An Extended Cooperative Transaction Model for XML

Francis Gropengießer, Kai-Uwe Sattler
Department of Computer Science and Automation
TU Ilmenau
P.O. Box 100565, D-98684 Ilmenau, Germany
{francis.gropengiesser|kus}@tu-ilmenau.de

ABSTRACT

In many application areas, for example in design or media production processes, several authors have to work cooperatively on the same project. Thereby, a frequently used data format is XML. In this paper, we address the special requirements of cooperative working on shared XML graph structures, such as early visibility of updates, multi-directional information flow, and parallel working. Since most existing transaction models are hardly applicable, we present a novel transaction model based on multi-level transactions and dynamic actions that meets these requirements. Additional advantages of this model are appropriate concepts for transaction synchronization and resolution of conflicts.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency, Transaction processing*; E.1 [**Data**]: Data Structures—*Trees*

General Terms

Design, Management, Theory

Keywords

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

1. INTRODUCTION

In many application scenarios, for example in media production, users need to work cooperatively on the same project. Within a cooperative workflow, every user is equal and they are all aware of 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 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.

PIKM'08, October 30, 2008, Napa Valley, California, USA.

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. Figure 1 shows an example scene graph.

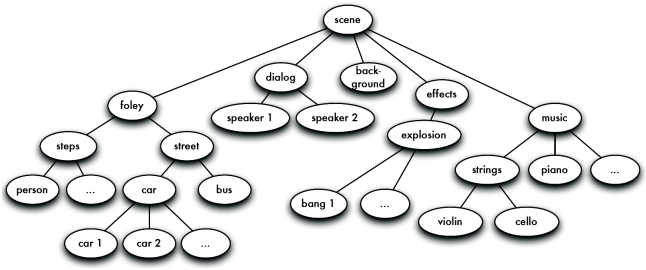

Figure 1: Example scene graph

In order to support transaction-oriented but cooperative working of several authors on the same XML data, we first have to consider the characteristics of media production processes, which are similar to the characteristics of common design processes, e.g., in CAD environments [13]. The duration of the transactions containing the operations of the authors is typically rather long. This results on the one hand in a late visibility of changes made by an author and on the other hand in a loss of up to a whole workday if a transaction has to be aborted. However, cooperation requires early visibility of changes and the possibility to discard single work steps.

A main problem of cooperation is that it conflicts with the serializability property. Cooperation means a multi-directional information flow while serializability only allows an uni-directional one. Relaxing this property might result in conflicts, which we have to treat accordingly.

Summarizing, a transaction model for cooperative media production processes has to fulfill at least the following requirements:

1. It has to enable an early visibility of changes as well as a multi-directional information flow. Thus, we have to abstain from isolation and even serializability.

2. Inconsistencies resulting from, for example, non repeatable reads, lost updates, or phantoms, and other

conflicts, which can occur due to lack of serializability have to be avoided or resolved.

3. The transaction model has to offer the possibility to discard single work steps. This requires relaxing atomicity.

4. Another challenge is to enable the work of several 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 and an appropriate synchronization concept, which together meet the requirements stated above.

2. RELATED WORK

Our work is based on extended transactions, which address the problems of long running transactions stated in Section 1. However, these extended transaction models are only applicable to the special use case described in Section 1 in a limited way, as shown subsequently.

Closed nested transactions [9, 10] require isolation between whole transaction trees and siblings within one transaction tree. (A transaction tree consists of a root transaction and several levels of sub transactions.) If every author is assigned one transaction tree, this means that changes of an author are only visible to others after the root transaction has committed. Thus, a cooperative workflow is inhibited.

Open nested transactions as well as their specialization, multi-level transactions [16], support cooperative working by relaxing the isolation property. One author could see the changes of another author before this 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 have very complex or even no reverse operations. In [5], problems of compensating transactions are discussed in more detail.

The Saga concept [3, 9] is based on open nested transactions. It relaxes the isolation property, too. However, to prevent possibly evolving cascading abortions and therefore reduce transaction recovery complexity, commutativity between sub transactions of different Sagas is required. Thus, no "real" information flow is possible between different authors, which means that no cooperative working can take place.

