Collaboration over Wiki Content

Luc André, Claudia-Lavinia Ignat, Gérald Oster
Université de Lorraine, INRIA Nancy - Grand Est, LORIA
54506 Vandœuvre-lès-Nancy, FRANCE
{luc.andre, claudia.ignat, gerald.oster}@loria.fr

ABSTRACT
Resolving conflicts is a key issue in collaborative editing. The difficulty of this task grows with the complexity of the document structure, and most of digital documents are not simple sequences of characters. This paper offers a new approach to handle conflict resolution automatically in a rich text document. This work uses Operational Transformation approach, and is based on a meaningful set of operations instead of primitive ones. This way, information about what the users precisely want to do is saved, and conflict resolution is more accurate.

ACM Classification Keywords
I.7.1 Document and Text Processing: Document and Text Editing; C.2.4 Computer-Communication Networks: Distributed Systems—*Distributed applications*; H.5.3 Information Interfaces and Presentation: Group and Organization Interfaces—*Collaborative computing, theory and models*

General Terms
Algorithms, Design, Human Factors

Author Keywords
Real-time editing, Wiki, Collaboration, CSCW

INTRODUCTION
The Web2.0 era is associated with the growing success of participatory collaborative tools for public users. Enterprises have integrated a part of these tools in their information systems to improve their productivity and to facilitate the emergence of a form of collective intelligence.

Among these tools, wikis are a good way to efficiently share a large amount of knowledge. Wikis are interlinked pages of rich text, which can be viewed and edited by every user. For instance, in a software development process, a wiki page can be dedicated to meetings, another to bug reports and a last to development progress. This way, everybody can keep an eye on when is the next meeting, who will attend, the minute of a previous meeting, what are the bugs found or fixed, and what work has been done on the project. But also edit the content without any restriction, thus allowing fast updates (wiki means fast indeed). All of this with a single tool - a wiki.

However, merging of parallel modifications on the same page is not automatically performed in most existing wiki systems such as Wikipedia. If two users are concurrently editing the same article, when they try to save their changes, the save of only one user succeeds and the other saves fail. The changes of the user whose save succeeds are published. The other users will be presented with two versions of the wiki page: the one that the user tried to publish and the last published version. Conflicts, i.e. concurrent changes on the same article, have to be manually resolved by users by re-typing or copying and pasting the changes they performed in the last version that was published. Moreover, users are not aware of other concurrent updates on the same article until they try to publish their own changes. The process of resolving conflicts might become very tedious and this can be critical when a deadline approaches and the document must be finalised.

A solution to this issue would be the real-time collaboration over wikis. Rather than requiring that users check periodically for updates, real-time collaboration is based on the principle of pushing modifications to users as soon as they are available. That is, updates are not submitted at the end of the edit on a wiki page, but each update is immediately sent at its generation, and integrated at its reception. Real-time collaboration is useful for brainstorming sessions, taking live meeting minutes, live discussion on a bug report or when editing an article when a deadline approaches.

Operational transformation approach [1, 6] has been identified as an appropriate approach to be used in real-time collaborative editing to maintain consistency of the copies of shared documents. In this approach local operations are executed immediately after they are generated and remote operations are transformed against the other operations. The transformations should be performed in such a manner that the intentions of the users are preserved and, at the end, the copies of the documents converge.

Wikis can be edited either as plain text documents with a special markup-based syntax or using a WYSIWIG (What You See Is What You Get) editor which allows content (text and graphics) of the wiki page to be displayed onscreen dur-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWCES'12, February 12, 2012, Seattle, Washington, USA.

ing editing in a form closely corresponding to the end result.

Regarding real-time collaborative editing for the first solution, any operational transformation approach with a set of transformations designed for plain text documents [2] could be applied. However, applying these algorithms for wikis cannot ensure that the special syntax will be preserved after integration of concurrent operations. This might also break the underlying hierarchical structure of the wiki.

Existing software such as GoogleDocs or CoWord [10] are WYSIWYG editors but they do not take advantage of the underlying structure of the document. Basically they see the document as a linear sequence of elements and high level operations are translated into primitive insert, delete and update operations on basic elements. This approach leads to simple algorithms, but semantics of user operations are lost. For instance, moving a block of text is obtained by deleting the block character by character followed by re-inserting each character of the block. If a user performs some modifications into a block of text that is concurrently moved, then these user changes are either completely lost or they are placed outside the context of the moved block of text. In this case user intention of modifying the paragraph is lost.

In this paper, we propose to use a traditional operational transformation approach for which we define operations with high-level semantic capturing user intentions when editing wiki content. We provide the required set of transformations for these operations.

This paper is organised as follows. First section presents existing approaches offering support to real-time collaboration over rich text documents. Second section presents our approach: our document model, our operations and the associated transformations. Third section discusses the benefits and the limitations of managing operations which captures high-level user intentions. Fourth presents our on-going validation in the context of the XWiki [1] system. Finally, last section concludes and presents our directions of future work.

RELATED WORK

CoWord [10] is a plug-in for Word using a strategy called Transparent Adaptation (TA), with Operational Transformation (OT) collaboration supporting technique. Basically, OT technique sends operations to other users when they are locally generated, and transforms received operations so they can be executed on a different state (than the state of the generation). TA technique converts every high-level operation to a sequence of primitive operations. OT technique is applied to this sequence of primitive operations. So TA is application-dependant, but not OT operations, allowing easy software updates : one just needs to turn new high-level operations into primitive ones. In CoWord, the set of primitive operations is Insert, Delete and Update. It is enough to create any Word (rich text) document, but the transformation of high-level operations into these ones erases all information about the original high-level operation. For instance, moving a character is transformed into one Delete operation followed by one Insert. Concurrently moving a sequence of ten characters while inserting a new character between the fourth and the fifth one will result in the ten characters moved (deleted then inserted), and the new single character inserted outside of its context. The expected result is to have the new character between the others, as the user intended to. Information about the move intention is discarded.

GoogleDocs [2] and Etherpad [3] work in a similar way. All edits are transformed into three basic types of changes: inserting text, deleting a range of text and applying styles to a range of text. Transformations are provided for all pairs of these types of changes. However, these operations are not enough for capturing user intention. For instance, the previous provided scenario leads to the deletion of the moved sequence of characters together with the new inserted character and the re-insertion of moved sequence of characters at the new position. The new inserted character disappears.

Docx2Go [4] is a framework for collaborative editing over XML documents on mobile devices where not all devices have the complete document. It adapts the Logoot [9] approach for XML documents. XML elements possess unique identifiers, the set of identifiers being ordered and dense. Docx2Go supports four types of operations at the element level: insert adds a new element at a specific location in the relative order; delete removes a specified element; update changes the internal contents of an element; and move changes the relative order of a specified element with respect to other elements. When one element is concurrently edited, the generated conflict can be resolved manually or automatically. In the case of a manual resolution the owner of the document is in charge of resolving the conflict. However, very often in collaborative editing multiple users edit the document and there is no explicit owner of the document. In the case of automatic resolution, when concurrent updates are performed on the same element, the element will be duplicated, each version of the element including the individual changes performed. However, if the element is a paragraph, the user will be provided with as many versions of the paragraph as the number of concurrent changes. The paper presents no awareness mechanism that would inform users about the different versions of the elements that were concurrently changed.

