Informatik - Forschung und Entwicklung manuscript No.
(will be inserted by the editor)

Francis Gropengießer · Katja Hose · Kai-Uwe Sattler

An Extended Transaction Model for Cooperative Authoring of XML Data

Eingegangen: date / Angenommen: date

Abstract In many application scenarios, for example in design or media production processes, several authors have to work cooperatively on the same project and consequently on the same data. In this context, a frequently used data format is XML. To enable cooperative authoring of shared XML graph structures, several requirements have to be fulfilled, e.g., early visibility of updates, multi-directional information flow, and processing data in parallel. Most transaction models proposed in the literature are hardly applicable in this context. In this paper, we propose a novel transaction model based on multi-level transactions and dynamic actions that meets these requirements. We describe the transaction model as well as its formal properties and discuss issues such as synchronization and logging.

Keywords ACTA, Dynamic Action, Cooperation, Media Production, Transaction, Tree Operations, XML

CR Subject Classification I.2.8 · I.2.3 · J.1 · K.3.2 · D.3.3 · D.1.6

1 Introduction

In many application scenarios, e.g., design or media production, it is necessary to allow users to work cooperatively on the same data. This means users are considered equal and have access to the current state of the project. They can exchange information in arbitrary directions without restrictions. In this way, it is possible for each user to adjust his own work to the current state of the project and to the work of the others. Each user is able to introduce his proposals and solutions and have them checked for conformance at an early stage.

The application scenario considered in this paper is the postproduction of movies as part of the media production process in a spatial sound system. 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. Fig. 1 shows an example scene graph.

In order to enable several authors to work cooperatively on the same XML data in a transactional context, we first have to consider the characteristics of media production processes, which are similar to the characteristics of common design processes in, for instance, CAD environments [17]. The durations of user transactions, which encapsulate the authors' operations, are typically rather long. Applying standard transaction models results in a late visibility of the authors' changes. Furthermore, if a transaction has to be aborted, the work of up to a whole workday might be lost. However, cooperative authoring requires not only changes to be visible at an early stage but also the possibility to discard single work steps that are part of a transaction.

One of the main problems of enabling cooperative authoring is that it conflicts with the serializability property of transactions. Cooperation requires multi-directional information flow while serializability only allows information flow in one direction (uni-directional information flow). Relaxing this property might result in conflicts, which we have to treat accordingly.

In summary, an appropriate transaction model for cooperative media production has to fulfill at least the following requirements:

1. It has to enable early visibility of changes as well as multi-directional information flow. Thus, we have to abstain from isolation and even serializability.
2. Inconsistencies resulting from, for example, non-repeatable read operations, lost updates, or the phan-

This work was supported by the DFG under grant SA782/15-1.

Francis Gropengießer
Department of Computer Science and Automation
TU Ilmenau
P.O. Box 100565, D-98684 Ilmenau, Germany
E-Mail: francis.gropengiesser@tu-ilmenau.de

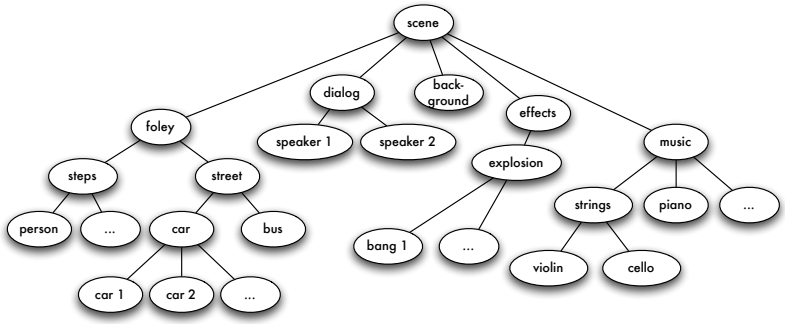

Fig. 1 Example Scene Graph

tom problem [22], as well as from other conflicts, which can occur due to the lack of serializability, have to be avoided or resolved.
3. The transaction model has to offer the possibility to discard single work steps, which requires relaxing atomicity.
4. Another challenge is to enable cooperative authoring for multiple authors on shared XML documents with the highest possible degree of parallelism. Thus, it is necessary to exploit the semantics of XML graph (tree) operations and to determine conflicts between these operations.

In this paper, we present a novel transaction model, describe an appropriate synchronization concept and a transaction recovery strategy, which altogether meet the requirements stated above. This paper is an extended version of [7] and structured as follows. In Section 2, we briefly review related work before Section 3 describes the system architecture. Section 4 presents the assumed data model and Section 5 describes semantic tree operations available for the data. Section 6 gives an informal description of the transaction model. In Section 7, we point out how to effect synchronization and illustrate our approach by giving a cooperative example in Section 8. Section 9 discusses logging and recovery. In Section 10, we comment on the properties of the developed model and Section 11 concludes the paper.

2 Related Work

Our work is based on the concept of extended transactions, which addresses the problems of long running transactions such as the late visibility of changes and the possible loss of great portions of work. However, as we will discuss below, extended transaction models are not fully applicable to our problem scenario as introduced above.

Closed nested transactions [14] require isolation between transaction trees and siblings of the same transaction tree – a transaction tree consists of a root transaction and several levels of subtransactions. If each author is assigned to a separate transaction tree, changes of a specific author are only visible to others after the root transaction was committed. Thus, a cooperative workflow is not possible.

Open nested transactions as well as their specialization, multi-level transactions [21], support cooperative authoring by relaxing the isolation property. An author can see the changes of another author before the other author finishes his work. However, transaction recovery in case of transaction errors or aborts is complex due to the use of compensating transactions. For example, sometimes it is hard or even impossible to find an appropriate compensating transaction because the operations that have to be undone are very complex or inverse operations do not exist. In [6], problems of compensating transactions are discussed in more detail.

The Saga concept [4] is based on open nested transactions. It relaxes the isolation property, too. However, in order to prevent possibly evolving cascading aborts and therefore to reduce transaction recovery complexity, commutativity between subtransactions of different Sagas is required. Thus, no "real" information flow is possible between different authors, which means no cooperativeness.

The ConTract model [20] is similar to the Saga concept but enables cooperative authoring because it does not require commutativity. However, the use of compensating transactions is required for transaction recovery.

The DOM transaction model [1] is an approach to combine open and closed nested transactions. This concept offers the possibility of open nesting of closed nested transactions. Thus, cooperative work is enabled, but again the use of compensating transactions is required.

The CONCORD model [19] is based on object versioning. Each author works on a local copy of an object. Cooperation is enabled by special relationships that can be committed by the authors. Version control systems (VCS) like CVS are also based on object versioning. They enable cooperation since an information flow between several authors in arbitrary directions is possible. However, VCS and the CONCORD model have one major drawback in common: changes of an author are not visible to other authors until he commits or starts

a special relationship. Furthermore, in VCS an update call is necessary after a commit in order to retrieve the current state of an object. In both models changes are not propagated automatically. However, cooperation requires visibility of changes at an early stage so that all authors are always aware of the most current state of the project. This is necessary because only in this way they are able to decide about further work steps within the project.

The dynamic action model [15] extends the traditional transaction model with an additional state *completed*, as shown in Fig. 2.

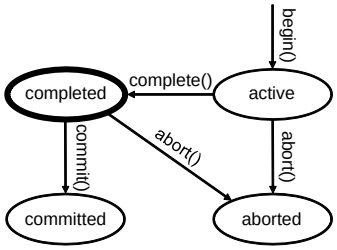

Fig. 2 Dynamic Action State Model

Before an action can proceed to the *committed* state, it has to be in the *completed* state. This means that all operations of this action are fulfilled and it is ready to commit. It can only be aborted due to the abort of actions it depends on. Only if it is proved that all actions it depends on are committed, this action can commit. Thus, committed actions never have to be aborted and so the durability property is ensured. Thereby, the use of compensating actions can be avoided. Based on this model, two extended action models have been developed.

