Kumawat and Khunteta *Technol Innov Educ (2016) 2:9*
DOI 10.1186/s40660-016-0015-3

Technology, Innovation and Education

RESEARCH Open Access

Inclusive transformation consistency control algorithm in distributed system

Santosh Kumawat[1*] and Ajay Khunteta[2]

*Correspondence: santoshkumawat82@ymail.com
[1] School of Engineering and Technology, Poornima University, IS-2027 To 2031 Ramchandrapura P.O. Vidhani Vatika Sitapura Extension, Jaipur, Rajasthan 303905, India
Full list of author information is available at the end of the article

Abstract

Operational transformation (OT) is the most effective method for consistency and concurrency control in multi-user groupware applications. This study proposes a new string-based OT algorithm to address the challenge of swapping and transposing two deletions. It has removed the faults of previous existing algorithm swapDD (ABTS: a transformation-based consistency control algorithm for wide-area collaborative applications, collaborative computing: networking, applications and worksharing 1–10, 2009). Existing algorithm swapDD fails totally in transposing two deletions if the first operation region is included in the second operation region or the second operation string is covered by the first operation string. In addition, swapDD has not considered partial overlapping between two deletions in swapping and fails at boundary conditions. New proposed algorithm works well in all possible cases of transposing two deletions. It handles overlapping and splitting of operations.

Keywords: Inclusive transformation algorithm, Distributed systems, Concurrency control, Consistency control, Groupware system

Background

Real-time groupware systems, such as multi-player game, and real-time computer conferencing in the area of computer-supported cooperative work have multiple users where the actions of all users must be propagated to all other users.

Groupware systems are multi-user systems that provide an interface to a multi-user shared environment, which require sharing of data, fine-granularity, concurrency control, and fast response times. Concurrency control protocols are needed to repair inconsistencies in the multi-user transactions and areas of computing systems, such as database systems, distributed systems, and groupware systems. Therefore, there are specific requirements (Sun et al. 1998): high local responsiveness, unconstrained interaction, real-time communication, and consistency.

Theorem 1 *In a consistent shared environment which has replicated data after execution of all operations, all have the same data.*

Traditional concurrency control methods, such as locking, transactions, single active participant, dependency detection, and reversible execution, may cause the loss of interaction results and were not suitable for distributed interactive applications that demand fast local response satisfying user intentions, intention consistency, and convergence.

Over the past decade, operational transformation (OT) has become an established acceptable method for consistency maintenance in group editors. Compared with alternative concurrency control methods, OT has been found uniquely promising in better way achieving convergence, causality, and intention preservation without killing responsiveness and concurrent work (Shao et al. 2009). OT allows users to edit any part of the shared data at any time. Local operations are always executed as soon as they are generated by the user. Remote operations are transformed before execution to repair inconsistencies. Most of the existing OT algorithms only support primitive character operations like insert and delete. Only a few OT algorithms support string primitive operations like insert and delete.

Review of OT algorithms

Operational transformation algorithms have been studied over the past 25 years. OT algorithms correctness cannot be formally proved due to informal condition called "intention preservation." OT algorithms only consider two primitive character-based operations like insert and delete.

We have reviewed a number of major OT algorithms for consistency maintenance in real-time group editors, including the distributed operation transformation (dOPT) algorithm (Ellis and Gibbs 1989), the generic operational transformation (GOT) algorithm (Sun and 1998), GOT optimized (GOTO) algorithm (Sun et al. 1998), state difference transformation (SDT) algorithm (Li and Li 2006), SCOT2 (Suleiman et al. 1998), SCOT 3/4 algorithm (Vidot et al. 2000), adopted (adOPTed) algorithm (Ressel et al. 1996), admissibility-based transformation (ABT) algorithm (Li and Li 2010), ABT-undo (ABTU) algorithm (Shao et al. 2010), admissibility-based sequence transformation (ABST) (Sun and 1998), and admissibility-based transformation with strings (ABTS) algorithm (Shao et al. 2009).

On categorizing all existing OT algorithms on the basis of major existing algorithms, such as dOPT, adOPT, GOT, GOTO, SDT, SOCT2, SOCT3/4, ABT (Li and Li 2007), and then further classified on the basis of area of operation, such as undo, char, string, web, graph and so on, we get that only three algorithms support string handling—GOT, GOTO and ABTS. We conclude that ABTS supports for string handling and is better than GOT and GOTO, because it has less time complexity and space complexity. In addition, ABTS is based on ABT framework, which can be formally proved. We conclude that ABTS is the best string-based OT algorithm as has less time and space complexity than GOT and GOTO (see Fig. 1). This study is focused on string-based OT algorithm based on ABT framework and removed the faults of ABTS algorithm.

System model and notations

In a multi-user system on starting of session, the shared data are replicated at all sharing sites. In OT, local operations are executed immediately without delay, and local operations are propagated to remote sites in the background, so local operations execution do not suffer. The shared data are like a linear string 's' of atomic characters and positions 'p' in the string that starts from zero and consider two only primitive string operations, called,

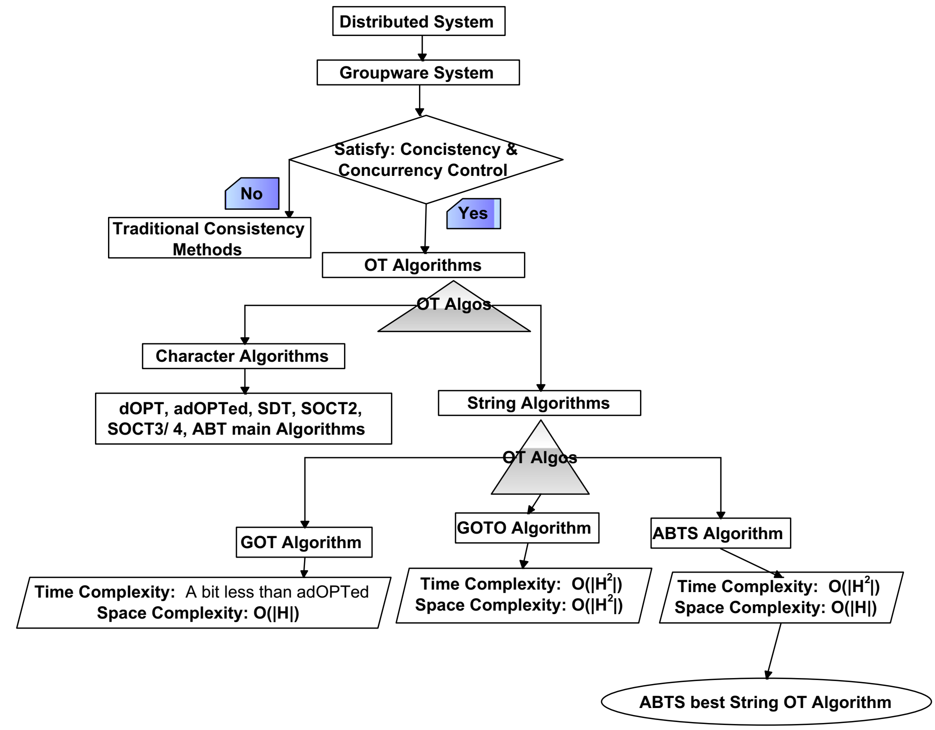

Fig. 1 Flow of problem justification process

insert(p, s) and delete(p, s). Here, insert(p, s) insert 's' at location 'p' in given string definition. In addition, delete(p, s) delete 's' at location 'p' in given string definition. The operations o1 and o2 are contextually serialized, denoted by $o1 \rightarrow o2$, if $o2'$s position is defined in the resulting state of applying o1 (but no other operation). The standard notations are summarized in Table 1, where a few standard notations are taken from (Shao et al. 2009).

