for supporting concurrent execution of *Update* operations, in addition to *Insert* and *Delete* operations, on objects of arbitrary types and complex structures in collaborative word processors. This work has been motivated by and conducted in the context of the CoWord project,[1] which aims to convert MS Word into a real-time collaborative word processor without changing its source code [23]. For issues related to the mapping from application-specific operations to generic *Update*, *Insert*, and *Delete* operations, the reader is referred to [23]. The focus of this paper is on the technical issues and solutions involved in transforming the generic *Update* operation for both consistency maintenance and group undo. This OT extension is relevant not only to word processors but also to a range of interactive applications that can be modelled as editors [3].

The rest of this paper is organized as follows. First, the basic OT extension strategy and *Update*-related transformation functions are discussed in Section 2. Then, a novel conflict resolution technique, called Multi-Version Single-

[1]CoWord Demo: http://reduce.qpsf.edu.au/coword.

Display (MVSD), is devised and integrated in the OT framework in Section 3. The extended OT is applied to support consistency maintenance and group undo in Section 4 and Section 5, respectively. In Section 6, the MVSD technique is compared with alternative conflict resolution techniques, and prior OT extensions for supporting *update*-like operations are discussed. Finally, major contributions of this work are summarized in Section 7.

2. EXTENDING OT FOR UPDATES

2.1 Basic strategy

The OT component in a collaborative editor is a complex system and is divided into two layers [5, 17, 21, 18]: the high-level transformation control algorithms, and the low-level transformation functions. Transformation control algorithms are responsible for determining which operation should be transformed against other operations according to their concurrency relationships; and transformation functions determine how to transform one operation against another operation according to operation types, parameters and other relationships. To extend OT for supporting *Updates*, our strategy is to keep the high-level control algorithms unchanged, but add new *Update*-related transformation functions. This strategy allows us to localize the extension and reduce the complexity.

The design of transformation functions, however, is not completely independent of control algorithms: different control algorithms impose different requirements on transformation functions [20, 18], and transformation functions must fit in the framework of the corresponding control algorithms. Moreover, since OT is at the core for supporting both consistency maintenance and group undo [17, 18], the design of transformation functions for *Updates* must consider the needs for both consistency maintenance and group undo. In this paper, the design and analysis of the transformation functions for *Updates* are discussed in the framework of GOTO [20] and AnyUndo [18], which are the OT control algorithms in the CoWord system. However, the major issue – conflict resolution – is general, and the solutions reported in this paper are adaptable to the framework of other OT systems as well.

2.2 Notations for primitive operations

We assume that the objects in a document are *addressable* from a linear address space. Addresses in this linear address space ranges from 0 to $N-1$, where N is the total number of objects in the document. Based on this linear address space, three primitive editing operations are defined and denoted as follows:

1. *I(p, obj)* denotes an *Insert* operation to create object *obj* at position *p*.
2. *D(p, obj)* denotes a *Delete* operation to remove object *obj* at position *p*.
3. *U(p, key, new-value, old-value)* denotes an *Update* operation to change the attribute *key* of the object at position *p* from *old-value* to *new-value*.

The *dot* notation is used to refer a parameter of an operation O, e.g. $O.p$ refers to the p parameter, $O.obj$ refers to the *obj* parameter, $O.key$ refers to the *key* parameter, etc.

It should be pointed out that the parameters of the three primitive operations are defined for the purpose of supporting both *do* and *undo*. For example, the *obj* parameter is not necessary for performing a *Delete*, but needed to save the deleted object for generating an inverse operation for undo [18]. Also, the *old-value* parameter is not needed for performing an *Update*, but needed to save the updated attribute value for undoing an *Update*. Under the above definitions, *Insert* and *Delete* are inverse operations to each other; and the inverse of an *Update* is obtained by swapping its *new-value* and *old-value*, as shown below:

1. $\overline{I}$*(p, obj) = D(p, obj).*
2. $\overline{D}$*(p, obj) = I(p, obj).*
3. $\overline{U}$*(p, key, new-value, old-value) =*
   U(p, key, old-value, new-value).

For the sake of simplicity, we have chosen to restrict the scope of an individual operation to only one object (at position p). In real systems, these operations can cover a sequence of objects by using one additional parameter *length*. For transformation functions defined for a sequence of objects (or string-wise operations for text editing), the reader is referred to [21].

2.3 Transformation functions for Updates

Two types of transformation function have been used in OT systems based on a linear history buffer [21]: one is the *Inclusion Transformation* function – $IT(O_a, O_b)$, which transforms operation O_a against operation O_b in such a way that the impact of O_b is effectively included in the parameters of O_a; and the other is the *Exclusion Transformation* function – $ET(O_a, O_b)$, which transforms O_a against O_b in such a way that the impact of O_b is effectively excluded from the parameters of O_a. In this paper, only IT functions for *Updates* are presented because they are representative in addressing the major technical issues and solutions involved. The reader can derive corresponding ET functions based on the techniques presented in this paper and [21].

The definition of five *Update*-related (inclusion) transformation functions are given in Fig. 1. The *Update* operation has an important property: the execution of an *Update* has no effect on the linear address space, which is different from *Insert* and *Delete* whose execution changes the linear address space of the document. This property has helped simplify the design of *Update*-related transformation functions.

When an *Insert* is transformed against a concurrent *Update*, no change is made to the *Insert* since the *Update* has no effect on the linear address space (see function *Tiu*). However, when an *Update* is transformed against an *Insert*, the *Update*'s position is incremented by one if the *Update*'s position is greater than or equal to the *Insert*'s position (see function *Tui*).

The transformation between an *Update* and a *Delete* is similar to that between an *Update* and an *Insert*, except when an *Update* and a *Delete* target the same object, i.e., *U.p == D.p*, a subroutine *applyUpdate(U, D.obj)*[2] is invoked to apply the *Update* to the deleted object in function *Tdu*;

[2]The application of the *Update* to the deleted object is needed for supporting undoing *Delete*. The definition of *appplyUpdate( )* is application-specific, and shall not be covered in this paper.

```
Tiu(I, U)
{
  return I;
}

Tui(U, I)
{ if(U.p >= I.p) U.p++;   //shift to right by one
  return U; }

Tdu(D, U)
{ if(D.p == U.p) applyUpdate(U, D.obj);
  return D; }

Tud(U, D)
{ if(U.p > D.p) U.p--;    //shift to left by one
  else if (D.p == U.p) setNULL(U);
  return U; }

Tuu(Ua, Ub)
{ if(Ua.p != Ub.p)||(Ua.key != Ub.key)
     return Ua;     // for compatible operations
  return conflictResolution(Ua,Ub); }
```

Figure 1: Update-related IT functions.

or a subroutine *setNULL( )* is invoked to set the *Update* to *NULL*[3] in function *Tud*.

