A Transaction Model for XML Databases

Stijn Dekeyser, Jan Hidders and Jan Paredaens
University of Antwerp

June 14, 2003

Abstract. The hierarchical and semistructured nature of XML data may cause complicated update behavior. Updates should not be limited to entire document trees, but should ideally involve subtrees and even individual elements. Providing a suitable scheduling algorithm for semistructured data can significantly improve collaboration systems that store their data — e.g. word processing documents or vector graphics — as XML documents. In this paper we show that concurrency control mechanisms in CVS, relational, and object-oriented database systems are inadequate for collaborative systems based on semistructured data. We therefore propose two new locking schemes based on *path locks* which are tightly coupled to the document instance. We also introduce two scheduling algorithms that can both be used with any of the two proposed path lock schemes. We prove that both schedulers guarantee serializability, and show that the conflict rules are necessary.

Keywords: XML, semistructured data, path lock, serializability, scheduler, concurrency control

1. Introduction

Semistructured data (Abiteboul, 1997; Abiteboul et al., 1999) is an important topic in Information Systems research that has been studied extensively — especially regarding query languages (Abiteboul et al., 1997; Buneman et al., 2000; Florescu et al., 1997) — in the past and which has regained importance due to the popularity of XML (Bray et al., 2000). Even though XML is not meant to replace traditional database systems, lately an interest in native XML databases has surfaced. Consequently, all features present in relational and object-oriented databases are revisited in the context of semistructured data. One such feature is the inclusion of an update language in XML databases. Possibly due to the existence of the DOM (Wood et al., 1998) interface which enables the user to change XML data in a procedural way, relatively little research has been done regarding update languages for XML; current proposals are (Lehti, 2001; Tatarinov et al., 2001).

A second feature that must be present in any XML database, is a concurrency control mechanism that allows multiple users to query and update a document simultaneously. Concurrency control (Weikum and Vossen, 2002) has been extensively studied in the context of traditional database management systems such as RDBMSs and OODBMSs

(Bernstein et al., 1987; Gray, 1978; Papadimitriou, 1986). This literature has introduced important concepts such as locks, transactions, schedulers, etc. Protocols such as two phase locking have been proposed to ensure serializability (Eswaran et al., 1976) — the central theoretical notion of correctness for concurrent database systems — of schedules of data manipulation language actions. Also, locking mechanisms such as predicate locking (Eswaran et al., 1976), hierarchical locking (Gray et al., 1975; Gray et al., 1976) and tree based locking protocols (Silberschatz and Kedem, 1980) have been introduced to suit special needs and to increase the level of concurrency that is allowed by schedulers.

The result of this research has been the incorporation of efficient scheduling algorithms in traditional DBMSs which ensure serializability and which allow a high degree of concurrency.

In contrast, research in the field of semistructured databases so far has been focussed on single-user environments, where documents are created, stored and altered by one user. Thus, if one wants to collaborate on an XML document, an existing collaboration system such as CVS (Cederqvist et al., 1993) has to be used. These systems, however, have a very coarse granularity and do not make use of locking schemes — multiple users can check out and make changes only upon entire documents simultaneously. The CVS system will then attempt to resolve conflicts automatically, and if it fails to do so, solicit help from the user.

Using concurrency control methods of existing systems As already mentioned, relational database systems have long since dealt with update conflicts in a satisfactory manner. Consequently, one way to offer concurrent update access to an XML document would be to store the document in relational database tables, and then use the existing locking scheme from the RDBMS. Unfortunately, as we will show in Section 3, this method — while guaranteeing serializability — often causes locks that are too restrictive.

Several existing XML-enabled relational database systems, such as Oracle 9i and Microsoft SQL Server, do use either variants of the traditional relational locking mechanisms, or provide conflict detection based on optimistic concurrency control.

Object-Oriented databases (Khoshafian, 1993) also offer concurrency control. A significant portion of research (Barghouti and Kaiser, 1991; Loomis, 1995), however, deals with supporting long-lived transactions, thereby relaxing the serializability requirement. Of more interest to our work is the research that looks at locking hierarchical objects (Cellary et al., 1988; Khoshafian, 1993). In such systems, locking can be based

on a hierarchy of collections, extents, objects etc. Protocols similar to those of relational databases extended with *intentional locks* are used. The remarks made in Section 3, are therefore also valid for OODBMSs.

Finally, the *Lore* database management system for semistructured data (Abiteboul et al., 1997; McHugh et al., 1997) did not contain support for concurrency control that utilizes semantical information, although this was mentioned in "future work". Instead, it used page-based strict two phase locking. The authors of Lore pointed out that "the semistructured nature [would] require us to rethink some aspects of traditional solutions" to concurrency control.

