# PIECEMEAL: A FORMAL COLLABORATIVE EDITING TECHNIQUE GUARANTEEING CORRECTNESS

Stijn Dekeyser
*University of Southern Queensland*
stijn.dekeyser@usq.edu.au

Jan Hidders
*Delft University of Technology*
a.j.h.hidders@tudelft.nl

**ABSTRACT**

While collaboration on documents has been supported for several decades by a variety of systems and tools, in recent months a renewed interest is apparent through the appearance of new collaborative editors and applications. Some of these distributed groupware systems are plug-ins for standalone word processors, while others have a purely web-based existence. Most exemplars of the new breed of systems are based on Operational Transformations, although some are using traditional version management tools and still others utilize document-level locking techniques.
All current techniques have their own drawbacks, creating opportunities for new methods. In this paper we present a novel collaborative technique for documents which is based on transactions, schedulers, conflicts and locks. It is not meant to replace existing techniques; rather, it can be used in specific situations where the alternatives are less attractive. While our approach is highly formal with an emphasis on proving desirable properties such as guaranteed correctness, the work is part of a university-industry linkage project which aims to fully implement our technique.

**KEYWORDS**

Collaborative editing, transactions and scheduling, locking, distributed computing, theory.

## 1. INTRODUCTION

While some lesser known approaches exist (e.g. Dekeyser et al, 2004 and Oster et al, 2007), collaboration on documents is typically done in two distinct ways. In the asynchronous setting collaborators typically check out a document from a central repository, make changes off-line over an extended time, and then check in their version of the document at which time they are typically asked to solve inconsistencies with the version currently stored in the repository. This type of collaboration has been possible since the introduction of the Source Code Control System (SCCS) in the early seventies of the previous century. Successors of SCCS have included RCS, CVS, and Subversion. The last of these is still in widespread use and for example provides the collaboration functionality of systems such as ICE (Sefton et al, 2006).

The second approach to collaboration, found in most modern CSCW (Computer-Supported Cooperative Work) systems, involves synchronous collaboration where editors are aware of other users' changes while working on their own content. This setting requires on-line communication and uses a replicated architecture: shared documents are replicated at local sites such that each works on their own local copy and changes are propagated to other users. Such systems are called *real-time*: the response for local operations is quick and the latency for remote operations is relatively low. Examples of real-time editors include CoWord (Xia et al, 2004), Google Docs, and Google Wave (which is more than just a real-time editor, but that is beyond the scope of this paper). The newest such systems call themselves *really* real-time: local operations are applied immediately, and remote operations are applied within seconds. EtherPad, recently acquired by Google, is an example of a really real-time editor.

## 1.1 Current Techniques

We briefly discuss existing collaboration techniques and highlight some of their strengths and drawbacks. Due to a lack of space we limit this discussion to the two dominant methods.

### 1.1.1 Version Control systems

As discussed above, Subversion is currently the most widely used versioning control system, but has its foundations in SCCS. The underlying collaboration technique is well understood and robust, although not grounded on a formal model or supported by a theory of correctness. The technique makes use of a so-called *diff* between two text documents, at the level of individual lines. At its most basic level, a diff is represented by lines that are marked as additions, lines that are marked as deletions and lines that are not marked and that are used as a context to help decide where a change was made in a document. This implicit notion of a context is one that we will formalize in our approach.

In contrast to techniques based on locking, the use of a diff cannot fully prevent conflicts between document versions. This is the main weakness of the version control approach for collaboration; users are asked to resolve conflicts if they occur. Other weaknesses include: the limitation to asynchronous communication and the lack of visual feedback on collaboration in the client. The main strength is in the technique's simplicity and that it does not require on-line connection and can synchronize documents that have been edited off-line over lengthy periods.

Compared to this approach, our proposed technique, dubbed *Piecemeal*, will maintain the simplicity (at least for clients), and seamlessly offer both real-time synchronous as well as off-line asynchronous operation.