Nested dynamic actions [16] have an architecture similar to the closed nested transactions described above. They require serializability between whole action trees. If each author is assigned a separate action, this means that there exists only a uni-directional information flow, which prevents cooperativeness.

Nested dynamic actions for cooperative applications [13] do not enforce serializability within groups of action trees, also called cooperation groups. Thus, cooperative work is enabled. However, this principle is impractical for our use case due to the following reason. Though the work can be divided into several groups, cooperation between different groups should still be possible, but is not allowed in this model. Furthermore, resolving conflicts, which may occur due to the lack of serializability, requires intervention by the users. Users have to specify objects for which a cooperative access has to be forbidden, because otherwise a conflict could occur. Such a low-level intervention cannot be expected from an end user.

In summary, all mentioned transaction or action models are not fully applicable to the media production use case described in Section 1. Thus, there is a need for a novel cooperative transaction model, which allows an early visibility of changes, a multi-directional information flow, and the possibility to discard single work steps. It should also prevent inconsistencies which can occur due to relaxation of isolation. Furthermore, transaction recovery should be possible without using compensating transactions.

A further approach for distributed cooperative working on XML data is the operational transformation presented in [3; 18]. In this approach, a user performs an operation on his local copy of the XML data. This operation is then broadcasted to other users who execute it on their local copies of the data. In order to preserve data consistency while executing concurrent operations on shared data items, an algorithm for the transformation of those operations is used that produces the same successor state S' of a data item independent of the execution order of two involved concurrent operations, starting from the same state S of this data item. The main problem of this approach is to find always the correct transformation function. Another problem is that lost updates can occur which can only be solved with a special undo operation.

3 Architecture

As described in Section 1, our main goal is to support a transaction-oriented, but cooperative workflow of several authors on shared XML documents, e.g., scene descriptions in media production processes. Meeting all the requirements of cooperative processes is not possible with a single system architecture. Thus, our goal is to develop a flexible *framework* that can be adapted to different system environments. The differences of these environments can be characterized as follows:

- The *coupling* degree of participating systems describes the degree of centralization of the data management. Data management is either more centralized (closed coupling) or more decentralized (loose coupling). Authors, for example, can work in the same building or distributed all over the world on the same data. In the first case all updates are executed together on a central database server while in the second case updates are first executed on local copies of the data and then propagated to a central database server.
- *Propagation* denotes the points in time when changes are made visible to other authors. Thus, changes can be propagated after every update operation or retrieved on demand like in CVS or SVN systems.
- The third dimension concerns the fulfillment of the *ACID* properties. In this respect, we can distinguish to what extend we can guarantee or relax these properties.

Ignoring the fact that these dimensions are not fully orthogonal to each other, we can illustrate them as sketched in Fig. 3 – spanning a three-dimensional solution space.

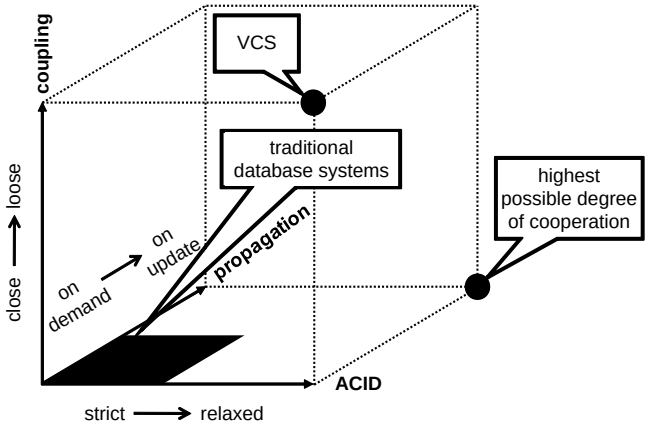

Fig. 3 Solution Space of Possible System Environments

Within this design space, we have to examine different working scenarios and system architectures concerning the possible degree of cooperation. Regarding our definition of cooperation, the maximum degree of cooperation can be reached in a closely coupled system environment with relaxed ACID properties and an immediate propagation of changes. Traditional database systems are closed coupled since users must always be connected to the database in order to query or update data. However, they are less cooperative since they enforce strict ACID properties. Only isolation can be relaxed slightly by using different isolation levels. Updates are propagated either on demand or on update. While the first propagation strategy is the normal case, the latter one occurs if a user has a sensitive cursor on data updated by another user. In this case, he immediately gets aware of these updates. Version control systems (VCSs) are loosely coupled, because users work on local copies of the data and do not need to be connected to the central data server all the time. Furthermore, VCSs do not enforce any ACID properties. They are not cooperative since users only become aware of changes by explicitly requesting and retrieving updated data.

Based on this design space, we can define three main scenarios [12]:

- **Single user:** This is the simplest and in fact non-cooperative case where a single author is working on his own scene document. Here, only basic database system functionalities like persistence and recovery are required, advanced techniques for distributed and cooperative work are not needed.
- **Workgroup:** In this scenario, multiple authors work together on the same scene and their updates have to be synchronized. In this scenario, a central repository is needed. It can be extended by local caches to reduce data traffic within the network.
- **Workspace:** This is a large-scale scenario where many authors work on the same data distributed all over the world. In contrast to the workgroup scenario, we cannot assume permanent connections to a central repository so that all authors work on their own workspaces. This requires a decoupled synchronization technique similar to version control systems like CVS.

For each scenario, we want to discover to what extend cooperativeness can be achieved by exploiting sophisticated transaction models, synchronization concepts, and recovery strategies. Therefore, we need a flexible system architecture.

The focus of this paper is the workgroup scenario. For this scenario, we have chosen a *client/server* architecture, where several authors (*clients*) are connected to a single *server* (Fig. 4).

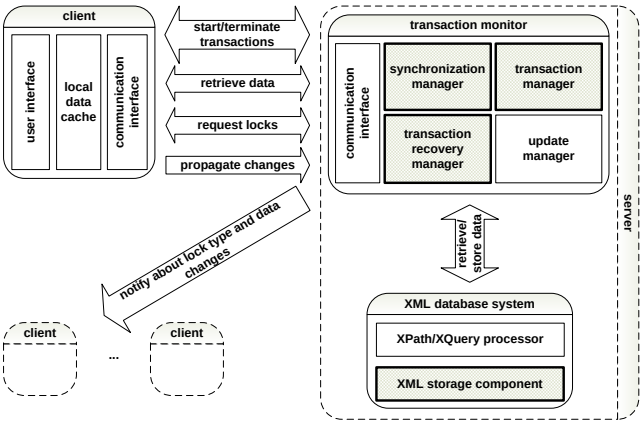

Fig. 4 System Model

The *clients* have a local *cache*, where they store copies of the XML subtrees they want to work on. Local operations are always executed on the data in this *cache*. The advantage of this procedure is that if an operation fails, no data has yet been propagated to the *server*.

The *server* consists of two major components – the *transaction monitor* and an *XML database system*. The transaction monitor consists of four major components:

- The *transaction manager* implements the transaction model and manages the transactions of all *clients*. A new transaction model based on nested dynamic actions is described in Section 6.
- The *synchronization manager* implements the synchronization strategy. In Section 7, we describe a lock-based synchronization protocol in more detail.
- The *transaction recovery manager* implements a transaction recovery strategy, which is used to handle aborts of transactions. For this purpose, we describe

a logging approach for our transaction model in Section 9.
- The *update manager* implements the publisher-subscriber pattern. Each client registers at the *update manager* in order to receive notifications about changes on data items. The registration is performed automatically by retrieving the parts of an XML document an author wants to work on. If the lock type of a data item or the data item itself is changed, all registered clients are notified by the server.

