Exhaustive Search and Resolution of Puzzles in OT Systems Supporting String-Wise Operations

Chengzheng Sun, Yi Xu, Agustina
School of Computer Science and Engineering, Nanyang Technological University, Singapore

ABSTRACT
OT (Operational Transformation) is a consistency maintenance technique for real-time collaborative editors and a continuous topic of research in CSCW for over two-decades. One main challenge in OT research is to detect and resolve puzzles ─ characteristic collaboration scenarios or transformation cases in which an OT system may fail to produce consistent results. A long-standing open issue is whether we can find and solve all possible puzzles in OT systems supporting string-wise operations under well-defined consistency requirements. In this paper, we present our research that solves this open problem and a set of concrete string-wise transformation functions capable of solving all known puzzles and preserving well-defined consistency requirements. Our results contribute to the advancement of state-of-the-art of OT fundamental knowledge and technological innovation.

Author Keywords
Operational Transformation; consistency maintenance, concurrency control; real-time collaborative editing.

ACM Classification Keywords
H.5.3 [**Information Systems**]: Group and Organization Interfaces – Synchronous Interaction, Theory and Model.

INTRODUCTION
Real-time collaborative editors allow multiple geographically dispersed people to edit shared documents and see each other's updates instantly [1,4,6,24,26]. One major challenge in building such systems is consistency maintenance of documents in the face of concurrent editing [6,24,33]. Operational Transformation (OT) was invented to address this challenge [6], and has been an active CSCW research subject for over two decades [1-8,11-34]. Due to its unique combination of non-blocking, fine-grained concurrency, and unconstrained interaction properties, OT is increasingly adopted in Internet/cloud-based industrial applications [9], e.g. Google Docs[1] and Codoxware[2].

[1] http://docs.google.com

[2] https://www.codoxware.com

CSCW '17, February 25-March 01, 2017, Portland, OR, USA
© 2017 ACM. ISBN 978-1-4503-4335-0/17/03…$15.00
DOI: http://dx.doi.org/10.1145/2998181.2998252

One major OT research challenge is to detect and resolve puzzles – characteristic editing scenarios or transformation cases in which an OT system may fail to produce consistent results [15,23,24,29]. Searching and resolving challenging puzzles has been driven by both academic curiosity and practical needs: as OT is increasingly applied to a wider range of real-world applications [1,26,33] and even mission-critical applications [18], the correctness of its core algorithms is becoming more and more important.

OT puzzles are notoriously difficult to detect since they are often hidden in complex scenarios and take varying forms under different concurrency situations [28,29]. In the history of OT puzzle search and research, an experimental approach has been commonly used, in which researchers devise intricate scenarios to explore puzzles and devise solutions for specific OT systems (concrete examples can be found in late sections). This approach has proven its effectiveness in detecting and solving well-known OT puzzles, including the *dOPT* puzzle in OT control algorithms [14,17,23,24], and the *False-Tie* (*FT*) puzzle in OT transformation functions [24]. However, the ad hoc nature of this approach makes it unsuitable for correctness verification: ad hoc search can detect the existence of puzzles, but cannot prove the absence of puzzles.

To prove the absence of puzzles, it is necessary to conduct *exhaustive* search of OT puzzles under well-defined conditions. The main challenge here is the *infinite* possibilities in a collaborative editing session [12]: there may exist an arbitrary number of operations with complex concurrency relations; operations may have arbitrary positional relations in a shared document consisting of arbitrary data objects. This *infinite-possibility* challenge can be and has been addressed by using a "*divide and conquer*" strategy [6,12, 23], which divides an OT system into two separate subsystems and applies different methods to verify them: one subsystem consists of *control algorithms* that are *generic* and determine which operations should be transformed with which others; the other contains *transformation functions* that are *application-specific* and perform actual transformations between two operations. In [12], *mathematic induction* is used to prove the correctness of control algorithms based on generic concurrency or context conditions [10,27]. The verification of control algorithms by mathematic induction effectively reduces the OT system verification problem, which involves an infinite number of operations with complex concurrency relations, to the problem of verifying transformation functions, which involves a limited and small number of operations with simple concurrency and context relations. Then, a different

method, called *exhaustive-case-checking*, can be applied to verify transformation functions.

Exhaustive-case-checking requires exhaustive case enumeration, which is non-trivial. One often-cited reason for using computer-aided software tools, such as *theorem-provers* [8] and *model checkers* [5], to assist exhaustive-case-checking is that transformation property verification is an exponential *explosion* problem that is too complex and "even impossible" for manual checking [5,8]. A state explosion phenomena was reported in [5], where hundreds of thousands states had to be checked to verify a scenario with a few operations (e.g. 331,776 states for merely 4 operations), which exceeded the capability of the model checker and failed to achieve exhaustive puzzle search.

Further research found the state explosion problem is not inherent to the transformation property verification itself, but depends on how collaborative editing sessions are modelled in verification. For example, the state explosion problem in [5] was mainly due to its specific modelling, in which concurrency relations (taken care of by control algorithms) and positional relations (responsible by transformation functions) among operations were mixed, rather than separated from each other, in the verification process ─ a common pitfall in the history of OT puzzle search [12,23,28]. In [29], a novel modeling and verification method were proposed that can cover all possible transformation scenarios under the same data and operations models as [5] with less than 100 verification cases, which can be efficiently checked even manually.

All prior OT verification works [5,8,29] have focused on a basic operation model with two *character-wise* operations: *insert* or *delete* of a *single* character. A more general operation model supports *string-wise* operations: *insert* or *delete* of a string of characters [26,33]. Past research has found designing and verifying transformation functions for string-wise operations is significantly more challenging than for character-wise functions [24,28]. There has been no prior published work, to our best knowledge, on exhaustive search of transformation puzzles under the string-wise operation model, let alone on resolving discovered puzzles, which is what this work focuses on.

The practical benefits of supporting string-wise operations in collaborative editing systems are multifold [15,17,20, 21,24,26,28,33]. First, the string-wise operation model is more powerful than the character-wise operation model in representing comprehensive user interactions and complex data structures in real-world applications, such as office productivity tools [26] and 2D and 3D computer-aided design systems [1, 30]. Second, OT systems supporting string-wise operations are generally more efficient than those that support only character-wise operations because the cost of invoking a transformation function for either string-wise or character-wise operations is basically the same, but a single string-wise transformation can represent a large number of character-wise transformations, and hence significantly reduce the total cost [20,28]. Third, supporting string-wise operations is a necessary condition for supporting *operation compression* [22] or *operation composition* [33] ─ an effective way of reducing the number of accumulated operations and associated transformations, which is not only useful in real-time co-editing, but also particularly useful in non-real-time or mobile collaborative editing systems [22]. Last but not least, it turned out (from this research) that the string-wise operation model is a necessary condition for resolving transformation puzzles at the transformation function level (see a later section on puzzle solutions).

In the rest of the paper, we first introduce basic OT knowledge related to this work. Then, we present formalization of string-wise data and operation models, combined-effects for concurrent operations and a general transformation matrix for verification. Next, we discuss our verification discoveries and puzzle solutions under the string-wise operation model, provide the design of a set of string-wise transformation functions capable of solving all transformation puzzles, and compare this work with prior work. Finally, we summarize major contributions in this work and suggest future directions.

OT BASICS

Concurrency Relations among Operations

In real-time collaborative editors, operations generated by multiple users may have causal or concurrent relations [6,23]. Informally, concurrent operations are *independent* in the sense that they have no impact on each other, whereas two operations are causally *dependent* when one operation may have impacts on the other, which could occur when the execution of one operation *happened-before* the generation of the other operation [10]. In OT-based collaborative editors, concurrent operations can be executed in different orders under the control of OT; but causally related operations must be executed in their causal orders, which can be enforced by using any suitable distributed computing techniques [6,24]. In this paper, we focus on verifying transformation functions for concurrent operations.

Basic Ideas for Consistency Maintenance

The basic idea for consistency maintenance in OT is to transform an operation according to the effect of other *concurrent* operations so that the transformed operation can achieve the correct effect and maintain document consistency [23]. This can be illustrated by using a text editing scenario shown in Figure 1.

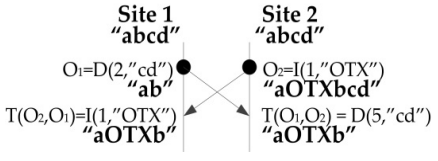

Figure 1. Basic idea of consistency maintenance in OT.

The initial document state *"abcd"* is replicated at two sites; two operations: $O_1 = D(2, \text{"cd"})$ (to delete *"cd"* starting at

position 2) and O_2 = $I(1,\text{“OTX”})$ (to insert *“OTX”* at position 1), are generated at Site 1 and Site 2, respectively. At Site 1, O_1 is first executed as-is to produce *“ab”*. Then, O_2 arrives and is transformed (by using function T) against O_1, i.e. $T(O_2,O_1) = I(1,\text{“OTX”})$, which makes no change to O_2 because the position of O_1 is at the right side of the position of O_2. Executing $I(1,\text{“OTX”})$ on *“ab”* produces a new state *“aOTXb”*. At Site 2, O_2 is first executed as-is to produce *“aOTXbcd”*. Then O_1 arrives and is transformed against O_2, i.e. $T(O_1,O_2) = D(5,\text{“cd”})$, where the position of O_1 is increased (to 5) to compensate the *right-shifting* effect caused by O_2. Executing $D(5,\text{“cd”})$ on *“aOTXbcd”* produces *“aOTXb”*, which is identical to the final state at Site 1 and also achieves the original effects of both O_1 and O_2. However, if the original $O_1 = D(2,\text{“cd”})$ were executed on *“aOTXbcd”*, the two characters starting at 2, i.e. *“TX”*, would have been deleted, resulting in a divergent state *“aObcd”*, which has been rescued by OT in this case.

The scenario in Figure 1 also illustrates one general transformation property [6,17,23], which is specified below.