### 1.1.2 Operational Transformations

Most real-time and real real-time collaborative editors make use of the Operational Transformation technique (Ellis and Gibbs, 1989) as it is a highly optimistic method that fundamentally allows any operation to proceed at the local copy and then relies on appropriate transformation of the operation at remote sites. The transformation of an operation typically involves modifying the position in the local document where the operation needs to be applied. Hence, as in the previous approach, the technique is strongly tied to position in a sequential list of characters.

The consistency model used by OT relies on three properties: (1) *convergence* which means that ultimately copies of shared documents will become identical when all operations have been applied everywhere; (2) *causality preservation* which, simplified, means that dependent operations are executed in the same order on all sites; and (3) *intention preservation* which means that the intention of an operation is maintained at all sites (Wang et al, 2002).

The main advantages of OT are: that it is highly suitable for real-time collaboration because its optimistic approach results in high speed, that it does not require locks nor a central server, and that under most circumstances it works well (Sun and Ellis, 1998). However, the drawbacks of OT are also significant: there is no guarantee for correctness and the consistency model is problematic (Sun and Sun, 2009), implementation of a correct transformation for operations is difficult and has in fact been proven to be faulty in many systems (Imine et al, 2006), supporting undo operations is problematic (Ressel et al, 1996; Ressel and Gunzenhauser, 1999), and there is no inherent support for merging of document versions produced in an off-line manner.

In comparison to OT, the proposed Piecemeal technique will seamlessly support both real-time as well as off-line operation, will formally guarantee correctness and show that its rules are as strict as they need to be but not stricter, and will enable simple, specification driven implementation. Piecemeal has the additional benefit that support for undo actions is comparatively straightforward. Measured against OT, our approach has a number of perceived drawbacks which we briefly address:

- **Central server**. While OT allows peers to communicate without a server, Piecemeal employs a scheduler that accepts or disallows operations. While this may be seen as a bottleneck, it is instructive to note that current OT-based really real-time editors are web-based: a central web server relays communication from client to client. Due to cloud-type implementation this setup does not lead to a bottleneck, and our approach would be comparable.
- **Stricter strategy**. Our approach uses locks to allow a scheduler to determine whether an operation can proceed or not. Such a strategy is by nature conservative and strict, even if applied by an optimistic scheduler, but leads to guaranteed correctness and is simpler to implement correctly.
- **Overhead**. Our server keeps track of locks, which is an overhead not incurred in OT. However, note that the number of locks in a collaborative editor is orders of magnitude less than in a database environment, as documents have relatively few concurrent editors and each editor

typically holds only a few locks at a time due to the nature of text processing. Furthermore, our approach allows the user to decide on the size of context, making it possible to reduce the number of (read) locks held. Similarly, the size of constituent pieces in a document (the granularity of locking) is a parameter in our approach.

In (Sun and Sosič 1999) the authors show that integrating locks in OT cannot solve any of the three problems of convergence, causality violation and intention violation. Crucially, what enables us to do so using locking (without the use of OT) is that we use a fundamentally different document model (where position has been substituted by unique identifiers for 'blocks') and our operations are dependent on a context.

# 2. THE THEORY OF PIECEMEAL

The focus of this paper lies firmly in providing a solid theoretical foundation for Piecemeal such that both correctness and necessity of the proposed locking technique can be proven. Due to space constraints we refer to (Dekeyser and Hidders 2010) for proofs and more details.

Our approach is inspired both by CVS and its successors, and traditional concurrency control theory for relational databases.

## 2.1 Documents and Operations

To simplify the final correctness proof, we start from a relatively simple model and subsequently refine it. At first, documents will be modeled as general graphs where the nodes represent blocks that have a unique identifier and contain data (text, markup, or both) and the edges represent the fact that a block immediately precedes another block. The delimiters of a block (and hence its size) can be set by the system prior to the document being shared for collaborative editing. Since blocks have identifiers, clients will need to be able to manage globally unique identifiers. Operations on such graphs are modeled as graph-manipulation operations that add and remove edges. To create a new block (e.g. a paragraph of text), it will need to be positioned in the document by creating edges that connect it to existing blocks the document. Later in the paper, when documents have been further restricted, this means that the new block will need a new incoming and outgoing edge and existing edges will need to be deleted