For storing XML data persistently, the transaction monitor makes use of an *XML database system*. This database system has to fulfill the following requirements:

- For data retrieval, *XPath* or *XQuery* are used, updates are performed using an appropriate language such as XUpdate.
- Since our transaction monitor handles transaction aborts only, the XML storage component has to offer a system recovery component.
- Furthermore, we make some assumptions on the data model, which are described in more detail in Section 4.

Based on these assumptions, transactions in the workgroup scenario are processed as follows. An author retrieves the XML subtree he wants to work on with an XPath/XQuery expression and stores it in a local data cache. Note that only this part of the whole XML data available at the server is visible to the author at this moment. In order to view more data, he has to issue another XPath/XQuery expression. Now, he can execute operations (Section 5) on the read XML subtree. The execution of operations requires the registration of the enclosing transactions at the transaction manager. With a lock-based strategy, the client has to request all necessary locks from the synchronization manager. If he can acquire the locks, this might cause changes in the lock types for other clients working on the same data. Then, they are notified by the server. Afterwards, the client executes the intended operations locally in the cache. After the corresponding transactions are terminated successfully, the changes are propagated to the server. There, they are logged for transaction recovery. Furthermore, the updates are immediately propagated to other clients working on the same data.

A first prototype of this approach is implemented and demonstrated in [8]. The server component is based on Oracle Berkeley DB XML[1] and uses SOAP for communicating with clients. The clients provide a graphical user interface for visualizing the XML tree and performing updates on nodes.

[1] `http://www.oracle.com/database/berkeley-db/xml/index.html`

4 Data Model

In this section, we describe the data format used for XML processing. Due to simplicity, an XML document is assumed to be a tree $T = (N, E, nl, nv)$. N is the set of nodes, which are represented by IDs unique in the database. Nodes are connected with directed edges. $E \subset N \times N$ is the set of all directed edges in T. Furthermore, T contains two labeling functions. The first one, $nl : N \rightarrow NL$, assigns node labels to nodes from the set of all possible node labels NL. This function is represented as a set of pairs $(node, node\ label)$. Node labels can be, for example, element names or attribute names. The second function, $nv : N_v \rightarrow NV$, where $N_v \subseteq N$, assigns node values to nodes from the set of all possible node values NV. This function is represented as a set of pairs $(node, node\ value)$. Node values can be, for example, attribute values or simple text. Considering attributes as individual nodes with a node value is possible since we are able to separate them from their corresponding elements, e.g., with the help of the taDOM (tailored DOM) specification [9]. In taDOM, attributes are represented as single nodes grouped under an additional node *attribute root node*. Note that we do not make any assumptions on the ordering of nodes at the same tree level. The approach presented in the following sections support ordered as well as unordered trees.

All XML documents are stored in an XML database for further processing. In order to avoid forests, we assume a database to resemble a tree structure $DB = (N', E', nl',\ nv')$, which is constructed according to Equations 1 – 7. $T_i = (N_i, E_i, nl_i, nv_i)$ with $N_{v_i} \subseteq N_i$ corresponds to an XML document (tree) within the database. NL_i and NV_i are its corresponding node label and node value sets.

$$N' = \bigcup_{i=1}^{n} N_i \cup DBroot, n \in \mathbb{N} \quad (1)$$

$$E' = \bigcup_{i=1}^{n} E_i \cup \quad (2)$$

$$\{(DBroot, r_i) | r_i \text{ is the root of } T_i, 1 \leq i \leq n\}, n \in \mathbb{N}$$

$$N'_v = \bigcup_{i=1}^{n} N_{v_i}, n \in \mathbb{N} \quad (3)$$

$$NL' = \bigcup_{i=1}^{n} NL_i \cup \{DBRootNode\}, n \in \mathbb{N} \quad (4)$$

$$NV' = \bigcup_{i=1}^{n} NV_i, n \in \mathbb{N} \quad (5)$$

$$nl' = \bigcup_{i=1}^{n} nl_i \cup \{(DBroot, DBRootNode\}, n \in \mathbb{N} \quad (6)$$

$$nv' = \bigcup_{i=1}^{n} nv_i, n \in \mathbb{N} \quad (7)$$

The node set N' consists of all node sets of all XML documents within the database and an additional node $DBroot$ (Equation 1). This node is exclusive and must not be changed or deleted. The set of directed edges E' consists of all edges of all XML documents (Equation 2). Furthermore, E' is extended by additional directed edges between the exclusive node $DBroot$ and the root nodes of all XML documents. The root node of a tree is the top-level node, which is only source and not sink of edges. The set N'_v of nodes, which can be assigned a node value, is the union of all such nodes in the XML database (Equation 3). The set of possible node labels NL' is the union of all node label sets within the database and the additional label $DBRootNode$ (Equation 4). Furthermore, the set of possible node values NV' is the union of all possible node values within the database (Equation 5). The set nl' consists of all pairs $(node, node\ label)$ of all XML documents and the additional pair for $DBroot$ (Equation 6). The last equation (Equation 7) defines set nv' as the set of all pairs $(node, node\ value)$ of all XML documents.

To illustrate these definitions, consider the following example database tree DB, which is also shown in Fig. 5:

- $N' = \{DBroot, 1, 2, 3, 4, 5, 6, 7, 8\}$
- $E' = \{(DBroot, 1), (DBroot, 6), (1, 2), (1, 3), (2, 4), (4, 5), (6, 7), (6, 8)\}$
- $nl' = \{(DBroot, DBRootNode), (1, scene), (2, music), (3, foley), (4, attribute\ root\ node), (5, volume), (6, scene), (7, dialog), (8, effects)\}$
- $nv' = \{(5, \text{“20”})\}$

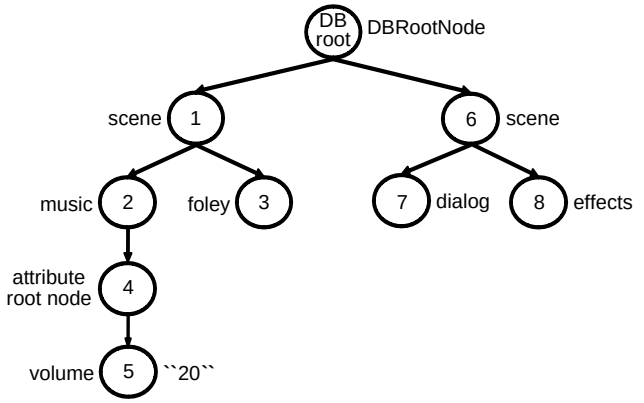

Fig. 5 Example Database Tree DB

5 Operations and Operation Sequences

Our work aims at increasing the number of authors that can work in parallel on shared XML documents. Implicitly, this means reducing the conflict set of operations. However, this cannot be achieved by applying a simple read/write model, because it would be too restrictive. Thus, we extend this model with semantic tree operations. Before specifying high-level tree operations, we specify low-level tree operations operating directly on node/edge level.

Low-level read operations of nodes and edges are implemented by the following operations:

- The operation $strReadNode(n)$, with $n \in N'$, is used to retrieve the structural information of a node and returns a pair $(n, node\ label) \in nl'$.
- The content of a node is read using the operation $contReadNode(n)$ with $n \in N'_v$. It returns a pair $(n, node\ value) \in nv'$.
- The operation $readEdge(m, n)$, with $m, n \in N'$, returns $true$ if $m, n \in E$ and $false$ otherwise.