Convergence Property 1 (CP1): *Given O_1 and O_2 defined on the document state S, let O_1^{O2} represent $T(O_1,O_2)$, and O_2^{O1} represent $T(O_2,O_1)$. An OT system must ensure:*

$$S \circ O_1 \circ O_2^{O1} = S \circ O_2 \circ O_1^{O2}$$

which means applying O_1 and O_2^{O1} in sequence on S has the same effect as applying O_2 and O_1^{O2} on S.

All transformation functions are required to preserve CP1, which ensures the same document is produced by executing two concurrent operations in different orders.

An Example OT Puzzle

Preserving CP1 turned out to be *insufficient* to ensure convergence in OT systems. Under some mysterious or puzzle scenarios, an OT system, capable of preserving CP1, may fail to produce correct transformation results, leading to inconsistent document states. In Figure 2, we illustrate a well-known puzzle, which involves three concurrent operations at three collaborating sites. This puzzle is named the *False-Tie (FT)* puzzle as it always involves an *insert-insert-tie* that is not originally generated by users but created by transformation [24,28].

To illustrate the inconsistency problem, we focus on Site 1 and 2, where O_3 is transformed against O_1 and O_2 in different orders. The initial document *“abc”* is replicated at all sites. At Site 1, $O_1 = I(2,\text{“Y”})$ is first executed as-is to produce *“abYc”*. Then, $O_2 = D(1,\text{“b”})$ arrives and is transformed against O_1 to get $O_2^{O1} = T(O_2,O_1) = D(1,\text{“b”})$ (no change); executing O_2^{O1} produces *“aYc”*. Finally, $O_3 = I(1,\text{“X”})$ arrives and is transformed against O_1 and O_2^{O1} in sequence, i.e. $O_3^{O1,O2} = T(I(1,\text{“X”}),D(1,\text{“b”})) = I(1,\text{“X”})$; executing $O_3^{O1,O2}$ will produce *“aXYc”*. At Site 2, O_2 is first executed as-is to produce *“ac”*. Then, O_1 arrives and transformed against O_2 to get $O_1^{O2} = T(O_1,O_2) = I(1,\text{“Y”})$; executing O_1^{O2} produces *“aYc”*. Finally, O_3 arrives and is transformed against O_2, and O_1^{O2} in sequence, $O_3^{O2,O1} = T(T(O_3,O_2),O_1^{O2}) = T(I(1,\text{“X”}),\ I(1,\text{“Y”}))$, which involves an *insert-tie* and this is an FT as $I(1,\text{“X”})$ was not original but created by transformation. Suppose the FT is resolved by a tie-breaking rule that gives O_1 higher priority than O_3, i.e. to shift O_3's position: $O_3^{O2,O1} = I(2,\text{“X”})$. Executing $O_3^{O2,O1}$ will produce *“aYXc”*, which is clearly inconsistent with *“aXYc”* at Site 1.

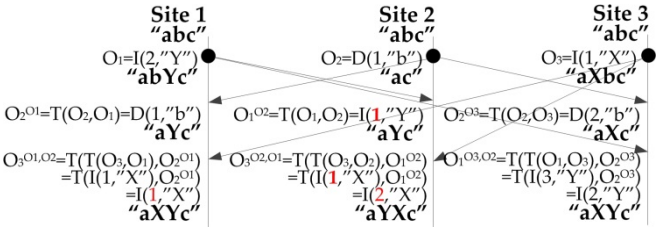

Figure 2. A False-Tie puzzle with three concurrent operations.

The FT puzzle may look trivial, but was difficult to detect and resolve. The FT puzzle in Figure 2 could have been masked if the underlying tie-breaking rule was reversed, i.e. to give O_3 higher priority than O_1. Under the reversed rule, however, the FT puzzle would manifest itself in a slightly different scenario by swapping the position parameters of O_1 and O_3, which means simply changing the tie-breaking rule cannot resolve the FT puzzle.

Follow-up research found that the FT puzzle was due to violation of another transformation property [15,17], named *Convergence Property 2* (*CP2*), as specified below.

Convergence Property 2 (CP2): *Given O_1, O_2 and O_3 defined on the same state, let O_2^{O1} represent $T(O_2,O_1)$ and O_1^{O2} represent $T(O_1,O_2)$. An OT system must ensure:*

$$T(T(O_3,O_1),O_2^{O1}) = T(T(O_3,O_2),O_1^{O2})$$

which means transforming O_3 against O_1 and O_2^{O1} produces the same result as transforming O_3 against O_2 and O_1^{O2}. Alternatively, CP2 can be expressed as:

$$T(O_3^{O1},O_2^{O1}) = T(O_3^{O2},O_1^{O2})$$

where $O_3^{O1} = T(O_3,O_1)$, and $O_3^{O2} = T(O_3,O_2)$.

Essentially, CP2 requires the same operation is produced in transforming one operation against two other concurrent operations in different orders. As shown in Site 1 and Site 2 in Figure 2, transforming O_3 against O_1 and O_2 in different orders produced two different operations: $O_3^{O1,O2} = I(1,\text{“X”})$ and $O_3^{O2,O1} = I(2,\text{“X”})$,which is a clear case of CP2-violation.

In [17,28], it has been established that CP1 and CP2 are two *necessary* and *sufficient* conditions to achieve convergence in OT systems that allow operations to be transformed in arbitrary orders. Prior work [29] has proven that FT is the *only* transformation property violation puzzle that exists in OT systems supporting character-wise operations.

Two questions arise: (1) Is there any other puzzle, in addition to FT, that violates CP1 or CP2 in OT systems supporting string-wise operations? (2) How to solve FT and other puzzles (if any) at the transformation function level? In the rest of this paper, we present details of our research which leads to our answers to these open questions.

DATA AND OPERATION MODEL FORMALIZATION

The data model used by string-wise operations is the same as that for character-wise operations [29], which is described below for self-containedness. The operation model for string-wise operations is more general than that for character-wise operations.

Data Model Specification

A document state S is expressed as a string of characters:

$$S = \text{“}c_0\, c_1\, c_2 \ldots c_{p-1}\, c_p\, c_{p+1} \ldots c_{n-1}\text{”}$$

where c_p, $0 \le p \le n-1$, represents a character. The following notations are used for convenience of expression:

1. $|S|$ is the length of the string S.
2. $S[p]=c_p$, $0 \le p \le |S|-1$, is the character at position p in S.
3. $S[i, j]=\text{“}c_i\, c_{i+1} \ldots c_{j-1}\, c_j\text{”}$, $0 \le i \le j \le |S|-1$, represents a substring of characters with positions from i to j, inclusively, in S. For notation convenience, $S[i, j]$ represents an empty string in case that $i > j$.
4. $S[i, j] + S[m, n] = \text{“}c_i\, c_{i+1} \ldots c_{j-1}\, c_j\, c_m\, c_{m+1} \ldots c_{n-1}\, c_n\text{”}$, represents the concatenation of two sub-strings.

Operation Model Specification

The two string-wise operations are specified below:

1. $I(p,s)$: to insert a string of characters s at position p;
2. $D(p,s)$: to delete a string of characters s at position p.

Definition 1. **Effect of an Insert Operation.** *Given an insert I(p, s) defined on a state S, where $0 \le p \le |S|$. The effect of I(p, s) on S is to convert S into a new state S′:*

$$S' = S \circ I(p, s) = S[0, p-1] + s + S[p, |S|-1],$$

with the following $S' \leftrightarrow S$ mapping relation:

1. $S'[0, p-1] = S[0, p-1]$, *when* $p \ne 0$;
2. $S'[p, p+|s|-1] = s$;
3. $S'[p+|s|, |S'|-1] = S[p, |S|-1]$, when $p \ne |S|$.

Informally, the effect of an insert $I(p, s)$ is to insert a sequence of characters s at position p in S, which will right-shift characters at and on the right of $S[p]$ by $|s|$ positions, but keep characters on the left of $S[p]$ unchanged.

Definition 2. **Effect of a Delete Operation.** *Given a delete D(p, s) defined on a state S, where $0 \le p \le |S|-1$. The effect of D(p, s) on S is to convert S into a new state S′:*

$$S' = S \circ D(p, s) = S[0, p-1] + S[p+|s|, |S|-1],$$

with the following $S' \leftrightarrow S$ mapping relation:

1. $S'[0, p-1] = S[0, p-1]$, *when* $p \ne 0$;
2. $S'[p, |S'|-1]=S[p+|s|, |S|-1]$, *when* $p \ne |S|-1$.

Informally, the effect of $D(p, s)$ is to delete a sequence of characters s in the range from p to $p+|s|-1$ in S, which will left-shift characters on the right of $S[p+|s|-1]$ by $|s|$ positions, but keep those on the left of $S[p]$ unchanged.

Exhaustive Coverage of Positions and States

Under string-wise data and operation models, a document state S may have an arbitrary length $|S|$; an operation O may have an arbitrary position p in the range $0 \le p \le |S|$, directly insert or delete an arbitrary number of characters expressed in the string parameter s, and indirectly affect the positions of an arbitrary number of characters in S, as defined in Definition 1 and 2. The combinations of these arbitrary parameters present major challenges to achieving exhaustive coverage of all possibilities in OT verification. The solution lies in the following key insights.

First, the effects of O on S can be partitioned into a *limited* number of *uniform* effect ranges (or positions). The effect of O in a range is uniform in the sense that all positions in the same range are affected by O in the same way; effects in different ranges may be different.

1. The insert operation has three uniform effect ranges :
   (1) $S[0, p-1]$: *non-effect range* from the beginning to $p-1$;
   (2) $S[p]$: *insert-effect position* at p; and
   (3) $S[p+1, |S|-1]$: *right-shift-effect range* from $p+|s|$ to the end of S.
2. The delete operation has three uniform effect ranges :
   (1) $S[0, p-1]$: *non-effect range* from the beginning to $p-1$;
   (2) $S[p, p+|s|-1]$: *delete-effect range* between p and $p+|s|-1$; and
   (3) $S[p+|s|, |S|-1]$: *left-shift-effect range* from $p+|s|$ to the end of S.

