Optimistic Synchronization of Cooperative XML Authoring using Tunable Transaction Boundaries

Francis Gropengießer, Kai-Uwe Sattler
TU Ilmenau
P.O. Box 100565, D-98684 Ilmenau, Germany
{*francis.gropengiesser|kus*}*@tu-ilmenau.de*

***Abstract*—Design applications, e.g., CAD or media production, often require multiple users to work cooperatively on shared data, e.g., XML documents. Using explicit transactions in such environments is difficult, because designers usually do not want to consider transactions or ACID. However, applying transactions in order to control visibility of changes or specify recovery units, is reasonable, but determining transaction boundaries must be transparent for the designer. For this reason we propose a novel approach for the automatic determination of transaction boundaries which considers the degree of cooperation designers want to achieve. Furthermore, we present an optimistic synchronization model based on the traditional backward oriented concurrency control (BOCC) algorithm, in order to synchronize the determined transactions in multi-user environments. It exploits the semantics of tree operations on XML data and enforces a correctness criterion weaker than serializability. As our evaluation shows, when multiple users work cooperatively on shared data, this model significantly reduces the number of transaction aborts in comparison to the traditional BOCC approach.**

***Keywords*-transaction, optimistic synchronization, boundaries, cooperation, XML, semantic tree operations**

I. Introduction

Nowadays, many applications, e.g., CAD or media production tools, operate on semi-structured, hierarchical data such as XML. In this paper, we focus on sound design based on the wave field synthesis technique [1]. It enables the creation of more realistic surround sounds for movies or concerts. In this context, the task of a sound designer is to animate static objects and to define their locations/movements as well as the characteristics of the surroundings, for example to give the listener the impression to be in a cave or in a concert hall. The produced meta information is stored as a scene graph in XML format.

The overall goal of our work is to enable multiple users to work cooperatively on the same XML data in a closely coupled client/server environment called workgroup [4]. We use transactional semantics in order to enable cooperative XML authoring. The reasons for choosing transactional semantics instead of just using version control systems like CVS or SVN is that transactions allow i) an exact controlling of visibility of changes and ii) a clear specification of recovery units in terms of failures and aborts.

Specifying transactions in design environments is not an easy task. In contrast to, e.g., bank applications, where predefined transactions exist, e.g., for withdrawal, in design environments, the transactions are constructed during the interaction of a designer with the system. Usually, designers are not interested in specifying transactions. Hence, this task must be performed transparently by the system. Thereby, several questions arise such as i) what are indivisible operation sequences and hence, ii) when to start/commit a transaction.

Another challenge is the synchronization of concurrent transactions in cooperative environments in order to guarantee consistent data. Using lock-based protocols, e.g., [3], [9], in situations where conflicts are relatively rare produces unnecessary overhead caused by acquiring and releasing locks. In order to solve this problem, several non-locking models have been proposed. Some examples are timestamp ordering algorithms [2], multiversion concurrency control [11], or backward oriented concurrency control (BOCC) [6]. The main problem of these solutions is that they enforce strict serializability, which hampers cooperation. A model that allows for cooperative processing of data is the operational transformation [8]. However, two problems exist with respect to this approach. First, it is difficult to always find the correct transformation function. Second, lost updates can occur which can only be solved with a special undo operation.

This paper considers the problems of automatically determining transaction boundaries and synchronizing the resulting transactions in cooperative design environments. It makes the following contributions:

- a novel approach which allows for an automatic determination of transaction boundaries dependent on the user specified degree of cooperation and
- a novel optimistic synchronization strategy based on the traditional BOCC approach [6]. It is suitable for cooperative environments, because it i) considers nested transactions and hence, supports an early visibility of changes and multi-directional information flows between different authors, and ii) reduces the number of transaction aborts by relaxing serializability and exploiting the semantics of tree operations on XML data.

Note that these approaches are not limited to XML but can also be applied to generally tree-structured data. Due to space limitations we mainly focus on the synchronization model and provide only an impression of the solution for the automatic determination of transaction boundaries.