To transform two *Updates*: *Ua* against *Ub*, if they are targeting different objects (*Ua.p != Ub.p*), or changing different attributes (*Ua.key != Ub.key*), then they are compatible and no change needs to be made to *Ua*, as shown in function *Tuu*. Otherwise, they conflict with each other, and a subroutine *conflictResolution(Ua, Ub)* (to be defined later) is invoked to resolve the conflict.

The need to resolve conflict between two concurrent *Updates* brings up the major technical challenge in extending OT for *Updates*. In existing OT techniques, transformation functions for *Insert* and *Delete* are based on a simple transformation technique: (1) compare the positional parameters of two concurrent operations to determine their relative positions (left or right) in the linear address space; and (2) assume one of them has been executed to determine its impact on the other operation's position and adjust that the position accordingly. This transformation technique for *Insert* and *Delete* is no longer relevant to the transformation of *Updates* since *Updates* do not have any impact on each other's positional references. However, conflicting *Updates* may have impact on each other's attribute parameters. What is needed is a new technique for transforming attribute parameters of *Updates* so that consistent effects of conflicting operations can be achieved.

3. CONFLICT RESOLUTION IN OT

3.1 Conflict and compatible relationship

Conflict and compatible relationships among editing operations have been studied in detail in the GRACE project for collaborative graphic editing [19]. Following GRACE, a precise definition of conflict relationship among *Updates* is given below.

[3] *NULL* is an empty operation without any effect on the document or in transformation.

Definition 1. *Conflict relation "$\otimes$".* Two *Updates Ua* and *Ub* conflict with each other, expressed as $Ua \otimes Ub$, if and only if: (1) they are concurrent;[4] (2) they are targeting at the same object, i.e. $Ua.p = Ub.p$; and they are updating the same attribute, i.e. $Ua.key = Ub.key$. □

Compared to GRACE conflict relationship Definition 3 in [19], the above conflict definition differs in two points: First, objects are identified by positional references to meet the basic OT requirement, rather than by a special object identification scheme in GRACE. Second, when two *Updates* are targeting the same object and changing the same attribute to the *same value*, they are defined as conflicting operations here, rather than as compatible operations in GRACE. The second point helps simplify the design and integration of a conflict resolution solution in the OT framework.

3.2 The GRACE Multi-Versioning technique

Given two concurrent *Updates*: *Ua* and *Ub*. If they conflict, there are three possible types of combined effects that can be achieved to ensure consistency [12, 19]:

1. *Null-effect*: none of the operations has any final effect on the target object.

2. *Single-operation-effect*: only one operation, either *Ua* or *Ub*, has a final effect on the target object.

3. *All-operations-effect*: all operations have final effects on the target object.

In GRACE [19], a Multi-Versioning (MV) technique was devised to achieve the *all-operations-effect*: multiple versions of the same object are created to accommodate the effects of multiple conflicting *Updates*. Compared to the *null-effect* and *single-operation-effect* approaches [7, 8], the MV technique has the advantage of being able to preserve all users' work even in the face of conflict. This technique provides users with a complete picture about what other users intended to do in the situation of conflict, so that they could better assess the situation and react accordingly.

The main technical challenge to the MV technique is how to maintain multiple versions internally and how to present multiple versions on the user interface. In GRACE, a sophisticated algorithm has been developed to ensure consistent creation and manipulation of multiple versions; and a simple interface strategy has been used to present multiple versions: multi-versioned objects are highlighted so that users can differentiate them from single-versioned objects [19].

3.3 Problems in applying GRACE MV to OT

Under the OT framework, an *Update* may change the attribute of an existing object, but have no effect on existing objects' positional references. This property has been explored to simplify the design of *Update*-related transformation functions (see Section 2.3). However, if the GRACE MV technique were directly applied in the OT framework, then the execution of conflicting *Updates* may result in new object versions created in the document, which would have equivalent effects as inserting new objects in the linear address space. This would violate the *Update*'s property of

[4] Two operations are concurrent if they are generated without any knowledge of each other. For formal definitions of operation concurrency, the reader is referred to [9, 21].

having no effect on the linear address space, and hence significantly complicate the design of *Update*-related transformation functions.

Moreover, for the GRACE MV technique to work, the identifier of an object must contain sufficient information in order to trace multiple versions of the same object [19]. In GRACE, we had the freedom to design any identification scheme as long as it meets the need. However, under the OT framework, all objects are identified by simple positional references, which contain no information for tracing multiple versions of the same object; and we cannot freely modify OT's linear identification scheme because OT's functioning is intimately dependent on it. This means the GRACE MV technique cannot be directly applied to the OT framework.

3.4 Multi-Version Single-Display

To avoid significantly complicating the OT framework in adopting the basic MV technique for conflict resolution, we devised a new MV technique, called Multi-Version Single-Display (MVSD). The basic idea of MVSD is the following: when an object is updated by conflicting operations, multiple versions of the target object shall be maintained internally (similar to GRACE), but only one version is displayed at the user interface (different from GRACE). Moreover, all versions of an object can be displayed (one by one) by invoking a multi-version management scheme (to be discussed in Section 5.4). The major merit of MVSD is that it fits very well in the OT framework (explained below).

3.4.1 Multi-version maintenance under OT

Since only one version is displayed at a time in MVSD, there is no need to actually create the multiple versions of an object in the document as long as (1) sufficient information for alternative versions are maintained internally, and (2) mechanisms exist in the system to dynamically create and display any version at any time. Under the condition of creating one version at a time in a document, we can keep a single position in the OT linear address space for all versions of the same object, and map alternative versions to the same position one by one at different times. This strategy allows us to preserve the *Update*'s property of having no effect on the OT linear space (thus keeping the simplicity of *Update*-related transformation functions), and to use a single positional reference to identify multiple versions (at different times).