As we can see, there is no suitable mechanism that offers an automatic resolution of conflicts that reflects closely user intentions. We claim that an approach that preserves user intentions should contain a rich set of operations for which intentions are precisely specified. Combined effects of pairs of these operations have to be provided.

APPROACH

System architecture

Figure 1 gives an overview of the architecture of the wiki system. It consists of a wiki server that stores all wiki pages and a set of clients. Each time a user wants to edit a wiki

[1] http://www.xwiki.com/

[2] https://docs.google.com/

[3] http://etherpad.com/

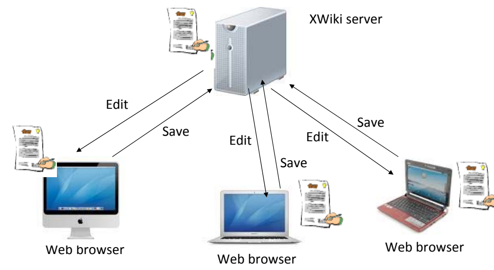

Figure 1. Architecture of the wiki system

page, a replica of the edited page is created at the client side by downloading its content from the server.

As soon as a client performs a change on his replica, this change is sent to the server that processes it by potentially performing some transformations and then sends it to the clients which at their turn might transform it before execution.

The proposed operational transformation approach is composed of an integration algorithm and a set of transformation functions. Any integration algorithm such as Jupiter [3], COT [7] or SOCT4 [8] can be used. In our implementation we used Jupiter algorithm. Transformation functions for the defined set of operations are proposed in the appendix of this paper and briefly described in the next subsections.

Document model and operations

Our approach uses one operation for one intention. To fully express the hierarchical organisation of the document, it is represented as a tree. The root is a "document" node, the level below is "paragraph" nodes, and then is the content. Content nodes are either text nodes, or style node with a unique child – a text node – and attributes representing the styles. Figure 2 gives an illustration of our document model.

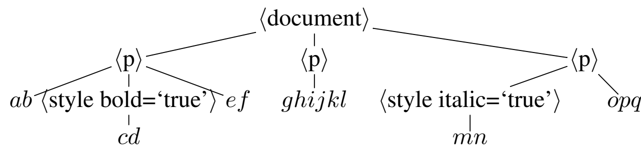

Figure 2. Document model.

On this document hierarchical model, we define the following operations:

- create a new paragraph with an empty text node.

  newP(int position, *int id*)

  position : index of the paragraph in the document

  id : unique site identifier that generated the paragraph

- merge two adjacent paragraphs (the children of the right paragraph are appended to the left one)

  mergeP(int position, *int leftchildren, int rightchildren*)

  position : index of the left paragraph in the document

  leftchildren : number of children in the left paragraph

  rightchildren : number of children in the right paragraph

- split one paragraph in two

  splitP(int position, int[] path, *int id, boolean splitleft*)

  position : split index in the text node of the paragraph to be split

  path : path to the text node where the split will occur

  id : unique site identifier

  splitleft : true if the split was generated at a position greater than 0 in the text node.

- move one paragraph

  moveP(int startposition, int endposition)

  startposition : index of the paragraph to be moved in the document

  endposition : index where the paragraph is moved

- insert one character in a text node

  insertText(int position, int[] path, *int id*)

  position : insertion index in the text node

  path : path to the text node

  id : unique site identifier

- delete one character in a text node

  deleteText(int position, int[] path)

  position : deletion index in the text node

  path : path to the text node

- add/delete a style to some text (bold, underlined, ...)

  textStyle(int start, int end, int[] path, String param, String value, *int id, boolean addStyle, boolean splitleft, boolean splitright*)

  start : start index of the style in a text node

  end : end index of the style in a text node

  path : path to the text node

  param : name of the style

  value : value of the style

  id : unique site identifier

  addStyle : true if the text node had no style at the generation

splitleft : true if start index is not zero

splitright : true if end index is not the length of the text node

- delete a subtree (keeping a tombstone)

  deleteTree(int[] path)

  path : path to the node to be deleted

Parameters presented in italic are not required to execute the operation, but they refer to required information for transformations. These parameters are updated during transformations. The document illustrated in Figure 2 can be created by executing the following operations: user1 creates a new paragraph, inserts “abcdef”, applies bold style to “cd”, while user2 creates another paragraph and inserts “ghijkl”, and while user3 creates another paragraph, inserts “mnopq” and applies italic style to “‘mn”.

We intend to extend these operations to handle headings, lists, tables, links and images. We do not need to create one new operation for every new intention. For instance, headings can be managed like paragraphs, but with “h1” nodes instead of “p” nodes. Links can be stored in a “style” node, with “link” as parameter, and “http://....” as value. However, this will require to update the presented transformation functions.

Overview of the proposed transformation functions

As we have eight operations, we must define sixty four transformations. Due to space limitation, we only provide a brief overview of our transformations. Interested reader can refer to the appendix for the full set of transformation functions.

When a newP operation is transformed, the insertion position is updated if some paragraphs were concurrently moved, merged, split or inserted. The parameter “id” is used to break some ties. The tricky case occurs when the new paragraph is inserted between two concurrently merged paragraphs. In this case it is appended after the merged paragraphs.

mergeP operations are transformed according to the same principles. However they are more tricky cases to deal with. If one paragraph was inserted between the merged ones, we need to first move the inserted one, then to merge the two other paragraphs. And if one of the merged was concurrently moved, we need to move the second before merging. leftchildren and rightchildren parameters respectively store the number of children of the left and the right paragraphs before the merge occurs. It is useful for instance to know in which child we have to insert a character in the case of a concurrent insertion.

splitP is more complex to handle, since it needs a whole path (not just a paragraph position) and a character position. The more difficult cases are those which involve an operation that modifies path and position. For instance, starting from document illustrated in Figure 2, transformation of splitP(2, 1/0) – split after “h” – with textStyle(1, 4, 1/0, underlined, true) – underline “hij” – leads to splitP(1, 1/1/0). For the sake of clarity, parameters presented in italic were omitted.

moveP operation works in the same way as newP and mergeP operations – with paragraph positions –. The tricky case appears when merge or move involving the same paragraphs concurrently occurred with a moveP operation. In such a case, the tie is broken using the end position parameter.

insertText operation is close to splitP operation, as it carries a position and a path parameters, but does not change the structure of the tree. To transform an insertion operation, the new path is computed according to concurrent newP, mergeP, splitP, moveP, textStyle operations and the new position is computed regarding concurrent splitP, insertText, deleteText, textStyle operations. The same principle is applied for deleteText operation except that a tie breaker parameter is no needed “id”, since two concurrent deletions of the same characters never conflict: the character is deleted anyway.

textStyle is the most complex operation to manage. It carries a path, two character positions, and can split a node into three others, while adding a child to one of them. For the sake of simplicity we only explain how transformation is performed when two concurrent textStyle operations occur. If they do not target the same paragraph, no transformation is performed. Else, if they do not target the same child, the path of the second textStyle operation has to be updated if the first one split its child into two or three children. Else, – they target the same child –, in this case, the two operations may overlap, so the transformation of one textStyle operation may result into two or three textStyle operations. It has also to check if the two styles are compatibles. If they are not, “id” is used as the tie breaker, but the discarded style is still applied (with the winner one as parameter) to apply the structure modifications. Other parameters – presented in italic – are, “addStyle” to store that a “style” node was created, “splitleft” to store that there were some characters in the child before “start” index, and “splitright” after “end” index. A generated textStyle(1,4...) indeed creates a node at the left of the child, while textStyle(0,3...) does not. So if a concurrent deleteText(0,...) turns textStyle(1,4...) into textStyle(0,3...) without any information, documents may diverge.

deleteTree operation is a tombstone-based deletion and is easy to transform – only path has to be updated –. However, it implies to write carefully some operations. For instance, if two paragraphs are merged while one is deleted, the expected result is to have the children of the deleted paragraph invisible, and the others visible. In this case the merge operation is not transformed, and the operation itself checks if the paragraph is a tombstone, and turns its children into tombstone too if needed, then move them for the merge. And the deleteTree operation is turned into as many deleteTree as the deleted paragraph has children. That is one reason why we need to store “leftchildren” and “rightchildren” as mergeP parameters.