Contribution Since we cannot simply put XML documents in relational or object-oriented databases without losing a high degree of concurrency, what is needed is a profound adaption of the work done in the field of these traditional databases to suit the needs of a semistructured database.

In this paper we therefore propose a simple data model to represent an XML document and detail two perspectives on the transaction model for an XML database. We also propose two new locking schemes based on *path locks* and introduce two scheduling algorithms that can both be used with any of the two proposed path lock schemes. We prove that both schedulers guarantee serializability, and show that the conflict rules are necessary.

Organization In Section 2 we present a running example that will appear throughout the paper, and also introduce two use cases for concurrency control for semistructured data. We show in Section 3 that traditional solutions from relational and object oriented database systems are not sufficient to support the use cases. This result gives the motivation for the research presented in this paper.

Section 4.1 gives the first of two perspectives on the data model and the query- and update language; this perspective is for users manipulating XML documents. In Section 4.2 we present the second perspective, to be used by the XML database's scheduling algorithm. We therefore also define the notions of actions, transactions and schedules.

Section 5 introduces two different locking schemes that describe which operations in the transactions need which locks and defines when such locks conflict. The complexity of checking for conflicts in both schemes is given, and we show that the two schemes are equivalent regarding the conflicts they detect.

The definitions of two schedulers are given in Section 6. In Section 7 we show that both schedulers guarantee their output schedules to be serializable. We also characterize the data structure needed by

the schedulers, and show that the conflict rules given in Section 5 are necessary. We end with a short conclusion in Section 8, in which we also briefly outline possible future work.

2. Running Example and Use Cases for Concurrency

In this paper we will use a running example to clarify important concepts.

EXAMPLE 1 (Running Example). *Figure 1 shows a part of an XML document representing genealogical information.*

We assume no given DTD for the document; the scheduling algorithm makes use of the instance, not the schema of the database.

```
<doc>
 <person id="1", age="55">
      <name>Peter</name>
      <addr>Parkl7</addr>
      <child>
           <person id="3", age="22">
                <name>John</name>
                <addr>Unistr1</addr>
                <hobby>swim</hobby>
                <hobby>cycling</hobby>
           </person>
      </child>
      <child>
           <person>
                <name>David</name>
           </person>
      </child>
 </person>
 <person id="2", age="43", spouse="1">
      <name>Mary</name>
      <hobby>paint</hobby>
 </person>
</doc>
```

Figure 1. A fragment of an XML document D.

To show that existing concurrency control methods are insufficient, and to illustrate what our transaction model contributes, we will make use of two use cases.

2.1. Use Cases for Concurrency

The use cases describe transactions from different users or processes that need to be allowed to run concurrently. This means that the individual actions of these transactions should not conflict with one another.

USE CASE 1. *Consider that a user U has accessed document D of Figure 1 and has "seen" (queried) elements* `<hobby>` *appearing at any level under elements* `<child>` *which themselves can be found at any depth in the XML tree representing D. Stated differently, using the XPath surface syntax, user U has issued the query* `//child//hobby`*. As the answer to this query, user U has received all node-identifiers of elements* `<hobby>` *that appear under some element* `<child>`*. In this case, these hobbies correspond to hobbies* swim *and* cycling*, but not to* paint *since that* `hobby` *does not appear under a* `child` *element.*

It should now be possible for a second user V to make changes to the `<hobby>` *element representing* paint *(because it does not appear in the result of U's query); e.g., changing it to* `painting`*.*

Use case 1 assumes that the query language can express simple XPath queries that make use of the `child-of` and `descendant-of` axes. No branching (use of conditions) is needed. The update language in this scenario is assumed to be able to change the CDATA between element tags. In particular, the element to be changed is a leaf element.

USE CASE 2. *Consider that a user U has accessed document D of Figure 1 and has "seen" (queried) elements* `<hobby>` *appearing at any level under elements* `<person>` *which themselves are children of the* `<doc>` *element. This in effect means that user U has issued the XPath query* `/doc/person//hobby`*.*

It should now be possible for a second user V to create a `<person>` *element which is a child of the element* `<doc>`*.*

On the other hand, if V is allowed to subsequently create a `<hobby>` *element at any level under this newly created* `<person>`*, this may lead to inconsistencies.*

The second use case implies the same assumptions with regard to the query language as the first use case. The update language in this scenario is assumed to be able to create leaf elements.

3. Insufficiency of Existing Concurrency Control Methods

In the introduction we have described how current document collaboration systems such as CVS do not offer a locking scheme of adequate granularity. In this section, we show that storing XML data in a relational database and using the RDBMS's concurrency mechanisms is also insufficient. Stated differently, we show that the use cases fail.