II. Preliminaries

In order to understand the approaches proposed in this paper, we briefly sketch some basic concepts. We start with

a characterization of the data model before going into details on the tree operations and the transaction model. A detailed description of all concepts can be found here [4].

A. Data Model

We assume that XML data has a tree structure following the taDOM specification [5]. A tree consists of *nodes* with *node ids*, *node labels* and *node values*. Nodes are connected via *directed edges* denoting parent-child relationships. A tree can be ordered or unordered. If a tree is ordered, the ordering of nodes on the same level within the tree is important. Otherwise the ordering is negligible. We support both, ordered and unordered trees.

Figure 1 depicts an example for our data structure. We use it as a running example throughout the paper.

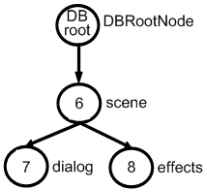

Figure 1. Example Database Tree

B. Operations

Processing of XML data (represented as a tree) is enabled by several semantic tree operations that we have defined so far.

In order to read a node, there exists a $strReadNode(n)$ (SRN) that retrieves the $node\ label$ of a node n and a $contReadNode(n)$ (CRN) that returns the $node\ value$ of a node n. The existence of an edge (p, n) between two nodes p and n can be tested by issuing the operation $readEdge(p, n)$ (RE).

The manipulation of XML data is possible with the help of some update operations. An $edit(n, new\ node\ value)$ (E) is used to assign a new $node\ value$ to a node n. The operation $insert(k, n, node\ label)$ (I) adds a new child node k with a given $node\ label$ to node n. Thereby, a new edge between n and k is inserted. The operation $move(n, m)$ (M) assigns node m as new $parent$ to node n so that the existing edge (p, n) between n's old parent p and n is removed and the new edge (m, n) is inserted. A $delete(n)$ (D), where n has no children, removes node n and the edge (p, n) between n and its parent p from the tree. Thereby, also the $node\ label$ and the $node\ value$ of node n are removed.

Note that the defined operations are more low-level than DOM or XPath operations. Hence, it is straightforward to map DOM/XPath operations onto the introduced operations. Furthermore, using such low-level operations instead of DOM/XPath operations has one big advantage: all models based on these operations can be easier adapted to general tree-structured data and are not limited to XML.

In addition to the above mentioned operations, we also defined high-level read and update operations, e.g., on subtrees. They are used for a more comfortable processing of XML data. Since they are implemented by the low-level operations defined above, we do not consider them in the following.

Table I shows the compatibility of the operations with respect to the nodes or the edge our operations consider. Thereby, $\surd$ states that the operations are fully compatible, $-$ that they are not compatible at all, and $+$ that they are only compatible if the tree they are performed on is considered unordered. Compatible operations are not in conflict and can be executed in parallel.

	SRN	CRN	RE	E	I	M	D
SRN	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$-$
CRN	$\surd$	$\surd$	$\surd$	$-$	$\surd$	$\surd$	$-$
RE	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$-$	$-$
E	$\surd$	$-$	$\surd$	$-$	$\surd$	$\surd$	$-$
I	$\surd$	$\surd$	$\surd$	$\surd$	$+$	$+$	$-$
M	$\surd$	$\surd$	$-$	$\surd$	$+$	$+$	$-$
D	$-$	$-$	$-$	$-$	$-$	$-$	$-$

Table I
COMPATIBILITY OF OPERATIONS

C. Transaction Model

The transaction model is based on dynamic actions [7] as well as on multi-level transactions [10] with a maximum transaction tree depth of four. Figure 2 shows the set of nested transactions resulting from a high-level read operation (reading a subtree rooted by the XML node with node id 6) and a low-level delete operation (deleting the node with id 8) on the XML tree of Figure 1. Figure 2 illustrates that the transaction model implements a top-down decomposition, which corresponds to: operation sequences (level 0, root transaction) $\rightarrow$ single operation sequence (level 1) $\rightarrow$ operation on subtree or node (level 2) $\rightarrow$ operation on a single node or edge (level 3). The root transaction is started by the designer and ends when he decides to finish his work. Within the root transaction several subtransactions at level 1, which represent operation sequences, are executed. Subtransactions at level 2 encapsulate the operations defined in the sequence at level 1. Subtransactions at level 3 represent the decomposition of operations on subtrees into operations on nodes and edges. The nesting enables visibility of changes after the completion of a subtransaction at level 1 and aborting of single operations on nodes or edges.