Definition 1 o1 and o2 are contextually equivalent o1||o2, o1Uo2 and if input is o1 and output then output should be $o2 \rightarrow o1'$.

Definition 2 If we have $exec(o_i)$, then all $exec(o_{i-1})$ must be completed then only o_i satisfy causality.

Definition 3 If o1Uo2, then IT(o1,o2) satisfy admissibility. It does not have inconsistent order at shared environment.

Algorithms

The basic swap functions for swapping two primitive operations insert and delete exist in (Shao et al. 2009). Given two operations o1 and o2, where $o1 \rightarrow o2$, function swap(o1, o2) transposes them into $o1'$ and $o2'$, such that $o2' \rightarrow o1'$. Depending on their types, insert (I) and delete (D), we call different swapping functions. The basic swap function for swapping primitive operations two deletions is swapDD (Shao et al. 2009). Here, swapDD and MGswapDD take two string operations o1 and o2 as parameters. Here,

Table 1 Standard notations

Notations	Description
o.id	Id of site that generate operation o
o.type	Type of operation o, i.e., either insert or delete
o.pos	Position of operation o
o.str	String insert/delete by o
o1 → o2	o1 occurs before o2
o1‖o2	o1 and o2 are concurrent
o1Uo2	o1 and o2 are contextually equivalent
o1 → o2	o1 and o2 are contextually serialized
[o1,o2]	An ordered list of two operations o1 and o2
<o1,o2>	Two operations in sequence
|L|	Number of objects in list L
L1.L2	Concatenation of two lists L1 and L2
s [i:len]	Substring of string s start from position i of length len
sq	A sequence is a special list in which all elements are operations that are contextually serialized
sq = <o1, o2,…, on>	sq = <o1, o2,…, on > , where o1 - > o2- > …- > on
< >	An empty sequence
L = [a, b, c], it has L = [a]· [b, c] = [a, b]· c	A sequence is a special list in which all elements are operations that are contextually serialized
|sq| = n	The number of elements in sequence sq = n
sq = < o1 > . < o2,… on >	All elements of sequence are contextually serialized
R1 = [o1.start, o1.end]	Operation region of operation o1 s R1 which start from o1.start & end at o1.end
o.Substring (i,len)	Substring of o start from i of length len
o.Substring (i)	Substring of o start from i position in o

o1.type = o2.type = delete. Before swapping, we have o1 → o2 and after swapping, we get o1′ and o2′, so that we can have o2′ → o1′.

Algorithm swapDD

Algorithm swapDD (o1, o2) transposes two deletions o1 and o2. There are three cases considered by (Shao et al. 2009). First, if o2.pos ≥o1.pos, it means that o2 is to delete a substring on the right side of the substring o1.str deleted by o1. Hence, if we execute o2 before o1 instead, then o2.pos should consider o1.str, because it has not been deleted yet. Therefore, o2 position shifted right by length of o1.str.

Second, if o2.pos + |o2.str| ≤o1.pos, it means that o2.str is completely on the left side of o1.pos. Hence, if o2 get executed before o1 instead, o1.pos should be shifted to the left, because o2.str has already been deleted.

Third, as in lines 6–12, o1.str is completely covered by o2.str. Then, if o2 get executed before o1 instead, o2.str is divided into three parts, among which the middle overlapping part is to be deleted by o1. The remaining left and right parts, as divided by position o1.pos, are deleted by two suboperations o2L and o2R, respectively. At last, finally, o1.pos should be set to o2.pos due to the deletion of o2L.str.

Algorithm swapDD (o_1, o_2): (o_2', o_1')

1. $o_1' \leftarrow o_1$; $o_2' \leftarrow o_2$;
2. if o_2.pos > = o_1.pos then

3. $o_2'.pos \leftarrow o_2.pos + |\, o_1.str|$
4. else if $o_2.pos + |\, o_2.str| \leq o_1.pos$ then
5. $o_1'.pos \leftarrow o_1'.pos - |\, o_2.str|$
6. else
7. $o_{2L} \leftarrow o_{2R} \leftarrow o_2$
8. $o_{2L}.str \leftarrow o_2.str[0{:}\ o_1.pos - o_2.pos]$
9. $o_{2R}.pos \leftarrow o_1.pos + |\, o_1.str|$
10. $o_{2R}.str \leftarrow o_2.str[o_1.pos - o_2.pos{:}]$
11. $o_2'.sol \leftarrow [o_{2L}, o_{2R}]$
12. $o_1'.pos \leftarrow o_2.pos$
13. endif
14. return(o_2', o_1')

Failure of algorithm swapDD

Algorithm swapDD fails in most of cases in swapping two deletions. Failure of algorithm swapDD in various conditions is highlighted in the following cases:

Case 1: *If o2.pos ≥o1.pos* In this case, swapDD fails at boundary condition means that if $o_2.pos = o_1.pos$, it fails totally (lines 2–3 of algorithm swapDD).

Case 2: *If there exist partial overlapping between deletion operations o1 and o2 regions* Here, partial overlapping between o_1 and o_2 means region of o_1 and o_2 overlaps to each other. In addition, we can say that $o_1.str$ partially overlaps by $o_2.str$. There can be either overlapping along the left border of o_1 with o_2 or overlapping along the right border of o_1 with o_2. In this case, lines 2–3 of algorithm swapDD execute for the right overlapping of $o_1.str$ with $o_2.str$, and lines 6–13 of algorithm swapDD execute for the left overlapping of $o_1.str$ with $o_2.str$ and it totally fails. As per the details of this algorithm given in (Shao et al. 2009), it has not discussed partial overlapping between two deletion operations o_1 and o_2 but in algorithm not put required conditions to avoid partial overlapping of o_1 and o_2. Therefore, either it has not considered the partial overlapping of o_1 and o_2 in swapDD just by assumption or it totally fails in this case.

Case 3: *If $o_1.str$ completely overlaps by $o_2.str$* In this case, swapDD lines 6–13 get executed, and it gives total wrong output in all cases. Ideally as per algorithm swapDD theory specified in (Shao et al. 2009), it should divide $o_2.str$ into three parts, among which the middle overlapping part is to be deleted by o1. However, it fails in splitting $o_2.str$ in the remaining left and right parts which are to be deleted by two suboperations o_{2L} and o_{2R}, respectively.

Case 4: *If o2.str completely overlaps by o1.str* This case is not discussed in theory of swapDD given in (Shao et al. 2009), but if we have this case, lines 2–3 of swapDD get executed and give total faulty result.

Therefore, it is concluded that swapDD fails totally in swapping two deletions if there exist partial or total overlapping of $o_1.str$ by $o_2.str$. In addition, in a few cases, it fails totally at boundary conditions.