**Definition 1** (Instance and Operation). Given the set of blocks (or nodes) $\boldsymbol{N}$, an *instance* is a set $I \subseteq \boldsymbol{N} \times \boldsymbol{N}$. The set of all instances is $\boldsymbol{I}$. An *operation* is a tuple $o = (P, D, A)$ where $P, D, A \subseteq \boldsymbol{N} \times \boldsymbol{N}$ are respectively *pattern edges*, *deleted edges* and *added edges*, such that $D \cap A = \emptyset$.

Operations, through the $A$ and $D$ edges, encapsulate what needs to change in the document graph. The edges in $D$ are removed, and the edges in $A$ are added. The $P$ edges are the context in which these changes occur. They are the formalization of the *diff* tools' non-marked lines and will correspond to read locks in our concurrency model. It is important to note that client editors may determine the size (and even the location) of the context for every individual operation, although the set of edges in the pattern will need to be at least a subset of the set of edges that are to be deleted although we formally do not require this at this stage.

The semantics of an operation $o = (P, D, A)$ is defined as a partial function $o : \boldsymbol{I} \rightarrow \boldsymbol{I}$ such that $o(I) = (I - D) \cup A$ if $P \subseteq I$ and undefined otherwise. The concatenation of two such partial functions $o_1$ and $o_2$ is denoted as $o_1 \circ o_2$ and defined such that $(o_1 \circ o_2)(I) = o_1(o_2(I))$ if $o_1(o_2(I))$ is defined, and undefined otherwise.

There is a clear conflict between two operations if one removes the edges that the other one requires. Operation $o_1 = (P_1, D_1, A_1)$ is said to *disable* operation $o_2 = (P_2, D_2, A_2)$ if $P_2 \cap D_1 \neq \emptyset$.

**Theorem 1**. Operation $o_1$ disables $o_2$ iff $o_2 \circ o_1$ is undefined for all instances.

The proof is given in [TR].

**Corollary**. Two operations $o_1$ and $o_2$ are mutually disabling iff both $o_1 \circ o_2$ and $o_2 \circ o_1$ are undefined for all instances.

## 2.2 Graph Locking

Transaction-oriented concurrency control is usually based on the commutativity of operations that do not conflict, i.e., the fact that their order in a schedule can be changed without affecting the final outcome of the schedule. It is this notion that is used in concurrency control theory to show the correctness of a scheduler by demonstrating for example that under the absence of cycles in the conflict graph the schedule can be serialized, i.e., the operations can be moved around in the schedule without changing its semantics to an order where the operations of the transactions are not interleaved. The challenge here is to come up with a notion of conflict that is optimal in the sense that the scheduler will disallow as little as possible schedules (and therefore stop certain operations from executing) that are in fact serializable, and thereby allows the maximum amount of parallelism.
We therefore now proceed with defining conflicts and showing operation commutativity in the relatively simple case where documents are graphs as defined in Definition 1. In subsequent sections we will refine this model.

**Definition 2** (Conflicting operations). Two operations $o_1 = (P_1, D_1, A_1)$ and $o_2 = (P_2, D_2, A_2)$ are said to conflict if at least one of the following holds:

**C1.** $P_1 \cap D_2 \neq \emptyset$ **C2.** $P_2 \cap D_1 \neq \emptyset$
**C3.** $P_1 \cap A_2 \neq \emptyset$ **C4.** $P_2 \cap A_1 \neq \emptyset$
**C5.** $D_1 \cap A_2 \neq \emptyset$ **C6.** $D_2 \cap A_1 \neq \emptyset$

Note that mutually disabling operations are also conflicting operations. The following theorem shows that if we only consider not-mutually disabling operations it holds that non-conflicting operations indeed do commute and that all commuting operation are non-conflicting or mutually disabling, i.e., for those combinations of operations the defined of conflict is theoretically optimal.