This *effect-uniformity* provides the foundation for reducing arbitrary possibilities to a fixed number of possibilities in operation effects and document states. The significance of this insight will become clear when we define combined-effects of concurrent operations and derive exhaustive verification cases later.

Second, $S' \leftrightarrow S$ mappings in Definitions 1 and 2 establish one-to-one positional correspondences between S' and S, and serves as a formalism of representing arbitrary lengths and characters in S' (the new state) in terms of S (the old state), and vice versa, without knowing specific lengths and value of S. This formalism provides the foundation for checking the equivalence of two states, without the knowledge of specific contents of these states, and has played a crucial role in verification, as detailed in [29].

COMBINED-EFFECTS

In collaborative editing, it is crucial to define combined-effects for concurrent operations, independent of execution orders. Like the effect of a single operation (Definitions 1 and 2), the combined-effects of concurrent operations are also specified in terms of document states. Exhaustive specification of combined-effects for string-wise operations is significantly more complex than character-wise operations. As this is a foundation of this verification work and "*the devil is in the details*", we provide detailed description of the key issues and our solutions below.

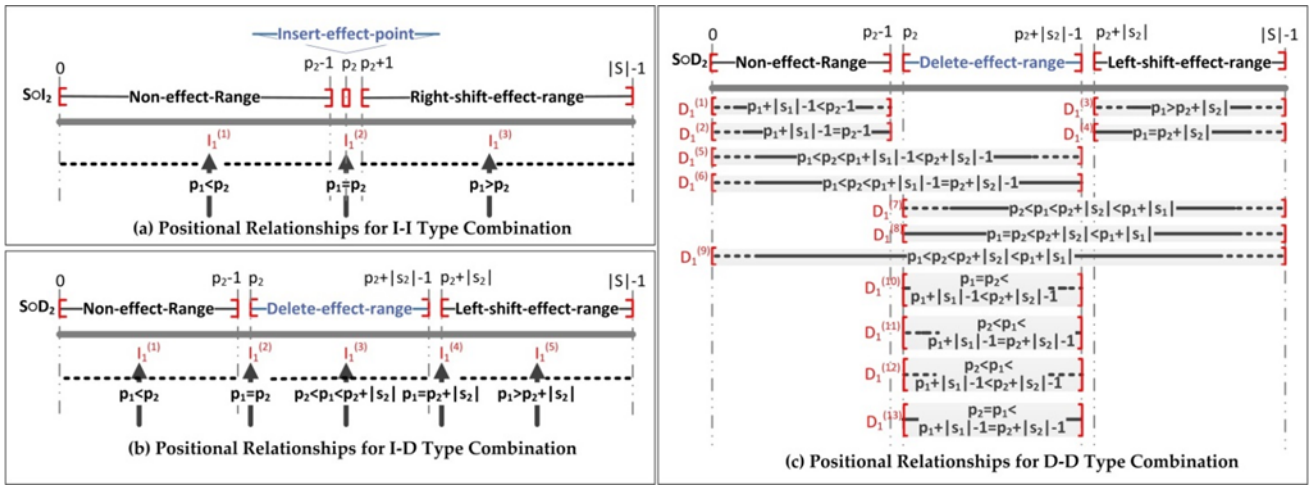

Figure 3. Positional relations for each type combination.

General Requirements and Case Analyses

To specify combined-effects under given data and operation models, we need to determine all possible cases in which two concurrent operations may have different combined-effects, and define specific effects for each case. To support verification, the combined-effect specification should meet two general requirements:

1. *Exhaustiveness*: combined-effect cases must exhaustively cover all possible operation type combinations and positional/object-reference relations between any two concurrent operations; and
2. *Uniformity:* the combined-effect of two concurrent operations under each case must be the same and hence can be expressed by the same effect expression.

Under the string-wise operation model, there are two operation types: *I* and *D,* which make four possible type *permutations*: *I-I, I-D, D-I* and *D-D*. For the purpose of combined-effect specification, we only need to consider three different type *combinations*[3] (as illustrated in Figure 3): *I-I*, *D-D*, and *I-D* as combined-effects of two concurrent operations are independent of their execution orders.

We specify the combined-effects of any two string-wise operations O_1 and O_2 under the column $CE1.0(O_1,O_2)$[4] in Table 1 (in next page). For convenience of comparison, we use the same table to specify the corresponding *Transformation Matrix* $TM1.0(O_1,O_2)$ (to be discussed later), which requires considering all four type permutations because transformation results depend on transformation orders. Consequently, those rows corresponding to *I-D* and *D-I* share the same $CE1.0(O_1,O_2)$ specifications.

We elaborate the exhaustive case derivation for all type combinations and definitions of combined-effects for all cases in following three sub-sections.

Combined-Effects of *Insert-Insert*

For the *I-I* type combination ($I(p_1,s_1)$ and $I(p_2,s_2)$), their positional relations can be exhaustively covered by three cases: $p_1 < p_2$, $p_1 = p_2$, $p_1 > p_2$, as illustrated in Figure 4 (a); and their combined-effect under each case can be uniformly specified by the combined-effect expression in Cases 1, 2 and 3, respectively, in Table 1.

In every case, the combined-effect retains the original effects of both operations, i.e. both s_1 and s_2 are inserted at their original positions. Case 2 (insert-insert-tie: $p_1 = p_2$) is special: two alternative combined-effects: "$s_1 + s_2$" and "$s_2 + s_1$" are specified as valid combined-effects. An OT system is free to support any but only one of them. Such a combined-effect is not the only possibility for the insert-insert-tie case. Other possibilities are: only one result is accepted as valid; none of them is accepted; or s_1 and s_2 are merged into one string if they are the same [6]. In prior work [24], the same combined-effect defined in $CE1.0(O_1, O_2)$ has been chosen due to its ability of preserving all operation effects and its intuitiveness to end-users.

All three positional relations and combined-effects in the string-wise operation model are similar to that in the character-wise model [29], except that the inserted object is not a single character but a string of characters.

Combined-Effects of *Insert-Delete*

For the *I-D* type combination ($I(p_1, s_1)$ and $D(p_2, s_2)$), their positional relations can be exhaustively covered by five cases and each case can be uniformly specified by the same combined-effect expression, as shown in Cases 4—8 (and 13—9), respectively, in Table 1.

As illustrated in Figure 3(b), the five positional relations are derived by: (1) dividing the linear data space into three

[3] Type permutations (orders matter) are different from type combinations (orders do not matter), e.g. *D-I* and *I-D* are different type permutations, but the same type combination.
[4] $CE1.0(O_1,O_2)$ is one instance of the general $CE(O_1,O_2)$.

O_1O_2	Positional Relation	CE1.0(O_1,O_2)	TM1.0(O_1,O_2)	#
I(p_1,s_1) I(p_2,s_2)	$p_1<p_2$	S[0,p_1-1]+s_1+S[p_1,p_2-1]+s_2+S[p_2,|S|−1]	I(p_1, s_1)	1
	$p_1=p_2$	S[0,p_1-1]+s_1+s_2+S[p_1,|S|−1]	I(p_1, s_1) priority(O_1)>priority(O_2)	2
		S[0,p_1-1]+s_2+s_1+S[p_1,|S|−1]	I(p_1+|s_2|, s_1) priority(O_1)<priority(O_2)	
	$p_1>p_2$	S[0,p_2-1]+s_2+S[p_2,p_1-1]+s_1+S[p_1,|S|−1]	I(p_1+|s_2|, s_1)	3
I(p_1, s_1) D(p_2,s_2)	$p_1<p_2$	S[0,p_1-1]+s_1+S[p_1,p_2-1]+S[p_2+|s_2|,|S|−1]	I(p_1, s_1)	4
	$p_1=p_2$	S[0,p_1-1]+s_1+S[p_2+|s_2|,|S|−1]		5
	$p_2<p_1<p_2+\|s_2\|$	S[0,p_2-1]+S[p_2+|s_2|,|S|−1]	*Null*	*6*
	$p_1=p_2+\|s_2\|$	S[0,p_2-1]+s_1+S[p_1,|S|−1]	I($p_1-\|s_2\|$, s_1)	7
	$p_1>p_2+\|s_2\|$	S[0,p_2-1]+S[p_2+|s_2|,p_1-1]+s_1+S[p_1,|S|−1]		8
D(p_1,s_1) I(p_2,s_2)	$p_1+\|s_1\|<p_2$	S[0,p_1-1]+S[p_1+|s_1|,p_2-1]+s_2+S[p_2,|S|−1]	D(p_1, s_1)	9
	$p_1+\|s_1\|=p_2$	S[0,p_1-1]+s_2+S[p_2,|S|−1]		10
	$p_1<p_2<p_1+\|s_1\|$	S[0,p_1-1]+S[p_1+|s_1|,|S|−1]	*D(p_1, S[p_1,p_2-1]+s_2+S[p_2, p_1+|s_1|−1])*	*11*
	$p_1=p_2$	S[0,p_2-1]+s_2+S[p_1+|s_1|,|S|−1]	D(p_1+|s_2|, s_1)	12
	$p_1>p_2$	S[0,p_2-1]+s_2+S[p_2,p_1-1]+S[p_1+|s_1|,|S|−1]		13
D(p_1,s_1) D(p_2,s_2)	$p_1+\|s_1\|<p_2$	S[0,p_1-1]+S[p_1+|s_1|,p_2-1] +S[p_2+|s_2|,|S|−1]	D(p_1, s_1)	14
	$p_1+\|s_1\|=p_2$	S[0, p_1-1]+S[p_2+|s_2|,|S|−1]		15
	$p_1>p_2+\|s_2\|$	S[0,p_2-1]+S[p_2+|s_2|,p_1-1]+S[p_1+|s_1|,|S|−1]	D($p_1-\|s_2\|$, s_1)	16
	$p_1=p_2+\|s_2\|$	S[0, p_2-1]+S[p_1+|s_1|,|S|−1]		17
	$p_1<p_2<p_1+\|s_1\|<p_2+\|s_2\|$	S[0, p_1-1]+S[p_2+|s_2|,|S|−1]	*D(p_1,S[p_1, p_2-1])*	*18*
	$p_1<p_2<p_2+\|s_2\|=p_1+\|s_1\|$	S[0, p_1-1]+S[p_1+|s_1|,|S|−1]	*D(p_1, S[p_1, p_2-1])*	*19*
	$p_2<p_1<p_2+\|s_2\|<p_1+\|s_1\|$	S[0, p_2-1]+S[p_1+|s_1|,|S|−1]	*D(p_2,S[p_2+|s_2|,p_1+|s_1|−1])*	*20*
	$p_1=p_2<p_2+\|s_2\|<p_1+\|s_1\|$	S[0, p_1-1]+S[p_1+|s_1|,|S|−1]	*D(p_1,S[p_2+|s_2|,p_1+|s_1|−1])*	*21*
	$p_1<p_2<p_2+\|s_2\|<p_1+\|s_1\|$	S[0, p_1-1]+S[p_1+|s_1|,|S|−1]	*D(p_1, S[p_1,p_2-1]+S[p_2+|s_2|,p_1+|s_1|−1])*	*22*
	$p_2=p_1<p_1+\|s_1\|<p_2+\|s_2\|$	S[0, p_2-1]+S[p_2+|s_2|,|S|−1]	*Null*	*23*
	$p_2<p_1<p_1+\|s_1\|=p_2+\|s_2\|$	S[0, p_2-1]+S[p_2+|s_2|,|S|−1]		*24*
	$p_2<p_1<p_1+\|s_1\|<p_2+\|s_2\|$	S[0, p_2-1]+S[p_2+|s_2|,|S|−1]		*25*
	$p_2=p_1<p_1+\|s_1\|=p_2+\|s_2\|$	S[0, p_2-1]+S[p_2+|s_2|,|S|−1]		26