C. Algorithm MGswapDD

```
MGswapDD(o1, o2)
{
Step1: o1'←o1;
        o1'.pos=o1.pos;
        o1'.str=o1.str;
Step2: o2'←o2;
        o2'.pos=o2.pos;
        o2'.str=o2.str;
Step3: if(o2.pos>(o1.pos+|o1.str|))
        {
Step4: o2'.pos=o2.pos+|o1.str|;
        }
Step 5: else if ((o2.pos+|o2.str|)<o1.pos)
        {
Step6: o1'.pos=o1'.pos-|o2.str|;
        }
Step7: elseif(o2.pos>o1.pos&&o2.pos<=(o1.pos+|o1.str|) &&(o2.pos+|o2.str|)>(o1.pos+|o1.str|))
        {
Step8: o2'.pos=o2.pos+((o1.pos+|o1.str|)-o2.pos);

Step9: o2'.str=o2.Substring((o1.pos+|o1.str|)-o2.pos);
         }
Step10: elseif(o2.pos<o1.pos&&(o2.pos+|o2.str|)>=o1.pos && o1.pos+|o1.str|>(o2.pos+|o2.str|)
)
        {
Step11: o2'.str=o2.Substring (0, (o1.pos-o2.pos));
Step12: o1'.pos=o1.pos-|o2'.str|;
        }
Step13: else
        {
 Step14: if(((o1.pos+|o1.str|)>=(o2.pos+|o2.str|))&&(o1.pos<= o2.pos))
{
Step15: o2'←null;
}
Step16 else {
Step17: o2Lpart.str←o2; o2Lpart.str=o2;
Step18: o2Lpart.pos=o2.pos;
Step19: o2Rpart.str←o2; o2Rpart.str=o2;
Step20: o2Rpart.pos=o2.pos;
Step21: o2Lpart.str=o2.Substring (0, o1.pos-o2.pos);
Step22: o2Rpart.pos=o1.pos+|o1.str|;
Step23: o2Rpart.str=o2.Substring (o1.pos-o2.pos+|o1.str|);
Step24: o2'.sol←[o2Lpart, o2Rpart];
Step25: o1'.pos=o2.pos;
       }
endif
   } endif    }
```

Algorithm MGswapDD

The new proposed algorithm MGswapDD has removed all faults of the existing algorithm swapDD and is working well in all possible cases of swapping two deletions. It works well at all boundary conditions. It has also considered the partial overlapping of operations o1.str and o2.str. Also if o1.str completely overlaps by o2.str or o2.str completely overlaps by o1.str, then also it works well totally. Thus, it considers well overlapping and splitting of operations. The MGswapDD is practically implemented in lab and works well on partial or total overlapping of operations. In addition, it works well on not overlapping operations and boundary conditions.

Algorithm MGswapDD is for swapping and transposing two deletions. The process of MgswapDD is explained in the following points. Here, if we have o $\leftarrow$ null means, o is initialized to null and will not perform any operation:

1. *From Line 3* if(o2.pos > (o1.pos + |o1.str|)), means if o2 lies completely on the right side of o1 then, if we execute o2 before o1 instead, then o2.pos should consider o1.str, because it has not been deleted yet. Therefore, o2 position shifted right by length of o1.str.
2. *From Line 5* if ((o2.pos + |o2.str|) < o1.pos), means o2.str is completely on the left side of o1.pos. Hence, if o2 get executed before o1 instead, o1.pos should be shifted to the left, because o2.str has already been deleted. Therefore, o1 shift left equal to length of o2.str.
3. *From Line 7* if (o2.pos > o1.pos&&o2.pos $\leq$ (o1.pos + |o1.str|)&&(o2.pos + |o2.str|) > (o1.pos + |o1.str|)), means o1.str overlaps partially with o2.str along its right boundary, then o1.str and o1 position will remain unchanged and o2′ position will shift right by the length of overlapping region of o1.str and o2.str. In addition, o2′.str will be set to not overlapping part of o2 string. Here, the overlapped region gets deleted by o1′, and o2′ deletes the remaining not overlapping region of o2.str.
4. *From Line10* if(o2.pos < o1.pos&&(o2.pos + |o2.str|) $\geq$o1.pos && o1.pos + |o1.str| > (o2.pos + |o2.str|), means o1.str overlaps partially with o2.str along its left boundary then the overlapped region gets deleted by o1, and o2 deletes the remaining not overlapping region. Here, o2′ string will reduced to not overlapping part of o2 string by deducting the overlapped region from the existing o2 string. In addition, o1′ position is shifted right by length of o2′ string, since o2′ is already deleted since after swapping, we have $o_2' \rightarrow o_1'$.
5. *From lines 13*–25 get executed if none of the above conditions are true. Line 14 check if o2.str completely covered by string o1.str. If o1 and o2 delete the same substring of given string sequence 's' which lie at the same position, then also condition at line 14 is true. In this case, o2 initialized to null, and o1 deletes the o1.str from o1 position. Lines 16–25 are executed if o1.str is completely covered by o2.str. Then, if o2 get executed before o1 instead, o2.str is divided into three parts, among which the middle overlapping part is to be deleted by o1. The remaining left and right parts are deleted by two suboperations o2L and o2R of o2′, respectively. At last, o1′.pos should be set to o2.pos due to the deletion of o2L.str. Therefore, if o1 is totally overlapped by o2 string, then the overlapping region gets deleted by o1, and o2 deletes its remaining regions left and right called o2Lpart and o2Rpart, respectively, which are separated by o1 region.

Correctness proof

In multi-user environment, practically, we have implemented ABTS and MGswapDD in lab using Qualnet and ASP.Net software.

Algorithm swapDD

Case 1: *If $o_2pos = o_1.pos$* In this case, swapDD fails at boundary condition means that if o_2.pos = o_1.pos, it fails totally (lines 2–3 of algorithm swapDD).

For example, let s = "TheGodHelpAllEqually." Here, suppose o_1 = delete(3, "God-Help") and o_2 = delete(3, "God"). Therefore, condition at line 2 *if(o_2.pos ≥o_1.pos)* is true, since o_2.pos = o_1.pos, so by line 3, we get o_2'.pos = o_2.pos + $|o_1.str|$ so we get o_2'.pos = 3+7 = 10. Here, in given string definition s, we apply o_2' = delete(10, "God"); the operation fails since at starting position '10' substring "God" not found (see Fig. 2). Therefore, swapDD fails totally.

If we implement new proposed MGswapDD for the same inputs like case 1, we get o1′ = o1 and o2′ = null, so get right input because if o1′ get executed, then no need to execute o2′ because o2 string get deleted by o1′ since o2 string is overlapped by o1 string (see Fig. 3).

Case 2: *If there exist partial overlapping between deletion operations o1 and o2 regions* Here, partial overlapping between o1 and o2 means region of o1 and o2 overlaps to each other.

For example, let s = "TheBirdsAreFlyingInTheSky"

Let o_1.str = "BirdsAreFlying" and o_1.pos = 3,$|o_1.str|$ = 14

o_2.str = "FlyingInTheSky" and o_2.pos = 11.