DISCUSSIONS

Our approach captures user intentions into the design of the set of operations. However, the number of operations is

higher than in other existing approaches and therefore transformations are multiple and complex.

One can claim that chances that users edit at the same time the same part of a document leading to conflicts are not very high. Even if conflicts occur in the real-time collaboration their automatic resolution can be quickly noticed and an undesired resolution effect can be manually repaired.

But in an asynchronous collaboration where users can perform changes on their local copies of the document and then publish them at a later time conflicting situations become more critical. In this collaboration mode users are not informed about concurrent changes of other users. When users publish their changes and have to merge them with concurrent changes performed by other users a lot of conflicts might have to be resolved. A manual conflict resolution mechanism might be very tedious for a user. An automatic resolution mechanism must take into account user intentions because otherwise a non meaningful result might be obtained. It is in this asynchronous collaboration that our approach offers a powerful solution compared to existing approaches based on primitive operations. Consider the simple case when a user moves some text and keeps editing it, while another user moves it elsewhere and edits it as well. In most existing approaches the final result will be two modified pieces of duplicated text. These pieces of text are meant to be manually merged, so users have to notice it and to properly merge them. Our approach prevents this situation as it captures the semantics of move and modify operations. In this simple example no duplication is performed.

EVALUATION

Correctness of transformation functions

The transformation functions have to satisfy condition C_1 [5]. Being given two concurrent operations op_1 and op_2, the relation $op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$ must hold. The notation $op_1 \circ op_2$ denotes the sequence of operations containing op_1 followed by op_2. This property expresses an equivalence between two sequences of operations, i.e. when the sequences of operations op_1 followed by $T(op_2, op_1)$ and op_2 followed by $T(op_1, op_2)$ are applied on the same initial document state, the same document state is produced.

We have designed our transformation functions in order to respect C_1 condition. We are working on formally proving that C_1 is respected by using SPIKE theorem prover.

Implementation

We are currently integrating our approach in the XWiki system. XWiki is a java-based wiki that runs on a servlet container such as Tomcat or Jetty. The Jupiter approach required that the same algorithm is running on the server side and on the client side. In our implementation, the client side algorithm code is exactly the same as the server side algorithm code except that it is compiled using the GWT [4] from Java to Javascript.

[4]Google Web Toolkit http://http://code.google.com/webtoolkit/

We propose to show a demonstration of our wiki extended for real-time collaboration during the workshop.

CONCLUSION AND FUTURE WORK

This paper presented a new approach to create operations for OT technique in the field of collaborative editing. This approach is based on high level operations instead of primitive operations, and is designed to capture user intentions. We now have to enlarge our set of operations and to prove the correctness of our transformations.Then our next investigation will be to do real case studies to measure if the added value is worth the cost -in operations and transformations- of this approach.

REFERENCES

1. C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. *SIGMOD Record : Proceedings of the ACM SIGMOD Conference on the Management of Data - SIGMOD '89*, 18(2):399–407, May 1989.

2. A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving Correctness of Transformation Functions in Real-Time Groupware. In *Proceedings of the European Conference on Computer-Supported Cooperative Work - ECSCW 2003*, pages 277–293, Helsinki, Finland, September 2003. Kluwer Academic Publishers.

3. D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the 8th Annual ACM Symposium on User interface and Software Technology - UIST '95*, pages 111–120, Pittsburgh, PA, USA, November 1995. ACM Press.

4. K. P. Puttaswamy, C. C. Marshall, V. Ramasubramanian, P. Stuedi, D. B. Terry, and T. Wobber. Docx2go: collaborative editing of fidelity reduced documents on mobile devices. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 345–356, New York, NY, USA, 2010. ACM.

5. M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the International Conference on Data Engineering - ICDE'98*, pages 36–45, Orlando, Florida, USA, February 1998. IEEE Computer Society.

6. 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 Transactions on Computer-Human Interaction*, 5(1):63–108, March 1998.

7. D. Sun and C. Sun. Context-based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 20:1454–1470, 2009.

8. N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies Convergence in a Distributed Real-Time Collaborative Environment. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2000*, pages 171–180, Philadelphia, PA, USA, December 2000. ACM Press.

9. M. P. Weiss S., Urso P. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Proceeding of International Conference on Distributed Computing Systems (ICDCS)*, pages 404–412, June 2009.

10. S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen. Leveraging single-user applications for multi-user collaboration: the coword approach. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 162–171, New York, NY, USA, 2004. ACM.

APPENDIX

Utility functions

```
// path comparison functions
diff (int [] t1, int [] t2) :-
  // return true if paths are different
  if (t2.length != t1.length) {
    return true;
  }
  for (int i = 0; i < t1.length; i++) {
    if (t1[i] != t2[i]) {
      return true;
    }
  }
  return false;


inf (int [] t1, int [] t2) :-
  // return true if path t1 is before path t2
  // Depth First Search
  int i = 0;
  while (i < t1.length && i < t2.length) {
    if (t1[i] < t2[i]) {
      return true;
    }
    if (t1[i] > t2[i]) {
      return false;
    }
    i++;
  }
  return (t1.length < t2.length);


// path update functions
addP(int[] path, int nb) :-
  // add nb to the paragraph index of the path
  int [] tab = new int[path.length];
  tab[0] = path[0] + nb;
  for (int i = 1; i < path.length; i++) {
    tab[i] = path[i];
  }
  return tab;


setP(int [] path, int nb) :-
  // set to nb the paragraph index of the path
  int [] tab = new int[path.length];
  tab[0] = nb;
  for (int i = 1; i < path.length; i++) {
    tab[i] = path[i];
  }
  return tab;


addC(int[] path, int pos, int nb) :-
  // add nb to the index at depth pos of the path
  int [] tab = new int[path.length];
  for (int i = 0; i < path.length; i++) {
    tab[i] = path[i];
  }
  tab[pos] = tab[pos] + nb;
  return tab;


addLevel(int[] path) :-
  // add one level to the path
  int [] tab = new int[path.length + 1];
  for (int i = 0; i < path.length; i++) {
    tab[i] = path[i];
  }
  tab[tab.length - 1] = 0;
  return tab;
```

```
reference(int []  path,  int []  ref )  :−
   //  compute a new path, as if  ref  was the path 0/0/0
   //  used when a transformation involves TreeSplitP
   int []  tab = addP(path, 1);
   int  i  = 1;
   while (ref [ i ]  == tab[ i ])  {
      tab[ i ]  = 0;
      i++;
   }
   tab[ i ]  = tab[ i ]  − ref[ i ];
   return tab;
```