Low-level update operations of nodes and trees are implemented by the following operations:

- The operation $edit(n, new\ node\ value)$, with $n \in N'_v$ and $new\ node\ value \in NV'$ is used for editing the content of a node. The new pair $(n, new\ node\ value)$ replaces the old pair $(n, node\ value)$ in nv', i.e., $(n, new\ value)$ is removed from nv' and $(n, new\ node\ value)$ is added to nv'.
- The operation $move(n, m)$, with $n, m \in N'$ and $n \neq m$, assigns m as new $parent$ to n so that the existing edge (p, n) between n's old parent p and n is removed from E' and the new edge (m, n) is added to E'.
- The operation $insert(n, p, node\ label)$, with $p \in N'$, $n \notin N'$, and $node\ label \in NL'$, adds a new child node n to node p. The operation extends N' with n, E' with the pair (p, n), and nl' with the pair $(n, node\ label)$.
- The operation $delete(n)$, with $n \in N'$ and n having no children, removes node n from N' and the edge (p, n) between n and its parent p from E'. Furthermore, the pair $(n, node\ label)$ is removed from nl' and the pair $(n, node\ value)$, if it exists, is removed from nv'.

With respect to $move$ and $delete$, note that all accessible nodes have a parent due to the existence of the exclusive node $DBroot$.

Based on these low-level operations, we define high-level operations, which are useful to give an author comfortable tools, for example, to read or delete subtrees – a $subtree(n)$ is a part of a tree rooted by node n and contains n, all of n's descendants, and the corresponding edges. High-level operations are mapped to low-level operations and thus offer the possibility to abort single steps of an operation on subtrees without having to abort the entire operation.

High-level read operations on nodes and subtrees are implemented by the following operations:

- The operation $readNode(n)$, with $n \in N'$, is used to read a node. This operation is implemented either

by $strReadNode(n)$ followed by $contReadNode(n)$ or only by $strReadNode(n)$. Section 7 provides more details on this issue.
- The operation $readSubtree(n)$, with $n \in N'$, reads all *accessible* nodes and edges within the subtree rooted by node n. This operation is mapped to a sequence of $readNode$ and $readEdge$ operations, which are executed on all *accessible* nodes and edges within the subtree. Section 7 defines what *accessible* means in this context.

High-level update operations on nodes and trees are implemented by the following operations:

- The operation $insertSubtree(S, n)$, with $n \in N'$, inserts a tree $S = (N'', E'', nl'', nv'')$ with $nv'' = \emptyset$ and attaches it as a subtree to node n – node contents can be created by future edit operations on the nodes. This method is implemented by a series of low-level *insert* operations. S's nodes and their corresponding edges are inserted from *top to bottom*, i.e., starting with S's root node and finishing with S's leaf nodes. For the insertion of nodes at the same level we assume no ordering.
- The operation $deleteSubtree(n)$, with $n \in N'$, deletes subtrees rooted by n from a tree DB. It is implemented by a sequence of low-level *delete* operations on each node in N'. N''s nodes and their corresponding edges are deleted from *the bottom to the top*. Thus, we start with the leaves and finish with the root node. For the deletion of nodes at the same level within the subtree, we assume no ordering.

Although, in principle, the application (i.e., the user) determines the order in which the operations are executed, we make some restrictions to guarantee several useful properties that we discuss in detail below. A valid operation sequence corresponds to the grammar defined by Equations 8 – 12. In these equations, $m, n, r, t, u, v, w \in N'$ and $k \notin N'$ are denoting nodes. The subtrees rooted by m and n do not overlap, i.e., $subtree(n) \nsubseteq subtree(m)$ and $subtree(m) \nsubseteq subtree(n)$. Furthermore, node t is a part of $subtree(n)$, v and r are descendants of n, $v \neq t$, r is a *leaf* node, u is a part of $subtree(m)$, and w is not part of $subtree(n)$.

$$\langle OS \rangle ::= \langle O_1 \rangle | \langle O_2 \rangle | \langle O_3 \rangle | \langle O_4 \rangle \quad (8)$$

$$\langle O_1 \rangle ::= readSubtree(n)[edit(t, node\ value)| move(v,t)|insert(k,t,node\ label)|delete(r)| deleteSubtree(v)|insertSubtree(S,t)] \quad (9)$$

$$\langle O_2 \rangle ::= readNode(n)[edit(n, node\ value)| insertSubtree(S,n)|insert(k,n,node\ label)] \quad (10)$$

$$\langle O_3 \rangle ::= readSubtree(n)\ readSubtree(m)\ move(v,u) \quad (11)$$

$$\langle O_4 \rangle ::= readSubtree(n)\ readNode(w)\ move(v,w) \quad (12)$$

Operation sequences are isolated from each other by applying the synchronization model described in Section 7. Changes are propagated right after an operation sequence is executed.

Using such operation sequences has several advantages. Before a data item can be changed, it is always read. Thus, there are no "blind" updates. Together with the isolation of an operation sequence, this prevents inconsistencies resulting from lost updates, non-repeatable reads, or phantoms (Requirement 2 as stated in the introduction) because update operations only depend on the data actually read in the previous step. Other conflicts, for example at a semantic level resulting from situations where an author disagrees with the work of another author, can be discovered by reading the data before changing it. Resolving such conflicts is possible by offering an "undo" function. This function strongly depends on the used transaction recovery technique – a detailed discussion is beyond the scope of this article. Another advantage of using operation sequences is that changes are visible after at most one high-level update operation is executed. This increases concurrency (Requirement 4).

6 Transaction Model

Our transaction model is based on dynamic actions as well as on multi-level transactions. For our solution, a maximum nesting depth of four with respect to multi-level transactions is sufficient. The reason is that the transaction model only covers the operation sequences defined in Section 5 and implements a top-down decomposition – corresponding to: operation sequences (level 0) → single operation sequence (level 1) → operation on subtree or node (level 2) → operation on node (level 3). To illustrate the model, assume a user executes the operation sequence OS (Equation 13) on the data structure depicted in Fig. 5:

$$\langle OS \rangle ::= readSubtree(6)\ delete(8) \quad (13)$$

The resulting transaction structure is shown in Fig. 6. It consists of the following four transaction levels:

- **root transaction (level 0):** This transaction is started by an author and ends when he decides to finish and "commit" his work. Within the root transaction no operation is executed. Instead, all operations are wrapped into subtransactions. If the root transaction is aborted, all of its subtransactions are aborted, too. In contrast, subtransactions can be aborted without causing the abortion of the root transaction. Note that this property fulfills Requirement 3.
- **subtransactions at levels 1-3:** Each operation sequence is represented by a subtransaction at level 1. As indicated in the previous section, the principle is to divide operation sequences into subtree operations

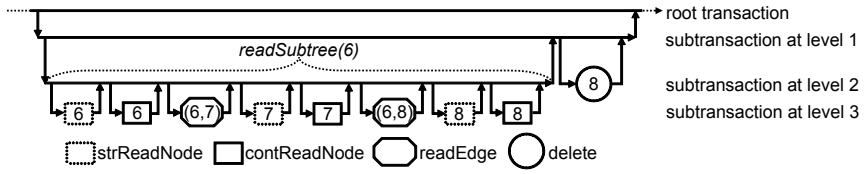

Fig. 6 Example Transaction Structure

and subtree operations into operations on nodes. This results in at most two more subtransaction levels. If a subtransaction aborts, all ancestor subtransactions are aborted as long as the subtransaction at level 1 is not in the state *completed*. This ensures that an operation sequence is either completely executed or not at all. However, after the completion of the subtransaction at level 1, descendant transactions of this subtransaction can be aborted – ancestor and descendant transactions are formally defined below. This reduces the number of cascading aborts. Changes resulting from subtransactions at level 2 or 3 are only visible to their parent subtransactions. However, once a subtransaction at level 1 is completed, these changes are visible to all users/transactions. Thus, isolation is relaxed between root transactions and changes are visible to other users at an early stage (Requirement 1).