The ConTract model [9, 15] is similar to the Saga concept, but it enables cooperative working of several authors, because it does not require commutativity. However, the use of compensating transactions is required for transaction recovery.

The DOM transaction model [1, 9] is an approach to combine open and closed nested transactions. This concept offers the possibility of open nesting of closed nested transactions. Thus, cooperative working is enabled. But again, the use of compensating transactions is required.

The CONCORD model [9, 14] is based on object versioning. Every 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, respectively. Furthermore, in VCS an update call is necessary after a commit, in order to retrieve the current state of an object at all. In both models there are no fixed time points when changes are propagated. 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 this way they are able to decide about further work steps within the project.

The dynamic action model [11] extends the traditional transaction model with an additional state *completed*, as shown in Figure 2. 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 abortion of actions it depends on. Only if it is proved, that all actions it depends on are committed, this action can commit, too. 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.

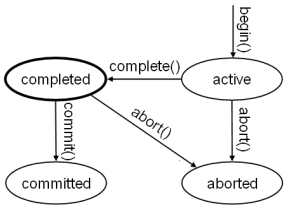

Figure 2: Dynamic action state model

Nested dynamic actions [9, 12] have an architecture similar to the closed nested transactions described above. They require serializability between whole action trees. If every author is assigned one action, this means that there is only an uni-directional information flow possible between the authors. Thus, cooperative working is prevented.

Nested dynamic actions for cooperative applications [9] abolish serializability within groups of action trees, also called cooperation groups. Thus, cooperative working is enabled. However, this principle is impractical for our use case due to the following fact. 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 result from the lack of serializability, needs the intervention of the authors. They have to specify objects for which a cooperative access has to be forbidden, because otherwise a conflict would occur. Such a low-level intervention cannot be expected from a designer.

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

3. 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 $T = (N, E, nl, nv)$. Thereby, N is the set of nodes, which are represented by database wide unique IDs. 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 single 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 [6]. In taDOM, attributes are represented as single nodes grouped under an additional node *attribute root node*.

All XML documents are stored in an XML database for further processing. Since we do not want to treat forrests, we assume the XML database to be a tree $DB = (N', E', nl', nv')$. DB is constructed as shown in Equations 1 – 7. Thereby, $T_i = (N_i, E_i, nl_i, nv_i)$, $N_{v_i} \subseteq N_i$, is one 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 of $(node, node\ label)$ pairs nl' consists of all such pairs of all XML documents and the additional labeling pair for $DBroot$ (Equation 6). The last equation (7) states that the set of $(node, node\ value)$ pairs nv' consists of all such pairs of all XML documents.

Summarizing, we give a short example of a database tree DB:

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

Figure 3 depicts the described example tree DB.

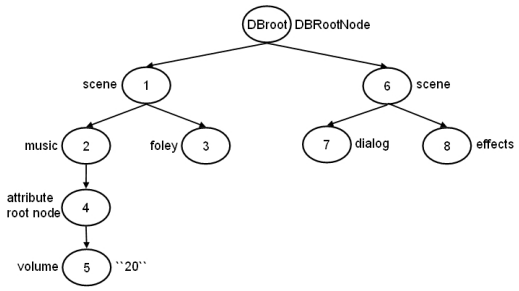

Figure 3: Example database tree DB

4. OPERATIONS AND OPERATION SEQUENCES

One goal of our work is to increase the degree of parallel working of several authors on shared XML documents. Implicitly, this means reducing the conflict set of operations. However, this cannot be achieved with the simple read/write model, since it is too restrictive. Thus, we extend this model with semantic tree operations.

First, we specify low-level tree operations. They are used for directly processing the XML tree.

Low-level reading of nodes or edges is implemented by the following operations:

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

Low-level updating of nodes or trees is implemented by the following operations:

- The operation $edit(n, new\ node\ value)$, where $n \in N'_v$ and $new\ node\ value \in NV'$, is used for editing the content of a node. Thereby, the existing pair $(n, node\ value)$ is removed from nv' and the new pair $(n, new\ node\ value)$ is added to nv'.
- The operation $move(n, m)$, where $n, m \in N'$ and $n \neq m$, assigns m as new *parent* to n. Thereby, the existing edge (p, n) between the parent p of n and n is removed from E' and the new edge (m, n) is added to E'.
- With $insert(n, p, node\ label)$, where $p \in N'$, $n \notin N'$ and $node\ label \in NL'$, a new node n can be added to p. Thereby, the operation extends N' with n, E' with the pair (p, n), and nl' with the pair $(n, node\ label)$.
- The operation $delete(n)$, where $n \in N'$ and n has no children, removes a node from N' and the edge (p, n) between the parent p of n and n 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'.