Table 1. Combined-Effects *CE1.0*(O_1,O_2) and Transformation Matrix *TM1.0*(O_1,O_2).
Positional relationship expressions in Figure 3 and this table are equivalent but some may take different forms, e.g."$p_1+|s_1|-1 = p_2-1$" is reduced to "$p_1+|s_1| = p_2$". Special cases that exist in the string-wise model but not in the character-wise model are highlighted in *italic* font (with *red* color).

uniform effect ranges of $D(p_2, s_2)$, plus two *border* positions of $D(p_2, s_2)$: p_2 — the first position of the *delete-effect* range of D; and $p_2+|s_2|$ — the first position of the *left-shift-effect* range of D; then (2) mapping the position of $I(p_1, s_1)$ into one of these five positions/ranges. The combined-effects under these cases in $CE1.0(O_1,O_2)$ are elaborated below.

Combined-Effects of *Insert-Delete*

For the *I-D* type combination ($I(p_1, s_1)$ and $D(p_2, s_2)$), their positional relations can be exhaustively covered by five cases and each case can be uniformly specified by the same combined-effect expression, as shown in Cases 4—8 (and 13—9), respectively, in Table 1.

As illustrated in Figure 3(b), the five positional relations are derived by: (1) dividing the linear data space into three uniform effect ranges of $D(p_2, s_2)$, plus two *border* positions of $D(p_2, s_2)$: p_2 — the first position of the *delete-effect* range of D; and $p_2+|s_2|$ — the first position of the *left-shift-effect* range of D; then (2) mapping the position of $I(p_1, s_1)$ into one of these five positions/ranges. The combined-effects under these cases in $CE1.0(O_1,O_2)$ are elaborated below:

1. When $I(p_1, s_1)$ falls in the *non-effect* range of $D(p_2, s_2)$, i.e. $p_1 < p_2$, the combined-effect is to insert s_1 at p_1 and delete s_2 from the original document state.
2. When $I(p_1, s_1)$ targets at the first position of the *delete-effect* range of $D(p_2, s_2)$, i.e. $p_1 = p_2$, the combined-effect is the same as the prior case, i.e. to insert s_1 at p_1 and delete s_2 from the original state, though their expressions look different.
3. When $I(p_1, s_1)$ falls in the rest of the *delete-effect* range of $D(p_2, s_2)$, i.e. $p_1 > p_2$ and $p_1 < p_2+|s_2|$, the combined-effect is to delete s_2 from the original state, together with s_1 inserted by $I(p_1, s_1)$.
4. When $I(p_1, s_1)$ targets at the first position of the *left-shift-effect* range of $D(p_2, s_2)$, i.e. $p_1 = p_2+|s_2|$, the combined-effect is defined to the same as the next case, though expressions in two cases look different.
5. When $I(p_1, s_1)$ falls in the rest of the *left-shift-effect* range of $D(p_2, s_2)$, i.e. $p_1 > p_2+|s_2|$, the combined-effect is to delete s_2 from the original state and to insert s_1 at a new position which is left-shifted from p_1 by $|s_2|$.

Among the five cases, the third case, where the insert position falls in the middle of the *delete-effect* range, occurs only between string-wise operations, but not character-wise operations [28]. There are two alternatives for defining the combined-effect under this case. One is to preserve the original effects of both *D* and *I*, i.e. to delete all characters in the *delete-effect* range of *D*, and to preserve the string *s* inserted by *I*, which was chosen in [24].

The other alternative, chosen in this work and [26], is to extend the *delete-effect* range of *D* to include the string *s* inserted by *I*; so after having executed both operations, the string *s* inserted by *I* is absorbed by *D* (becomes invisible) but can be restored (becomes visible) if *D* is undone [15, 25]. This combined-effect should be differentiated from the one that completely ignores the string *s* inserted by *I* even if *D* is undone. This alternative is chosen because it avoids splitting one *D* operation into two *sub-D* operations, which is one undesirable consequence of the first alternative [24], thus simplifying the design and implementation of string-wise transformation functions while preserving reasonable semantics for combined-effects under such circumstances.

Apart from this *I-D* overlapping (the 2nd) case, there are two other special combined-effects for string-wise *I-D* operations: when the insert is at the two borders of the delete (the 3rd and 4th cases). The reason for distinguishing the two border positions of the delete is that they are *critical* for defining different combined-effects, which may cause or avoid puzzles, as will become clear late when we discuss verification discoveries and puzzle resolution.

Combined-Effects of *Delete-Delete*

For the *D-D* type combination ($D(p_1,s_1)$ and $D(p_2,s_2)$), exhaustive and uniformed coverage of their positional relations requires 13 cases for string-wise operations, whereas only 3 cases are needed for character-wise operations [29]. As illustrated in Figure 4 (c), the 13 positional relations are derived as follows. First, we divide the linear data space into three uniform effect ranges according to $D(p_2, s_2)$, i.e. $[0, p_2-1]$, $[p_2, p_2+|s_2|-1]$, and $[p_2+|s_2|, |S|-1]$, respectively. Second, we systematically map the *delete-effect range* of $D(p_1, s_1)$, i.e. $[p_1, p_1+|s_1|-1]$, into the three ranges of $D(p_2, s_2)$.

The 13 cases in Figure 3 (c) (corresponding to Combined-effect Cases 14—26 in Table 1) are elaborated as follows:

1. There are four cases, in which the *delete-effect* ranges of $D(p_1, s_1)$ and $D(p_2, s_2)$ are *disjoint*:
   (1) in two cases (denoted as $D_1^{(1)}$ and $D_1^{(2)}$ in Figure 3 (c)), the *delete-effect range* of $D(p_1, s_1)$ falls in the *non-effect* range of $D(p_2, s_2)$; in Case $D_1^{(2)}$, the two *delete-effect* ranges are *adjacent*, i.e. the last position in the *delete-effect* range of $D(p_1, s_1)$ is at the last position of the *non-effect* range of $D(p_2, s_2)$, i.e. $p_1+|s_1|-1 = p_2-1$;
   (2) in two cases ($D_1^{(3)}$ and $D_1^{(4)}$), the *delete-effect range* of $D(p_1, s_1)$ falls in the *left-shift-effect* range of $D(p_2, s_2)$; and in Case $D_1^{(4)}$, the two *delete-effect* ranges are *adjacent*, i.e. the first position of the *delete-effect* range of $D(p_1, s_1)$ is equal to the first position of the *left-shift-effect* range of $D(p_2, s_2)$, i.e. $p_1 = p_2+|s_2|$.
2. There are eight cases, in which the *delete-effect* ranges of $D(p_1, s_1)$ and $D(p_2, s_2)$ are *partially overlapping*:
   (1) in two cases ($D_1^{(5)}$ and $D_1^{(6)}$), the *delete-effect range* of $D(p_1, s_1)$ spans across the *non-effect* and the *delete-effect* ranges of $D(p_2, s_2)$; $D_1^{(6)}$ has the *last-delete* of $D(p_1, s_1)$ equal to the *last-delete* of $D(p_2, s_2)$, i.e. $p_1+|s_1|-1 = p_2+|s_2|-1$;
   (2) in two cases ($D_1^{(7)}$ and $D_1^{(8)}$), the *delete-effect range* of $D(p_1, s_1)$ spans across the *delete-effect* and the *left-shift-effect* ranges of $D(p_2, s_2)$; $D_1^{(8)}$ has the *first-deletes* of both $D(p_1,s_1)$ and $D(p_2,s_2)$ equal, i.e. $p_1 = p_2$;
   (3) in one case ($D_1^{(9)}$), the *delete-effect* range of $D(p_1, s_1)$ spans across all three effect ranges of $D(p_2, s_2)$, i.e. the *first* and *last* delete positions of $D(p_1, s_1)$ fall in the *non-effect* and the *left-shift-effect* ranges of $D(p_2, s_2)$, respectively;
   (4) in three cases ($D_1^{(10)}$, $D_1^{(11)}$ and $D_1^{(12)}$), the *delete-effect* range of $D(p_1, s_1)$ fall within the *delete-effect* range of $D(p_2, s_2)$; $D_1^{(10)}$ has the same *first* delete positions for both operations, i.e. $p_1 = p_2$; and $D_1^{(11)}$ has the same *last* delete positions for both, i.e. $p_1+|s_1|-1 = p_2+|s_2|-1$.
3. There is one case ($D_1^{(13)}$), in which the *delete-effect* ranges of $D(p_1, s_1)$ and $D(p_2, s_2)$ are *exactly overlapping*, i.e. $p_1 = p_2$ and $p_1+|s_1|-1 = p_2+|s_2|-1$.