Here, o_1 overlaps with o_2 along its right boundary. And if we execute swapDD; condition at line 2 is correct that is (o_2.pos ≥o_1.pos), since 11 > 3, so enter in if block

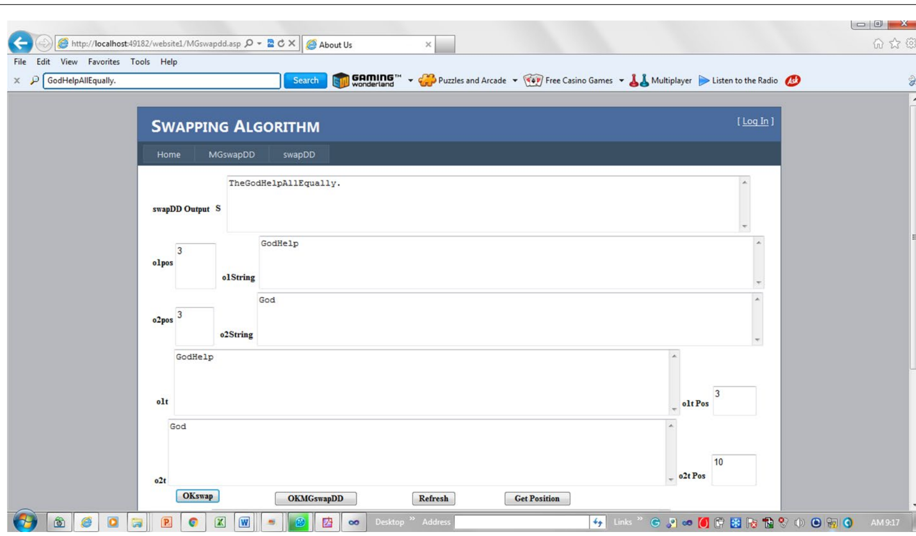

Fig. 2 Practical implementation of swapDD in lab for inputs of case 1 (wrong output)

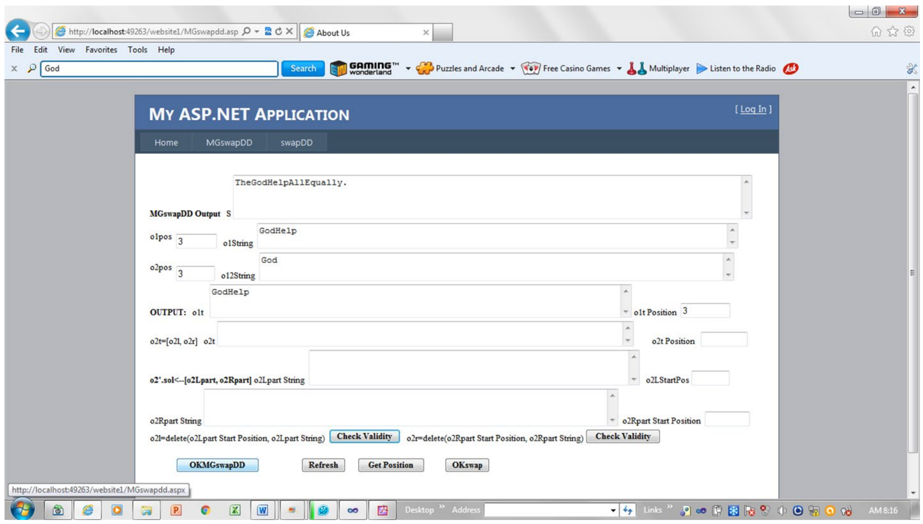

Fig. 3 Practical implementation of MGswapDD in lab (right output)

and execute the code at line 3 that are $o_2'.pos = o_2.pos + |o_1.str|$, so here, we get $o_2'.pos = 11 + 14 = 25$, so we get $o_2' =$ delete (25, "FlyingInTheSky"). The operation o_2' fails since at starting position '25' substring "FlyingInTheSky" not found. Even position '25' not exist in given 's'. Thus, swapDD fails totally (see Fig. 4).

When we implement MGswapDD in lab practically for inputs of case 2, we get $o1' = o1$. $o_2'.pos = 17$ and $o2'.str$="InTheSky" which give right output, because there exist no overlapping in $o1'$ and $o2'$ and both lie at given position in string 's' (see Fig. 5).

Case 3: *If $o_1.str$ completely overlaps by $o_2.str$* In this case, swapDD lines 6–13 get executed and it gives total wrong output in all cases.

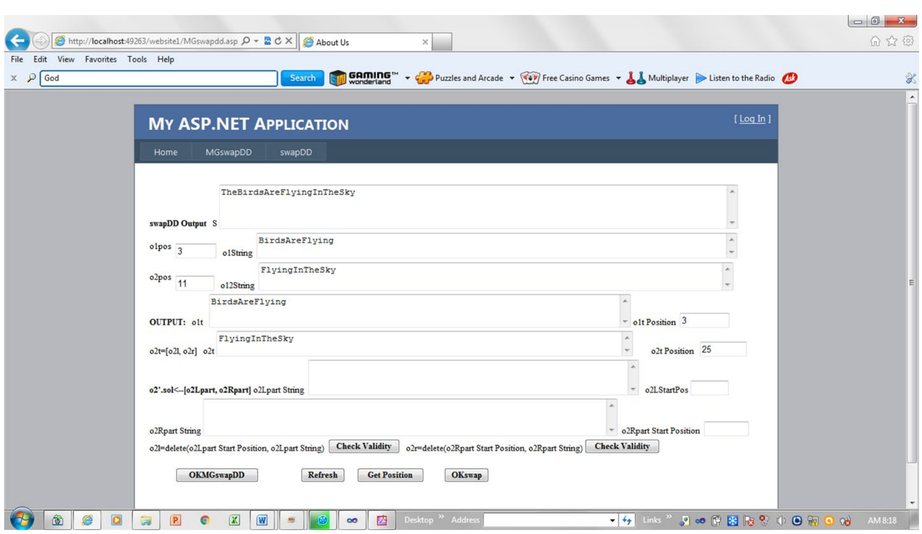

Fig. 4 Practical implementation of swapDD in lab for inputs of case 2 (wrong output)

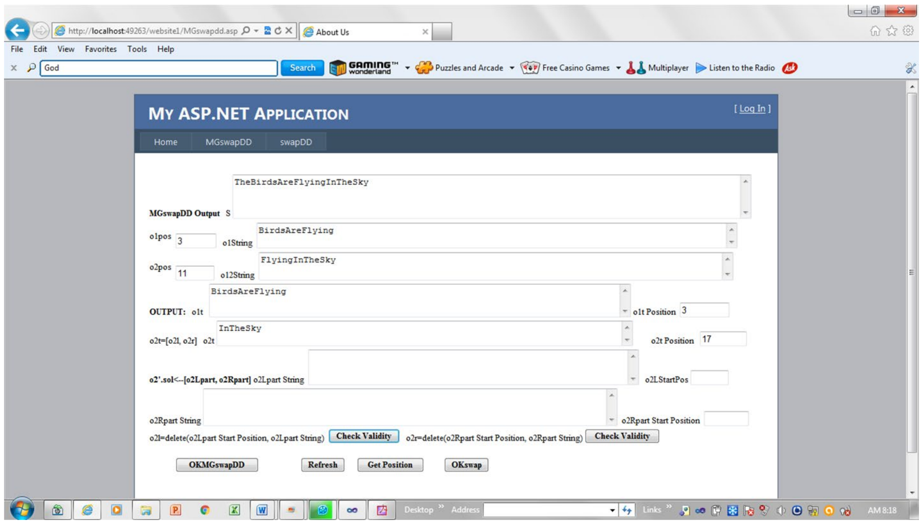

Fig. 5 Practical implementation of MGswapDD in lab for inputs of case 2 (right output)

For example, s = "WorkNotO nlyHardButGoodAlso".