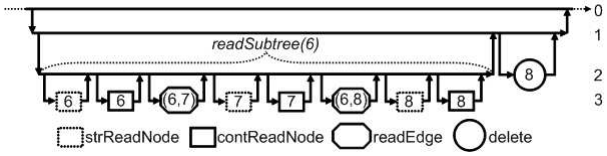

Figure 2. Example Transaction Structure

III. TRANSACTION BOUNDARIES

In the previous section we described the basic structure of our transaction model. Now, we clarify how operations performed by a designer are encapsulated into transaction boundaries.

Concerning transaction levels 0, 2, and 3, there is no need for further investigation. A root transaction is started when a designer starts his work and is completed when he decides to finish his work. Furthermore, the transaction levels 2 and 3 perform an automatic decomposition of an operation sequence assigned to a subtransaction at level 1 into single operations on nodes and edges.

The transaction level we have to consider is level 1. By deciding about transaction boundaries of a subtransaction at level 1 we control the visibility of changes. That means that the earlier we complete a subtransaction at level 1, the earlier changes are visible to other designers and the higher is the achieved degree of cooperation. Thereby, we only have to specify the complete points, because the beginning of a subtransaction at level 1 within a root transaction is triggered by the first user command after the completion of the previous subtransaction at level 1 (if one exists) within the same root transaction. So, the overall question is when to complete a subtransaction at level 1 without explicit interaction of a designer. The designer just issues operations and our system transparently encapsulates them into transactions.

In order to answer the question we specify a set of rules. A subtransaction T is completable, iff:

- Every data item that is updated within T was previously read within T. This is necessary in order to avoid so-called "blind writes".
- No high-level operation executed in T, e.g., the deletion of a subtree, will be split across two subtransactions at level 1 due to the completion of T. This is necessary, because in our use case high-level operations are treated as indivisible units during their execution.

If both rules are fulfilled, T can be completed. In this case, we have the highest possible degree of cooperation, because every update is immediately visible to other designers. However, we also want to support a variable degree of cooperation $0 \leq DC \leq 1$. This is a user specified threshold. By falling below this value, T finally has to be completed. In this way, a user can specify, how cooperative he wants to work with respect to other users working on the same data. Thereby, 1 means highly cooperative and hence, more cooperative than other users, and 0 means as cooperative as the other users.

The decision, whether T has to be completed (CN_T) is made according to the following equation:

$$CN_T = \begin{cases} \text{false}, & \text{single-user} \\ 1 - \frac{\sum_{u \in T} g_u}{k * g_{max}} \leq DC, & \text{multi-user, } |N| = 0 \\ 1 - \frac{\sum_{u \in T} g_u}{\frac{\sum_{S \in N} \sum_{u \in S} g_u}{|N|}} \leq DC, & \text{multi-user, } |N| > 0 \end{cases} . \quad (1)$$

Thereby, let u be an update operation within a subtransaction at level 1, N be the set of subtransactions at level 1 (containing update operations) of those users working on the same part of the data tree as the user that executes T, and S be a subtransaction within N. The variable g_u denotes the semantic importance of an update u. We assume updates on higher regions of the tree more important to other users than updates on the leaves of the tree. Thus, g_u equals the level of the data item within the tree, starting with level 1 at the leaves and level n at the root of the data tree. Hence, g_{max} equals n. However, other measurements for weighting update operations are conceivable.