The above strategy implies that we have to maintain state information of alternative object versions outside the document itself. In the OT framework, there is a History Buffer (HB) used to save all executed operations in sequence [20]. The current document state, including the current states of all objects, can be determined by sequentially applying the operations in HB to the initial document state. In other words, HB is the data structure for OT to maintain the document state, as well as object states. Therefore, it is natural to use HB to maintain state information of multiple versions, and this can be achieved by saving all conflicting operations (in proper forms) in HB.

3.4.2 Single displayed version selection

Apart from the issues of maintaining and identifying multiple versions in the OT framework, another technical issue is how to ensure consistent selection of the single displayed version at all collaborating sites. This issue is addressed by devising a *Priority Assignment Scheme* (Definition 2), which assigns *priorities* to the new and old attribute values of each *Update*. We require the priorities assigned to the *new-values* of different *Updates* to be totally and causally ordered [9, 21]. One way of generating such priorities in distributed collaborative systems is to use the total ordering relationship defined by the *State Vector* and the site identifier of an operation [21]. Without losing generality, we use a sequence of positive integers (from 0 to infinite) to represent the totally and causally ordered priorities in this paper for the sake of conciseness.

Definition 2. *Priority Assignment Scheme (PAS):* When an *Update* is created and executed at the local site, its *new-value* parameter shall be assigned with the current highest priority available in the priority sequence, and its *old-value* shall be assigned with the lowest priority (i.e. zero). □

The priority assigned to an *Update*'s *new-value* at the time of its generation is called the *initial priority* of this operation. Based on PAS, the following *Single Displayed Version Effect* (SDVE) for conflicting *Updates* is defined.

Definition 3. *Single Displayed Version Effect (SDVE).* Given a group of conflicting *Updates* $G = \{U1, U2, ..., Un\}$. After executing all operations in G in any order, their combined effect on the single displayed object version must be the effect of the operation whose initial priority is the highest among all operations in G. □

It should be stressed that the priority of an *Update* is determined by the underlying system, is independent of the operation type and parameters, and does not reflect the user's role in their collaborative work. The rationale behind this design is the principle of separating the mechanism from the policy: the low-level single version display selection (the mechanism) should be kept simple and generic, and the high-level user needs can be better addressed by collaboration policy modules on top of the generic mechanism. After all, all versions are maintained internally in MVSD and any version can be selected for display by invoking a high-level multi-version management scheme (to be discussed in Section 5.4).

To illustrate the basic idea of MVSD in combination with the PAS and SDVE schemes, consider the following three concurrent *Updates* (Note: a pair *(value, priority)* is used to denote an attribute *value* with its associated *priority*):

1. *U1(p, Color, (Red, 3), (Dark, 0))*,
2. *U2(p, Color, (Green, 2), (Dark, 0))*, and
3. *U3(p, Color, (Blue, 1), (Dark, 0))*,

which set the *Color* attribute of the same object at position p from *Dark* to *Red* (by *U1*), to *Green* (by *U2*), and to *Blue* (by *U3*), respectively. Clearly, these three operations conflict with each other, i.e. $U1 \otimes U2 \otimes U3$. Therefore, three versions of the target object, colored by *Red, Green*, and *Blue*, respectively, shall be maintained internally by the system. According to the priorities denoted in these operations and SDVE, the single displayed object version should have a *Red* color since *U1.new-value = (Red, 3)* has the highest initial priority. How to transform *Updates* to achieve the effect determined by SDVE is discussed in the next section.

3.5 Transformation rules for Updates

If a group of conflicting *Update* operations are known in advance, then achieving the correct SDVE result is simply a matter of selecting and executing the *Update* with the highest initial priority. However, in a distributed collaborative computing environment, it is infeasible to know all concurrent *Updates* before the execution of an *Update*. This is because that a local *Update* must be executed immediately to achieve high responsiveness, regardless of whether there exist any concurrent *Updates* generated at other sites. Moreover, concurrent *Updates* may arrive and be executed in different orders at different sites. Therefore, the challenge is how to transform concurrent *Updates* so that (1) the execution of transformed *Updates* in any order can achieve the same SDVE result; and (2) the transformed *Update* parameters carry sufficient information for supporting group undo and for creating alternative versions later.

Consider the two conflicting *Updates U1* and *U2* in our previous example. According to their priority assignments, no matter in which order they are executed, they must achieve the same SDVE result: setting the single displayed object version's color to *Red*. Suppose they are executed in the order of *U2* followed by *U1*. After the execution of *U2*, the object's color is set to *Green*. We can achieve the correct SDVE result by executing *U1* as-is, i.e. without any change to *U1.new-value*. In this case, the effect of *U2* is naturally masked by *U1*. In another execution order: *U1* followed by *U2*, the object's color is set to *Red* after the execution of *U1*. We can achieve the correct SDVE result by *propagating U1.new-value* to *U2.new-value* or by *voiding U2.new-value*.

Generally, if a group of *Updates* are executed in the ascending order of their initial priorities, then the correct SDVE result can be achieved by their natural *masking* effects without any change to their attribute parameters. However, if an *Update* with a higher initial priority is executed before any *Update* with a lower initial priority, the correct SDVE result can be achieved by either propagating the higher priority operation's *new-value* to that of the lower priority operation, or voiding the lower priority operation's *new-value*. The *new-value propagation* strategy is favored because of the consideration in transforming the *old-value* parameter for supporting group undo (see below). Based on this analysis, the following general transformation rule for the *new-value* parameter of an *Update* is defined.

Definition 4. *Transformation Rule for New Value (TRNV):* Given two conflicting *Updates Ua* and *Ub*. Suppose *Ub* is executed before *Ua*. In transforming *Ua* against *Ub*, if the priority of *Ua.new-value* is lower than the priority of *Ub.new-value*, then *Ua.new-value* must be replaced by *Ub.new-value*; otherwise, no change is made to *Ua.new-value*. □

The purpose of the *old-value* parameter of an *Update* is to keep track of the *background* attribute value corresponding to the current *new-value* parameter of the same operation. This background attribute value is needed for generating the correct inverse operation for undoing an *Update*. When an *Update* was initially generated, its *old-value* recorded the background attribute value before its generation. When *Ua* is transformed against *Ub*, *Ua.old-value* should be replaced by *Ub.new-value* since the latter carries the *current* background attribute value of *Ua* (thanks to the *new-value propagation* strategy used in TRNV).[5] Therefore, we have the following general transformation rule for the *old-value* parameter of an *Update*.