Let us now give a basic formal description of the transaction model. For this purpose, we use the ACTA framework [2] and introduce necessary dependencies. We begin with some basic definitions.

Definition 1 (Nested Transaction). A nested transaction resembles a tree structure. The top of the tree (level 0) is called *root transaction*. All transactions at levels $1, ..., n$, with $n \in \mathbb{N}$ and n denoting the maximum tree depth, are called *subtransactions*. The set of *children* of a transaction t consists of all subtransactions at $level(t) + 1$ that are directly wrapped into transaction t. The set of transaction t's *descendants* consists of all transactions at $level(t)+1, ..., n$, with $n \in \mathbb{N}$, that are wrapped into t. Transactions at level n are called *leaves*. The *parent* of a transaction t is the transaction at $level(t) - 1$ that t is directly wrapped into. The set of *ancestors* of a transaction t consists of all transactions at $level(t) - 1, ..., 0$ that t is wrapped into. A *subtransaction tree* T of a *nested transaction* consists of *subtransaction* t (root of T) and all its *descendants*.

Definition 2 (History). A history H contains all events invoked by the transactions. Furthermore, these events are sorted within H according to their execution order. The predicate $e_1 \rightarrow e_2$ indicates that event e_1 precedes e_2 and that both events are contained in H. $P(H, p) = H_p$ is the projection of H on p where p denotes a data item da. Then, H_{da} contains all operations executed on da. The *state* s_{da} of data item da can be defined as $s_{da} = state(s_{da_0}, H_{da})$. This means that the current *state* of a data item da is the result from executing H_{da} on its initial state s_{da_0}. The value of $return(s_{da}, o[da]) = Output$ corresponds to the output of operation $o[da]$ executed on da – the output depends on da's current state s_{da}.

In the following definition, the transaction model is specified with the help of ACTA Axioms.

Definition 3 (Transaction Model).

$$t \text{ is a root or a subtransaction} \tag{14}$$

$$\forall t(t \text{ is } \textit{failure atomic}) \tag{15}$$

$$r \text{ is a root transaction} \tag{16}$$

$$s \text{ is a subtransaction} \tag{17}$$

$$Desc_t \text{ is the set of all descendants of } t \tag{18}$$

$$Anc_s \text{ is the set of all ancestors of } s \tag{19}$$

$$Children_t \text{ is the set of all children of } t \tag{20}$$

$$o_t[da] \text{ is an operation of transaction } t \tag{21}$$

Axioms 14–21 define all variables and sets used below.

$$SE_t = \{Begin_t,\ Complete_t,\ Abort_t,\ Commit_t\} \tag{22}$$

Axiom 22 shows the special event set SE of a transaction (dynamic action).

$$(Begin_t \in H) \Rightarrow (\neg(Begin_t \rightarrow Begin_t) \wedge \neg(Complete_t \rightarrow Begin_t) \wedge \neg(Commit_t \rightarrow Begin_t) \wedge \neg(Abort_t \rightarrow Begin_t)) \tag{23}$$

Axiom 23 states that a transaction is only allowed to begin if it has not already begun and has not yet been completed, committed, or aborted.

$$(Complete_t \in H) \Rightarrow ((Begin_t \rightarrow Complete_t) \wedge \neg(Complete_t \rightarrow Complete)_t \wedge \neg(Abort_t \rightarrow Complete_t) \wedge \neg(Commit_t \rightarrow Complete_t)) \tag{24}$$

Furthermore, a transaction can only be completed if it has begun and has not been completed, committed, or aborted (Axiom 24).

$$(Abort_t \in H) \Rightarrow ((Begin_t \rightarrow Abort_t) \wedge \neg(Commit_t \rightarrow Abort_t)) \tag{25}$$

A transaction can only be aborted if it has begun and has not been committed (Axiom 25). Note that a transaction in state completed can still be aborted.

$$(Commit_t \in H) \Rightarrow ((Complete_t \rightarrow Commit_t) \wedge \neg(Commit_t \rightarrow Commit_t) \wedge \neg(Abort_t \rightarrow Commit_t)) \tag{26}$$

Axiom 26 states that a transaction can only commit if it has been completed and has not yet been committed or aborted.

$$(Begin_s \in H) \Rightarrow \forall t \in Anc_s((Begin_t \rightarrow Begin_s) \wedge \neg(Abort_t \rightarrow Begin_s) \wedge \neg(Complete_t \rightarrow Begin_s)) \quad (27)$$

A subtransaction can only begin if its ancestor transactions are active (Axiom 27).

$$\forall s_{t_i} \in Children_t, 2 \leq i \leq n((Begin_{s_{t_i}} \in H) \Rightarrow (Complete_{s_{t_{i-1}}} \rightarrow Begin_{s_{t_i}})) \quad (28)$$

Axiom 28 states that all child transactions of a transaction t are executed in a sequential order.

$$(Complete_s \in H) \Rightarrow \forall s' \in Desc_s((Complete_{s'} \rightarrow Complete_s) \wedge \neg(Abort_{s'} \rightarrow Complete_s)) \quad (29)$$

If a subtransaction wants to proceed to state completed, all its descendant transactions must already be in this state and must not have been aborted (Axiom 29).

$$(Complete_r \in H) \Rightarrow \forall s \in Desc_r((Complete_s \rightarrow Complete_r) \vee (Abort_s \rightarrow Complete_r)) \quad (30)$$

In contrast, a root transaction can be completed even if its descendant transactions have been aborted (Axiom 30). However, in both cases the descendant transactions must not be active.

$$(Abort_t \in H) \Rightarrow \forall s \in Desc_t(Abort_s \in H) \quad (31)$$

If a transaction is aborted, all descendant transactions are aborted, too (Axiom 31).

$$vital(t, t') \Leftrightarrow ((t' = s) \wedge (t \in Desc_{t'}) \wedge (Complete_{t'} \notin H)) \quad (32)$$

A transaction t is *vital* to a transaction t' iff t' is a subtransaction, t is a descendant of t', and t' is still active (Axiom 32). If a transaction t is *vital* to a transaction t', then the abort of t leads to the abort of t'.

$$ID(t, t') \Leftrightarrow \exists da(\exists o_{t'}(\exists o_t((o_{t'}[da] \rightarrow o_t[da]) \wedge ((s'_{da} = return(s_{da}, o_{t'}[da])) \vee (s'_{da} = state(s_{da}, o_{t'}[da]))) \wedge ((Output = return(s'_{da}, o_t[da])) \vee (s''_{da} = state(s'_{da}, o_t[da])))))) \quad (33)$$

Axiom 33 states that a transaction t is said to be information-dependent (ID) on t' iff both transactions execute at least one operation on a shared object. The output or the produced state of the operation in transaction t depends on the state produced or returned by the operation of transaction t'. For example, an *edit* operation depends on the result of the preceding *contRead* operation on the same node. Note that *contRead* returns the *state* of the node, which corresponds to the *node value*. This implicitly means that the *contRead* operation depends on the *state* (*node value*) of the node produced by a previous *edit* operation. This results in chains of information-dependent transactions. Note that in our model such dependencies only exist between subtransactions at levels 2 and 3, because only these transactions are allowed to execute operations.

$$AD(t', t) \Leftrightarrow ((t' \in Children_t) \vee ID(t', t) \vee vital(t, t')) \quad (34)$$

A transaction t' is said to be abort-dependent (AD) on a transaction t iff t' is a child transaction of t or t' is information-dependent on t or t is *vital* to t' (Axiom 34). This means that t' has to be aborted if t aborts.

$$(Commit_t \in H) \Rightarrow \forall t'(AD(t, t') \Rightarrow (Commit_{t'} \in H)) \quad (35)$$

A transaction can only commit if all transactions, it is abort-dependent on, are committed or commit together with the considered transaction (Axiom 35).