The first case of the formula describes a single-user system. Here, cooperation is obviously not necessary. Thus, completing a subtransaction can be coupled with a save point desired by the user. The second case describes the startup phase of a multi-user working session. Here, value $k \in \mathbb{N}$ specifies the maximum number of semantically highly important update operations that may be executed without completion, when there have not yet occurred any update transaction on the tree the user executing T is working on. It is an empirical value and is set by the system administrator for a working session. The last case describes a running working session. Here, the updates of T are compared with the average updates of all transactions executed by users working on the same part of the tree.

The decision about completions of subtransactions at level 1 is made centrally by the transaction manager running on the server. The transaction manager monitors the states of all transactions and decides according to the given rules.

IV. The Extended BOCC Approach

As already mentioned in the introduction, our optimistic synchronization model is based on the traditional BOCC algorithm. Since we assume that the reader is familiar with the BOCC concept, we only sketch the basic idea. The traditional BOCC approach requires that a transaction consists of three phases – read, validate, and write. In the read phase all operations (read and write) are performed on local copies of the data items. Thereby, every transaction holds a $ReadSet$, which contains all read objects, and a $WriteSet$, which contains all manipulated objects. In the validation phase, a transaction is checked against all interleaving transactions which have already been successfully validated, whether or not its $ReadSet$ intersects any $WriteSet$ of the successfully validated transactions. If there is no intersection, serializability is preserved and all changes are made persistent in the write phase.

The traditional BOCC approach has several disadvantages with respect to cooperative environments. First, it does not consider nested transactions. If validation is performed not before the root transaction is completed, this leads to a late visibility of changes and prevents multi-directional information flows between different designers. Second, the

traditional BOCC algorithm enforces strict serializability and does not consider semantic tree operations. This leads to unnecessary transaction aborts.

We tailor the traditional BOCC approach in two ways that make it suitable for cooperative environments. First, we develop a correctness criterion that relaxes serializability. Second, we define the right point for validation in order to allow an early visibility of changes.

A. Validation Criterion

The basic assumption of our validation criterion is that an update operation on a data item depends only on the previously executed read operations on the same data item. It does not depend on read operations performed on other data items. Thereby, a data item denotes either the $node\ value$, the $node\ label$, or an $edge$. For our operations, the following dependencies exist: An $edit(n, new\ node\ value)$ of node n depends only on the preceding $contReadNode(n)$. Furthermore, an $insert(k, n, node\ label)$ of a new node k at node n depends only on the previously executed $strReadNode(n)$. A $delete(n)$ of node n depends only on the preceding $strReadNode(n)$, $contReadNode(n)$ and $readEdge(p, n)$ (edge between the parent p of n and n). A $move(n, m)$ of node n to node m depends only on the preceding $strReadNode(n)$, $strReadNode(m)$ and $readEdge(p, n)$. Since high-level operations are implemented by low-level operations, there are no further dependency specifications necessary.

Our goal is to assure that a data item is not changed by another transaction between a read and a consecutive update of this item. This is possible by checking if update operations of different transactions conflict. If a conflict exists, the read data item has been changed. Otherwise the update operations work on different items. In this case, we neglect conflicts between read and update operations and hence, relax serializability.

We illustrate this idea with an example. Consider the following schedule, which contains two operation sequences executed by transaction 1 and 2 on the example tree in Figure 1:

$$\begin{aligned} &SRN_1(6)SRN_1(7)SRN_1(8)CRN_1(6) \quad (2)\\ &CRN_1(7)CRN_1(8)RE_1(6,7)RE_1(6,8)SRN_2(6)\\ &SRN_2(7)SRN_2(8)CRN_2(6)CRN_2(7)CRN_2(8)\\ &RE_2(6,7)RE_2(6,8)E_2(7,''...'')E_1(8,''...'') \end{aligned}$$