Definition 5. *Transformation Rule for Old Value (TROV):* Given two conflicting *Updates Ua* and *Ub*. Suppose *Ub* is executed before *Ua*. In transforming *Ua* against *Ub*, *Ua.old-value* must always be replaced by *Ub.new-value*. □

Basically, an unconditional *propagation* strategy is used in TROV to propagate the *new-value* of a previously executed operation to a later executed operation's *old-value*.

```
conflictResolution(Ua, Ub)
{
  if(Ua.new-value.priority < Ub.new-value.priority)
     Ua.new-value = Ub.new-value;
  Ua.old-value = Ub.new-value;
  return Ua;
}
```

Figure 2: Conflict resolution function.

Based on TRNV and TROV, the *conflictResolution()* function is sketched in Fig. 2. The *conflictResolution()* function is clean and simple, but the issues involved in its design are actually very intricate and its correctness can only be verified by applying it for consistency maintenance under the control of GOTO, and for group undo (and multi-version management) under the control of AnyUndo, which are discussed in the following sections.

4. EXECUTING UPDATES UNDER GOTO

According to the GOTO algorithm [20, 18], when an operation arrives at a site, HB is searched to find out all operations that are concurrent with the newly arrived operation. These concurrent operations are transformed and shifted to the right part of HB. Then, the newly arrived operation is transformed against them one by one. Finally, the transformed new operation is executed and appended at the end of HB.

To examine how the *conflictResolution()* function works when it is applied to a sequence of operations under the control of the GOTO algorithm, consider the three concurrent operations in our running example again. The three *Updates* may be transformed and executed in six different orders. The parameters of the transformed operations and the object's color after executing each transformed operation in all six orders are given in Table 1.

To explain the results in this table, consider Do-Order(2, 1, 3). First, *U2* is executed and the object's color is set to *Green*. Second, when *U1* arrives, it must be transformed against *U2*. Since *U1.new-value (= (Red, 3))* has a higher priority than *U2.new-value (= (Green, 2))*, *U1.new-value* remains unchanged but *U1.old-value* is replaced with *U2.new-value*. The execution of the transformed *U1* shall set the object's color to *Red*, which is obviously the correct SDVE result for *U2* and *U1*. Third, when *U3* arrives, it must be transformed against *U2* and *U1* in sequence. After transformation against *U2*, *U3.new-value* is replaced with *U2.new-*

[5]The purpose of transforming *Ua* against *Ub* is to adjust the parameters of *Ua* under the assumption that *Ua* was executed immediately after *Ub*, so *Ub.new-value* must represent the current background attribute value of *Ua*.

Do-Order	Transformed operations	Object Color
(1, 2, 3)	*U1(p, Color, (Red, 3), (Dark, 0))*	Red
	U2(p, Color, (Red, 3), (Red, 3))	Red
	U3(p, Color, (Red, 3), (Red, 3))	Red
(1, 3, 2)	*U1(p, Color, (Red, 3), (Dark, 0))*	Red
	U3(p, Color, (Red, 3), (Red, 3))	Red
	U2(p, Color, (Red, 3), (Red, 3))	Red
(2, 1, 3)	*U2(p, Color, (Green, 2), (Dark, 0))*	Green
	U1(p, Color, (Red, 3), (Green, 2))	Red
	U3(p, Color, (Red, 3), (Red, 3))	Red
(2, 3, 1)	*U2(p, Color, (Green, 2), (Dark, 0))*	Green
	U3(p, Color, (Green, 2), (Green, 2))	Green
	U1(p, Color, (Red, 3), (Green, 2))	Red
(3, 1, 2)	*U3(p, Color, (Blue, 1), (Dark, 0))*	Blue
	U1(p, Color, (Red, 3), (Blue, 1))	Red
	U2(p, Color, (Red, 3), (Red, 3))	Red
(3, 2, 1)	*U3(p, Color, (Blue, 1), (Dark, 0))*	Blue
	U2(p, Color, (Green, 2), (Blue, 1))	Green
	U1(p, Color, (Red, 3), (Green, 2))	Red

Table 1: Transformed operations and their execution effects on the object's color in different orders.

value since the latter has a higher priority, and *U3.old-value* is also replaced with *U2.new-value*. After transformation against *U1*, *U3.new-value* is replaced with *U1.new-value* since the latter has a higher priority, and *U3.old-value* is replaced with *U1.new-value*. The execution of the transformed *U3* shall reset the object's color to *Red*, which is again the correct SDVE result for these three operations.

An important observation can be drawn from the results in Table 1: after transforming and executing three operations in any order, the final color of the target object is always *Red*, which is the correct SDVE result for these three operations. In general, it can be verified that, for a group of conflicting *Update* operations $G = \{U1, U2, ..., Un\}$, after transforming them in any order under the control of the GOTO algorithm, the *new-value* and *old-value* parameters of the transformed operations in HB must be sorted in an ascending order of their priorities, thanks to the attribute value *propagation* strategy used in the *conflictResolution()* function. Particularly, the last transformed operation *U* must have a *new-value* with the highest priority among all operations in *G* (including *U*), and an *old-value* with the highest priority among all operations in *G* - {*U*} (excluding *U*). These properties are essential for ensuring the consistent SDVE result and for ensuring the correct undo effect for *Updates* (see the next section).

It is known that a transformation function *T* used by an OT control algorithm (like GOTO) that allows arbitrary execution and transformation paths should maintain the following two transformation properties for ensuring convergence [17, 20]:

TP1: $O_a \circ O'_b \equiv O_b \circ O'_a$

TP2: $T(T(O, O_a), O'_b) = T(T(O, O_b), O'_a)$

where O, O_a, and O_b are concurrent operations defined on the same state, $O'_a = T(O_a, O_b)$, and $O'_b = T(O_b, O_a)$. A formal proof of maintaining TP1 and TP2 by *Tuu* is beyond the space limitation of this paper, so we give a sketch of an informal proof as follows.

TP1 is maintained because the TRNV rule (Definition 4) ensures that the later executed operation, either O'_a or O'_b, shall carry the same *new-value* with the highest priority among O_a and O_b. TP2 is maintained because the TRNV and TROV rules (Definitions 4 and 5) jointly ensures that the last executed operation O shall carry a *new-value* with the highest priority among O, O_a, and O_b, and an *old-value* with the highest priority among O_a and O_b, regardless in which order O_a and O_b were transformed. The reader can use the results in Table 1 to check the validity of these properties. For example, Do-Order(1, 2, 3) and Do-Order(2, 1, 3) give the same final *U3(p, Color, (Red, 3), (Red, 3))*.