Transformations

```
//  OT for  Id
T(Operation o1, Id  o2) :−
   return o1;

T(Id  o1,  Operation o2) :−
   return o1;

//  OT for  composite
//  Composite is a  list   of  operations
//  needed when a transformation returns
//  more than one operation
T(Operation op, TreeCompositeOperation opl) :−
   Iterator <Operation> it = opl. list . iterator  ();
   Operation to = op;
   while ( it .hasNext()) {
      to  = T(to,   it .next ());
   }
   return to;

T(TreeCompositeOperationOperation opl, Operation op) :−
   ArrayList<Operation> l = new ArrayList<Operation>();
   Iterator <Operation> it = opl. list . iterator  ();
   Operation to = op;
   Operation n =  it .next ();
   l .add(T(n, to ));
   while ( it .hasNext()) {
      to  = T(to,  n);
      n =  it .next ();
      l .add(T(n, to ));
   }
   return new TreeCompositeOperation(l);

//  OT for  InsertText
T(insertText  o1,  insertText  o2) :−
   if  ( diff (o1.t ,  o2.t ))  {
      return o1;
   }
   if  (o1.p < o2.p) {
      return o1;
   }
   if  (o1.p == o2.p && o1.site < o2.site)  {
      return o1;
   }
   return new insertText(o1.p + 1,  o1.t ,  o1.text ,  o1.site );

T(deleteText o1,  insertText  o2) :−
   if  ( diff (o1.t ,  o2.t ))  {
      return o1;
   }
   if  (o1.p < o2.p) {
      return o1;
   }
   return new deleteText(o1.p + 1, o1.t );

T(newP o1, insertText o2) :−
   return o1;
```

```
T(mergeP o1, insertText o2) :−
  return o1;

T(splitP o1, insertText o2) :−
  if ( diff (o1.t, o2.t )) {
    return o1;
  }
  if (o1.p <= o2.p) {
    return o1;
  }
  return new splitP(o1.p + 1, o1.t, o1.site, o1. splitleft );

T(deleteTree o1, insertText o2) :−
  return o1;

T(textStyle o1, insertText o2) :−
  // style applied if the inserted character is at a bound of the style range
  if ( diff (o1.path, o2.t )) {
    return o1;
  }
  if (o1.start >= o2.p) {
    return new textStyle(o1.path, o1.start == o2.p ? o1.start : o1.start + 1, o1.end + 1, o1.param, o1.value, o1.siteId,
                         o1.addStyle, o1. splitleft , o1. splitright );
  }
  if (o1.end < o2.p) {
    return new textStyle(o1.path, o1.start, o1.end, o1.param, o1.value, o1.siteId, o1.addStyle, o1. splitleft , o1. splitright );
  }
  return new textStyle(o1.path, o1.start, o1.end + 1, o1.param, o1.value, o1.siteId, o1.addStyle, o1. splitleft , o1. splitright );

T(moveP o1, insertText o2) :−
  return o1;

// OT for DeleteText
T(insertText o1, deleteText o2) :−
  if ( diff (o1.t, o2.t )) {
    return o1;
  }
  if (o1.p <= o2.p) {
    return o1;
  }
  return new insertText(o1.p − 1, o1.t, o1.text, o1.site );

T(deleteText o1, deleteText o2) :−
  if ( diff (o1.t, o2.t )) {
    return o1;
  }
  if (o1.p == o2.p) {
    return new Id();
  }
  if (o1.p < o2.p) {
    return o1;
  }
  return new deleteText(o1.p − 1, o1.t);

T(newP o1, deleteText o2) :−
  return o1;

T(mergeP o1, deleteText o2) :−
  return o1;

T(splitP o1, deleteText o2) :−
  if ( diff (o1.t, o2.t )) {
    return o1;
  }
  if (o1.p <= o2.p) {
    return o1;
  }
  return new splitP(o1.p − 1, o1.t, o1.site, o1. splitleft );

T(deleteTree o1, deleteText o2) :−
  return o1;

T(textStyle o1, deleteText o2) :−
```

```
  if ( diff (o1.path, o2.t)) {
    return o1;
  }
  if (o1.start > o2.p) {
    return new textStyle(o1.path, o1.start − 1, o1.end − 1, o1.param, o1.value, o1.siteId, o1.addStyle, o1. splitleft , o1. splitright );
  }
  if (o1.end <= o2.p) {
    return o1;
  }
  return new textStyle(o1.path, o1.start, o1.end − 1, o1.param, o1.value, o1.siteId, o1.addStyle, o1. splitleft , o1. splitright );

T(moveP o1, deleteText o2) :−
  return o1;

// OT for NewP
T(insertText o1, newP o2) :−
  if (o1.t[0] < o2.p) {
    return o1;
  }
  int [] tab = addP(o1.t, 1);
  return new insertText(o1.p, tab, o1.text, o1.site );

T(deleteText o1, newP o2) :−
  if (o1.t[0] < o2.p) {
    return o1;
  }
  int [] tab = addP(o1.t, 1);
  return new deleteText(o1.p, tab);

T(newP o1, newP o2) :−
  if (o1.p < o2.p) {
    return o1;
  }
  if (o1.p == o2.p && o1.s < o2.s) {
    return o1;
  }
  return new newP(o1.p + 1, o1.s);

T(mergeP o1, newP o2) :−
  if (o1.p == o2.p) {
    return new TreeCompositeOperation(new moveP(o2.p, o2.p + 2), o1);
  }
  if (o1.p < o2.p) {
    return o1;
  }
  return new mergeP(o1.p + 1, o1.leftchildren, o1. rightchildren );

T(splitP o1, newP o2) :−
  if (o1.t[0] < o2.p) {
    return o1;
  }
  int [] tab = addP(o1.t, 1);
  return new splitP(o1.p, tab, o1.site, o1. splitleft );

T(deleteTree o1, newP o2) :−
  if (o1.t[0] < o2.p) {
    return o1;
  }
  int [] tab = addP(o1.t, 1);
  return new deleteTree(tab);

T(textStyle o1, newP o2) :−
  if (o1.path[0] < o2.p) {
    return o1;
  }
  int [] tab = addP(o1.path, 1);
  return new textStyle(tab, o1.start, o1.end, o1.param, o1.value, o1.siteId, o1.addStyle, o1. splitleft , o1. splitright );

T(moveP o1, newP o2) :−
  // if move at the same index of insert,
  // move index before insert index
  int sp = o1.sp;
  int ep = o1.ep;
```

```
  if (o2.p <= sp) {
    sp++;
  }
  if (o2.p < ep) {
    ep++;
  }
  return new moveP(sp, ep);

// OT for SplitP
T(insertText o1, splitP o2) :-
  if (! diff (o1.t, o2.t)) {
    if (o1.p < o2.p) {
      return o1;
    }
    int [] tab = new int[o1.t.length];
    tab[0] = o1.t[0] + 1;
    return new insertText(o1.p - o2.p - 1, tab, o1.text, o1.site);
  }
  if ( inf (o1.t, o2.t)) {
    return o1;
  }
  if (o1.t[0] > o2.t[0]) {
    int [] tab = addP(o1.t, 1);
    return new insertText(o1.p, tab, o1.text, o1.site);
  }
  int [] tab = reference(o1.t, o2.t);
  return new insertText(o1.p, tab, o1.text, o1.site);

T(deleteText o1, splitP o2) :-
  if (! diff (o1.t, o2.t)) {
    if (o1.p < o2.p) {
      return o1;
    }
    int [] tab = new int[o1.t.length];
    tab[0] = o1.t[0] + 1;
    return new deleteText(o1.p - o2.p, tab);