**Theorem 2** (Commutativity). For all operations $o_1$ and $o_2$ that are not mutually disabling it holds that $o_1$ and $o_2$ do not conflict iff $o_1 \circ o_2 = o_2 \circ o_1$.
The proof is given in [TR].

The conflict rules also have to deal with the case were both $o_1 \circ o_2$ and $o_2 \circ o_1$ are never defined, in which case the operations commute even though they conflict. We can show that this is indeed the only exception:

**Proposition 1**. For all operations $o_1$ and $o_2$ it holds that $o_1 \circ o_2 = o_2 \circ o_1$ iff $o_1$ and $o_2$ do not conflict or are mutually disabling.

A basic notion in transaction theory is that of schedule which represents a sequence of operations which would or would not be allowed by a scheduler. Since in practice a scheduler will only allow operations that have a defined result we will try to syntactically characterize such schedules.

**Definition 3** (Schedules). A *schedule* is a non-empty sequence $S = \langle o_1, \ldots, o_n \rangle$ of operations. A schedule is said to be *sound* if there is an instance $I$ such that $(o_n \circ \ldots \circ o_1)(I)$ is defined. A schedule $S = \langle o_1, \ldots, o_n \rangle$ is said to be *well-defined* if it holds that for all two operations $o_i = (P_i, D_i, A_i)$ and $o_j = (P_j, D_j, A_j)$ in $S$ such that $i < j$ and edges $(v_1, v_2) \in D_i \cap P_j$ there is an operation $o_k = (P_k, D_k, A_k)$ such that $i < k < j$ and $(v_1, v_2) \in A_k$.

**Theorem 3**. A schedule is sound iff it is well-defined.
The proof is given in [TR].

Piecemeal is a conservative concurrency control technique but not necessarily a pessimistic one. A document server may be implemented using either a commit scheduler or a conflict scheduler. The former one would

be pessimistic in the sense that as soon as an operation arrives that causes a conflict, the scheduler would make the originating transaction wait and try again. However, a more optimistic conflict scheduler would allow the conflicting operation to proceed, and then ensure that subsequent operations do not generate a cycle in the so-called conflict graph. Next to the normal conflict graph we also define a restricted conflict graph where the conflicts between mutually disabling operations are ignored, i.e., this graph indicates exactly when operations commute or not.

**Definition 4** (Conflict Graphs). The *conflict graph* of a schedule $S = \langle o_1, \ldots, o_n \rangle$ is $G_S = (V, E)$ where $V = \{1, \ldots, n\}$ and $E$ contains the edge $(i,j)$ iff $i < j$ and $o_i$ and $o_j$ conflict. The *restricted* conflict graph of a schedule $S = \langle o_1, \ldots, o_n \rangle$ is $G^r_S = (V, E)$ where $V = \{1, \ldots, n\}$ and $E$ contains the edge $(i,j)$ iff $i < j$ and $o_i$ and $o_j$ conflict and are not mutually disabling.

It can then be shown that the restricted graph is in terms of paths equivalent to the non-restricted graph.

**Theorem 4**. For every sound schedule $S$ and edge $(i,j)$ in $G_S$ there is a path from $i$ to $j$ in $G^r_S$.

It follows that an optimistic scheduler which bases its decisions on the presence of cycles, can use the simpler unrestricted conflict graph without disallowing serializable schedules and would therefore still be theoretically optimal.

## 2.3 From Graph over Cyclic Instance to Document Locking

As an intermediate step towards restricting instances to documents we first consider cyclic instances that consist of one or more disjoint simple cycles. Note that we can model a document as a single cycle with one special edge that connects the end node with the begin node and cannot be removed.

In the following, the set of nodes in a set of edges $X$ is denoted as $N_X$. The set of incoming and outgoing edges of a node $v$ in a set of edges $X$ is denoted as $in_X(v)$ and $out_X(v)$, respectively. The indegree and outdegree of a node $v$ in $X$ are denoted as $|in_X(v)|$ and $|out_X(v)|$, respectively.

**Definition 5** (Cyclic Instance). An instance $I$ is said to be *cyclic* if it is finite and for every node $v$ in $N_I$ it holds that $|in_I(v)| = |out_I(v)| = 1$.