5. UNDOING UPDATES UNDER ANYUNDO

5.1 Undo effect for Updates

According to the general definition of *Undo Effect* in [18], after an operation is undone, this operation's effect shall be eliminated, but the effects of all other operations shall be retained. Operations may be undone in any order, but after undoing all operations, the document should be restored to the original state as if these operations were never executed, which is specified as the *Undo Property* in [18].

The undo-effect of an operation is directly related to its do-effect. Apart from meeting the general undo-effect definition, we must take into account of the SDVE do-effect in defining the undo-effect of conflicting *Updates*. For any *Update Ux*, if *Ux*'s do-effect is visible on the current displayed object version, then undoing *Ux* should eliminate its visible do-effect and display the SDVE result determined by all non-undone operations performed on the same object attribute. However, if *Ux*'s do-effect is not visible on the current displayed object version, then undoing *Ux* should not have any visible undo-effect on the current displayed object, but shall have the visible effect when all operations with higher priorities have been undone. More precisely, we define the undo effect for *Updates* on the single displayed version as follows.

Definition 6. *Undo Effect on the Single Displayed Version* Given a group of non-undone *Updates* $G = \{U1, U2, ..., Un\}$ that target the same object and the same attribute. After undoing any operation *Ui*, $1 \le i \le n$, the undo effect on the single displayed version is determined by the SDVE result for (*G* - {*Ui*}). □

According to the above definition, if the SDVE results for (*G* - {*Ui*}) and *G* are the same, then, undoing *Ui* does not have a visible effect; otherwise, it has a visible effect. To illustrate, consider the three *Updates* in our running example again. These three *Updates* can be undone in six different orders. According to Definition 6, the undo-effects on the displayed object version's color in different orders are derived and summarized in Table 2.

To explain the results in Table 2, consider Undo-Order(3, 1, 2). First, after undoing *U3*, there should be no visible effect on the object's color *Red* since the do-effect for the two retained operations *U1* and *U2* is *Red* according to SDVE. Second, after undoing *U1*, the object's color should be changed from *Red* to *Green* since *U1*'s do-effect (*Red*) should be eliminated and the do-effect of operation *U2* (*Green*) should be retained and displayed. Third, when *U2* is undone, the displayed object's color should be restored to the original *Dark*, rather than *Blue*, since *U3* has already been

Undo-Order	Undone Operation	Undo Effect on Object Color
(3, 2, 1)	$\overline{U3}$	Red
	$\overline{U2}$	Red
	$\overline{U1}$	Dark
(2, 1, 3)	$\overline{U2}$	Red
	$\overline{U1}$	Blue
	$\overline{U3}$	Dark
(3, 1, 2)	$\overline{U3}$	Red
	$\overline{U1}$	Green
	$\overline{U2}$	Dark
(1, 3, 2)	$\overline{U1}$	Green
	$\overline{U3}$	Green
	$\overline{U2}$	Dark
(2, 3, 1)	$\overline{U2}$	Red
	$\overline{U3}$	Red
	$\overline{U1}$	Dark
(1, 2, 3)	$\overline{U1}$	Green
	$\overline{U2}$	Blue
	$\overline{U3}$	Dark

Table 2: Undo effects on object's color in different orders according to Definition 6.

undone, which means the previous invisible undo-effect of *U3* becomes visible now.

5.2 Undoing Updates in chronological order

According to the AnyUndo[6] algorithm in [18], when an operation is selected for undo, an inverse of the selected operation is generated and transformed against all following operations in HB. After executing the transformed inverse, the inverse is transposed back to the position immediately after the undone operation to make a do-undo-pair.

To examine how *Updates* are undone under the control of the AnyUndo algorithm, consider our running example again. Suppose the three *Update* operations are done in the order of *U2, U1, U3*, and then undone in the reverse chronological order: $\overline{U3}$, $\overline{U1}$, $\overline{U2}$. The transformed operations and their execution effects in doing these operations can be found in Do-Order(2, 1, 3) of Table 1. In the following, we explain the process of undoing these three operations in a reverse order. The inverse operations and their undo effects on the single displayed version achieved by the AnyUndo algorithm are shown in Table 3. To check the correctness, the results in Table 3 can be compared with the undo effects for Undo-Order (3, 1, 2) in Table 2, which are derived according to Definition 6.

Inverse Operations	Object Color
$\overline{U3}$*(p, Color, (Red, 3), (Red, 3))*	Red
$\overline{U1}$*(p, Color, (Green, 2), (Red, 3))*	Green
$\overline{U2}$*(p, Color, (Dark, 0), (Green, 2))*	Dark

Table 3: Inverse operations and their undo effects on the object's color in Undo-Order (3, 1, 2) under the control of the AnyUndo algorithm.

To undo *U3*, inverse $\overline{U3}$*(p,Color,(Red,3),(Red,3))* is generated by swapping *U3*'s *new-value* and *old-value* parameters. Then, $\overline{U3}$ is executed as-is since there is no operation in HB to be transformed against. The execution of $\overline{U3}$ shall reset the object's color to *Red*, which is clearly the correct undo-effect for $\overline{U3}$ in Undo-Order(3, 1, 2) in Table 2. Afterwards, *U3* and $\overline{U3}$ are coupled into a do-undo-pair.

To undo *U1*, inverse $\overline{U1}$*(p,Color,(Green,2),(Red,3))* is generated by swapping *U1*'s attribute value parameters. Then, $\overline{U1}$ is transformed against the *U3*-$\overline{U3}$ pair, which does not cause any change to $\overline{U1}$ because the do-undo-pair is treated as an *Identity* operation [18]. After the execution of $\overline{U1}$, the object's color shall be set to *Green*, which is clearly the correct undo-effect for $\overline{U1}$ in Undo-Order(3,1,2) in Table 2. Afterwards, *U1* and $\overline{U1}$ shall be coupled into a do-undo-pair.

To undo *U2*, inverse $\overline{U2}$*(p,Color,(Dark,0),(Green,2))* is generated by swapping *U2*'s attribute value parameters. Then, $\overline{U2}$ is transformed against the *U3*-$\overline{U3}$ pair and the *U1*-$\overline{U1}$ pair, which should not cause any change to $\overline{U2}$ for the same reason. After the execution of $\overline{U2}$, the object's color shall be set to *Dark*, which is clearly the correct undo-effect for $\overline{U2}$ in Undo-Order(3,1,2) in Table 2. Afterwards, *U2* and $\overline{U2}$ shall be coupled into a do-undo-pair.