  }
  if ( inf (o1.t, o2.t)) {
    return o1;
  }
  if (o1.t[0] > o2.t[0]) {
    int [] tab = addP(o1.t, 1);
    return new deleteText(o1.p, tab);
  }
  int [] tab = reference(o1.t, o2.t);
  return new deleteText(o1.p, tab);

T(newP o1, splitP o2) :-
  if (o1.p <= o2.t[0]) {
    return o1;
  }
  return new newP(o1.p + 1, o1.s);

T(mergeP o1, splitP o2) :-
  if (o1.p == o2.t[0]) {
    return new mergeP(o1.p, o1.leftchildren, o2.t[1] + (o2. splitleft ? 1 : 0));
  }
  if (o1.p == o2.t[0] + 1) {
    return new mergeP(o1.p + 1, o1.leftchildren - o2.t[1], o1. rightchildren );
  }
  if (o1.p < o2.t[0]) {
    return o1;
  }
  return new mergeP(o1.p + 1, o1.leftchildren, o1. rightchildren );

T(splitP o1, splitP o2) :-
  if (! diff (o1.t, o2.t)) {
    if (o1.p < o2.p) {
      return o1;
    }
    if (o1.p == o2.p) {
      return new Id();
    }
```

```
    int [] tab = new int[o1.t.length];
    tab[0] = o1.t[0] + 1;
    return new splitP(o1.p − o2.p, tab, o1.site, o1.splitleft);
  }
  if (inf(o1.t, o2.t)) {
    return o1;
  }
  if (o1.t[0] > o2.t[0]) {
    int [] tab = addP(o1.t, 1);
    return new splitP(o1.p, tab, o1.site, o1.splitleft);
  }
  int [] tab = reference(o1.t, o2.t);
  return new splitP(o1.p, tab, o1.site, o1.splitleft);

T(deleteTree o1, splitP o2) :−
  if (inf(o1.t, o2.t)) {
    return o1;
  }
  if (! diff(o1.t, o2.t)) {
    if (o2.p == 0) {
      int [] tab = new int[o1.t.length];
      tab[0] = o1.t[0] + 1;
      return new deleteTree(tab);
    }
    return o1;
  }
  if (o1.t[0] > o2.t[0]) {
    int [] tab = addP(o1.t, 1);
    return new deleteTree(tab);
  }
  int [] tab = reference(o1.t, o2.t);
  return new deleteTree(tab);

T(textStyle o1, splitP o2) :−
  if (inf(o1.path, o2.t)) {
    return o1;
  }
  if (! diff(o1.path, o2.t)) {
    if (o1.end < o2.p) {
      return o1;
    }
    if (o1.end == o2.p) {
      return new textStyle(o1.path, o1.start, o1.end, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, false);
    }
    int [] tab = new int[o1.path.length];
    tab[0] = o1.path[0] + 1;
    if (o1.start > o2.p) {
      return new textStyle(tab, o1.start − o2.p, o1.end − o2.p, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, o1.splitright);
    }
    if (o1.start == o2.p) {
      return new textStyle(tab, o1.start − o2.p, o1.end − o2.p, o1.param, o1.value, o1.siteId, o1.addStyle, false, o1.splitright);
    }
    // paragraph between start and end
    return new TreeCompositeOperation(new textStyle(o1.path, o1.start, o2.p, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, false),
         new textStyle(tab, 0, o1.end − o2.p, o1.param, o1.value, o1.siteId, o1.addStyle, false, o1.splitright));
  }
  // inf(o2.t,o1.path)
  if (o1.path[0] > o2.t[0]) {
    int [] tab = addP(o1.path, 1);
    return new textStyle(tab, o1.start, o1.end, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, o1.splitright);
  }
  int [] tab = reference(o1.path, o2.t);
  return new textStyle(tab, o1.start, o1.end, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, o1.splitright);

T(moveP o1, splitP o2) :−
  if (o1.sp == o2.t[0]) {
    if (o1.sp < o1.ep) {
      return new TreeCompositeOperation(new moveP(o1.sp, o1.ep + 1), new moveP(o1.sp, o1.ep + 1));
    } else {
      return new TreeCompositeOperation(new moveP(o1.sp, o1.ep), new moveP(o1.sp + 1, o1.ep + 1));
    }
  }
  int sp = o1.sp;
```

```
  int ep = o1.ep;
  if (o2.t[0] < sp) {
    sp++;
  }
  if (o2.t[0] < ep) {
    ep++;
  }
  return new moveP(sp, ep);

// OT for MergeP
T(insertText o1, mergeP o2) :-
  if (o1.t[0] < o2.p) {
    return o1;
  }
  int[] tab = addP(o1.t, -1);
  if (o1.t[0] > o2.p) {
    return new insertText(o1.p, tab, o1.text, o1.site);
  }
  tab[1] = o1.t[1] + o2.leftchildren;
  return new insertText(o1.p, tab, o1.text, o1.site);

T(deleteText o1, mergeP o2) :-
  if (o1.t[0] < o2.p) {
    return o1;
  }
  int[] tab = addP(o1.t, -1);
  if (o1.t[0] > o2.p) {
    return new deleteText(o1.p, tab);
  }
  tab[1] = o1.t[1] + o2.leftchildren;
  return new deleteText(o1.p, tab);

T(newP o1, mergeP o2) :-
  if (o1.p <= o2.p) {
    return o1;
  }
  return new newP(o1.p - 1, o1.s);

T(mergeP o1, mergeP o2) :-
  if (o1.p == o2.p) {
    return new Id();
  }
  if (o1.p == o2.p + 1) {
    return new mergeP(o1.p - 1, o1.leftchildren + o2.leftchildren, o1.rightchildren);
  }
  if (o1.p == o2.p - 1) {
    return new mergeP(o1.p, o1.leftchildren, o1.rightchildren + o2.rightchildren);
  }
  if (o1.p < o2.p) {
    return o1;
  }
  return new mergeP(o1.p - 1, o1.leftchildren, o1.rightchildren);

T(splitP o1, mergeP o2) :-
  if (o1.t[0] < o2.p) {
    return o1;
  }
  int[] tab = addP(o1.t, -1);
  if (o1.t[0] > o2.p) {
    return new splitP(o1.p, tab, o1.site, o1.splitleft);
  }
  tab[1] = tab[1] + o2.leftchildren;
  return new splitP(o1.p, tab, o1.site, o1.splitleft);

T(deleteTree o1, mergeP o2) :-
  if (o1.t.length == 1 && o1.t[0] == o2.p - 1) {
    // merge within a paragraph that will be deleted
    ArrayList<Operation> list = new ArrayList<Operation>();
    for (int i = 0; i < o2.leftchildren; i++) {
      int[] tab = new int[2];
      tab[0] = o1.t[0];
      tab[1] = i;
      list.add(new deleteTree(tab));
```

```
    }
    return new TreeCompositeOperation(list);
  }
  if (o1.t[0] < o2.p) {
    return o1;
  }
  if (o1.t[0] > o2.p) {
    return new deleteTree(addP(o1.t, −1));
  }
  // same paragraph
  if (o1.t.length == 1) { // merge within a paragraph to be deleted
    ArrayList<Operation> list = new ArrayList<Operation>();
    for (int i = 0; i < o2.rightchildren; i++) {
      int[] tab = new int[2];
      tab[0] = o1.t[0] − 1;
      tab[1] = i + o2.leftchildren;
      list.add(new deleteTree(tab));
    }
    return new TreeCompositeOperation(list);
  }
  int[] tab = addP(o1.t, −1);
  tab[1] = tab[1] + o2.leftchildren;