7 Synchronization

In order to synchronize concurrent operations, we apply a lock-based approach, which is inspired by the ideas presented in [5; 10; 11]. However, other synchronization methods such as optimistic models are conceivable, too.

In order to identify lock types and their compatibility, we need to consider the list of supported tree operations introduced in Section 5. As high-level operations are mapped to low-level operations, we only have to handle low-level operations. For the three different *read* operations we introduce the lock types SRL (*strReadNode*), CRL (*contReadNode*), and ERL (*readEdge*). EL denotes the lock type corresponding to the *edit* operation, the lock type DL is used in conjunction with *delete* operations, and IL denotes the lock type for the *insert* operation. In addition, we need an *intentional structure change lock* ISCL that protects a node against deletion.

The compatibility of these lock types can be illustrated with a compatibility matrix. As we support operations changing both nodes and edges, we have to distinguish conflicts on nodes and conflicts on edges. Table 1 considers the first case. The rows show the locks already held for a node and the columns show the locks requested on the same node. Symbol $\surd$ means that the locks are fully compatible and $-$ means that they are incompatible. Furthermore, $+$ denotes that two *insert* operations are only compatible if the order of the inserted nodes is

irrelevant. When working with ordered trees, two *insert* operations are not compatible at all. The choice which policy is used is up to the application.

	SRL	CRL	EL	DL	IL	ISCL
SRL	√	√	√	−	√	√
CRL	√	√	−	−	√	√
EL	√	−	−	−	√	√
DL	−	−	−	−	−	−
IL	√	√	√	−	+	√
ISCL	√	√	√	−	√	√

Table 1 Lock Compatibility Matrix for Node Operations

Table 2 shows the compatibility matrix for operations on edges. The rows again show the locks already held for an edge and the columns show the locks requested on the same edge.

	ERL	DL
ERL	√	−
DL	−	−

Table 2 Lock Compatibility Matrix with Respect to an Edge

Based on these fundamental considerations, we now specify which locks are necessary for which operation:

- The operations $strReadNode(n)$, $contReadNode(n)$, $readEdge(m, n)$, and $edit(n, node\ value)$ only need the corresponding lock type on the affected node n, or edge (m, n) respectively.
- The $insert(n, p, node\ label)$ and $insertSubtree(S, p)$ operations only require an IL on the specified target node p. For the inserted node n (the subtree S), no locks are required, because we assume that the new node is not visible (and thus accessible) to others before the insertion is completed.
- The operation $delete(n)$ requires a DL on the corresponding node n and on the edge between n and its parent node. Furthermore, an ISCL has to be acquired for the parent node. This prevents the deletion of the parent by another user. This is necessary since we assume an edge to exist only if both of its nodes exist. If a subtree shall be deleted using the operation $deleteSubtree(n)$, all of n's descendants and their edges have to be locked with DLs.
- The operation $move(n, m)$ requires a DL on the edge $(parent(n), n)$, an IL on the target node m, and ISCLs on n itself and its parent – the latter preventing both nodes from being deleted by another user/transaction.
- The operation $readNode(n)$ requires at least a SRL on node n. This implicitly means that a CRL is optional. If no CRL can be obtained, the low-level operation $contReadNode(n)$ is simply not executed.
- The operation $readSubtree(n)$ requires at least an SRL on node n. This implicitly means that a CRL on n and SRLs, CRLs, and ERLs on all descendant nodes with their corresponding edges are optional. Let us assume a user executes $readSubtree(1)$ on the example data structure in Fig. 5, while another user issues a delete operation: $delete(3)$. Proceeding as described above allows the first user to read all *accessible* nodes $(1, 2, 4, 5)$ with their corresponding edges $((1, 2), (2, 4), (4, 5))$ and thus increases concurrency. Proceeding in the traditional way causes an abort of the entire $readSubtree(1)$ operation because not all necessary locks could be acquired. Note that a *readSubtree* operation returns a connected set of nodes. This means, for example, the user who performs $readSubtree(1)$ cannot access $subtree(4)$ if another user is moving node 4 at the same time.

Next, we discuss the process of acquiring and releasing locks. We illustrate this procedure with the help of the example sequence of Equation 13.

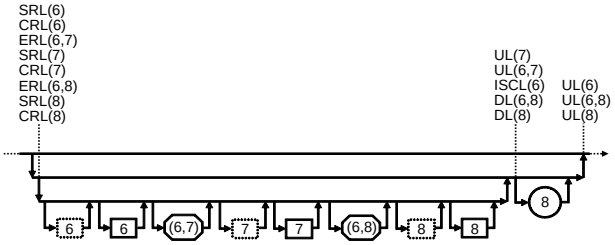

Fig. 7 Acquiring and Releasing Locks

As illustrated in Fig. 7, locks are acquired and released according to the following scheme:

- All necessary read locks are acquired atomically at the beginning of the first subtransaction at level 2.
- Afterwards, the locks are passed down to the subtransactions at level 3.
- After having entered the state completed, the subtransactions at level 3 pass the locks back to the subtransaction at level 2, which passes them on to the subtransaction at level 1 after completion.
- With the beginning of the second subtransaction at level 2 the read locks on the affected nodes and edges are tightened atomically. All other locks are released instantly since they are no longer used. Note that locks for update operations (EL, DL, IL, ISCL) can only be acquired for nodes that are already locked with an appropriate read lock.
- After having entered the state completed, the subtransaction at level 2 passes the locks back to the subtransaction at level 1, which releases them after completion.

A special case occurs if several authors hold read locks on the same node or edge. Then, lock tightening requires

special attention since some update operations such as edit or delete are exclusive. In this case, we proceed as follows:

- The exclusive lock is granted to the first author who tries to tighten his read lock into an exclusive lock.
- For all other read operations on the considered data item, the corresponding subtransactions are aborted.

Aborting the subtransactions is no disadvantage due to the following fact. Within the subtransaction at level 1, which encloses the affected subtransaction, at most two read operations could have been executed. Thus, there is no need for a transaction rollback with the help of the recovery manager. The affected transactions are simply deleted by the transaction manager and a new subtransaction at level 1 is started.

Using this synchronization model has several advantages:

- It does not require serializability of root transactions and thus enables a multi-directional information flow (Requirement 1).
- As already mentioned in Section 5, it prevents inconsistencies resulting from lost updates, non-repeatable reads, or phantoms (Requirement 2) due to the following reasons. The data is read before it is updated since update locks can only be acquired by a transaction if it already holds appropriate read locks on the corresponding data items. Thus, update operations only depend on actually read data items. Furthermore, no other transaction can update the data between a read and an update subtransaction at level 2, because read locks are held until they are tightened. Thus, subtransactions at level 1 are isolated from each other.
- Another advantage is that a *readSubtree* operation can be executed partially on those parts of the subtree that are structurally stable. This increases concurrency because only the actually changed parts of a subtree are not accessible (Requirement 4). Furthermore, it is possible to read and update a node that is being moved by a direct jump to this node with the help of the *readNode* operation. This increases parallelism since a node can be moved and edited at the same time by two different users (Requirement 4).
- Furthermore, requesting locks atomically prevents deadlocks, because a transaction is aborted if it cannot acquire the requested locks. Note that this does not conflict with the partial execution of, for instance, a *readSubtree* operation.