Let o_1 = delete(7, "Only"); o_2 = delete(4, "NotOnlyHard")

Here, on executing swapDD lines 6–13, it will get executed, and we get from line7: $o_{2L} \leftarrow o_2$ and $o_{2R} \leftarrow o_2$. From line 8: o_{2L}.str $\leftarrow$ o_2.str [0:o_1.pos−o_2.pos], so we get o_{2L}.str $\leftarrow$ o_2.str [0:7−4] = "Not";

Also from line 9, we get o_{2R}.pos $\leftarrow$ o_1.pos + |o_1.str|; so we get o_{2R}.pos $\leftarrow 7 + 4 = 11$. And from Line 10 we get o_{2R}.str $\leftarrow$ o_2.str [o_1.pos−o_2.pos:]; so we get o_{2R}.str $\leftarrow$ o_2.str[7 − 4:]; so we get o_{2R}.str $\leftarrow$ "OnlyHard". Therefore, we get o_{2R} = delete (11, "OnlyHard") but in given 's' at position 11 "OnlyHard" not exist so o_2'.sol $\leftarrow [o_{2L}, o_{2R}]$ also fails totally (see Fig. 6).

When we practically implemented MGswapDD in lab for inputs of case 3, we get o1′.str = o1.str and o1′.pos = 4. We get o2L′.str = "Not" and o2L′.pos = 4, o2R′.str = "Hard",

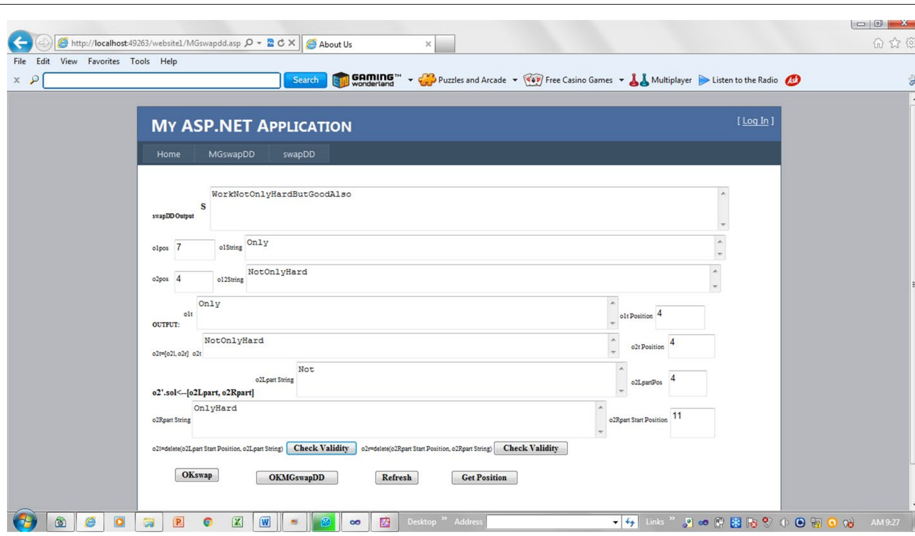

Fig. 6 Practical implementation of swapDD in lab for inputs of case 3 (wrong output)

o2R’.pos = 11 which give desired output, because o1 overlapped completely by o2 and overlapped region of o2 get deleted by o1′ so o2′ split in o2L’ and o2R’ to get desired output (see Fig. 7).

Case 4: *If o2.str completely overlaps by o1.str* If o_2.str completely overlaps by o_1.str, then in this case, swapDD lines 2–3 get executed and it gives total wrong output in all cases.

For example:
Let s = “GodHelpThoseWhoHelpThemselves”
o_2 = delete (12, “Who”);
o_1 = delete (3, “HelpThoseWhoHelp”);
Here on executing swapDD lines 2–3 will get executed and we get wrong output.
o1’ = “HelpThoseWhoHelp” o1’ position = 3 o2’ = “Who” o2’ position = 28
Algorithm swapDD failed. Thus, o_2'.sol ← $[o_{2L}, o_{2R}]$ fails totally (see Fig. 8).

When we practical implement MGswapDD in lab for inputs of case 4, we get o1′.pos = “HelpThoseWhoHelp” and o1′.pos = 3. Also o2′ = null, because o2 overlapped by o1 and o1′ delete the overlapped region so no need to execute o2′ (see Fig. 9).

Algorithm MGswapDD

Case 1: *If o2.pos = o1.pos* Here, in this case on executing MGswapDD lines 13–25, it will get executed and will give right result.

For example, let s = “TheBirdsAreFlyingInTheSky”
o_1 = delete(3, “BirdsAreFlying”); o_2 = delete(3, “Birds”)
Condition at line 14 is correct so switch to line 15. Condition if $(((o_1.pos + |o_1.str|) \geq (o_2.pos + |o_2.str|))\&\& (o_1.pos \leq o_2.pos))$. Here, we get if((3 + 14) ≥ (3 + 5)&&3 ≤3) returns true so code at line 15 that is $o_2' \leftarrow$ null get executed means o_2' will not execute any

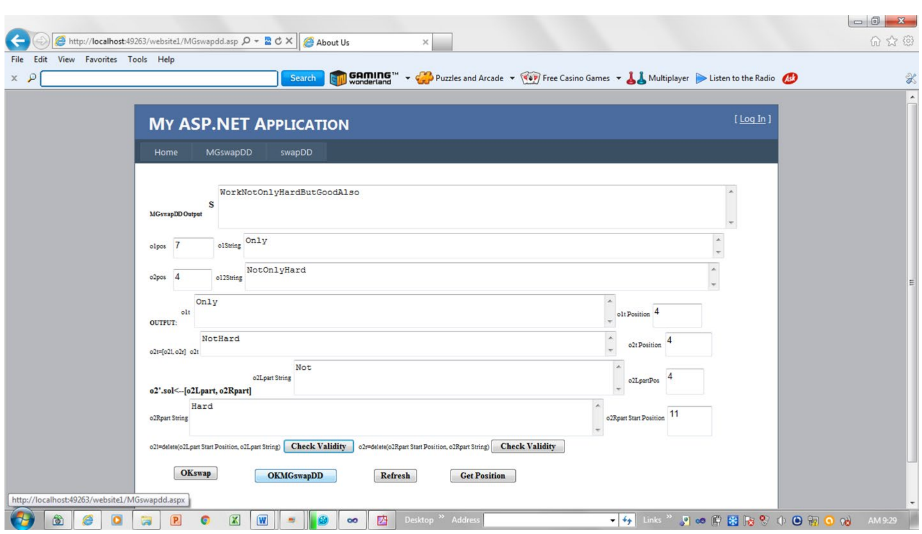

Fig. 7 Practical implementation of MGswapDD in lab for inputs of case 3 (right output)