  return new deleteTree(tab);

T(textStyle o1, mergeP o2) :−
  if (o1.path[0] < o2.p) {
    return o1;
  }
  int[] tab = addP(o1.path, −1);
  if (o1.path[0] > o2.p) {
    return new textStyle(tab, o1.start, o1.end, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, o1.splitright);
  }
  tab[1] = o1.path[1] + o2.leftchildren;
  return new textStyle(tab, o1.start, o1.end, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, o1.splitright);

T(moveP o1, mergeP o2) :−
  int sp = o1.sp;
  int ep = o1.ep;
  if (sp == o2.p) { // move of the right merged paragraph
    if (ep == sp − 1) {
      // if move just before left merged paragraph,
      // cancel move and keep merge
      return new Id();
    }
    // else move the merged paragraph
    if (ep > sp) {
      ep−−;
    }
    return new moveP(sp − 1, ep);
  }
  if (sp == o2.p − 1) { // move of the left merged paragraph
    if (ep == sp + 2) {
      // if move just after right merged paragraph,
      // cancel move and keep merge
      return new Id();
    }
    // else move the merged paragraph
    if (ep > sp) {
      ep−−;
    }
    return new moveP(sp, ep);
  }
  if (o2.p < sp) {
    sp−−;
  }
  if (o2.p < ep) {
    ep−−;
    // if move end between merged paragraphs,
    // move after the merged paragraphs
  }
  return new moveP(sp, ep);

// OT for DeleteTree
```

```
T(Operation o1, deleteTree o2) :-
  return o1;

// OT for Style
T(insertText o1, textStyle o2) :-
  if (o1.t[0] != o2.path[0]) {
    return o1;
  }
  if (o1.t[1] < o2.path[1]) {
    return o1;
  }
  if (o1.t[1] == o2.path[1]) { // same path
    if (o1.p < o2.start) {
      return o1;
    }
    if (o1.p == o2.start || o1.p <= o2.end) {
      int[] tab = addC(o1.t, 1, o2.splitleft ? 1 : 0);
      if (o2.addStyle) {
        tab = addLevel(tab);
      }
      return new insertText(o1.p - o2.start, tab, o1.text, o1.site);
    }
    int[] tab = addC(o1.t, 1, o2.splitleft ? 2 : 1);
    if (o2.addStyle) {
      tab = addLevel(tab);
    }
    return new insertText(o1.p - o2.end, tab, o1.text, o1.site);
  }
  // o1.t[1]>o2.path[1]
  int d = 0; // shift
  if (o2.splitleft) {
    d++;
  }
  if (o2.splitright) {
    d++;
  }
  int[] tab = addC(o1.t, 1, d);
  return new insertText(o1.p, tab, o1.text, o1.site);

T(deleteText o1, textStyle o2) :-
  if (o1.t[0] != o2.path[0]) {
    return o1;
  }
  if (o1.t[1] < o2.path[1]) {
    return o1;
  }
  if (o1.t[1] == o2.path[1]) { // same path
    if (o1.p < o2.start) {
      return o1;
    }
    if (o1.p == o2.start || o1.p < o2.end) {
      int[] tab = addC(o1.t, 1, o2.splitleft ? 1 : 0);
      if (o2.addStyle) {
        tab = addLevel(tab);
      }
      return new deleteText(o1.p - o2.start, tab);
    }
    int[] tab = addC(o1.t, 1, o2.splitleft ? 2 : 1);
    if (o2.addStyle) {
      tab = addLevel(tab);
    }
    return new deleteText(o1.p - o2.end, tab);
  }
  // o1.t[1]>o2.path[1]
  int d = 0; // shift
  if (o2.splitleft) {
    d++;
  }
  if (o2.splitright) {
    d++;
  }
  int[] tab = addC(o1.t, 1, d);
  return new deleteText(o1.p, tab);
```

```
T(newP o1, textStyle o2) :−
  return o1;

T(mergeP o1, textStyle o2) :−
  if (o1.p == o2.path[0]) {
    int d = 0;
    if (o2. splitleft ) {
      d++;
    }
    if (o2. splitright ) {
      d++;
    }
    return new mergeP(o1.p, o1.leftchildren, o1. rightchildren + d);
  }
  if (o1.p == o2.path[0] + 1) {
    int d = 0;
    if (o2. splitleft ) {
      d++;
    }
    if (o2. splitright ) {
      d++;
    }
    return new mergeP(o1.p, o1.leftchildren + d, o1. rightchildren );
  }
  return o1;

T(splitP o1, textStyle o2) :−
  if (o1.t [0] != o2.path[0]) {
    return o1;
  }
  if (o1.t [1] < o2.path[1]) {
    return o1;
  }
  if (o1.t [1] == o2.path[1]) { // same path
    if (o1.p < o2.start) {
      return o1;
    }
    if (o1.p == o2.start) {
      int [] tab = addC(o1.t, 1, o2. splitleft ? 1 : 0);
      if (o2.addStyle) {
        tab = addLevel(tab);
      }
      return new splitP(0, tab, o1.site , false);
    }
    if (o1.p < o2.end) {
      int [] tab = addC(o1.t, 1, o2. splitleft ? 1 : 0);
      if (o2.addStyle) {
        tab = addLevel(tab);
      }
      return new splitP(o1.p − o2.start, tab, o1.site , o1. splitleft );
    }
    int [] tab = addC(o1.t, 1, o2. splitleft ? 2 : 1);
    if (o1.p == o2.end) {
      return new splitP(0, tab, o1.site , false);
    }
    return new splitP(o1.p − o2.end, tab, o1.site , o1. splitleft );
  }
  // o1.t[1]>o2.path[1]
  int d = 0; // shift
  if (o2. splitleft ) {
    d++;
  }
  if (o2. splitright ) {
    d++;
  }
  int [] tab = addC(o1.t, 1, d);
  return new splitP(o1.p, tab, o1.site , o1. splitleft );

T(deleteTree o1, textStyle o2) :−
  if (o1.t [0] != o2.path[0]) {
    return o1;
  }
```

```
  if (o1.t.length == 1) {
    return o1;
  }
  if (o1.t[1] < o2.path[1]) {
    return o1;
  }
  int d = 0; // shift
  if (o2.splitleft) {
    d++;
  }
  if (o2.splitright) {
    d++;
  }
  if (o1.t[1] > o2.path[1]) {
    int[] tab = addC(o1.t, 1, d);
    return new deleteTree(tab);
  }
  ArrayList<Operation> list = new ArrayList<Operation>();
  for (int i = 0; i <= d; i++) {
    int[] tab = addC(o1.t, 1, i);
    list.add(new deleteTree(tab));
  }
  if (list.size() == 1) {
    return list.get(0);
  }
  return new TreeCompositeOperation(list);

T(textStyle o1, textStyle o2) :-
  if (o1.path[0] != o2.path[0]) {
    return o1;
  }
  if (o1.path[1] < o2.path[1]) {
    return o1;
  }
  int d = 0; // shift
  if (o2.splitleft) {
    d++;
  }
  if (o2.splitright) {
    d++;
  }
  if (o1.path[1] > o2.path[1]) {
    int[] tab = addC(o1.path, 1, d);
    return new textStyle(tab, o1.start, o1.end, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, o1.splitright);
  }
  // o1.path=o2.path
  ArrayList<Operation> list = new ArrayList<Operation>();