Regarding *move* and *delete*, it should be noted that all accessable nodes have a parent due to the exclusive node $DBroot$.

Next, we specify high-level operations. They are useful within the application layer to give an author comfortable tools to, for example, read a subtree or delete a subtree. (A $subtree(n)$ is a part of a tree rooted at node n, which comprises n, all descendants of n, and the corresponding edges.) However, these methods are not directly executed on the tree but mapped to low-level operations. This offers the possibility to abort single parts of an operation on a subtree without aborting the entire operation. The reason for this becomes clear in the following sections.

High-level reading of nodes or subtrees is implemented by the following operations:

- With operation $readNode(n)$, $n \in N'$ a node is read. This operation is implemented either by a $strReadNode(n)$ followed by a $contReadNode(n)$ or only by a $strReadNode(n)$, depending on the locks already applied to n. We will describe this in more detail in Section 6.
- The operation $readSubtree(n)$, $n \in N'$ reads all *accessible* nodes and edges within the subtree rooted at 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. We will describe this in more detail in Section 6 .

High-level updating of nodes or trees is implemented by the following operations:

- The operation $insertSubtree(S, n)$, $n \in N'$, can be used to insert a tree $S = (N'', E'', nl'')$ as subtree of node n. This method is implemented by single low-level *insert* operations. Thereby, a certain order is defined. All nodes of S with their corresponding edges are inserted from *top to bottom*. Thus, we start with the root of S and finish with the $leafs$ of S. (A $leaf$ is a node which is not a source of edges.) For the insertion of nodes of the same level within S we assume no ordering.
- With $deleteSubtree(n)$, $n \in N'$, we can delete a whole subtree, rooted at n, from a tree DB. This method is implemented by low-level *delete* operations on every node of the subtree. Thereby, a certain order is defined. All nodes of the subtree with their corresponding edges are deleted from *the bottom to the top*. Thus, we start with the leafs and finish with the root. For the deletion of nodes of 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 issued, we make some restrictions to guarantee several properties mentioned below. Thus, a valid operation sequence is constructed according to the grammar given in Equations 8 – 12. Thereby, $m, n, r, t, u, v, w \in N'$ and $k \notin N'$ are nodes. Furthermore, it holds that neither $subtree(n) \subseteq subtree(m)$ nor $subtree(m) \subseteq subtree(n)$. Node t is a part of $subtree(n)$, v and r are descendants of n, $v \neq t$, r is a $leaf$, u is a part of $subtree(m)$, and w is not a part of $subtree(n)$.

$$< OS > ::= < O_1 > | < O_2 > | < O_3 > | < O_4 > \quad (8)$$

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

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

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

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

Each operation sequence is isolated from each other with the help of the synchronization model described in Section 6. Changes are propagated right after an operation sequence was finished.

Using such operation sequences has several advantages. Before changing a data item it is always read. Thus, there exist no "blind" updates. Together with the isolation of an operation sequence this prevents the authors from inconsistencies resulting from lost updates, non repeatable reads, or phantoms (Requirement 2) because update operations only depend on the data actually read in the previous step. Other conflicts, for example on a semantic level, can also be discovered by reading the data before changing it. To resolve them, an author can abort the conflicting operations of another author. A further advantage of using operation sequences is that changes are visible after at most one high-level update operation. This increases concurrency (Requirement 4).