Combined-effect expressions under all 13 cases in $CE1.0(O_1,O_2)$ share one common *union* effect property, i.e. *deleting the ranges covered by either operation and leaving other ranges untouched.* When two delete operations overlap in *delete-effect* ranges (9 cases in total), called a *delete-delete-tie*, the characters covered by both operations will be deleted only once. Such combined-effects cannot be achieved by any serialization scheme. As will become clear late, these special positional relationships and combine-effects in the string-wise model may become the cause of puzzles that do not exist in the character-wise model.

Integrated CE-CP1 Property

In [23,26], the notion of *operation intention* was introduced and informally described as "*the effect achieved by applying an operation on the document state from which the operation was generated*" [26]. The single operation effects in Definitions 1 and 2 provide *formal* specifications of *individual* operation intentions, whereas $CE1.0(O_1,O_2)$ provides a *formal* specification for operation intentions between two *concurrent* operations in the string-wise operation model.

Given two concurrent operations O_1 and O_2 defined on the same document state, an OT system must ensure that the states achieved by executing O_1 and O_2 in different orders will not only be equal to each other (as required by CP1), but will also be equal to the state defined by $CE(O_1,O_2)$, as

expressed by the following CE-CP1 property, which was introduced in prior work [2,29,30]:

$$CE(O_1,O_2) = S \circ O_1 \circ O_2^{O1} = S \circ O_2 \circ O_1^{O2}.$$

It is worth highlighting that the three expressions in CE-CP1 are all document state representations; the mutual equivalence of them is a stronger requirement than CP1 alone as CE-CP1 captures both convergence and intention preservation requirements [23,26], whereas CP1 captures the convergence requirement only.

TRANSFORMATION MATRIX

Given a specific $CE(O_1,O_2)$ and the general requirement CP1, we can specify a *Transformation Matrix* $TM(O_1,O_2)$ which represents the CE-CP1-preserving results of transforming O_1 against O_2 under all possible operation type permutations and positional relationships. From $TM(O_1,O_2)$, it is straightforward to design concrete transformation functions. Conversely, a $TM(O_1,O_2)$ can also be derived from existing transformation functions. In this work, $TM(O_1,O_2)$, rather than specific transformation functions, is used for verification, which allows us to verify both existing and new transformation functions whose transformation results are captured in $TM(O_1,O_2)$. Based on $CE1.0(O_1,O_2)$ and CP1, we give a specific transformation matrix $TM1.0(O_1,O_2)$[5] in Table 1. One set of transformation functions derived from $TM1.0(O_1,O_2)$ is given in APPENDIX-I-1.

In most cases, CP1 is guaranteed if the combined-effect in $CE1.0(O_1,O_2)$ is achieved. In two cases, however, special measures have been taken to meet the integrated CE-CP1 requirement. In the *insert-insert-tie* case, a priority-based tie-breaking rule is used. In *delete-delete-tie* cases, the delete-effect range of the transformed delete is its original delete-effect range less the overlapping delete-effect range; when two delete-effect ranges are exactly overlapping, the final result is the special operation *Null* [6].

VERIFICATION DISCOVERIES

We have developed a software tool, named *OTX* (*OT eXplorer*), to automate the verification process. OTX is efficient in checking a large number of verification cases under the string-wise operation model, and especially useful in supporting design and verification of solutions to the discovered puzzles. For the working principles, schemes and verification processes underlying OTX, the reader is referred to APPENDIX-II and III. For more detailed description of OTX structures and functionalities, the reader is referred to [29]. In this section, we report our discoveries and solutions obtained with the help of OTX. It is worth highlighting that these results are independent of the used tool, e.g. manual checking can obtain the same results but just much slower.

CE-CP1 Verification Discoveries

CE-CP1 involves two independent operations O_1 and O_2, which have four operation type permutations: *I-I*, *I-D*, *D-I* and *D-D*. Based on the positional relationships for every operation type combination in Figure 3, the four type permutations have a total of 26 possible positional relationships (see Table 1). Each of the 26 relationships can make a verification case for CE-CP1. However, careful examination reveals that nearly half of those 26 cases are redundant. After eliminating redundant cases, there are a total of **14** distinctive and still exhaustive CE-CP1 verification cases to be checked. Full details of exhaustive case derivation are provided in APPENDIX-II .

With the support of OTX, we checked all 14 transformation cases and found *CE-CP1 is preserved under every case.* This result is the first to establish CE-CP1 correctness for transformation functions defined on the string-wise operation model. Full details of the CE-CP1 verification results are provided in APPENDIX-III-1.2.

CP2 Verification Discoveries

CP2 involves three independent operations O_1, O_2 and O_3, which have $2^3 = 8$ different type permutations. The exhaustive analysis of possible positional relationships among three independent string-wise operations is based on the same ideas of that for two independent operations. In total, we have derived **818** different combinations of operation type permutations and position relationships. Each of them could make a CP2 verification case. After eliminating redundant cases (see details in APPENDIX-II), there are a total of **422** distinctive CP2 verification cases to be checked. In contrast, there are only 58 CP2 verification cases under the character-wise operation model [29].

With the help of OTX, we checked all 422 cases, and discovered that *TM1.0* can preserve CP2 under **413** cases but violates CP2 in **9** cases. Full details of CP2 verification results are provided in APPENDIX-III-1.4. In the following, we discuss the discovered CP2-violation cases in detail.

The False-Tie Puzzle

The first CP2-violation case occurs when transforming O_1 against O_2 and O_3 in different orders under the following conditions: $O_1 = I(p_1, s_1)$, $O_2 = D(p_2, s_2)$, $O_3 = I(p_3, s_3)$, and $(p_1 - |s_2|) = p_2 = p_3$. This case is a general description of the well-known FT puzzle, which may occur under both char-wise and string-wise operation models [24,29]. As a matter of fact, the FT puzzle illustrated in Figure 2 is a special instantiation of the general conditions with specific operations: $O_1 = I(2, \text{“}Y\text{”})$, $O_2 = D(1, \text{“}b\text{”})$, and $O_3 = I(1, \text{“}X\text{”})$, where $((p_1=2) - (|s_2|=1)) = (p_2=1) = (p_3=1)$, generated from the same state “*abc*”.

The False-Border Puzzle

The other 8 CP2-violation cases form one new category of puzzles, which occur only under the string-wise operation model. Every case in this new category of puzzles involves one insert operation and two delete operations, i.e. $O_1 = I(p_1, s_1)$, $O_2 = D(p_2, s_2)$, and $O_3 = D(p_3, s_3)$. Originally, the

[5] $TM1.0(O_1,O_2)$ is one instance of the general $TM(O_1,O_2)$.

[6] *Null* operation has the following properties: $S \circ Null = S$; $T(Null, O) = Null$; $T(O, Null) = O$.

position of O_1 is in the middle of the *delete-effect* range of O_2 (or O_3). After transforming O_2 with O_3, the position of O_1 is no longer in the middle, but at a *border* position of the transformed operation O_2^{O3} (or O_3^{O2}). Under such circumstances, the string inserted by O_1 may be absorbed or preserved, depending on the execution and transformation orders among the three operations, which leads to CP2 violation and divergent results. We name this new category of puzzles as *False-Border* (*FB*) puzzles as the trouble is always related to inserting at a *border* position, which is not original but caused by transformation. The three operations in each instance of FB puzzles may take one of the 4 general positional patterns, as illustrated in Figure 4.

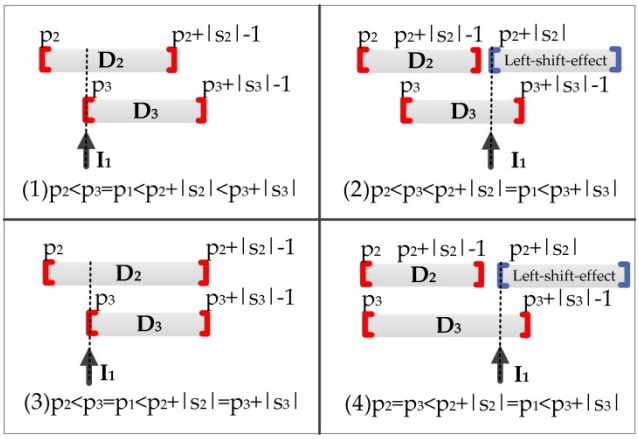

Figure 4. Positional patterns for False-Border puzzles.

The 8 CP2-violation cases correspond to the combinations of the 4 general positional patterns with 2 distinctive transformation patterns of these three operations: (1) the insert operation is transformed against the two delete operations in different orders; and (2) one delete operation (with the insert operation falling in the middle of its delete-effect range) is transformed against the insert operation and another delete operation (with the insert operation falling at its critical border position) in different orders. Other transformation patterns do not generate distinctive cases for CP2 verification. Detailed descriptions of these 8 CP2-violation cases can be found in APPENDIX-III-1.5.

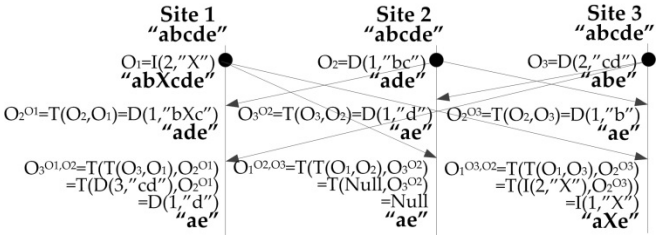

Figure 5. A False-Border puzzle.