  if (o1.start < o2.start) {
    if (o1.end <= o2.start) { // style1 before style2
      return new textStyle(o1.path, o1.start, o1.end, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, o1.end == o2.start ? false : true);
    }
    list.add(new textStyle(o1.path, o1.start, o2.start, o1.param, o1.value, o1.siteId, o1.addStyle, o1.splitleft, false));
    if (o1.end >= o2.end) { // style2 into style1
      if (!(o1.param.equals(o2.param)) || o1.siteId < o2.siteId) {
        // no conflict between s1 and s2, or s1 win
        list.add(new textStyle(!o2.addStyle ? addC(o1.path, 1, o1.splitleft ? 2 : 1) : addLevel(addC(o1.path, 1, o1.splitleft ? 2 : 1)),
                               0, o2.end - o2.start, o1.param, o1.value, o1.siteId, false, false, false));
      } else { // apply o2 style to split the String
        list.add(new textStyle(!o2.addStyle ? addC(o1.path, 1, o1.splitleft ? 2 : 1) : addLevel(addC(o1.path, 1, o1.splitleft ? 2 : 1)),
                               0, o2.end - o2.start, o1.param, o2.value, o1.siteId, false, false, false));
      }
      if (o1.end != o2.end) {
        list.add(new textStyle(addC(o1.path, 1, o1.splitleft ? 3 : 2), 0, o1.end - o2.end, o1.param, o1.value, o1.siteId,
                               o1.addStyle, false, o1.splitright));
      }
      return new TreeCompositeOperation(list);
    }
    // o1.end<o2.end && o1.end>o2.start
    if (!(o1.param.equals(o2.param)) || o1.siteId < o2.siteId) {
      // no conflict between s1 and s2, or s1 win
      list.add(new textStyle(!o2.addStyle ? addC(o1.path, 1, o1.splitleft ? 2 : 1) : addLevel(addC(o1.path, 1, o1.splitleft ? 2 : 1)),
                             0, o1.end - o2.start, o1.param, o1.value, o1.siteId, false, false, true));
    } else { // apply o2 style to split the String
```

```
list .add(new textStyle(!o2.addStyle ? addC(o1.path, 1, o1. splitleft ? 2 : 1) : addLevel(addC(o1.path, 1, o1. splitleft ? 2 : 1)),
                        0, o1.end − o2.start, o1.param, o2.value, o1.siteId , false, false, true ));
  }
  return new TreeCompositeOperation(list);
}
if (o1.start == o2.start ) {
  if (!( o1.param.equals(o2.param)) || o1.siteId < o2.siteId) {
    // no conflict between s1 and s2, or s1 win
    list .add(new textStyle(!o2.addStyle ? addC(o1.path, 1, o1. splitleft ? 1 : 0) : addLevel(addC(o1.path, 1, o1. splitleft ? 1 : 0)),
                        0, (o1.end >= o2.end) ? o2.end − o2.start : o1.end − o1.start, o1.param, o1.value, o1.siteId ,
                        false, false, o1.end <= o2.end ? false : true));
  } else { // apply o2 style to split the String
    list .add(new textStyle(!o2.addStyle ? addC(o1.path, 1, o1.start == 0 ? 0 : 1) : addLevel(addC(o1.path, 1, o1.start == 0 ? 0 : 1)),
                        0, (o1.end >= o2.end) ? o2.end − o2.start : o1.end − o1.start, o1.param, o2.value, o1.siteId ,
                        false, false, o1.end <= o2.end ? false : true));
  }
  if (o1.end > o2.end) {
    list .add(new textStyle(addC(o1.path, 1, o1. splitleft ? 2 : 1), 0, o1.end − o2.end, o1.param, o1.value, o1.siteId, o1.addStyle, false, o1. splitright ));
  }
  if ( list .size () == 1) {
    return list .get (0);
  }
  return new TreeCompositeOperation(list);
}
// o1.start>o2.start
if (o1.start >= o2.end) {// style1 after style2
  return new textStyle(addC(o1.path, 1, o2. splitleft ? 2 : 1), o1.start − o2.end, o1.end − o2.end, o1.param, o1.value, o1.siteId,
                    o1.addStyle, o1.start == o2.end ? false : true, o1. splitright );
}
if (o1.end <= o2.end) {// style1 between style 2
  if (!( o1.param.equals(o2.param)) || o1.siteId < o2.siteId) {
    // no conflict between s1 and s2, or s1 win
    return new textStyle(!o2.addStyle ? addC(o1.path, 1, o2. splitleft ? 1 : 0) : addLevel(addC(o1.path, 1, o2. splitleft ? 1 : 0)),
                        o1.start − o2.start, o1.end − o2.start, o1.param, o1.value, o1.siteId , false, true, o1.end == o2.end ? false : true);
  } else { // apply o2 style to split the String
    return new textStyle(!o2.addStyle ? addC(o1.path, 1, o2. splitleft ? 1 : 0) : addLevel(addC(o1.path, 1, o2. splitleft ? 1 : 0)),
                        o1.start − o2.start, o1.end − o2.start, o1.param, o2.value, o1.siteId , false, true, o1.end == o2.end ? false : true);
  }
}
// last case : s2<s1<e2<e1
if (!( o1.param.equals(o2.param)) || o1.siteId < o2.siteId) {
  // no conflict between s1 and s2, or s1 win
  list .add(new textStyle(!o2.addStyle ? addC(o1.path, 1, o2. splitleft ? 1 : 0) : addLevel(addC(o1.path, 1, o2. splitleft ? 1 : 0)),
                        o1.start − o2.start, o2.end − o2.start, o1.param, o1.value, o1.siteId , false, true, false ));
} else { // apply o2 style to split the String
  list .add(new textStyle(!o2.addStyle ? addC(o1.path, 1, o2. splitleft ? 1 : 0) : addLevel(addC(o1.path, 1, o2. splitleft ? 1 : 0)),
                        o1.start − o2.start, o2.end − o2.start, o1.param, o2.value, o1.siteId , false, true, false ));
}
list .add(new textStyle(addC(o1.path, 1, o2. splitleft ? 3 : 2), 0, o1.end − o2.end, o1.param, o1.value, o1.siteId, o1.addStyle, false, o1. splitright ));
return new TreeCompositeOperation(list);

T(moveP o1, textStyle o2) :−
  return o1;

// OT for moveP
T(insertText o1, moveP o2) :−
  if (o1.t [0] == o2.sp) {
    if (o2.sp >= o2.ep) {
      return new insertText(o1.p, setP(o1.t , o2.ep), o1.text , o1.site );
    } else {
      return new insertText(o1.p, setP(o1.t , o2.ep − 1), o1.text , o1.site );
    }
  }
  if (o1.t [0] < o2.sp) {
    if (o1.t [0] < o2.ep) {
      return o1;
    } else {
      return new insertText(o1.p, addP(o1.t, 1), o1.text , o1.site );
    }
  }
  // o1.t[0]>o2.sp
  if (o1.t [0] < o2.ep) {
    return new insertText(o1.p, addP(o1.t, −1), o1.text , o1.site );
```

```