Semistructured data can be stored in relational databases in many different ways (Deutsch et al., 1999; Florescu and Kossmann, 1999). For all these different representations, the locking mechanisms of RDBMSs may cause locks that are too restrictive. Central in our proof is the fact that the parent-child relationship is typically modelled within one relation. Though this relationship can indeed be split up in several tables, in general one table will — because of XML's semistructured nature (elements can exist at any nesting depth) — contain arbitrarily many tuples that model the parent-child relationship. For our discussion it is therefore appropriate to abstract this into one relation.

We will examine the following locking mechanisms utilized by relational database systems. First, we start with the simplest case where entire tables are locked to prevent *phantoms* (Date, 2000) from occuring. Secondly, as an improvement to the first system, we investigate predicate locks. To conclude the comparison with RDBMSs, we look at intention locks and tree locking.

3.1. Table locking

In this approach the entire table representing the hierarchy will be locked in case of query that traverses the hierarchy. If this were not the case, a phantom could occur; i.e., an INSERT update could generate a tuple that should be in the result of the original query. Since we require serializability, we cannot permit phantoms.

A lock on the entire parent-child relationship table makes it impossible, however, to add an element in a subtree that has not been read by any user. This will be made clear in the following property.

PROPERTY 1. *Use Case 1 fails when document D is represented by an* Edge Table *(*Florescu and Kossmann, 1999*) in an RDBMS, and the database system employs the table locking mechanism.*

Proof. Using the *Edge Table* approach proposed by (Florescu and Kossmann, 1999) to convert XML data to the relational model, we get the table presented in Table I.

Table I. A fragment of the Edge Table for XML document D.

source	ordinal	name	target
1	1	age	55
1	5	child	**3**
1	6	child	**4**
3	5	hobby	swim
3	6	hobby	cycling
2	4	hobby	paint
...	...	...	...

We are obliged — since we require serializability and thus do not allow phantoms that could be introduced if row-level locking was used — to lock the entire Edge Table for user U's query.

Using an RDBMS table locking scheme on this and other relational representations of D would prohibit the user V from performing his update because the entire table is locked. □

3.2. Predicate Locks

Predicate locks (Eswaran et al., 1976; Weikum and Vossen, 2002) have been introduced to fix the problem mentioned above. There is no longer any need to lock an entire table; phantoms cannot occur because predicates are given that describe the tuples that have been selected in an INSERT, UPDATE or DELETE query. New or altered tuples that satisfy these predicates cannot be added to the table, thus eliminating the threat of phantoms.

Predicate locks have two problems in view of this work, however. First, they are rarely implemented in commercial relational databases systems because they are prohibitively expensive. Indeed, testing satisfiability for even simple predicates (i.e. consisting of Boolean combinations of comparisons between a field of a tuple and a constant) is NP-complete (Hunt and Rosenkrantz, 1979). Surprisingly, this even holds true for predicates that do not contain disjunction. Thus, storing XML in existing RDBMSs and using the product's locking scheme will almost certainly not offer the benefits of predicate locks.

The second and most important problem is that while predicate locks come very close to capturing the expressiveness of our proposed locking scheme, they still fall short. This is because predicates do not take the

containment hierarchy into account; locks work over the entire table representing the parent-child relationship.

PROPERTY 2. *Use Case 2 fails when document D is represented by an Edge Table in an RDBMS, and the database system employs the predicate lock mechanism.*

Proof. Property 1 presented a partial translation of D to the Edge Table approach. Assume that the root node `<doc>` has the internal element identifier 0 such that in the relational representation of D it is identified by `source='0'`.

When a user poses the query `/doc/person//hobby`, this has the informal meaning that an element `<person>` may be added directly 'under' `<doc>` as long as no `<hobby>` element is later added somewhere under `<person>`.

To evaluate the query in a relational database that uses predicate locks, consider that the query processor starts by reading all the `<person>` children of the `<doc>` node such that in a next phase it can recursively look for `<hobby>` nodes under the `<person>` nodes. This first read query will result in the predicate lock `source='0'` $\wedge$ `name='person'`. Clearly, this is not what we want, since this predicate lock means that no-one can insert a new `<person>` element under the root, regardless of whether a `<hobby>` element gets inserted under it in a later phase or not. □

In (Dekeyser and Hidders, 2002a) we show that one can fundamentally change the process by which predicate locks are set such that the new relational locking mechanism closely mimics the way our proposed system works. However, such extentions require the re-evaluation of each active query each time an update is performed. Additionally, the satisfiability problem remains. For all these reasons, predicate locks in relational database systems are unsuited for the kind of locking mechanism we would like to use for XML documents.