8 Example of Cooperative Processing

For a better understanding of the proposed concepts, we sketch a cooperative workflow (Fig. 8) between two authors (clients) working on the same subtree. Both authors execute the same operation sequence *readNode*(5) *editNode*(5, *node value*) several times. The main focus of this example is to show the core features of cooperation – multi-directional information flow between authors and immediate propagation of updates. The first box in Fig. 8 shows the following:

1. In the beginning, both clients retrieve the same subtree indicated by an XPath expression. Thus, the root transactions are started.
2. Client 1 starts the subtransactions at level 1 and 2, which are necessary to effect the operation sequence stated above.
3. Now, client 1 requests the necessary SRL and CRL lock in order to read node 5. If the locks are granted, client 2 is immediately notified by the server about the lock type changes on node 5.
4. Next, client 1 executes the *readNode* operation on node 5. This operation is decomposed in a *strReadNode* and a *contReadNode* operation as described in Section 6. Due to space limitations we abstain from sketching the enclosing subtransactions at level 3.
5. After having read node 5 successfully, the subtransaction at level 2 is completed and its locks are passed to the subtransaction at level 1.

At this point, client 1 is able to edit node 5. Client 2 is informed that someone else is working on the same subtree. Note that this is very important in a cooperative environment. Every author should always be up-to-date with respect to the state of the project. The process of editing node 5 is shown in the second box:

1. Client 1 starts a subtransaction at level 2. Note that the root transaction and started subtransaction at level 1 are still running.
2. Furthermore, the owned read locks are tightened to an EL lock. This lock type change is immediately propagated to client 2.
3. Now, client 1 executes the *edit* operation on node 5.
4. Afterwards, the subtransaction at level 2 is completed and its EL lock is passed to the subtransaction at level 1.
5. With the completion of the subtransaction at level 1, the EL lock is released and the result of the *edit* operation is propagated to the server. The server then notifies client 2 about the lock type change and sends him the new node value. Now, both clients know the most current state of the subtree.

The last box shows a multi-directional information flow between both clients in a short form. First, client 2 executes the operation sequence defined above. Then, client 1 executes it again. Note, the root transactions of both clients are still running. Thus, we abstain from serializability between root transactions.

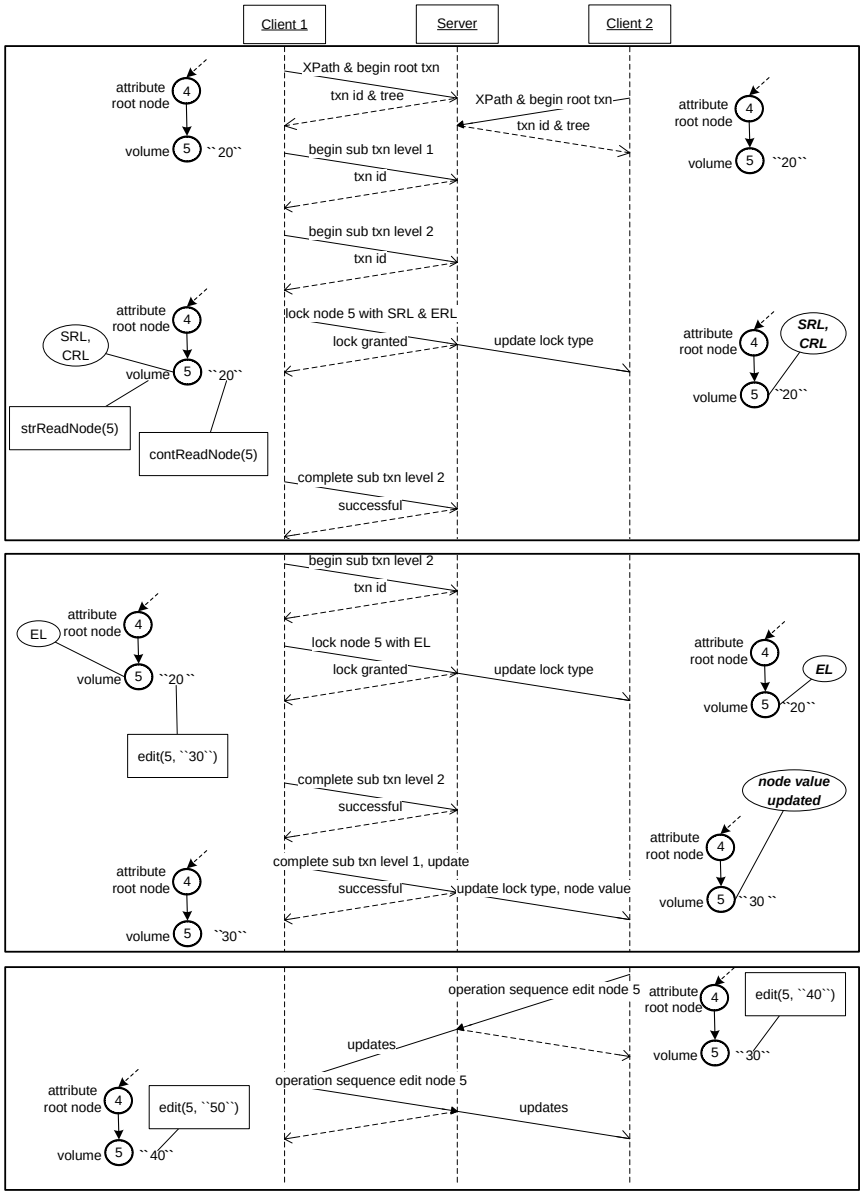

Fig. 8 Cooperational Example

9 Transaction Logging and Recovery

Transaction recovery is necessary to guarantee failure atomicity. Aborted transactions have to be rolled back and thus have no effect on the persistent database. We have chosen a logging technique as our first-class solution, because it is very easy to implement and avoids the problems of compensating transactions described in Section 2. However, it is possible to extend the transaction recovery manager shown in Fig. 4 with other recovery strategies.

Let us first describe the log information. As defined in Section 6, all operations are wrapped into subtransactions at levels 2 or 3. Since only low-level operations are executed on the data, logging the transactions containing them is sufficient. Furthermore, only low-level update operations are interesting for transaction recovery because only they change the content or the structure of XML documents. Logging transactions containing only a low-level read operation is not necessary in practice since reading a data item has no effect on its content or structure. Fig. 9 shows the log with the log entries for every transaction containing an update operation.

The log entries are ordered with respect to the changed node and the execution order of the transactions. They contain the following information:

– The transaction ID (*txnID*) of the corresponding transaction is logged.

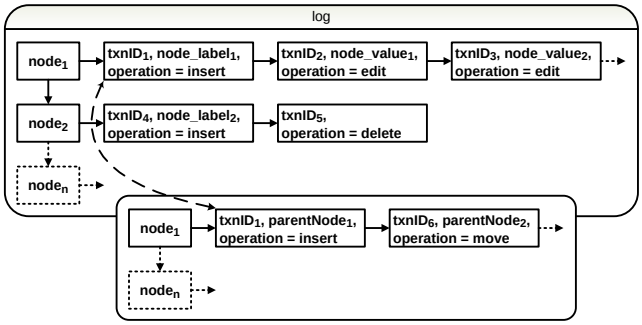

Fig. 9 Example Log

- For each node, its value and its label are captured. Each edit operation creates a new version of the node value. The node label is only stored once, because it is created with an insert operation and must not be changed afterwards.
- Furthermore, for each node its list of parents is captured. An insert operation enqueues the first parent. Every move operation enqueues a new parent.

In summary, the log contains the following major information:

- the order of executed update subtransactions with respect to the changed node or edge and
- the history of changes on nodes and edges.