  } else {
    return o1;
  }

T(deleteText o1, moveP o2) :−
  if (o1.t [0] == o2.sp) {
    if (o2.sp >= o2.ep) {
      return new deleteText(o1.p, setP(o1.t, o2.ep));
    } else {
      return new deleteText(o1.p, setP(o1.t, o2.ep − 1));
    }
  }
  if (o1.t [0] < o2.sp) {
    if (o1.t [0] < o2.ep) {
      return o1;
    } else {
      return new deleteText(o1.p, addP(o1.t, 1));
    }
  }
  // o1.t[0]>o2.sp
  if (o1.t [0] < o2.ep) {
    return new deleteText(o1.p, addP(o1.t, −1));
  } else {
    return o1;
  }

T(newP o1, moveP o2) :−
  // if move and create target the same location, then move is put before creation
  if (o2.sp < o1.p) {
    if (o2.ep <= o1.p) {
      return o1;
    }
    return new newP(o1.p − 1, o1.s);
  }
  // o2.sp>=o1.p
  if (o2.ep <= o1.p) {
    return new newP(o1.p + 1, o1.s);
  }
  return o1;

T(mergeP o1, moveP o2) :−
  // cancel move if
  // move of the right merged paragraph before the left,
  // move of the left merged paragraph after the right.
  // if move between the merged, move it after the merge
  if (o2.sp == o1.p) { // move of the right merged paragraph
    if (o2.ep == o2.sp − 1) {
      // if move of the right merged paragraph just before the left ,
      // cancel move, keep merge.
      return new TreeCompositeOperation(o2, o1);
    }
    // else move the left and merge
    if (o2.ep > o2.sp) {
      return new TreeCompositeOperation(new moveP(o2.sp − 1, o2.ep − 1), new mergeP(o2.ep − 1, o1.leftchildren, o1.rightchildren));
    } else {
      return new TreeCompositeOperation(o2, new mergeP(o2.ep + 1, o1.leftchildren, o1.rightchildren));
    }
  }
  if (o2.sp == o1.p − 1) { // move of the left merged paragraph
    if (o2.ep == o2.sp + 2) {
      // if move of the left merged paragraph just after the right ,
      // cancel move, keep merge.
      return new TreeCompositeOperation(o2, o1);
    }
    // else move the right and merge
    if (o2.ep > o2.sp) {
      return new TreeCompositeOperation(o2, new mergeP(o2.ep, o1.leftchildren, o1.rightchildren));
    } else {
      return new TreeCompositeOperation(new moveP(o2.sp + 1, o2.ep + 1), new mergeP(o2.ep + 1, o1.leftchildren, o1.rightchildren));
    }
  }
  if (o1.p < o2.sp) {
    if (o1.p == o2.ep) {
```

```
        // if move end between the merged paragraphs,
        // move after the merge
        return new TreeCompositeOperation(new moveP(o2.ep, o2.ep + 2), o1);
      }
      if (o1.p < o2.ep) {
        return o1;
      }
      // o1.p>o2.ep
      return new mergeP(o1.p + 1, o1.leftchildren, o1. rightchildren );
    }
    // o1.p>o2.sp
    if (o1.p == o2.ep) {
        // if move end between the merged paragraphs,
        // move after the merge
      return new TreeCompositeOperation(new moveP(o2.ep − 1, o2.ep + 1), new mergeP(o1.p − 1, o1.leftchildren, o1.rightchildren));
    }
    if (o1.p < o2.ep) {
      return new mergeP(o1.p − 1, o1.leftchildren, o1.rightchildren );
    }
    // o1.p>o2.ep
    return o1;

T(splitP o1, moveP o2) :−
    if (o1.t [0] == o2.sp) {
      if (o2.sp >= o2.ep) {
        return new splitP(o1.p, setP(o1.t, o2.ep), o1.site , o1. splitleft );
      } else {
        return new splitP(o1.p, setP(o1.t, o2.ep − 1), o1.site , o1. splitleft );
      }
    }
    if (o1.t [0] < o2.sp) {
      if (o1.t [0] < o2.ep) {
        return o1;
      } else {
        return new splitP(o1.p, addP(o1.t, 1), o1.site , o1. splitleft );
      }
    }
    // o1.t[0]>o2.sp
    if (o1.t [0] < o2.ep) {
      return new splitP(o1.p, addP(o1.t, −1), o1.site , o1. splitleft );
    } else {
      return o1;
    }

T(deleteTree o1, moveP o2) :−
    if (o1.t [0] == o2.sp) {
      if (o2.sp >= o2.ep) {
        return new deleteTree(setP(o1.t, o2.ep));
      } else {
        return new deleteTree(setP(o1.t, o2.ep − 1));
      }
    }
    if (o1.t [0] < o2.sp) {
      if (o1.t [0] < o2.ep) {
        return o1;
      } else {
        return new deleteTree(addP(o1.t, 1));
      }
    }
    // o1.t[0]>o2.sp
    if (o1.t [0] < o2.ep) {
      return new deleteTree(addP(o1.t, −1));
    } else {
      return o1;
    }

T(textStyle o1, moveP o2) :−
    if (o1.path[0] == o2.sp) {
      if (o2.sp >= o2.ep) {
        return new textStyle(setP(o1.path, o2.ep), o1.start , o1.end, o1.param, o1.value, o1.siteId , o1.addStyle, o1. splitleft , o1. splitright );
      } else {
        return new textStyle(setP(o1.path, o2.ep − 1), o1.start , o1.end, o1.param, o1.value, o1.siteId , o1.addStyle, o1. splitleft , o1. splitright );
      }
```

```
  }
  if (o1.path[0] < o2.sp) {
    if (o1.path[0] < o2.ep) {
      return o1;
    } else {
      return new textStyle(addP(o1.path, 1), o1.start , o1.end, o1.param, o1.value, o1.siteId , o1.addStyle, o1. splitleft , o1. splitright );
    }
  }
  // o1.path[0]>o2.sp
  if (o1.path[0] < o2.ep) {
    return new textStyle(addP(o1.path, 1), o1.start , o1.end, o1.param, o1.value, o1.siteId , o1.addStyle, o1. splitleft , o1. splitright );
  } else {
    return o1;
  }

T(moveP o1, moveP o2) :-
  if (o1.sp == o2.sp) {
    if (o1.ep == o2.ep) {
      return new Id();
    }
    if (o1.ep < o2.ep) {// tie : higher end win (o2)
      if (o1.sp < o1.ep) {
        return new moveP(o1.ep - 1, o2.ep);
      }
      if (o2.ep < o1.sp) {
        return new moveP(o1.ep, o2.ep + 1);
      }
      // o1.e1<sp<o2.ep
      return new moveP(o1.ep, o2.ep);
    }
    if (o1.ep > o2.ep) {// tie : higher end win (o1)
      return new Id();
    }
  }
  int sp = o1.sp;
  int ep = o1.ep;
  if (o2.sp < o1.sp) {
    sp--;
  }
  if (o2.sp < o1.ep) {
    ep--;
  }
  if (o2.ep <= o1.sp) {
    sp++;
  }
  if (o2.ep < o1.ep) {
    ep++;
  } else {
    if (o2.ep == o1.ep) {// tie : higher start is after lower one
      if (o1.sp > o2.sp) {
        ep++;
      }
    }
  }
  return new moveP(sp, ep);
```