5. TRANSACTION MODEL

Our transaction model is based on dynamic actions as well as multi-level transactions with a maximum tree depth of four. We start with an example to give a short impression of how it is structured. Assume, a user executes the operation sequence OS (Equation 13) on the data structure depicted in Figure 3:

$$< OS > ::= readSubtree(6)\ delete(8) \quad (13)$$

The resulting transaction structure is shown in Figure 4. As

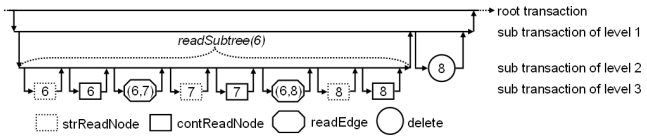

Figure 4: Example transaction structure

it is illustrated, the structure consists of the following four transaction levels:

- **Root transaction:** This transaction is started by the author and ends when he decides to finish his work. Within this transaction no operation is executed. All operations are wrapped into sub transactions. If the root transaction is aborted, subsequently all of its sub transactions are aborted, too. However, sub transactions can be aborted without causing the abortion of the root transaction. Note, that this property fulfills Requirement 3.
- **Sub transaction of level 1-3:** Each operation sequence is represented by one sub transaction of level 1. The principle is, as mentioned in the last section, to divide operation sequences into single operations on subtrees and single operations on subtrees into single operations on nodes. This results in at most two more sub transaction levels. If a sub transaction aborts, all ancestor sub transactions are aborted as long as the sub transaction of level 1 is not in the state *completed*. This ensures that an operation sequence is either completely executed one time or not at all. However, after the completion of the sub transaction of level 1, descendant transactions of this sub transaction can be aborted. (Ancestor and descendant transactions are defined in Section 7.) This reduces the number of cascading aborts. Changes resulting from sub transactions of level 2 or 3 are only visible to their parent sub transactions. However, after a sub transaction of level 1 was 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).

This description of the transaction model is quite concise. We describe it with the help of some ACTA Axioms in more detail in Section 7.

6. SYNCHRONIZATION

In order to synchronize concurrent operations a lock-based approach is applied, which is inspired by the ideas presented in [4, 7, 8]. However, other synchronization methods, for example optimistic models, are also conceivable.

In order to identify lock types and their compatibility, we need to consider the list of supported tree operations that were introduced in Section 4. Since high-level operations are not directly executed, we only have to treat the 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 for the *edit* operation. The lock type DL is assigned to the *delete* operation. IL denotes the lock type for the *insert* operation. Additionally, we need an *intentional structure change lock* ISCL. Its use is described below.

The compatibility of the different lock types and the according operations can be summarized in a compatibility matrix. Since there exist operations on nodes and edges, we have to distinguish conflicts on nodes and conflicts on edges within two compatibility matrices. Table 1 considers the first case. The rows show the locks already applied to the node and the columns show the locks requested on this node. Symbol $\surd$ means that the locks are fully compatible and $-$ means that they are not compatible. Furthermore, $+$ denotes that two *insert* operations are only compatible if ordering of the inserted nodes is negligible. Table 2 shows

	SRL	CRL	EL	DL	IL	ISCL
SRL	$\surd$	$\surd$	$\surd$	$-$	$\surd$	$\surd$
CRL	$\surd$	$\surd$	$-$	$-$	$\surd$	$\surd$
EL	$\surd$	$-$	$-$	$-$	$\surd$	$\surd$
DL	$-$	$-$	$-$	$-$	$-$	$-$
IL	$\surd$	$\surd$	$\surd$	$-$	$+$	$\surd$
ISCL	$\surd$	$\surd$	$\surd$	$-$	$\surd$	$\surd$

Table 1: Lock compatibility matrix with respect to a node

the compatibility matrix with respect to an edge. The rows, again, show the locks already applied to an edge and the columns show the locks requested on this edge.

	ERL	DL
ERL	$\surd$	$-$
DL	$-$	$-$

Table 2: Lock compatibility matrix with respect to an edge

Next, we give a short insight in how the locking protocol works. First, we take a closer look on what lock types are needed for each operation:

- The operations $strReadNode(n)$, $contReadNode(n)$, $readEdge(m,n)$, and $edit(n, node\ value)$ only need the according lock type on the affected node n or edge (m,n).
- The $insert(n,p,node\ label)$ and $insertSubtree(S,p)$ operations only require an IL on the specified destination node p. For the inserted node n or subtree S no locks are required, since we assume that the inserted node or subtree only appears after the insertion was completed. Thus, until then they are not accessible for other users/transactions.
- The operation $delete(n)$ requires a DL on the affected node and on the edge between the parent node and this node. Furthermore, an ISCL has to be acquired for the parent node. This prevents the parent from being deleted by another user. This is necessary, since we assume that an edge can only exist if both corresponding nodes exist. If a whole subtree shall be deleted with the help of the operation $deleteSubtree(n)$, all descendants of n with their corresponding edges have to be locked with DL, too.
- The operation $move(n,m)$ requires a DL on the edge $(parent(n),n)$, an ISCL on the parent of n and n itself, and an IL on the destination node m. The ISCL on n and its parent node prevents 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 a SRL on node n. This implicitly means that a CRL on n and SRL, CRL, and ERL on all descendant nodes with their corresponding edges are optional. The operations on nodes or edges for which the appropriate lock types could not be acquired are simply not executed. Assume that a user executes $readSubtree(1)$ on the example data structure in Figure 3 while another user performs $delete(3)$. Proceeding in the described way allows the first user to read all *accessible* nodes $(1, 2, 4, 5)$ and the 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* returns a connected set of nodes. This means, for example, the user who performs $readSubtree(1)$ cannot see $subtree(4)$ if another user is moving node 4 at the same time.

In the following, we describe the process of acquiring and releasing locks within the transaction model with the help of the example operation sequence in Equation 13. As illus-

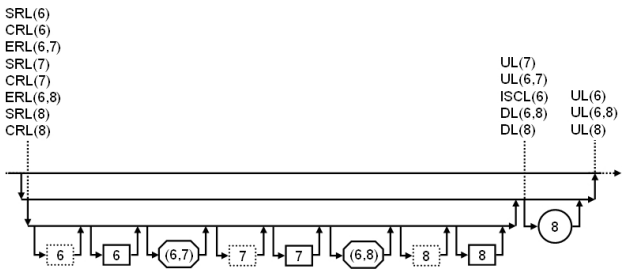

Figure 5: Acquiring and releasing locks

trated in Figure 5, locks are acquired and released according to the following scheme:

- The necessary read locks are acquired atomically at the beginning of the first sub transaction of level 2.

- Thereafter, the locks are passed to the sub transactions of level 3.

- After having entered the state completed, the sub transactions of level 3 pass the locks back to the sub transaction of level 2, which itself passes them to the sub transaction of level 1 after completion.

- With the beginning of the second sub transaction of 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 sub transaction of level 2 passes the locks back to the sub transaction of level 1, which itself releases them after completion.

Using this synchronization model has several advantages:

- It requires no serializability of root transactions and thus enables a multi-directional information flow (Requirement 1).

- It prevents us, as already mentioned in Section 4, from inconsistencies resulting from lost updates, non repeatable reads, or phantoms (Requirement 2) due to the following facts. 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 sub transaction of level 2, since read locks are hold until they are tightened. Thus, sub transactions of level 1 are isolated from each other.

- Another advantage is that a *readSubtree* operation can be executed partially on those parts of the subtree which 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 which 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, for example, moved and edited at the same time by two different users (Requirement 4).

- Furthermore, atomically requesting locks prevents deadlocks, since a transaction is aborted if it cannot get the requested locks. (Note that this does not conflict with the partial execution of, e.g., a *readSubtree* operation.)

7. FORMAL SPECIFICATION

The formal specification of the developed transaction model is necessary for proving fundamental properties. For this purpose, we use the ACTA framework [2] and introduce the necessary dependencies. We start with some basic definitions.

Definition 1 (nested transaction). A nested transaction can be viewed as a *tree*. The top of the *tree* (*level* 0) is called *root transaction*. All transactions on *level* $1, ..., n$, where $n \in \mathbb{N}$ and n is the maximum tree depth, are called *sub transactions*. The set of *children* of a transaction t consists of all *sub transactions* of $level(t)+1$ which are directly wrapped into transaction t. The set of *descendants* of a transaction t consists of all transactions of $level(t) + 1, ..., n$, $n \in \mathbb{N}$ which are wrapped into t. Thereby, transactions on level n are $leafs$. The *parent* of a transaction t is the transaction of $level(t) - 1$ in which it is directly wrapped into. The set of *ancestors* of a transaction t consists of all transactions of $level(t) - 1, ..., 0$ in which t is wrapped into. A *sub transaction tree* t of a *nested transaction* T consists of *sub transaction* $t \in T$ and all of its *descendants*.