From this example, we can see that the inverse operation generated for undoing any *Update* is always correct, which is due to TROV embedded in *conflictResolution()*.

5.3 Undoing Updates in any order

Apart from the reverse chronological undo order, the AnyUndo algorithm is capable of undoing operations in any order. The illustration of undoing *Updates* in any order is quite intricate and requires intimate knowledge on the *Inverse-Property-Preservation* capability of the AnyUndo algorithm [18]. In the following example, we shall give a sketch of undoing *U1* without undoing *U3* after executing the three operations in the order of *U2, U1, U3* (see Do-Order(2, 1, 3) in Table 1).

When *U1* is selected for undo, if its inverse $\overline{U1}$ was transformed against *U3* by means of a normal IT function, then an incorrect operation $\overline{U1}'$ *= (p, Color, (Red, 3), (Red, 3))* would be obtained, which shall produce incorrect undo effect (i.e. resetting the object color to *Red*). However, thanks to the *Inverse-Property-Preservation* capability of the AnyUndo algorithm, the IP3-preserving IT function *IP3P-IT* [18] is invoked instead. What happens inside the *IP3P-IT* function is the following: First, *U1* and *U3* are found to be concurrent at Step (1). Then, *U1* and *U3* are virtually *transposed* (by invoking a pair of ET and IT functions) at Step (1)-(b) and (c). After this transposition, *U1 = (p, Color, (Red, 3), (Green, 2))*, which has the same parameters as the *U1* in Do-Order(2, 3, 1) (see Do-Order(2, 3, 1) in Table 1). In other words, the overall effect of Step (1)-(b) and (c) is equivalent to converting Do-Order(2, 1, 3) into Do-Order(2, 3, 1). Finally, the *IP3P-IT* function returns (at Step (1)-(d)) the correct inverse $\overline{U1}$*(p, Color, (Green, 2), (Red, 3))*, which can be executed to achieve the correct selective undo effect, i.e. setting the object color to *Green*. The reader may refer to Section 7 of [18] to check its correctness.

As illustrated in this example, the transformation effect of Step (1)-(b) and (c) of the *IP3P-IT* function is equivalent to transforming and re-ordering (i.e. to transpose) the operation to be undone with the next concurrent operation in HB. The same effect can be achieved more elegantly and efficiently at the AnyUndo algorithm control level: to undo any operation (*Update*, *Insert*, or *Delete*), the selected oper-

[6]By AnyUndo, we mean the ANYUNDO-X algorithm in [18].

ation is first transposed to the end of all concurrent operations (if any) in HB. After that, an inverse of this operation is generated and transformed against all following operations (if any) which are not concurrent with this operation. This new version of the AnyUndo algorithm has been implemented in the Generic Collaboration Engine of the CoWord system [23].

5.4 AnyDisplay by AnyUndo

According to MVSD, after executing a group of conflicting *Updates*, only one version (with the highest priority) is displayed. To display the version for a particular *Update*, all conflicting *Updates* with higher initial priorities must be undone. In our running example, the version for *U2* can be displayed if *U1* is undone, and the version for *U3* can be displayed if both *U1* and *U2* are undone.

Since conflicting operations are concurrent and can be executed in any order, the selective undo capability of the AnyUndo algorithm, as shown in the previous section, is essential to support selective display of any version at any time. Based on the AnyUndo algorithm, the following AnyDisplay algorithm is defined.

```
ALGORITHM 1. AnyDisplay(U, HB)
//U is the Update operation whose version is to be displayed.
//HB is the history buffer saving executed operations.

1. Search HB to find Updates {U1, U2, ..., Un}, that
   meet the following conditions: for $1 \leq i \leq n$, (1) Ui
   conflicts with U; (2) Ui's initial priority is higher than
   U's initial priority; and (3) Ui is not undone.

2. For each operation Ui, invoke AnyUndo(HB, Id(Ui))[7]
   to undo Ui.

3. If U was undone before, invoke AnyUndo(HB, Id(U))
   to redo U.                                            □
```

It should be pointed out that our work on multi-version management in MVSD is restricted to the basic mechanisms that have been implemented in CoWord [23]. The user can use the CoWord's selective undo capability to display alternative versions. This is a rather low level user interface for multi-version management. Work is on the way to investigate and design a higher level user interface which allows the user to browse alternative versions without any knowledge of the relationship between undo and multi-version management.

6. DISCUSSIONS

6.1 MVMD, MVSD, and SVSD

Using the MV technique to resolve conflicts was pioneered by the TIVOLI system [12] and the GRACE system [19]. Both TIVOLI and GRACE MV techniques can be regarded as a kind of *Multi-Version Multi-Display* (MVMD) technique since they maintain and display multiple versions of the same object at the same time on the user interface. The traditional *Single-Operation-Effect* approach [7, 8] to conflict resolution can be regarded as a kind of *Single-Version Single-Display* (SVSD) technique since it maintains and displays only one operation's effect. The MVSD technique proposed in this paper is different from both MVMD and SVSD since it maintains multiple versions at all times, but displays only one version at a time on the user interface.

[7]The definition of the AnyUndo() function can be found in [18].

To illustrate the differences between these three techniques, consider the scenarios in Fig. 3. The history of an object *Obj* is represented as a tree of state nodes. In such an object state tree, a root node, labelled by *Obj[ ]*, represents the initial state of *Obj*; an intermediate node, labelled by *Obj[Ux, Uy]*, represents a state with operations *Ux* and *Uy* performed on the object *Obj*; a leaf node represents one version of *Obj*; and an arrow represents a state transfer triggered by the operation labelling the arrow. If a leaf node is drawn as a filled (dark) circle, it represents a displayed object version; if a leaf node is drawn as an empty circle, it represents an invisible version.

There are four *Updates*: *U0, U1, U2, U3*, performed on *Obj*, where *U0* is first performed, and then three concurrent and conflicting operations *U1, U2, U3* are generated and performed. After executing all operations under MVMD, MVSD, and SVSD, three different *Obj* state trees are shown in Fig. 3 (MVMD-a), (MVSD-a), and (SVSD-a), respectively.[8] The state trees for both MVMD and MVSD have three leaf nodes, representing three versions maintained by them; but the state tree for SVSD has only one leaf node, representing the single version maintained. On the other hand, the tree for MVMD has three filled circles representing three versions displayed at the same time; but the trees for both MVSD and SVSD have only one filled circle representing only one version displayed.