We now define what a cyclically sound operation is. Intuitively this is an operation where at least for one instance all additions are real additions, i.e., the edges are not in the instance, the deletions are real deletions, and the nodes not appearing in the pattern or the delete set are completely new. These restrictions are chosen such that they represent what might be expected of a correct operation that is specified by the user.

**Definition 6** (Cyclically Sound Operation). An operation $o = (P, D, A)$ is said to be *cyclically sound* if there is at least one cyclic instance $I$ such that $I \cap A = \emptyset$, $D \subseteq I$, $N_I \cap (N_A - N_{P \cup D}) = \emptyset$ and $o(I)$ is defined and is a cyclic instance.

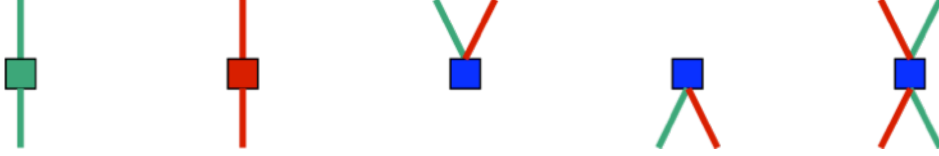
The problem with Definition 6 is that it suggests that soundness of an operation must be tested by checking the instance. Since the instance can be very large, it would be better if could determine soundness by looking only at the operation itself. The following definition is a syntactic approximation of a cyclically sound operation.

**Definition 7** (Cyclically Well-formed Operation). An operation $o = (P, D, A)$ is said to be *cyclically well-formed* if it holds that $P$, $D$ and $A$ are finite, for every node $v \in N_{P \cup D}$ it holds that $|in_{P \cup D}(v)| \leq 1$ and $|out_{P \cup D}(v)| \leq 1$, $P \cap A = \emptyset$ and for every node $v \in N_D \cup N_A$ one of the following holds:

**Cwf1**: $|in_A(v)| = 1$, $|out_A(v)| = 1$, $|in_{P \cup D}(v)| = 1$, $|out_{P \cup D}(v)| = 0$
**Cwf2**: $|in_A(v)| = 0$, $|out_A(v)| = 0$, $|in_D(v)| = 1$, $|out_D(v)| = 1$
**Cwf3**: $|in_A(v)| = 1$, $|out_A(v)| = 0$, $|in_D(v)| = 1$, $|out_D(v)| = 0$
**Cwf4**: $|in_A(v)| = 0$, $|out_A(v)| = 1$, $|in_D(v)| = 0$, $|out_D(v)| = 1$

**Cwf5**: $|in_A(v)| = 1$, $|out_A(v)| = 1$, $|in_D(v)| = 1$, $|out_D(v)| = 1$

The following theorem states that the syntactical notion of cyclically soundness coincides with the syntactical notion of cyclically well-formedness.



**Theorem 5**. An operation is cyclically sound iff it is cyclically well-formed.

The preceding theorem states that a cyclically well-formed expression might return a correct result. It can be shown that under the given restrictions for the input instance it in fact must return a correct result.

**Theorem 6**. If an operation $o = (P, D, A)$ is cyclically well-formed and $I$ a cyclic instance such that $D \subseteq I$, $A \cap I = \emptyset$, $N_I \cap (N_A - N_{P \cup D}) = \emptyset$ and $o(I)$ is defined, then $o(I)$ is a cyclic instance.

A reasonable restriction on operations is one where the delete set is a subset of the pattern; i.e., it is checked whether all edges that are to be deleted are indeed present in the instance. This restriction is useful in the remainder of the theory; hence, we give this property a name.

**Definition 8** (Well-guarded Operation). An operation $(P, D, A)$ is said to be well-guarded if $D \subseteq P$.

The following theorem states that if the result of a well-guarded cyclically well-formed operation is defined and the new nodes are indeed new nodes, then the result is a cyclic instance.