Definition 2 (history). A history H contains all events invoked by the transactions. Furthermore, these events

are ordered within H according to their execution order. The predicate $e_1 \rightarrow e_2$ indicates, iff it is *true*, that the event e_1 precedes e_2 and that both events are in H. $P(H,p) = H_p$ is the projection of H on p. Thereby, p may be, for example, 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 results from executing H_{da} on the initial state s_{da_0} of the data item. The $return(s_{da}, o[da]) = Output$ is the output of operation $o[da]$ when executed on da. Thereby, the output depends on the current state s_{da} of da.

Now, we describe the transaction model with the help of ACTA Axioms. Due to space limitations, we present only an extract of the whole specification.

Definition 3 (transaction model).

$$t \text{ is a root or a sub transaction} \quad (14)$$

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

$$r \text{ is a root transaction} \quad (16)$$

$$s \text{ is a sub transaction} \quad (17)$$

$$Desc_t \text{ is set of all descendants of } t \quad (18)$$

$$Anc_s \text{ is set of all ancestors of } s \quad (19)$$

$$Childs_t \text{ is set of all children of } t \quad (20)$$

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

$$SE_t = \{Begin_t,\ Complete_t,\ Abort_t,\ Commit_t\} \quad (22)$$

$$(Begin_t \in H) \Rightarrow (\neg(Begin_t \rightarrow Begin_t) \wedge \neg(Complete_t \rightarrow Begin_t) \wedge \neg(Commit_t \rightarrow Begin_t) \wedge \neg(Abort_t \rightarrow Begin_t)) \quad (23)$$

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

$$(Abort_t \in H) \Rightarrow ((Begin_t \rightarrow Abort_t) \wedge \neg(Commit_t \rightarrow Abort_t)) \quad (25)$$

$$(Commit_t \in H) \Rightarrow ((Complete_t \rightarrow Commit_t) \wedge \neg(Commit_t \rightarrow Commit_t) \wedge \neg(Abort_t \rightarrow Commit_t)) \quad (26)$$

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

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

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

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

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

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

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

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

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

Axiom 22 shows the special event set SE of a transaction (dynamic action). Axiom 23 states that a transaction is only allowed to begin if it has not already begun and has not been completed, committed, or aborted. Furthermore, a transaction can only complete if it has begun and has not been completed, committed, or aborted (Axiom 24). A transaction can only be aborted if it has begun and has not been committed (Axiom 25). Note that a transaction which is in state completed can still be aborted. Axiom 26 states that a transaction can only commit if it has been completed and has not already been committed or aborted. A sub transaction can only begin if its ancestor transactions are active (Axiom 27). The next Axiom (28) states that all child transactions of a transaction t are executed in a sequential order. If a sub transaction wants to proceed to state completed, all its descendant transactions must already be in this state and must not have been aborted (Axiom 29). In contrast, a root transaction can be completed even if their descendant transactions have been aborted (Axiom 30). However, in both cases the descendant transactions must not be active. If a transaction is aborted, all descendant transactions are aborted, too (Axiom 31). A transaction t is *vital* to a transaction t' iff t' is a sub transaction, 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'. The next Axiom (33) states that a transaction t is information dependent (ID) on t' iff both transactions execute at least one operation on a shared object. Thereby, the output or the produced state of the operation in transaction t depends on the state that is produced or returned by the operation of transaction t'. For example, an *edit* operation depends on the result of the preceding *contReadNode* operation on the considered node. Thereby, *contReadNode* returns the *state* of the node, which is nothing more than the *node value*. This implicitly means that the *contReadNode* operation depends on the *state* (*node value*) of the node which has been produced by another previous *edit* operation. This results in chains of information dependent transactions. Note that in our model such dependencies only exist between sub transactions of level 3 and 4, as only these transactions are allowed to execute operations. A transaction t' is 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. A transaction can only commit if all transactions it is *abort dependent* on are committed or commit together with the considered transaction (Axiom 35).

8. PROPERTIES

In this section, we describe some properties of the developed transaction model and synchronization concept, starting with a closer look at the ACID properties.