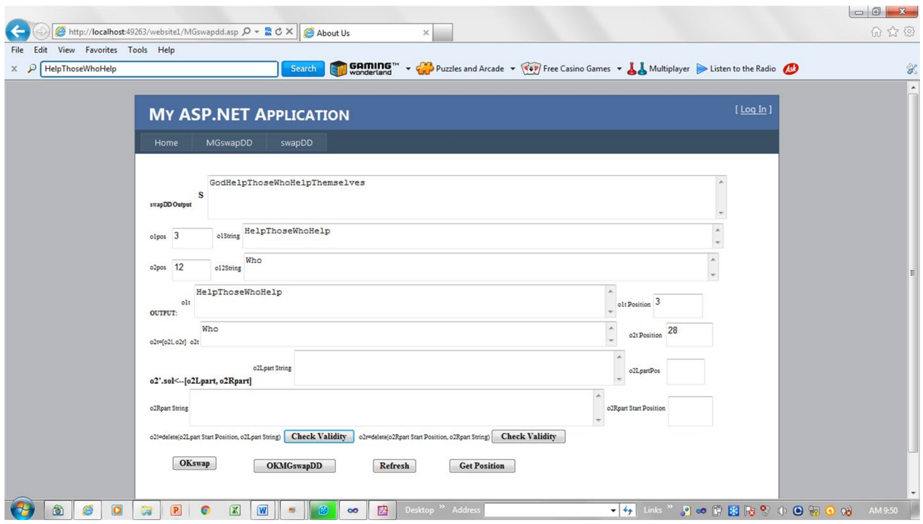

Fig. 8 Practical implementation of swapDD in lab for inputs of case 4 (wrong output)

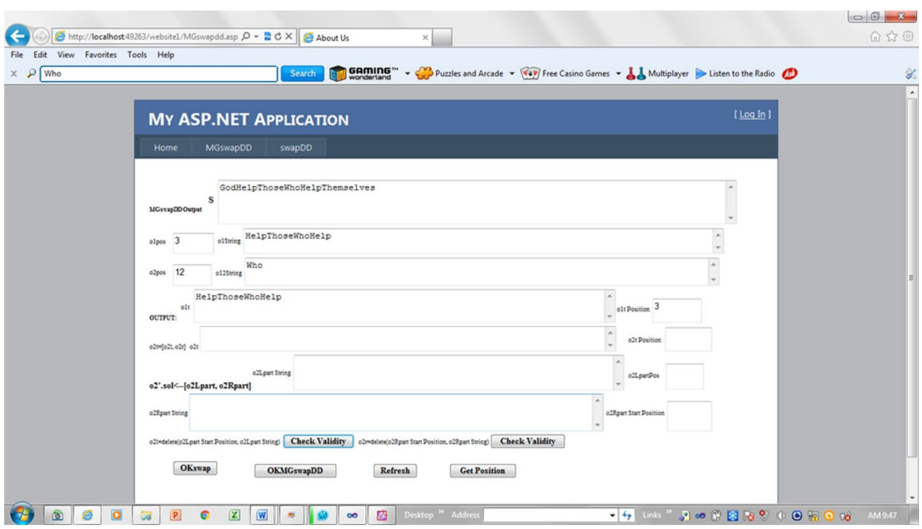

Fig. 9 Practical implementation of MGswapDD in lab for inputs of case 4 (right output)

operation both its string and position are null. And $o_1' \leftarrow o_1$ from line 1, so we get desired output "TheInTheSky" after execution of o_1' and o_2' where o_2' is null. It satisfies user intentions also (see Fig. 10).

Case 2: *If there exist partial overlapping between deletion operations o1 and o2 regions* Here, two cases are possible either o_1.str overlaps with o_2.str along its right border or left border.

First, we consider the case when o_1.str overlaps with o_2.str along its rightboundary. For example, let s = "GodPleaseHelpMeToTakeCareMyChild".

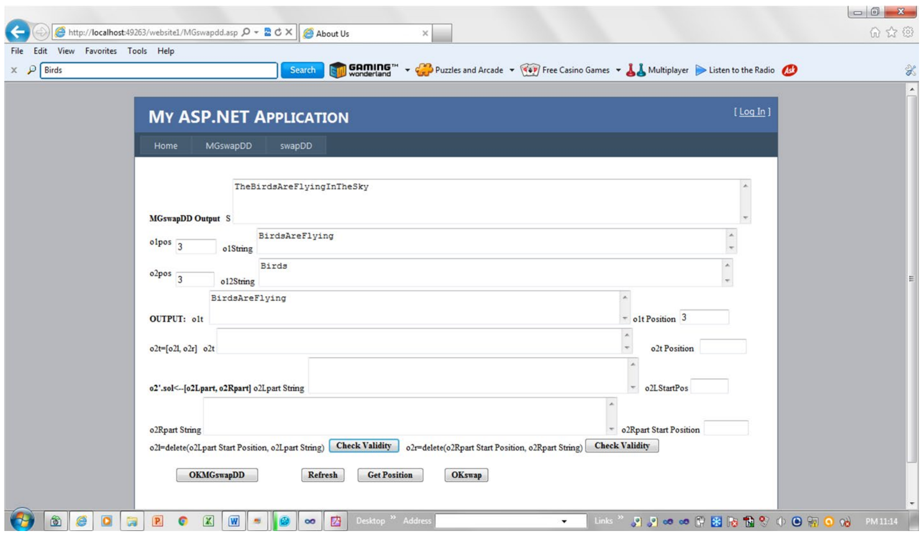

Fig. 10 Practical implementation of MGswapDD in lab for inputs of case 1 (right output)

$o_1 = (3, \text{“PleaseHelpMe”})$; $|o_1.str| = 12$ and $|o_2.str| = 12$; $o_2 = \text{delete}(13, \text{“MeToTakeCare”})$; $o_1.pos = 3$ and $o_2.pos = 13$.

Since on executing MGswapDD condition at line 7 is true that is if ($o_2.pos > o_1.pos \&\& o_2.pos \le (o_1.pos + |o_1.str|)$ && $(o_2.pos + |o_2.str|) > (o_1.pos + |o_1.str|)$) returns true, so lines 8 and 9 will get executed.

Step 8: $o_2'.pos = o_2.pos + ((o_1.pos + |o_1.str|) - o_2.pos)$;

Step 9: $o_2'.str = o_2.\text{Substring}((o_1.pos + |o_1.str|) - o_2.pos)$;

From step 8, $o_2'.pos = 13 + (3 + 12) - 13$; $o_2' = 15$;

Step 9: $o_2'.str = o_2.\text{Substring}(3 + 12 - 13) = o_2.\text{Substring}(2)$, so $o_2'.str =$ “ToTakeCare”. We get $o_2' = \text{delete}(15, \text{“ToTakeCare”})$ and it runs well since at position 15 “ToTakeCare” exist in given ‘s’. Therefore, the overlapped substring “Me” get deleted by o_1' and o_2' has deleted just unoverlapped part of o_2. Here, $o_1' \leftarrow$ o1 from line 1. So again, we get totally right output satisfying user intentions (see Fig. 11).