**Theorem 7**. For every cyclic instance $I$ and well-guarded cyclically well-formed operation $o = (P, D, A)$ such that $N_I \cap (N_A - N_{P \cup D}) = \emptyset$ and $o(I)$ is defined then $o(I)$ is a cyclic instance.

We are now ready to revisit the notion of conflicts. We redefine it so that it coincides with the notion of conflict in a locking protocol where read-locks are requested for edges in $P$ and write locks for edges in $A$ and $D$.

**Definition 9** (Lock-Conflicting Operations). Two operations $o_1 = (P_1, D_1, A_1)$ and o2 = $(P_2, D_2, A_2)$ are said to conflict if at least one of the following holds:

| | |
|---|---|
| **LC1.** $P_1 \cap D_2 \neq \emptyset$ | **LC2.** $P_2 \cap D_1 \neq \emptyset$ |
| **LC3.** $P_1 \cap A_2 \neq \emptyset$ | **LC4.** $P_2 \cap A_1 \neq \emptyset$ |
| **LC5.** $D_1 \cap A_2 \neq \emptyset$ | **LC6.** $D_2 \cap A_1 \neq \emptyset$ |
| **LC7.** $A_1 \cap A_2 \neq \emptyset$ | **LC8.** $D_1 \cap D_2 \neq \emptyset$ |

Note that the conditions LC1, …, LC6 are the same as C1, …, C6 from Definition 2 in Section 2.2 regarding conflicting operations in the setting of graph instances. Only the conditions LC7 and LC8 are new, indicating that when we look at cyclic instances, operations can cause a lock conflict when they both try to add or delete the same edges. This will then also imply an alternative notion of conflict graph.

**Definition 10** (Lock-conflict Graph). The lock-conflict graph of a schedule $S = \langle o_1, \ldots, o_n \rangle$ is $G^l_S = (V, E)$ where $V = \{1, \ldots, n\}$ and $E$ contains the edge $(i,j)$ iff $i < j$ and $o_i$ and $o_j$ lock-conflict.

It can then be shown that this lock-conflict graph is in terms of paths equivalent with the preceding conflict graphs if we restrict ourselves to cyclically sound schedules.

**Theorem 8**. For every cyclically sound schedule $S$ and edge $(i,j)$ in $G^l_S$ there is a path from $i$ to $j$ in $G_S$.

It therefore follows that although the notion of conflict is here more restrictive, the scheduler that uses it will still be theoretically optimal for cyclically sound schedules.

As already explained earlier, a document can be represented as a single cycle with a special irremovable edge that connects the last node with the first node. Unfortunately it is not possible to syntactically characterize the operations that preserve this property for instances as this is possible for cyclic instances. This is because the same operation might, given a single cycle as input, sometimes return a single cycle and sometimes several cycles. As an example, consider the operation $o = (P, D, A)$ with $P = D = \{(1,2), (3,4), (5,6)\}$ and $A = \{(1,6), (3,2), (5,4)\}$. Consider the instances $I_1$ = (1-2-3-4-5-6) and $I_2$ = (5-6-3-4-1-2). Then $o(I_1)$ results in the document (1-6) plus the separate cycles (2-3) and (4-5), but $o(I_2)$ = (5-4-1-6-3-2).

This of course does not mean that we cannot restrict the operations such that they always result in a document, but the above operation is generated by a fairly straightforward move that moves the blocks (4-1) from between the blocks 3 and 2 to between the blocks 5 and 6.

A consequence of this is that the document server will, in addition to the syntactical checks on the operation, still have to check whether the result of the operation still correctly represents a document. However, there are indexing and optimization techniques that can be used to avoid computing the complete result and checking whether it consists of a single cycle.

## 2.4 Working Offline – Operation Merging

The operations as defined in Section 2.1 are highly general in nature. We have seen that only some operations will be accepted and result in a document as defined in Section 2.3. Even so, they remain powerful and can encapsulate editing actions from diverse collaborative client editors on complex documents such as word processing files and serialized graphical content. Mapping client-side editing actions to Piecemeal operations can proceed on the basis of the needs of the editor and hence may depend on what sort of document is being edited.