Atomicity is relaxed since sub transactions can abort without the abort of the root transaction (Requirement 3). However, *failure atomicity* can be guaranteed with an appropriate transaction recovery model. Since our model is based on dynamic actions, which guarantee *durability* [11], the recovery model can be chosen freely. Thus, it is possible to, e.g., apply an object versioning model instead of compensating transactions.

Preserving *consistency* means meeting certain integrity constraints. The solely defined *integrity constraint* is preserving the document structure. Meeting this constraint is possible by simply checking if all operations, even in the case of an abort, keep the structure of the document.

Isolation and even *serializability* is 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 us from inconsistencies resulting from lost updates, non repeatable reads, or phantoms. Other conflicts that could occur can be resolved by aborting the conflicting transactions (Requirement 2).

Durability is guaranteed, since the model is based on dynamic actions, which themselves fulfill this property [11].

Besides the ACID properties we highlight some further features of the model. The exploitation of the semantics of the special tree operations allows a fine grained conflict specification and thus increases the degree of parallelism in the workflow (Requirement 4). Hence, it is possible that several operations, for example editing and moving nodes, can be executed in parallel on the same node.

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

Finally, dividing operations on subtrees into operations on nodes in combination with wrapping these node operations into single sub transactions allows fine grained information dependencies management. Hence, it is possible to abort parts of, e.g., an *insertSubtree* operation (Requirement 3). This also reduces the number of cascading aborts.

9. CONCLUSION

In this paper, we presented a novel multi-level 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. To achieve this goal we had to abstain from serializability. However, isolating single operation sequences and reading data before updating it prevents us from inconsistencies resulting from conflicts like lost updates, non repeatable reads, or phantoms. Other conflicts can be resolved by aborting the conflicting transactions. To increase the degree of parallelism we extended the simple read/write model with semantic tree operations. Thus, e.g., *move* and *edit* operations can be supported even on the same node. Finally, this paper also presented an extract of the formal specification of the developed transaction model.

Future work comprises the formal proof of correctness of the transaction and synchronization model as well as the development of a transaction recovery strategy. A next step will be the implementation of the transaction model, locking protocol, and recovery model as part of a whole design application. Thereby, the analysis of different system architectures will be an interesting point. Furthermore, the investigation of other synchronization techniques, e.g., an optimistic one, is a challenging research issue.

10. REFERENCES

[1] A. P. Buchmann, M. T. Ozsu, M. Hornick, D. Georgakopoulos, and F. A. Manola. A transaction model for active distributed object systems. In *Database Transaction Models for Advanced Applications*, pages 123–158, 1992.

[2] P. K. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Model. In *VLDB '91*, pages 103–112, 1991.

[3] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87*, pages 249–259, 1987.

[4] T. Grabs, K. Böhm, and H.-J. Schek. Xmltm: efficient transaction management for xml documents. In *CIKM '02*, pages 142–152, 2002.

[5] P. Greenfield, A. Fekete, J. Jang, and D. Kuo. Compensation is Not Enough. In *EDOC '03*, page 232, 2003.

[6] 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.

[7] M. P. Haustein and T. Härder. Adjustable transaction isolation in xml database management systems. In *XSym*, pages 173–188, 2004.

[8] S. Helmer, C.-C. Kanne, and G. Moerkotte. Evaluating lock-based protocols for cooperation on xml documents. *SIGMOD Rec.*, 33(1):58–63, 2004.

[9] A. Kruse. A cooperative lock protocol for nested dynamic actions — diploma thesis at the university of Bonn (in German). March 1997.

[10] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, 1981.

[11] E. Nett and M. Mock. How to Commit Concurrent, NonIsolated Computations. In *IEEE Workshop on Future Trends in Distributed Systems, Korea*, 1995.

[12] 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.

[13] M. H. Nodine, S. Ramaswamy, and S. B. Zdonik. A cooperative transaction model for design databases. In *Database transaction models for advanced applications*, pages 53–85, 1992.

[14] N. Ritter, B. Mitschang, T. Härder, M. Gesmann, and H. Schöning. Capturing Design Dynamics the Concord Approach. In *ICDE*, pages 440–451, 1994.

[15] H. Wächter and A. Reuter. The contract model. In *Database transaction models for advanced applications*, pages 219–263, 1992.

[16] 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.