A concrete FB puzzle can be created by instantiating the first positional pattern (in Figure 4) with: $O_1 = I(2,\text{“}X\text{”})$, $O_2 = D(1,\text{“}bc\text{”})$, and $O_3 = D(2,\text{“}cd\text{”})$, where $(p_2{=}1) < (p_3{=}2) = (p_1{=}2) < (p_2{+}|s_2|{=}3) < (p_3{+}|s_3|{=}4)$, generated from the same initial state *“abcde”*, as shown in Figure 5.

To explain, we focus on Sites 2 and 3, in which O_1 is transformed against O_2 and O_3 in different orders, resulting in divergent states:

1. At Site 2, O_2 is executed first, followed by O_3 and O_1:
   (1) O_2 is executed without transformation;
   (2) O_3 is transformed against O_2 to produce $O_3^{O2} = T(O_3,O_2) = D(1,\text{“}d\text{”})$, by *Case 20* of *TM1.0*(O_1, O_2);
   (3) O_1 is first transformed against O_2 to produce $O_1^{O2} = T(O_1,O_2) = Null$, by *Case 6* of *TM1.0*(O_1, O_2); and then against O_3^{O2} to produce $O_1^{O2,O3} = T(O_1^{O2}, O_3^{O2}) = Null$, according to *Null* operation properties.

   In above transformations, the effect of O_1 is absorbed since when O_1 is transformed against O_2, the position of O_1 falls into the middle of the delete-effect range of O_2.
2. At Site 3, O_3 is executed first, followed by O_2 and O_1:
   (1) O_3 is executed without transformation;
   (2) O_2 is transformed against O_3 to produce $O_2^{O3} = T(O_2,O_3) = D(1,\text{“}b\text{”})$, according to *Case 18* of *TM1.0*(O_1, O_2);
   (3) O_1 is first transformed against O_3 to produce $O_1^{O3} = T(O_1,O_3) = I(2, \text{“}X\text{”})$, according to *Case 5* of *TM1.0*(O_1, O_2); then against O_2^{O3} to produce $O_1^{O3,O2} = T(O_1^{O3}, O_2^{O3}) = I(1, \text{“}X\text{”})$, according to *Case 7*.

   In this sequence of transformations, the effect of O_1 is preserved because when O_1^{O3} is transformed against O_2^{O3}, the insert position of O_1^{O3} is at a border – the first position of the delete-effect range of O_2^{O3}.

In this scenario, CP2 is violated as $T(O_1^{O2},O_3^{O2}) \neq T(O_1^{O3}, O_2^{O3})$, or $Null \neq I(1, \text{“}X\text{”})$, which results in divergent final states: *“ae” ≠ “aXe”*.

RESOLVING PUZZLES

Based on the exhaustiveness of CP2 verification cases, we assert that *FT and FB puzzles are the only two possible categories of CP2-violation cases under TM1.0 in the string-wise operation model*. Therefore, we can achieve CP2-preservation by solving *FT* and *FB* puzzles at transformation function/matrix level, provided no new puzzle is induced by the devised solutions.

Design Objectives to Puzzle Solutions

From prior research and CP2 verification for *TM1.0*, we learned that *FT* and *FB* puzzles occur only under circumstances involving special operation types and positional relations, which are independent of most components of an OT system. Therefore, we set the following design objectives to puzzle solutions: (1) *isolation:* the solution should be isolated within the relevant components of an OT system, without disturbing the rest of the system; (2) *efficiency:* the solution should not cause additional overhead to the OT system; and (3) *verifiability*: the correctness of the solutions, i.e. CE-CP1 and CP2 preservation, can be formally verified.

Analyses of FT and FB Puzzles

Each instance of the FT involves three concurrent operations: two inserts and one delete; the two insert operations must target at the two critical borders of the

delete operation, respectively. The root of the problem is that the two insert operations are originally not tied, but become (falsely) tied when they are transformed with each other after the delete operation has removed the string between them. An FT solution cannot impose restrictions on the user in generating original operations, but can prevent transformed operations from having FT relations.

Similarly, each instance of the FB involves three concurrent operations: two deletes and one insert. For the FB to occur, the position of the insert must be at a critical border of one delete and in the middle of the delete-effect range of another delete. The root of the problem is twofold: (1) the insert was originally in the middle of a delete, but at a border position of the same delete after transformation; (2) the combined-effects in *CE1.0* are different when the insert is in the middle or at a critical border of the delete:

1. When an insert falls into the middle of the delete-effect range of a delete (Case 6 and Case 11 in Table 1), the inserted string is absorbed by the delete.
2. When an insert is at one of the two critical borders of a delete (Cases 5, 7, 10, 12 in Table 1), the inserted string is kept. Such combined-effects are the same as that under the char-wise operation model [29].

An FB solution cannot prohibit the user from generating original operations with potential FB relations, but can define the combined-effects of transformed operations to avoid FB relations, e.g. by eliminating the difference between the combined-effects of inserting at a border and in the middle of the delete-effect range of a delete operation.

One Uniformed Solution to Two Puzzles

Despite the different symptoms of FT and FB puzzles, they can be resolved by a single uniformed solution. The basic idea is to absorb an insert by transformation if it is at a critical border of a concurrent delete[7]. This basic idea works well regardless of whether the absorption is applied to one or both critical border positions. For simplicity, we choose to absorb the insertion at the first position of the delete-effect range of a delete operation, which effectively unifies the combined-effects between an insert and a delete when their positions are overlapping in the whole delete-effect range of the delete operation.

More precisely, we re-define the combined-effects of Case $I_1^{(2)}$ in Figure 3(b) as follows: When $I(p_1, s_1)$ targets at the first position of the *delete-effect* range of $D(p_2, s_2)$, i.e. $p_1 = p_2$, the combined-effect is to delete s_2 from the original state, together with s_1 inserted by $I(p_1, s_1)$.

To achieve the above new combined-effect, two transformation cases in Table 1 are re-defined accordingly:

1. In case that $O_1 = I(p_1, s_1)$, $O_2 = D(p_2, s_2)$, and $p_1 = p_2$, transforming O_1 against O_2 produces *Null,* i.e. the insert is effectively being absorbed.
2. In case that $O_1 = D(p_1, s_1)$, $O_2 = I(p_2, s_2)$, and $p_1 = p_2$, transforming O_1 against O_2 produces $D(p_1, s_2+s_1)$, i.e. to extend the delete-effect range of the delete to absorb the inserted string.

Based on above new definitions of the combined-effect and transformation cases, we have designed a new *Combined-Effects CE1.1*, and a corresponding *Transformation Matrix TM1.1*. Among all 26 possible transformation cases in Table 1, only two cases are different, as shown in Table 2. Full details of *TM1.1* are provided in APPENDIX-I-2,

O_1O_2	Positional Relation	*CE1.1*(O_1,O_2)	*TM1.1*(O_1,O_2)	#
$I(p_1, s_1)$ $D(p_2, s_2)$	$p_1=p_2$	$S[0,p_2-1]+S[p_2+\|s_2\|,\|S\|-1]$	Null	5
$D(p_1, s_1)$ $I(p_2, s_2)$	$p_1=p_2$	$S[0,p_1-1]+S[p_1+\|s_1\|,\|S\|-1]$	$D(p_1, s_2+s_1)$	12

Table 2. *CE 1.1* and *TM 1.1*. Only two cases are listed; other 24 cases are exactly the same as *CE1.0* and *TM1.0* in Table 1.

It is worth pointing out that the proposed FT and FB solution requires the OT system to have the capability of supporting string-wise operations. This is because the basic strategy of absorbing an insert (which may insert only one char) by a delete (which may originally delete only one char) would always result in a transformed delete with a delete-effect range covering more than one character. In other words, the string-wise operation model is the foundation of resolving FT and FB puzzles.

Verification of TM1.1

In previous sections, we have elaborated the exhaustive derivation of verification cases for CE-CP1 and CP2. We have used exactly the same set of cases to verify both *TM1.0* and *TM1.1*, except those cases affected by the two combined-effect changes listed in Table 2. Our *TM1.1* verification found no single violation case for either CE-CP1 or CP2 under the exhaustive set of verification cases, which hence established that *TM1.1* can preserve CE-CP1 and CP2 under all cases, which is the main outcome of this verification. Detailed verification results under all cases are provided in APPENDIX-III-2.

Furthermore, we have previously established that *FT* and *FB* puzzles are the only two categories of CP2-violation cases under the string-wise operation model. In the following, we provide theorems that establish the general correctness of *TM1.1*-based *FT* and *FB* solutions, together with illustration examples. These theorem proofs and illustrations provide additional insight into the inner working of *TM1.1*-based puzzle solutions and their capabilities of preserving CE-CP1 and CP2 in general.

Verification of the FT Solution

The following theorem establishes that the FT puzzle is resolved under *TM1.1*.

[7]The reader is referred to the section on "Combined-Effects of Insert-Delete" for discussion on alternative CE and justification of the absorbing effect for concurrent insert and delete.

Theorem 1. *Under TM1.1, the FT puzzle is resolved.*

Proof. According to the CP2 verification results for *TM*1.0, an FT puzzle may occur only when transforming one insert O_1 against a delete O_2 and another insert O_3 under the following conditions: $O_1 = I(p_1, s_1)$, $O_2 = D(p_2, s_2)$, $O_3 = I(p_3, s_3)$, and $(p_1 - |s_2|) = p_2 = p_3$. First, consider the path of transforming O_1 against O_2 and O_3 in sequence. Both O_1 and O_3 must first be transformed against O_2 before they can be transformed with each other, according to the context-equivalence requirement, which is enforced by OT control algorithms [12]. So, we have: $O_1^{O2} = TM1.1(O_1, O_2) = I(p_1 - |s_2|, s_1)$, which means O_1 is left-shifted by O_2; $O_3^{O2} = TM1.1(O_3, O_2) = Null$ (*Case 5* in Table 2), because O_3 is targeting at the first position of the delete-effect range of O_2 and hence absorbed by O_2. This effectively avoids the possibility of producing two transformed insert operations with a false-tie (one of the inserts has been absorbed during transformation). Finally, $O_1^{O2,O3} = TM1.1(O_1^{O2}, O_3^{O2}) = O_1^{O2} = I(p_1 - |s_2|, s_1)$.