This schedule contains two conflicts – one between $CRN_1(7)$ and $E_2(7,''...'')$ and one between $CRN_2(8)$ and $E_1(8,''...'')$. A corresponding conflict graph would contain a cycle and thus, the schedule is not serializable. However, we assume that $E_2(7,''...'')$ does not depend on the value retrieved by $CRN_2(8)$ and $E_1(8,''...'')$ does not depend on $CRN_1(7)$. Due to this assumption, this schedule becomes acceptable, because an inconsistent read does not lead to an update operation that might introduce an inconsistent database state. However, if transaction 1 would execute $E_1(7,''...'')$ instead of $E_1(8,''...'')$, the whole schedule becomes unacceptable, because the value retrieved by $CRN_1(7)$ is changed by $E_2(7,''...'')$. The succeeding execution of $E_1(7,''...'')$ would lead to an inconsistent database state.

The dependency assumption described above is applicable to many cases where XML documents are authored, because designers might issue a lot of read operations in order to navigate through the data tree. Most of the values are not further considered. If there is a need for more dependencies they have to be specified in addition. If, for example, $E_1(8,''...'')$ also depends on $CRN_1(7)$, this can easily be expressed by extending the edit operation to $E_1(8, 7,''...'')$. Then, the above schedule would become unacceptable.

Based on the above considerations, we formulate a validation criterion. For this purpose, we introduce the notion of $UpdateSets$. An $UpdateSet_{T_i}$ contains all update operations a transaction T_i has performed. Update operations can conflict but they need not, as it is shown in Table I. Let T_j be a transaction with transaction number $tn(T_j)$. T_j is successfully validated, iff $\forall T_i, tn(T_i) < tn(T_j)$ one of the following conditions holds:

1) T_i has completed its write phase before T_j starts its read phase.
2) $UpdateSet_{T_i}$ is not in conflict with $UpdateSet_{T_j}$ according to Table I.

The first case follows the traditional BOCC approach. Hence, if T_i and T_j are executed serially, T_j is validated successfully. The second case implies that validation is not successful if the transactions conflict in their $UpdateSets$. Otherwise, the transactions worked on different data items, serializability is preserved, and T_j is validated successfully.

By validating transactions following our validation criterion, we can guarantee that the database state produced by a schedule is equal to the result produced by a serial execution of the transactions involved in the schedule. However, this is only possible if all dependencies have been fully specified. Otherwise, it is possible that an update operation was performed in dependence on an inconsistent read operation.

By enforcing this kind of relaxed serializability, we can meet one of the most important requirements of cooperative environments, namely that all designers are always aware of the most current state of the project. This is possible because we only permit an interleaving of transactions if the data item a designer intends to update is not changed by another designer between the read and update operation.

Specifying semantically rich tree operations instead of using only simple read and write operations allows a fine-grained conflict/compatibility specification as it is shown in Table I. Multiple update operations can be executed in parallel. This number even increases if the data structure is assumed to be unordered. Reducing conflicts between update operations also reduces the number of conflicts between

transactions and thus the number of transaction aborts as our evaluation shows. Allowing highly parallel workflows is an important requirement of cooperative environments that we can meet this way.

B. Validation Time

In the previous subsection we answered the question how to validate. Now, we have to clarify when to validate in order to meet the requirements of cooperative environments. We argue that validation fits best right after a subtransaction at level 1 proceeds to the state $completed$. There are several reasons for this:

1) Validating root transactions is unacceptable, because they must be able to interleave randomly in order to allow multi-directional information flows between different designers. Furthermore, root transactions do not execute operations directly. Thus, there is no special need for validating them.
2) If validation of a subtransaction at level 1 fails, a relatively small amount of work progress is lost, depending on the desired degree of cooperation.
3) Changes of a designer are visible to other designers at an early stage, depending on the desired degree of cooperation.
4) Validating subtransactions at lower levels (2 and 3) would increase the validation overhead too much. In doing so, the benefit of optimistic synchronization compared to lock-based synchronization would be decreased.

Validation is performed in a centralized fashion at the server. The server is aware of all transactions that were executed and those that are still running. After a subtransaction at level 1 has been automatically completed according to equation 1, the $UpdateSet$ is propagated to the server. Then, validation is performed. If it is successful, all clients which are interested in the changes, i.e., working on the same data, are notified. Otherwise, the changes are discarded.