Given the fact that both MVSD and SVSD display only one version *Obj[U0, U1]* on the user interface, one may wonder what is the difference between them from the user's point of view? Their difference becomes clear when *U1* is undone. After undoing *U1* under MVMD, MVSD, and SVSD, three different *Obj* state trees are shown in Fig. 3 (MVMD-b), (MVSD-b), and (SVSD-b), respectively. In all three trees, the leaf node labelled by *Obj[U0,U1]* is changed from a filled circle to an empty circle, which means the corresponding version disappears from the user interface. However, in the MVMD tree, all other nodes remain unchanged; in the MVSD tree, an *alternative* leaf node labelled by *Obj[U0,U2]* is changed from an empty circle to a filled circle, which means the corresponding version is now displayed; and in the SVSD tree, an *intermediate* node labelled by *Obj[U0]* is changed from an empty circle to a filled circle, which means the previous state of the object is restored.

To summarize, an MVMD-based object state tree may have multiple leaf nodes which are all drawn as filled circles; a SVSD-based object state tree may have only one leaf node and it is drawn as a filled circle; and an MVSD object state tree may have multiple leaf nodes, but only one of them is drawn as a filled circle. In CoWord, the GOTO algorithm (for *group do*) is used to control the construction of MVSD-based object trees, and the AnyUndo algorithm (for *group undo*) is used to control the exploration of alternative branches (versions) of MVSD-based object trees by its fundamental *backtracking* capability.

6.2 OT extensions for update-like operations

There have been some prior work on extending OT to sup-

[8]Without losing generality, we assume the priority order of the three conflicting operations is: *U1* > *U2* > *U3*, and assume that SVSD selects *U1* for the final effect according to a priority scheme similar to what is proposed for MVSD in this paper.

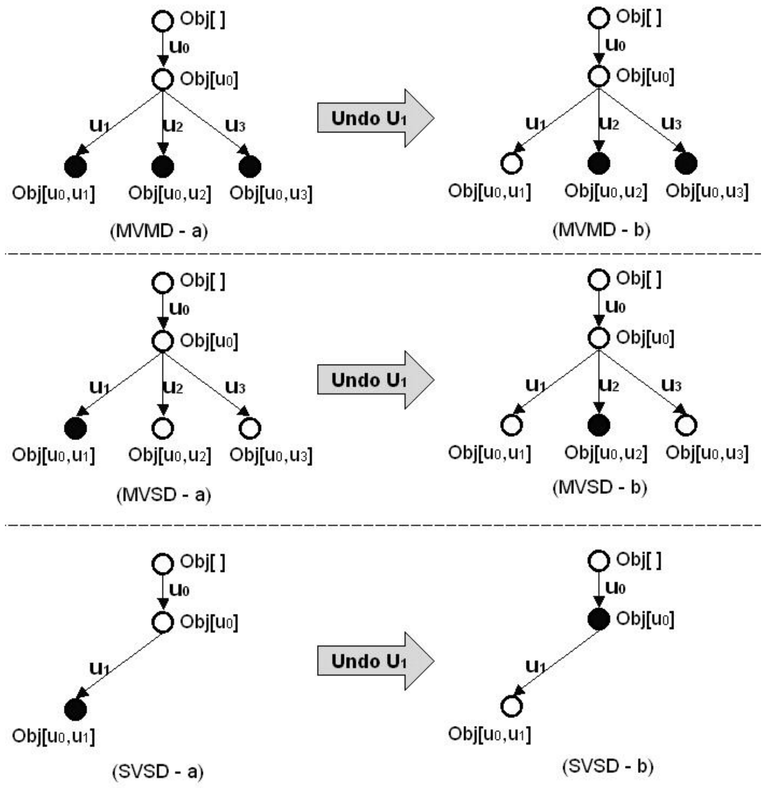

Figure 3: Comparison of different object state trees under MVMD, MVSD, and SVSD.

port operations that update values of data objects. Palmer et al [14] defined transformation functions for *set, format,* and *copy* operations that update the cell value or format in a distributed collaborative spreadsheet. In case of conflict (e.g. two concurrent operations setting the same cell to different values), one operation will "win" over the other to ensure convergence, and the "winner" is determined by a *total ordering* relation among operations.

Molli et al [11] defined a transformation function for an operation, named *ChangeAttribute*, which changes attribute values of XML documents. In case of conflict (e.g. two concurrent operations changing the same attribute of the same object to different values), the operation with the *maximum attribute value* will be the "winner" to ensure convergence.

Davis et al [2] defined transformation functions for a *change* operation that changes the content data or meta-data of a SGML/XML document. Conflict resolution was not addressed in [2].

In summary, prior OT extensions have addressed some issues in transforming *update*-like operations for consistency maintenance, but did not support group undo of *update*-like (or other) operations. In their approaches to conflict resolution [14, 11], there was no notion of MVSD and convergence was achieved by choosing one operation as the "winner", which is determined by a total ordering relation among conflicting operations.

In principle, any total ordering relation among conflicting operations can be used to determine a single "winner" to ensure convergence. However, if the chosen total ordering relation is inconsistent with the *causal ordering* relation [9, 5, 21] among these operations, a *causality-violation* effect may be produced. Taking the total ordering based on *different attribute values* (as in [11]) as the example, consider three *Update* operations: *U1 = U(p, Color, Red, Dark)*, *U2 = U(p, Color, Green, Red)*, and *U3 = U(p, Color, Blue, Dark)*, where *U2* was generated by a user who had seen the *Red* color set by *U1*, so *U2* is causally after *U1*, denoted as $U1 \rightarrow U2$; *U3* was generated by a user without any knowledge of the other two operations, so *U3* is concurrent with both *U1* and *U2*, denoted as $U3 \parallel U1$ and $U3 \parallel U2$. Since *U3* is changing the same object's attribute (color) to a different value (*Blue*), *U3* is conflicting with both *U1* and *U2*. Without losing generality, we assume the total ordering based on the color values is: $Blue < Green < Red$. If the "winner" is determined by this order (as in [11]), the final object's color shall be *Red* after executing these three operations. However, this *Red* result violates the effect determined by the causal relationship between *U1* and *U2*: the *Red* color by *U1* has been overwritten by a *causally-after* operation *U2* (with a *Green* color). We argue that the result for resolving conflicts among concurrent operations should preserve the effect determined by causal relationships among operations. For the above example, a final result of either *Blue* or *Green* respects causality, and one of them can be chosen to maintain both convergence and causality.