Second, consider the path of transforming O_1 against O_3 and O_2 in sequence. Following the same reasoning, we have: $O_1^{O3} = TM1.1(O_1, O_3) = I(p_1 + |s_3|, s_1)$, which means O_1 is right-shifted by O_3; and $O_2^{O3} = TM1.1(O_2, O_3) = D(p_2, s_3 + s_2)$ (*Case 12* in Table 2), which means the delete-effect range of O_2 is extended to absorb s_3 inserted by O_3. Finally, $O_1^{O3,O2} = TM1.1(O_1^{O3}, O_2^{O3}) = I(p_1 + |s_3| - |s_3 + s_2|, s_1) = I(p_1 - |s_2|, s_1)$. Comparing the results along the two transformation paths, we have $O_1^{O3,O2} = O_1^{O2,O3}$, and the theorem holds. □

To illustrate, we apply *TM1.1* to the same FT scenario in Figure 2, and obtain new results shown in Figure 6.

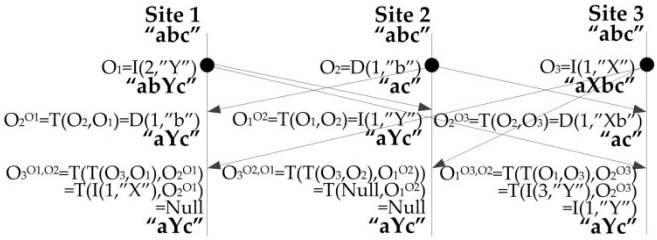

Figure 6. An FT puzzle solution example under *TM1.1*.

In this example, transforming O_3 against O_1 and O_2 in two different orders produces the same operation *Null*:

1. At Site 1, O_3 is transformed against O_1 and O_2 in sequence to produce $O_3^{O1,O2} = TM1.1(O_3^{O1}, O_2^{O1}) = Null$ (O_3^{O1} is absorbed by O_2^{O1}), where $O_3^{O1} = TM1.1(O_3, O_1) = I(1, "X")$; and $O_2^{O1} = TM1.1(O_2, O_1) = D(1, "b")$.
2. At Site 2, O_3 is transformed against O_2 and O_1 in sequence to produce $O_3^{O2,O1} = TM1.1(O_3^{O2}, O_1^{O2}) = Null$, where $O_3^{O2} = TM1.1(O_3, O_2) = Null$ (O_3 is absorbed by O_2); $O_1^{O2} = TM1.1(O_1, O_2) = I(1, "Y")$.

At the ends of the two sequences of transformation and execution, the final document states are the same: *"aYc"*, which achieves convergence and combined-effects as defined in *CE1.1*.

Verification of the FB Solution

The following theorem establishes that the FB puzzle is resolved under *TM1.1*.

Theorem 2. *Under TM1.1, the FB puzzle is resolved.*

Proof: According to the CP2 verification results for *TM1.0*, an FB puzzle may occur only under one of the 8 CP2-violation cases. We examine the case of transforming the insert operation against the two delete operations under the following positional pattern: $O_1 = I(p_1, s_1)$, $O_2 = D(p_2, s_2)$, $O_3 = D(p_3, s_3)$, and $p_2 < p_3 = p_1 < p_2 + |s_2| < p_3 + |s_3|$. First, consider the path of transforming O_1 against O_2 and O_3 in sequence. Since O_1 falls in the middle of the delete-effect range of O_2, O_1 will be absorbed by O_2 under *TM1.1*, which results in $O_1^{O2} = Null$, thus $O_1^{O2,O3} = Null$ based on the property of the *Null* operation. Second, consider the other path of transforming O_1 against O_3 and O_2 in sequence. Since O_1 is at the first position of the delete-effect range of O_3, O_1 will also be absorbed by O_3 under *TM1.1*, which also results in $O_1^{O3} = Null$, thus $O_1^{O3,O2} = Null$. Comparing the results along the two transformation paths, we have $O_1^{O3,O2} = O_1^{O2,O3}$, so the FB puzzle has been resolved in this case. Similar reasoning can be applied to other 7 cases to arrive at the same conclusion. So, the FB puzzle has been resolved in all possible cases, and the theorem holds. □

To illustrate, we apply *TM1.1* to the same FB scenario in Figure 5 and obtain new results shown in Figure 7.

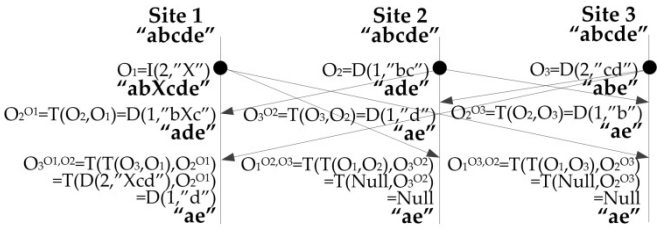

Figure 7. An FB puzzle solution under *TM1.1*.

In this example, transforming O_1 against O_2 and O_3 in two different orders produces the same *Null* operation:

1. At Site 2, O_1 is transformed against O_2 and O_3 in sequence: all steps are the same as that in Figure 5 and the final transformation result $O_1^{O2,O3} = Null$.
2. At Site 3, O_1 is transformed against O_3 and O_2 in sequence to produce $O_1^{O3,O2} = TM1.1(O_1^{O3}, O_2^{O3}) = Null$, where $O_1^{O3} = TM1.1(O_1, O_3) = Null$ (O_1 is absorbed by O_3), which illustrates how the FB puzzle is resolved in this scenario; and $O_2^{O3} = TM1.1(O_2, O_3) = D(1, "b")$, which is the same as in Figure 5.

At the ends of the two sequences of operation transformation and execution, the final document states are the same string "ae", which achieves convergence and combined-effects as defined in *CE1.1*.

Transformation Functions Based on TM1.1

Drawn insights from this verification work and guided by *TM1.1*, we have designed a set of string-wise transformation functions (in pseudo-code) in Table 3. The

following steps may be followed to convert *TM1.1* into transformation functions: (1) each operation type permutation corresponds to a transformation function, e.g. the *I-D* type permutation in *TM1.1* corresponds to the function $T(I(p_1,s_1), D(p_2, s_2))$ in Table 3; (2) each operation positional relation case serves as one *if*-condition in the function; multiple positional relation cases that share the same outcome are merged into one compound condition, e.g. the two cases, $p_1 = p_2 + |s_2|$ and $p_1 > p_2 + |s_2|$, are merged into a single compound condition $p_1 \geq p_2 + |s_2|$ as they produce the same output operation $I(p_1 - |s_2|, s1)$ in $T(I(p_1, s_1), D(p_2, s_2))$; (3) each distinctive output in *TM1.1* corresponds to one *return* statement in the function. The correspondences between the cases in *TM1.1* and the function statements are explicitly annotated in Table 3. Cases 5 and 12 are specially highlighted (in bold) as they represent the only differences between *TM1.1*-based functions and *TM1.0*-based functions (see APPENDIX-I).

Based on the correspondence between *TM1.1* and *TM1.1*-based functions and the verification result for *TM1.1*, it follows logically that the *TM1.1-based* transformation functions in Table 3 are able to preserve CE-CP1 and CP2. To the best of our knowledge, this is the first set of published string-wise transformation functions, defined on the common string-wise data model, that have the verified capability of preserving CE-CP1 and CP2.

```
T(I(p1,s1), I(p2,s2)) {
  if p1<p2 or (p1=p2 and priority(O1) > priority(O2)) //Case 1, 2-a
  return I(p1,s1) ;
  else return I(p1+|s2|,s1);} //Case 2-b, 3
```

```
T(I(p1,s1), D(p2,s2)) {
  if p1< p2 return I(p1,s1); //Case 4
  else if p1≥(p2+|s2|) return I(p1-|s2|,s1); //Case 7, 8
  else return NULL;} //Case 5, 6
```

```
T(D(p1,s1), I(p2,s2)) {
 if (p1+|s1|)≤p2 return D(p1,s1); //Case 9, 10
 else if (p1>p2) return D(p1+|s2|,s1); //Case 13
 else if (p1=p2) return D(p1,s2+s1); //Case 12
 else return D(p1, S[p1,p2-1]+s2+S[p2, p1+|s1|-1]);} //Case 11
```

```
T(D(p1,s1), D(p2,s2)) {
  if (p1+|s1|)≤p2 return D(p1,s1); //Case 14, 15
  else if (p1≥(p2+|s2|) return D(p1-|s2|,s1); //Case 16, 17
  else if (p1<p2) and (p2<p1+|s1|) and (p1+|s1|≤p2+|s2|) //Case 18, 19
        return D(p1, S[p1, p2-1]);
  else if (p1≤p2) and (p1<p2+|s2|) and  (p2+|s2|<p1+|s1|) //Case 20, 21
        return D(p2, S[p2+|s2|,p1+|s1|-1]);
  else if  (p1<p2) and (p2+|s2|<p1+|s1|)  //Case 22
        return D(p1, S[p1,p2-1]+S[p2+|s2|,p1+|s1|-1])
  else return NULL;} //Case 23, 24, 25, 26
```

Table 3 *TM1.1*-based transformation functions capable of preserving CE-CP1 and CP2.

COMPARISON TO PRIOR WORK

String-Wise vs Character-Wise Verification

Verification of transformation functions for string-wise operations, reported in this paper, has been built on and significantly extended prior verification work on character-wise operations in [29]. From OT function design point of view, string-wise operations possess the following main differences from character-wise operations [28]:

1. a string-wise *delete* covers a deleting range, which may include the characters in the string as well as the *interval positions* between characters;
2. concurrent string-wise *delete* operations may *arbitrarily overlap* with each other and even with concurrent insert operations; and
3. a string inserted by a previous *insert* operation may be changed by follow-up *insert* and *delete* operations.