V. Evaluation

The main goal of our work is to enable cooperative processing of XML data. The proposed models for determination and synchronization of transactions accomplish this objective due to the following facts: Since validation is performed at the end of a subtransaction at level 1, there is no restriction (demand for serializability) with respect to the interleaving of root transactions. This means that designers can exchange information in arbitrary directions without restrictions. Furthermore, by specifying a high degree of cooperation according to equation 1 changes can be made visible at an early stage. This way, designers are always aware of the current state of the project.

Next, we show that by applying our extended BOCC algorithm in order to validate transactions, the number of transaction aborts can be reduced. Thereby, we also consider situations where many transactions contain update operations that are executed on shared data. In these situations, the probability that transactions conflict is very high. Although, optimistic synchronization models are not intended to be applied to such situations, we will show that even in such worst-case situations our extended approach can reduce the number of transaction aborts.

In order to show how the number of aborts is reduced by our extended algorithm compared to the traditional one, we first have to specify which of our tree operations is a read and which is a write operation. This is necessary because the traditional BOCC algorithm considers only simple read and write operations whereas we consider operation sequences consisting of semantic tree operations. Hence, we define a mapping from tree operations on simple read/write operations. The operations $strReadNode(n)$ and $contReadNode(n)$ are mapped on a $read(n)$, respectively. Furthermore, $readEdge(p, n)$ is mapped on a $read(p, n)$. An $edit(n, new\ node\ value)$ and an $insert(k, n, node\ label)$ are mapped on a $write(n)$, respectively. The operations $delete(n)$ and $move(n, m)$ are mapped on three write operations, $write(n)$, $write(p)$, and $write(p, n)$, whereby p is the parent node of n. Additionally, $move(n, m)$ is mapped on a $write(m)$ (m is the new parent of n) and write operations on every node and edge within the subtree rooted by n.

In order to measure our test results, we used a simulation where a randomly generated set of transactions was executed on a randomly generated XML document, represented as a tree that follows our data model specification of Section II-A. We tested trees with 500 nodes and variable tree depths, ranging from wider to deeper trees. The transactions comply with subtransactions at level 1 (Section II-C). Thereby, we used a high degree of cooperation, i.e., at most one high-level update operation within each transaction.

Before presenting experimental results, we first introduce the notion of *conflict rate*: Let W be a set of transactions. Then W can be divided into two disjoint sets of transactions C and F for which holds that every transaction in C conflicts with at least another transaction in C and every transaction in F conflicts with no transaction in W. The conflict rate is then defined as follows:

$$conflict\ rate = \frac{|C|}{|W|} * 100\% \tag{3}$$

The set of conflicting transactions (C) is determined based on the validation criterion of our new approach (conflicting $UpdateSets$). Thereby, we consider the tree, where the transactions are executed upon, as unordered.

We tested the following three different validation implementations: the traditional BOCC algorithm, our extended algorithm with unordered trees, and our extended algorithm with ordered trees. We measured the number of aborts with respect to five different conflict rates. The results shown in

the chart present averaged values resulting from 5 different test runs with different trees and different transaction sets.

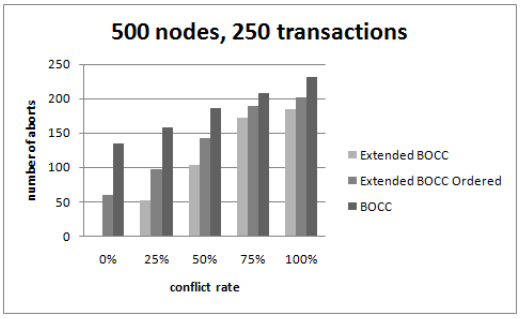

Figure 3. Based on Conflicts Between $UpdateSets$

Figure 3 presents the experimental results. Our new approach acts as a baseline and the other two algorithms are compared to this baseline. The maximum possible number of aborts when applying the extended BOCC algorithm on unordered trees equals the conflict rate. This number, however, does not necessarily have to be reached because the conflicting transactions could in fact be executed serially. In the chart we see that the extended BOCC algorithm on unordered trees has the fewest aborts followed by the extended BOCC on ordered trees and the traditional BOCC algorithm. This observation can be explained as follows:

- The extended BOCC algorithm on ordered trees considers more update operations to be incompatible than the algorithm on unordered trees. This is shown in Table I. Hence, more transactions are in conflict which might lead to more aborts.
- The traditional BOCC algorithm considers the $ReadSet_t$ of a transaction t that is validated. Since a transaction could contain a huge set of navigational read operations, the $ReadSet_t$ might become very large. Thus, there is a high potential that a transaction t is validated against writes on a data item that is in $ReadSet_t$. This issue might lead to more transaction aborts.

Summarizing, the results show that our approach leads to fewer transaction aborts than the traditional BOCC algorithm. However, this is only possible because we use a weaker correctness criterion for validation than the traditional BOCC approach. Our criterion ensures consistency if dependencies are fully specified. Furthermore, we consider tree operations and exploit their different semantics.

In this evaluation, we did not explicitly consider the overhead produced by our extended BOCC algorithm. However, it is obvious that it is equal or less than the one produced by the traditional approach because we do not have to hold a $ReadSet$ and a $WriteSet$ for each transaction but only an $UpdateSet$. This $UpdateSet$ is at most as big as the $WriteSet$ with respect to a transaction because of the mapping described above. Furthermore, we did not compare our optimistic approach with lock-based protocols regarding overhead or transaction throughput, due to the following reasons. First, it is obvious and common that optimistic approaches produce less overhead than lock-based protocols. Second, measuring transaction throughput makes little sense in our use case, because transactions are constructed during the user interaction and are not predefined. Hence, transaction throughput mainly depends on the user himself.

VI. Conclusion

In this paper, we introduced a novel approach for the automatic determination of transaction boundaries. This approach takes into account user specifications regarding cooperation requirements. Furthermore, we presented a novel approach for the synchronization of concurrent transactions in an optimistic fashion. Our approach is based on the traditional BOCC algorithm. The main difference to this approach is that validation follows a relaxed kind of serializability. By considering only conflicts between update operations and thereby exploiting the semantics of these tree operations, the number of aborts of concurrently executed transactions can be reduced – as our evaluation results show. Furthermore, the algorithm considers nested transactions and thus enables an early visibility of changes. It does not enforce serializability between root transactions and hence enables multi-directional information flows between different designers. Due to these features, our new approach is suitable for cooperative environments.

References

[1] A. J. Berkhout. A Holographic Approach to Acoustic Control. In *Journal Audio Eng. Soc.*, volume 36, pages 977–995. 1988.

[2] P. A. Bernstein and Newcomer. *Transaction Processing*. Morgan Kaufmann, 1997.

[3] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Readings in Database Systems (2nd ed.)*, pages 181–208, 1994.

[4] F. Gropengießer and K.-U. Sattler. An Extended Cooperative Transaction Model for XML. In *PIKM*, pages 41–48, 2008.

[5] M. P. Haustein and T. Härder. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. In *ADBIS*, pages 88–102, 2003.

[6] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. In *VLDB*, pages 351–351, 1979.

[7] E. Nett and B. Weiler. Nested Dynamic Actions - How to Solve the Fault Containment Problem in a Cooperative Action Model. In *Symposium on Reliable Distributed Systems*, pages 106–115, 1994.

[8] G. Oster, H. Skaf-Molli, P. Molli, and H. Naja-Jazzar. Supporting Collaborative Writing of XML Documents. In *ICEIS (4)*, pages 335–341, 2007.

[9] A. Silberschatz and Z. Kedem. Consistency in Hierarchical Database Systems. *J. ACM*, 27(1):72–80, 1980.

[10] G. Weikum and H.-J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. 1992.

[11] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2001.