Second, we consider the case when $o_1.str$ overlaps with $o_2.str$ along its left boundary.

For example, let s = “GodPleaseHelpMeToTakeCareMyChild”.

$o_2 = (3, \text{“PleaseHelpMe”})$; $|o_2.str| = 12$ and $|o_1.str| = 12$; $o_1 = \text{delete}(13, \text{“MeToTakeCare”})$; $o_2.pos = 3$ and $o_1.pos = 13$. Since on executing MGswapDD condition at line 10 is true that is so the given code will get executed. Therefore, condition at line 10 is as follows: if($o_2.pos < o_1.pos \&\& (o_2.pos + |o_2.str|) \ge o_1.pos$ && $o_1.pos + |o_1.str| > (o_2.pos + |o_2.str|)$) returns true so from Step 11: $o_2'.str = o_2.\text{Substring}(0, (o_1.pos - o_2.pos))$; so we get $o_2'.str = o_2.\text{Substring}(0, (13 - 3)) =$ “PleaseHelp”; here $|o_2'.str| = 10$; and from step Step 12: $o_1'.pos = o_1.pos - |o_2'.str|$; we get $o_1'.pos = 13 - 10 = 3$. Since after deletion by o_2' the o_1'. pos should left by the length of $o_2'.str$ as o_2' lies left of o_1' and is already deleted. Here, we get finally $o_1' = \text{delete}(3, \text{“PleaseHelpMe”})$ and $o_2' = \text{delete}(15, \text{“ToTakeCare”})$ and $o_2' \rightarrow o_1'$ work well after swapping of $o_1 \rightarrow o_2$.

In this case, also we get right output.

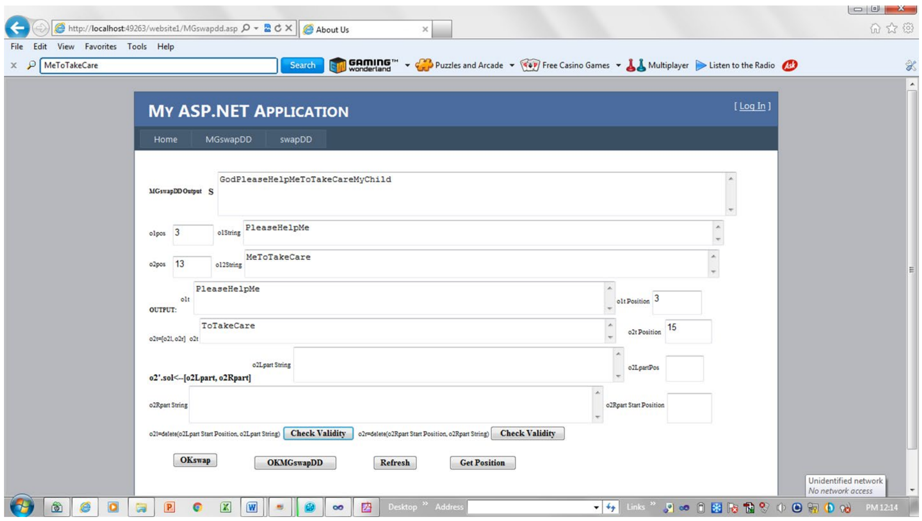

Fig. 11 Practical implementation of MGswapDD in lab for inputs of case 2 (right output)

Case 3: *If o_1.str completely overlaps by o_2.str* In this case, all the above conditions before line 13 are false so enter in else block at line 13. Here, condition at Step 14: if(((o_1.pos + |o_1.str|) ≥ (o_2.pos + |o_2.str|))&& (o_1.pos ≤ o_2.pos)) is false so enter in its else part. So lines 16 to 25 get executed.

For example, let s = "The Sun give us Heat and Light"; o_1 = delete(13, "us") and o_2 = delete(4, "Sun give us Heat"). From Step 17: $o_{2Lpart} \leftarrow o_2$; o_{2Lpart}.str = o_2.str; here, we have o_{2Lpart}.str = "Sun give us Heat". From Step 18: o_{2Lpart}.pos = o_2.pos; Here we have o_{2Lpart}.pos = 4. From Step 19: $o_{2Rpart} \leftarrow o_2$; o_{2Rpart}.str = o_2.str; here, we have o_{2Rpart}.str = "Sun give us Heat". From Step 20: o_{2Rpart}.pos = o_2.pos; here, we have o_{2Rpart}.pos = 4.
From Step 21: o_{2Lpart}.str = o_2.Substring (0, o_1.pos−o_2.pos); here, we have o_{2Lpart}.str = o_2.Substring (0,13−4) ≥ o_{2Lpart}.str = o_2.Substring (0,9) = "Sun give". From Step 22: o_{2Rpart}.pos = o1.pos + |o1.str|; here, we have o_{2Rpart}.pos = 13 + 2=15. From Step 23: o_{2Rpart}.str = o_2.Substring (o_1.pos−o_2.pos + |o_1.str|); here, we have o_{2Rpart}.str = o_2.Substring (13−4 + 2) = o_2.Substring (11) = "Heat". From Step 24: o_2'.sol ← [o_{2Lpart}, o_{2Rpart}]; so the left part "Sun give". get deleted by operation o_{2Lpart};right part "Heat" get deleted by o_{2Rpart} and the middle overlapping región "us" get deleted by o_1 and by *Step 25:* o_1'.pos = o_2.pos;so we get o_1'.pos = 4 since at first o_2' get executed and since o_{2Lpart} is already executed the position of o_1' shift left to o_2.pos. So $o_2' \rightarrow o_1'$ works correctly here (see Fig. 12).

Case 4: *If o_2.str completely overlaps by o_1.str* In this case, all the above conditions before line 13 are false so enter in else block at line 13. Here, condition at Step 14: if (((o_1.pos + |o_1.str|) ≥ (o_2.pos + |o_2.str|))&& (o_1.pos ≤ o_2.pos)) is true so enter in its if part. So step 15 will get executed.