Log entries are written after an update transaction is completed and the modified data is propagated to the server. Thereby, for transactions containing an edit, delete, or move operation only one entry is written. For an insert operation, we have to write two entries since on the one hand, the node with the given node label is created and, on the other hand, an edge between the parent node and the new node has to be inserted.

If a completed transaction is aborted, the corresponding entry is removed from the log. Furthermore, all transactions that are abort-dependent on this transaction are also aborted and their entries are removed. The new state of the XML documents is created with the information of the last not removed entries. Note that this log information is sufficient to reconstruct all XML documents within the database. All nodes can be created with the help of the node label and node value entries. The edges can be created using the information about the parent of a node. The current state of a node is represented by the last entry in the list of this node, the current edge by the last entry in the parent list of a node. If all transactions with respect to one node are committed, the current state of this node is saved in the XML database persistently. In this case, the log entries of these transactions are removed.

10 Properties

In this section, we discuss some properties of the proposed transaction model and the synchronization strategy, starting with the ACID properties.

Atomicity is relaxed since subtransactions can abort without aborting the root transaction (Requirement 3 in Sect. 1). However, *failure atomicity* can be guaranteed with an appropriate transaction recovery model. Since our model is based on dynamic actions, which guarantee *durability* [15], it is possible for instance to exploit an object versioning model instead of compensating transactions. In this paper, we presented a logging technique as a particular case of object versioning.

Preserving *consistency* means satisfying integrity constraints. The only kind of *integrity constraint* considered here is preserving the document structure. Satisfying this constraint is possible by simply checking if all operations – even in the case of an abort – preserve the structure of the document.

Isolation and even *serializability* are relaxed between root transactions. This allows an early visibility of object changes as well as a cooperative workflow between several authors (Requirement 1). However, individual operation sequences are isolated from each other. Together with the constraint that only read data can be changed, this prevents inconsistencies resulting from lost updates, non-repeatable reads, and phantoms.(Requirement 2).

Durability is guaranteed since the model is based on dynamic actions, which already fulfill this property[15].

In addition to the ACID properties, let us highlight some additional features of the model. Exploiting the semantics of special tree operations allows a fine-grained conflict specification and in this way increases the degree of parallelism in the workflow (Requirement 4). Hence, it is possible that several operations, e.g., updating and moving the same nodes, are executed in parallel.

Furthermore, locks are always acquired atomically. This implicitly means that, if not all locks can be obtained, the transaction is aborted. This prevents deadlocks.

Finally, splitting operations on subtrees into operations on nodes and enclosing these operations within separate subtransactions allows fine-grained information dependency management. Hence, it is possible to abort parts of, e.g., an *insertSubtree* operation (Requirement 3), which reduces the number of cascading aborts.

11 Conclusion

In this paper, we have presented a novel cooperative transaction model based on dynamic actions and multi-level transactions that fulfills the requirements of media production applications. The proposed model enables a high degree of cooperation between users in comparison to other models by making changes visible to other users at an early stage and allowing a multi-directional

information flow between them. In order to achieve this goal, we have to relax serializability. However, by isolating operation sequences and reading data before updating, the model prevents inconsistencies resulting from conflicts like lost updates, non-repeatable reads, or the phantom problem. Furthermore, we have extended the simple read/write model with semantic tree operations to increase the degree of parallelism. Thus, for instance *move* and *edit* operations can be supported even on the same node.

Our future work comprises the formal proof of correctness of the transaction, synchronization, and recovery model. A second step will be the integration of the presented approach into a cooperative sound design application. In this context, the analysis of different system architectures as well as the evaluation of other synchronization techniques like optimistic strategies are also important issues.

References

1. Buchmann AP, Ozsu MT, Hornick M, Georgakopoulos D, Manola FA (1992) A Transaction Model for Active Distributed Object Systems. In: Database Transaction Models for Advanced Applications, pp 123–158
2. Chrysanthis PK, Ramamritham K (1991) A Formalism for Extended Transaction Model. In: VLDB '91, pp 103–112
3. Ellis CA, Gibbs SJ (1989) Concurrency control in groupware systems. SIGMOD Rec 18(2):399–407
4. Garcia-Molina H, Salem K (1987) Sagas. In: SIGMOD '87, pp 249–259
5. Grabs T, Böhm K, Schek HJ (2002) XMLTM: efficient transaction management for XML documents. In: CIKM '02, pp 142–152
6. Greenfield P, Fekete A, Jang J, Kuo D (2003) Compensation is Not Enough. In: EDOC '03, p 232
7. Gropengießer F, Sattler KU (2008) An Extended Cooperative Transaction Model for XML. In: PIKM, pp 41–48
8. Gropengießer F, Hose K, Sattler KU (2009) Ein kooperativer XML-Editor für Workgroups. In: BTW
9. Haustein MP, Härder T (2003) taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. In: ADBIS, pp 88–102
10. Haustein MP, Härder T (2004) Adjustable Transaction Isolation in XML Database Management Systems. In: XSym, pp 173–188
11. Helmer S, Kanne CC, Moerkotte G (2004) Evaluating lock-based protocols for cooperation on XML documents. SIGMOD Rec 33(1):58–63
12. Hose K, Sattler KU (2007) Cooperative Data Management for XML Data. In: DEXA, pp 308–318
13. Kruse A (1997) A Cooperative Lock Protocol for Nested Dynamic Actions — Diploma Thesis at the University of Bonn (in German)
14. Moss JEB (1981) Nested Transactions: An Approach to Reliable Distributed Computing. PhD thesis
15. Nett E, Mock M (1995) How to Commit Concurrent, NonIsolated Computations. In: IEEE Workshop on Future Trends in Distributed Systems, Korea
16. Nett E, Weiler B (1994) Nested Dynamic Actions - How to Solve the Fault Containment Problem in a Cooperative Action Model. In: Symposium on Reliable Distributed Systems, pp 106–115
17. Nodine MH, Ramaswamy S, Zdonik SB (1992) A Cooperative Transaction Model for Design Databases. In: Database Transaction Models for Advanced Applications, pp 53–85
18. Oster G, Skaf-Molli H, Molli P, Naja-Jazzar H (2007) Supporting Collaborative Writing of XML Documents. In: ICEIS (4), pp 335–341
19. Ritter N, Mitschang B, Härder T, Gesmann M, Schöning H (1994) Capturing Design Dynamics the Concord Approach. In: ICDE, pp 440–451
20. Wächter H, Reuter A (1992) The ConTract Model. In: Database Transaction Models for Advanced Applications, pp 219–263
21. Weikum G, Schek HJ (1992) Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In: Database Transaction Models for Advanced Applications, pp 515–553
22. Weikum G, Vossen G (2001) Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers

Francis Gropengießer was born in Nordhausen, Germany, in 1982. He received his diploma (M.Sc.) from the Ilmenau University of Technology, Germany, in the beginning of 2008. Thereafter, he started his Ph.D. studies as a research associate with the Databases & Information Systems Group at the same university.

Katja Hose studied Computer Science at Ilmenau University of Technology and graduated in 2004. Afterwards, she joined the Databases & Information Systems Group in Ilmenau as a research associate. Her research interests are cooperative XML data management and distributed query processing in P2P networks.

Kai-Uwe Sattler is full professor and heads the Database and Information Systems group at the Faculty of Computer Science and Automation of the Ilmenau University of Technology, Germany. He received his Diploma (M.Sc.) in Computer Science from the University of Magdeburg, Germany. He received his Ph.D. in Computer Science in 1998 and his Habilitation (venia legendi) in Computer Science in 2003 from the same university. He has published five textbooks and more than 100 research papers. His current research interests include autonomic features in database systems and large-scale distributed data management.