However, the contents of individual operations need not only depend on editors' requirements. The generality of operations as allowed in our approach also makes it possible to vary the length or size of a single operation. Clients may decide to send minute changes to the server, resulting in a steady stream of tiny operations. Such operations may be seen by other clients (if they so wish) but only become 'permanent' upon committing a transaction. Alternatively, clients may decide to go offline and concatenate all changes into one sole operation. Such a 'merged' operation has the advantage of acting as a diff between the version of the document as last seen from the server, and the one that the offline client currently manages. In addition, the merged operation is not simply the union of the individual $P$, $A$, and $D$ sets; instead, it gives a condensed presentation of all changes since check-out occurred.

We present the intricate theory behind operation merging in (Dekeyser and Hidders 2010).

# 3. CONCLUSION

We have presented a novel conservative and strict technique for enabling collaborative editing of documents both on- and offline. Inspired by CVS's diff approach and its implicit context, we have given a formal foundation and proven that our technique guarantees correctness while limiting locking to the bare essential.
While requiring a central scheduler and incurring locking overhead, the approach offers an alternative to Operational Transformations in those situations where a correctness guarantee is critical, clients should be easy to implement (correctness being the responsibility of the document server), undo operations should be easy to deal with, and collaboration may occur in online and offline circumstances.

## ACKNOWLEDGEMENT

This work was partially funded by IWT, the Flemish agency for innovation by science and technology, under grant nr. 070171. In addition the authors wish to thank Xenit B.V. for their support and participation in the execution and implementation of this research.

## REFERENCES

Dekeyser, Stijn et al, 2004. A Transaction Model for XML Databases. *World Wide Web Journal*, 7:1, 29—57.

Dekeyser, Stijn and Hidders, Jan, 2010, A Notion of Serializability for Document Editing and Corresponding Optimal Locking Protocols, *Delft University of Technology technical report*, March, Available at: `http://www.st.ewi.tudelft.nl/~hidders/docs/piecemeal-tr.pdf`

Ellis, Clarence and Gibbs, S, 1989, Concurrency Control in Groupware Systems. *Proceedings of the ACM SIGMOD Conference on Management of Data*. Seattle, US, 399—407.

Imine, Abdessamad et al, 2006. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 351:2, 167—183.

Oster, Gerald et al, 2006, Data Consistency for P2P Collaborative Editing. *Proceedings of the 2006 ACM conference on Computer supported cooperative work (CSCW'06)*. Banff, Canada.

Oster, Gerald et al, 2007, Supporting Collaborative Writing of XML Documents. *Proceedings of the International Conference on Enterprise Information Systems (ICEIS'07)*. Madeira, Portugal.

Ressel, M. and Gunzenhauser, R, 1999, Reducing the Problems of Group Undo. *Proceedings of the ACM conference on Supporting Group Work (GROUP'99)*. Phoenix, US, 131—139.

Ressel, M. et al, 1996, An integrating, transformation-directed approach to concurrency control and undo in group editors. *Proceedings of the 1996 ACM conference on Computer supported cooperative work (CSCW'96)*. New York.

Sun, Chengzheng and Sosič, Rok, 1999, Optimal locking integrated with operational transformation in distributed real-time group editors. *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing (PODC'99)*. Atlanta, US, 43—52.

Sun, Chengzheng and Ellis, Clarence, 1998, Operational transformation in real-time group editors: issues, algorithms, and achievements. *Proceedings of the 1998 ACM conference on Computer supported cooperative work (CSCW'98)*. Seattle, US, 59—68.

Sun, David and Sun, Chengzheng, 2009. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Trans. Parallel Distrib. Systems*, 20:10, 1454-1470.

Wang, Xueyi et al, 2002, A New Consistency Model in Collaborative Editing Systems. *Proceedings of the 4th International workshop on Collaborative Editing*.

Xia, S. et al, 2004, Leveraging Single-User Applications for Multi-User Collaboration: the CoWord Approach. *Proceedings of the ACM conference on Computer supported cooperative work (CSCW'04)*. Chicago, US, 162—171.