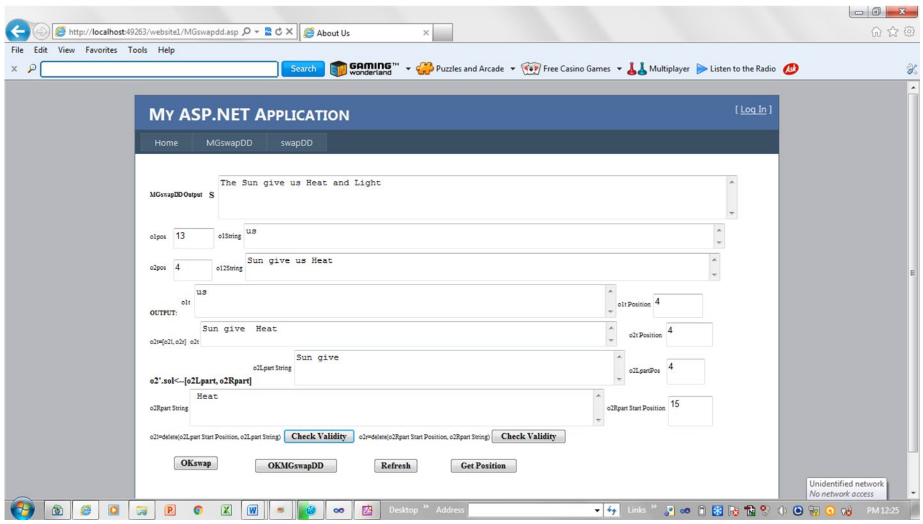

Fig. 12 Practical implementation of MGswapDD in lab for inputs of case 3 (right output)

For example, let s = "The God will help me always everywhere". o_1 = delete (4, "God will help me") and o_2 = delete (8, "will"). Here, condition at Step 14: if$(((o_1.pos + |o_1.str|) \geq (o_2.pos + |o_2.str|))\&\&(o_1.pos \leq o_2.pos))$ that is if$((4 + 15) \geq (8 + 4)\&\&4 \leq 8)$ is true so condition at line 15 get executed where o_2' is set to null means will not perform any operation and o_1 will remain as it is. If o_1 will execute the region of o_2 which is covered by o_1 will automatically deleted giving right output. Here, $o_2' \rightarrow o_1'$ is equal to execution of o_1' only, since o_2' is null and also $o_1' \leftarrow o_1$ from line 1 (see Fig. 13).

Conclusion

Operational transformation is the most optimistic method for concurrency and consistency control in muti-user groupware systems.

ABTS is the best string handling OT algorithm. The swapDD function of ABTS is proposed to swap two deletions, but swapDD fails totally if there exist partial overlapping between two deletions. In addition, it fails if one deletion operation string is totally covered by other deletion operation string. In few other cases, also swapDD fails at boundary conditions.

We propose a new algorithm MGswapDD to swap two deletions. It is also based on ABT framework and support string handling. It considers and works well in splitting and overlapping of operations. It works well on all boundary conditions also. It is practically implemented in lab also covering all possible cases of swapping two deletions. It gives totally right result if either there exist partial overlapping between two deletions or if one deletion operation string is totally covered by other deletion operation string. Therefore, in brief, it has removed all faults of the existing swapDD and work well in all possible cases of swapping two deletions.

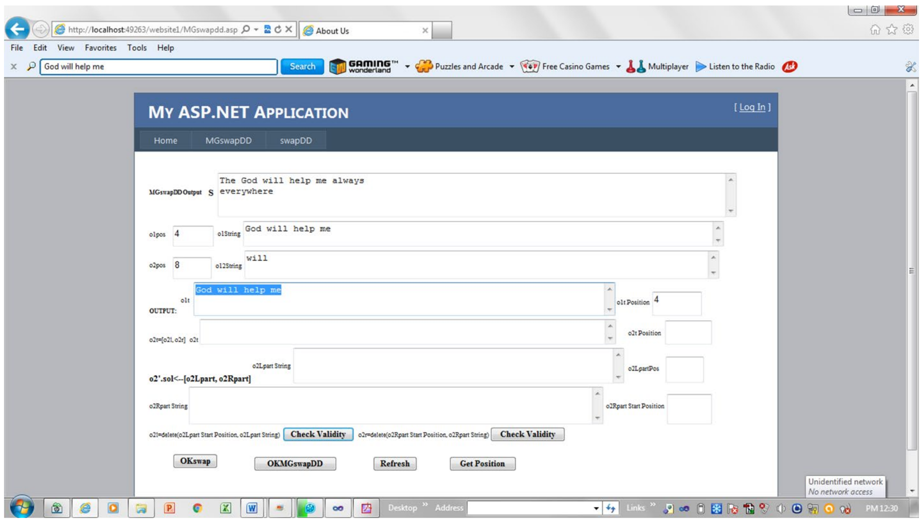

Fig. 13 Practical implementation of MGswapDD in lab for inputs of case 4 (right output)

Future work

Still there is scope to extend the support to other composite operations of string handling and char handling. Also there is need to support better data structures. A lot of work is done to reduce time complexity and space complexity. Still there is a scope to reduce time complexity and space complexity.

Authors' contributions

SK made substantial contributions to conception and design, acquisition of data, and analysis and interpretation of data; has been involved in drafting the manuscript or revising it critically for important intellectual content; has given final approval of the version to be published, and agrees to be accountable for all aspects of the work in ensuring that questions related to the accuracy or integrity of any part of the work are appropriately investigated and resolved. AK provided full guidance and support in all of the above works. Both authors read and approved the final manuscript.

Author details

[1] School of Engineering and Technology, Poornima University, IS-2027 To 2031 Ramchandrapura P.O. Vidhani Vatika Sitapura Extension, Jaipur, Rajasthan 303905, India. [2] Poornima College of Engineering, Jaipur, Rajasthan, India.

Acknowledgements

The paper is dedicated to my mother Meera Devi and daughter Harshita Kumawat.

Competing interests

The authors declare that they have no competing interests.

Received: 4 January 2016 Accepted: 13 May 2016
Published online: 10 June 2016

References

Ellis CA, Gibbs SJ (1989) Concurrency control in groupware systems. In ACM SIGMOD 1989 Preceedings, p 399–407, Portland Oregon

Li D, Li R (2006) An approach to ensuring consistency in peer-to-peer real-time group editors. Comput Support Co-op Work (2008) 17:553–611. doi:10.1007/s10606-005-9009

Li R, Li D (2007) A new operational transformation framework for real-time group editors. IEEE Trans Parallel Distrib Syst 18(3):307–319

Li D, Li R (2010) An admissibility-based operational transformation framework for collaborative editing systems. Comput Support Co-op Work 19:1–43. doi:10.1007/s10606-009-9103-1

Ressel M, Nitsche-Ruhland D, Gunzenha R (1996) An integrating, transformation-oriented approach to concurrency control and undo in group editors. Proc. ACM conf. computer supported cooperative work (CSCW 1996), p 288–297

Shao B, Li D, Gu N (2009) ABTS: A transformation-based consistency control algorithm for wide-area collaborative applications, collaborative computing: networking, applications and worksharing. CollaborateCom 2009. 5th International Conference on Nov. 2009 doi: 10.4108/ICST.COLLABORATECOM2009.8271. p1–10, 11–14

Shao B, Li D, Gu N (2010) An algorithm for selective undo of any operation in collaborative applications, in ACM
Suleiman M, Cart M, Ferrié J (1998) Concurrent operations in a distributed and mobile collaborative environment. Proceedings of the fourteenth international conference on data engineering, p 23–27
Sun C, Ellis C (1998) Operational transformation in real-time group editors: issues, algorithms, and achievements In ACM CSCW 1998, p 59–68
Sun C, Jia X, Zhang Y, Yang Y, Chen D (1998) Achieving convergence, causality-preservation, and intention preservation in real-time cooperative editing systems. ACM Trans Comput Hum Interact 5(1):63–108
Vidot N, Cart M, Ferrie J, Suleiman M (2000) Copies convergence in a distributed real-time collaborative environment. Proceedings of the 2000 ACM conference on computer supported cooperative work. ACM Press, New York, pp 171–180