Finally, it is worth pointing out that the above results ensure only *syntactic* consistency, which is concerned with whether the same and causality-preserving view of the shared document is maintained at all sites, but not *semantic* consistency [4, 20], which is concerned with whether the common view makes sense or not in the application context. If, in the above example, the application context requires that the user who issued *U3* be given the highest priority in conflict resolution, then *Blue* is the only result that is able to achieve both syntactic and semantic consistency. In the MVSD scheme, all versions of conflicting operations are maintained and can be selectively displayed. On top of this syntactic consistency maintenance mechanism, consistency policy modules can be built to select semantically correct versions according to the application context requirement.

7. CONCLUSIONS

The work presented in this paper has contributed to the advancement and integration of the following collaborative editing techniques: Operational Transformation (OT), Multi-Versioning (MV), and Group Undo (GU).

The main contribution to OT is a novel extension for supporting a generic *Update* operation for collaborative word processing. An innovation in this work is the Multi-Version Single-Display (MVSD) conflict resolution technique as the transformation technique for *Updates*. The MVSD technique also represents an advancement of the MV technique because it simplifies the internal maintenance of multiple versions and lends itself to the integration in the OT framework. Moreover, MVSD is suitable for transparent integration with existing commercial word processing systems because its *Single-Display* strategy naturally matches with the existing single-user interface of word processors. The OT extension for undoing *Updates* leverages existing GU solutions (the AnyUndo algorithm) to support both graphics editing and text editing. Moreover, using selective undo for multi-version management (the AnyDisplay algorithm) is an innovative application of the GU technique.

OT, MV, and GU were invented from research on different types of collaborative editor. This work has shown they can be integrated in the same framework for collaborative word processing. In the past decades, collaborative text and graphics editors have been seen as as research vehicles in developing the current state-of-the-art collaborative editing techniques. We believe that collaborative word processors shall be seen as useful applications and new research vehicles for advancing collaborative editing techniques in the future.

ACKNOWLEDGEMENT

The authors are grateful to Bo Begole and anonymous referees for their valuable comments and suggestions which have helped clarify some technical issues and improve the presentation of the paper.

8. REFERENCES

[1] J. Begole, M. Rosson, and C. Shaffer. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM Trans. on Computer-Human Interaction*, 6(2):95–132, 1999.

[2] A. Davis, C. Sun, and J. Lu. Generalizing operational transformation to the standard general markup language. In *Proc. of the ACM Conf. on Computer-Supported Cooperative Work*, pages 58 – 67, Nov. 2002.

[3] P. Dewan, R. Choudhary, and H. Shen. An editing-based characterization of the design space of collaborative applications. *Journal of Organizational Computing*, 4(3):219–240, 1994.

[4] P. Dourish. Consistency guarantees: exploiting application semantics for consistency management in a collaboration toolkit. In *Proc. of the ACM Conf. on Computer-Supported Cooperative Work*, pages 268–277, November 1996.

[5] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proc. of the ACM Conf. on Management of Data*, pages 399–407, May 1989.

[6] C. Ignat and M.C. Norrie. Customizable collaborative editor relying on treeOPT algorithm. In *Proc. of the European Conf. of Computer-supported Cooperative Work*, pages 315–324, Sept. 2003.

[7] R. Kanawati. LICRA: a replicated-data management algorithm for distributed synchronous groupware application. *Parallel Computing*, 22:1733–1746, Feb. 1997.

[8] A. Karsenty, C. Tronche, and M. Beaudouin-Lafon. Groupdesign: shared editing in a heterogeneous environment. *Usenix Journal of Computing Systems*, 6(2):167–195, Spring 1993.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of ACM*, 21(7):558–565, 1978.

[10] D. Li and R. Li. Transparent sharing and interoperation of heterogeneous single-user applications. In *Proc. of the ACM Conf. on Computer-Supported Cooperative Work*, pages 246–255, Nov. 2002.

[11] P. Molli, H. Skaf-Molli, G. Oster, and S. Jourdain. SAMS: synchronous, asynchronous, multi-sychronous environments. In *Proc. of the 7th Int. Conf. on CSCW Design*, 2002.

[12] T. Moran, K. McCall, B. van Melle, E. Pedersen, and F. Halasz. Some design principles for sharing in tivoli, a whiteboard meeting-support tool. In Saul Greenberg, editor, *Groupware for Real-time Drawings: A Designer's Guide*, pages 24–36. McGraw-Hill International(UK), 1995.

[13] D. Nichols, M. Dixon P. Curtis, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proc. of ACM Symposium on User Interface Software and Technologies*, pages 111–120, Nov. 1995.

[14] C. Palmer and G. Cormak. Operation transforms for a distributed shared spreadsheet. In *Proc. of the ACM Conf. on Computer-Supported Cooperative Work*, pages 69–78, Nov. 1998.

[15] A. Prakash and M. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. on Computer-Human Interaction*, 4(1):295–330, Dec. 1994.

[16] M. Ressel and R. Gunzenhäuser. Reducing the problems of group undo. In *Proc. of the ACM Conf. on Supporting Group Work*, pages 131–139, Nov. 1999.

[17] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proc. of the ACM Conf. on Computer-Supported Cooperative Work*, pages 288–297, Nov. 1996.

[18] C. Sun. Undo as concurrent inverse in group editors. *ACM Trans. on Computer-Human Interaction*, 9(4):309–361, December 2002.

[19] C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Trans. on Computer-Human Interaction*, 9(1):1–41, March 2002.

[20] C. Sun and C. A. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of the ACM Conf. on Computer-Supported Cooperative Work*, pages 59–68, Nov. 1998.

[21] 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. on Computer-Human Interaction*, 5(1):63–108, March 1998.

[22] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proc. of the ACM Conf. on Computer-Supported Cooperative Work*, pages 171–180, Dec. 2000.

[23] S. Xia, D. Sun, C. Sun, D. Chen, and H.F. Shen. Leveraging single-user applications for multi-user collaboration: the CoWord approach. In *Proc. of the ACM Conf. on Computer-Supported Cooperative Work*, Nov. 2004.