The above factors make simplistic design and verification methods that were suitable to character-wise operations become inapplicable to string-wise operations. As elaborated in previous sections (and in APPENDIX-II), exhaustive derivation of combined-effects and verification cases for string-wise operations require new insights and analytic techniques to cover all possible cases without causing verification space explosion.

This work is unique in discovering all puzzles, including the new FB puzzle, under the string-wise operation model, in providing efficient function level solutions to all discovered puzzles, and in verifying the correctness of the proposed solutions.

Alternative Puzzle Solutions

To our best knowledge, there has been no prior verification work on string-wise operations. However, there had been some prior attempts to resolve the FT puzzle under the char-wise operation model [5,8,11,13]. One prior approach was to invent special data models or structures to combat the fundamental positional shifting effects of insert/delete operations, e.g. injecting meta-data (e.g. tombstones [13]) into the sequential data model, or converting the sequential data model to a *Commutative Replicated Data Type* [16], that associates each char with an immutable and sequentially ordered identifier, so that concurrent insert or delete operations, equipped with such identifiers and maintenance and mapping schemes, can be executed in any order. Another approach was to invent new OT system components and structures (e.g. mixing control algorithms with transformation functions [11]) that are radically different from the common OT system structure that separates the generic control algorithms from application-specific transformation functions [6,14,17,23,24,27]. Though different from each other, those alternative approaches share one issue: they made major changes to key OT components, e.g. data models or generic algorithms, that are actually unrelated to the FT puzzle but have complex impact on the whole system. Consequently, such changes often brought in new problems (in both correctness and efficiency), that are, in our view, more complex and difficult than the original FT problem they were proposed to solve [28].

In contrast, the *TM1.1* proposed in this paper resolves both FT and FB puzzles under the string-wise model with *micro-*

surgeries on only two cases in a transformation matrix, without affecting the rest of an OT system. This solution preserves common data and operation models supported by existing OT systems [26], so the solution is readily adoptable to real-world applications powered by those OT systems. The *TM1.1* solution causes no additional overhead and induces no new puzzle as formally verified in the paper.

Past research has also found that CP2-violation can be resolved by a CP2-avoidance approach, i.e. designing suitable OT control algorithms that avoid requiring transformation functions to preserve CP2 [14, 27, 32, 34]. This approach has the advantage of being generic and applicable to collaborative applications under different data and operation models, but requires the OT control system to impose a total transformation order among operations [34]. In contrast, the *TM1.1-like* CP2-preserving approach provides transformation function solutions to OT systems that were designed without requiring any ordering constraint on operation propagation or transformation [17, 23,24]. However, the CP2-preserving approach requires designing different transformation functions for operation and data models that cannot be mapped into the string-wise model. From the verification point of view, CP2-avoiding control algorithms are more suitable for verification by mathematic proof [12,25,27,34], whereas the *TM1.1-like* CP2-preserving approach is subject to application-specific data and operation modeling and more suitable for verification by exhaustive-case-checking, as demonstrated in this work or using model checkers [12].

CONCLUSIONS

In this work, we have contributed an exhaustive puzzle search, resolution and correctness verification with respect to transformation properties *CE-CP1* (*Combined-Effects – Convergence Property 1*) and *CP2* (*Convergence Property 2*) for OT systems supporting string-wise operations.

First, we proposed a verification framework, comprising the modeling of string-wise operations, specifications of Combined-Effects (CE) and Transformation Matrix (TM) of concurrent operations, and methods for exhaustive derivation of transformation cases related to a given transformation property, which is able to cover all possible transformation cases, without causing exponential verification case explosion. Then, we conducted exhaustive checking of all possible transformation cases to verify whether a given *TM* is able to preserve *CE-CP1* and *CP2*. Moreover, we developed a software tool, called *OTX* (*OT eXplorer*), to automate the verification case checking and to support experimentation with alternative transformation solutions. Based on the exhaustiveness of the verification cases in our framework and the fact that no single violation case was found under the proposed transformation solution (*TM1.1*), we claim *our solution preserves CE-CP1 and CP2 under the string-wise data and operation model.* From this result, we can derive that *all OT puzzles, under CE-CP1 and CP2 and the string-wise model, have been discovered and resolved*, which solves a long-standing open issue in OT research and contributes to the advancement of OT fundamental knowledge and technological innovation.

In this work, we have demonstrated that *TM* is an effective tool not only for correctness verification but also for transformation function design. Guided by *TM1.1*, we have designed a set of string-wise transformation functions capable of preserving both *CE-CP1* and *CP2*. These functions would be of interest to practitioners who are seeking concrete transformation functions that are simple to implement, significantly more efficient than character-wise transformation functions, and having verified correctness with respect to established transformation properties.

Existing and emerging real-world applications, such as Google Docs, Codoxware, and cloud storage systems [3], provide exciting opportunities and fresh stimuli to future OT research and innovation. Our main ongoing and future work is to apply core OT techniques to advanced real-world applications that are based on data and operation models with domain-specific constraints on conflict resolution and consistency requirements.

ACKNOWLEDGEMENTS

This research is partially supported by an Academic Research Fund Tier 2 Grant (MOE2015-T2-1-087) from Ministry of Education Singapore. The authors wish to thank anonymous reviewers for their insightful and constructive comments and suggestions.

REFERENCES

1. Agustina, F. Liu, S. Xia, H.F. Shen, and C. Sun. CoMaya: Incorporating advanced collaboration capabilities into 3D digital media design tools. *ACM CSCW* (2008), 5-8.
2. Agustina and C. Sun. Dependency-conflict detection in real-time collaborative 3D design systems. *ACM CSCW* (2013), 715-728.
3. Agustina and C. Sun. Operational Transformation for Real-time Synchronization of Shared Workspace in Cloud Storage. *ACM GROUP* (2016).
4. J. Begole, M.B. Rosson, and C.A. Shaffer. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM TOCHI 6, 2* (1999), 95-132.
5. H. Boucheneb, and A. Imine. On model-checking optimistic replication algorithms. *FORTE on Formal Techniques for Distributed Systems* (2009), 73-89
6. C.A. Ellis, and S.J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD* (1989), 399-407.
7. N. Gu, J. Yang, and Q. Zhang. Consistency maintenance based on the mark & retrace technique in groupware systems. *ACM GROUP* (2005), 264-273.
8. A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. *ECSCW* (2003), 277-293.

9. M. Koch, and G. Schwabe. Interview with Jonathan Grudin on "Computer-Supported Cooperative Work and Social Computing". *Bus Inf Syst Eng.* DOI 10.1007/s12599-015-0377-1. Springer. Published online: 03 March 2015.
10. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM 21, 7* (1978), 558-565.
11. D. Li, and R. Li. Preserving operation effects relation in group editors. *ACM CSCW* (2004), 427-466.
12. Y. Liu, Y. Xu, S. Zhang, and C. Sun. Formal verification of operational transformation. *Proc. of the 19th International Symposium on Formal Methods* (2014), 432-448.
13. G. Oster, et al. Tombstone transformation functions for ensuring consistency in collaborative editing systems. *IEEE Conf. CollaborateCom* (2006), 1-10.
14. D. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. *ACM UIST* (1995), 111-120.
15. A. Prakash, and M. Knister. A framework for undoing actions in collaborative systems. *ACM TOCHI 1, 4* (1994), 295-330.
16. N. Preguiça, et al. A commutative replicated data type for cooperative editing. *IEEE ICDCS* (2009), 395-403.
17. M. Ressel, N. Ruhland, and R. Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. *ACM CSCW* (1996), 288-297.
18. C.S. Robert. Collaborative editing for mission control. *Third Int. Workshop on Collaborative Editing Systems*, in conjunction with *ACM GROUP* (2001).
19. B. Shao, D. Li, and N. Gu. A sequence transformation algorithm for supporting cooperative work on mobile devices. *ACM CSCW* (2010), 159-168.
20. B. Shao, D. Li, and N. Gu. An optimized string transformation algorithm for real-time group editors. *IEEE ICPADS* (2009), 376-383.
21. B. Shao, D. Li, and N. Gu. ABTS: A transformation-based consistency control algorithm for wide-area collaborative applications. *IEEE CollaborateCom* (2009).
22. H.F. Shen, and C. Sun. A log compression algorithm for operation-based version control systems. *IEEE Conf. on Computer Software and Application (2002),* 867-872.
23. C. Sun, and C.A. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. *ACM CSCW* (1998), 59-68.
24. 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 TOCHI 5, 1* (1998), 63-108.
25. C. Sun. Undo as concurrent inverse in group editors. *ACM TOCHI 9, 4* (2002), 309-361.
26. C. Sun, S. Xia, D. Sun, D. Chen, H.F. Shen, and W. Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM TOCHI 13, 4* (2006), 531-582.
27. D. Sun, and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE TPDS 20, 10* (2009), 1454-1470.
28. C. Sun. OTFAQ: Operational Transformation Frequently Asked Questions and Answers. http://www.ntu.edu.sg/home/czsun/projects/otfaq.
29. C. Sun, Agustina, and Y. Xu. Exploring operational transformation: from core algorithms to real-world applications. *ACM CSCW* (2011) *demo.*
30. C. Sun, H. Wen, and H. Fan. Operational transformation for orthogonal conflict resolution in collaborative 2-dimensional document editors. *ACM CSCW* (2012),1391-1400.
31. C. Sun, Y. Xu, and Agustina. Exhaustive search of puzzles in operational transformation. *ACM CSCW* (2014), 519-529.
32. N. Vidot, M. Cart, J. Ferrie, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. *ACM CSCW* (2000), 171-180.
33. D. Wang, A. Mah, and S. Lassen. Google wave operational transformation. *http://www.waveprotocol.org/whitepapers/operational-transform.*
34. Y. Xu, and C. Sun. Conditions and patterns for achieving convergence in OT-based co-editors. *IEEE Trans. on Parallel and Distributed Systems, 27, 3* (2016), 695-709.

APPENDIX of this paper can be found at http://www.ntu.edu.sg/home/czsun/cscw2017_337_Appendix.pdf