3.3. Hierarchical and Tree Locking Protocols

Hierarchical locking protocols (Gray et al., 1975; Gray et al., 1976), also known as "multigranularity locking protocols", are used for data that can be thought of as nested hierarchical granules and where it is important that we can place locks on granules at different levels in the hierarchy. Usually such protocols allow shared and exclusive locks at different levels but with the restriction that if a granule is to be locked then corresponding intention locks (or stronger locks) must be acquired

for all the ancestors, i.e., the granules that directly or indirectly contain this granule. Additionally, if a granule is to be extended with a new element then an exclusive lock on the granule must be acquired.

PROPERTY 3. *Use Case 1 fails when the database system that stores document D employs the hierarchical locking mechanism.*

Proof. If the hierarchical locking protocol is applied to D then a query like `//child//hobby` will require shared locks on the whole document tree and therefore disallow any update on it by other transactions. □

Object-oriented databases typically implement some version of the hierarchical locking protocol (Khoshafian, 1993). The granules in this case are, for example, classes, extents, objects, etc.

Tree Locking Another type of protocol that is often used for hierarchical data are so-called *tree locking protocols* (Silberschatz and Kedem, 1980). In these protocols locks do not hold for entire granules but only for nodes, i.e., when a node is locked its descendants are not also locked. However, there is the restriction that a lock can only be acquired for a node if an identical or stronger lock was already obtained for the parent of the node. Furthermore, as in hierarchical locking protocols, we need to acquire an (exclusive) lock on a node if we want to add or remove children (Lanin and Shasha, 1986).

Tree Locking protocols are used in multi user operating systems to allow concurrent access to the directory structure.

PROPERTY 4. *Use Case 1 fails when the database system that stores document D employs the tree locking mechanism.*

As for the hierarchical locking protocol, when using tree locking it holds that a query like `//child//hobby` will require shared locks on all the element nodes in the document tree and thereby block any update by other transactions. □

If use case 1 is altered such that the query does not make use of the `descendant-of` axis (e.g., `/doc/person/child/person/hobby`), the tree locking protocol would correctly process use case 1. This does not hold for use case 2, however, which fails whether or not the query makes use of the `descendant-of` axis.

For a comprehensive overview of Hierarchical Locking and Tree Locking protocols, see (Barghouti and Kaiser, 1991).

4. Two Perspectives on the Transaction Model

In this paper we present two perspectives on the transaction model: the user's perspective and the scheduler's perspective which extends the first with the necessary scheduling information. In the next section we give the user perspective before turning to the scheduler's perspective in Section 4.2.

Each perspective defines what transactions are. More precisely, we detail which operations can be used in transactions, and over which data model these operations are defined.

4.1. The User Perspective

Consider that users write programs in a high-level programming language which contains special operations to manipulate an XML document. The user's perspective of a transaction is simply a sequence of such operations.

In the present section, we give a simple model for representing XML documents and introduce a query and update language defined over this simple model. We then proceed with formally defining transactions as seen from the user's perspective.

4.1.1. *Data Model*

The data model we assume for XML documents is a simplification of the standard XPath data model (Fernández et al., 2002), which is itself based on the XML Information Set (Cowan and Richard, 2001). The basic notion in our data model is the xp-tree (XPath tree).

DEFINITION 1 (Xp-tree). *An xp-tree d is a tuple (N, B, r, ν) where N is a set of nodes, $B : N \times N$ is a binary relation representing the directed edges (branches) of the tree d, and $r \in N$ is the root node of d. The function ν maps nodes (except r) to strings representing the node's name.*

Seen from the user's perspective, an XML document is an instance of the xp-tree data model. We briefly discuss the differences between the XML data model and the xp-tree data model.

Translating an XML document to an xp-tree The xp-tree data model resembles the XML data model. For instance, references are treated as in XML: the attributes in which they appear are modelled as ordinary nodes in the xp-tree. However, there are a few important differences.

First, our model lacks an ordering of sibling nodes. This is not crucial, however, since an ordering can be simulated by using a skewed binary xp-tree.

Secondly, a single XML document may be represented by several xp-trees which are equal up to isomorphism. Stated differently, node identifiers in our model are only incidental; they do not remain persistent after a user process is finished manipulating the document.

Finally, in our model we do not distinguish between element, attribute or text nodes. It would be trivial, however, to add a *type* function to the xp-tree model to solve this. For simplicity, we have refrained from doing so in this paper. Instead, text appearing in an XML document is simulated in our model. As a consequence, in our model an 'element' node may contain several 'attributes' of the same name. Likewise, it is possible for several 'text' nodes to be adjacent to each other. Furthermore, updating text is simulated by replacing the node representing the orginal text with a new node that has a different name. In (Dekeyser and Hidders, 2002b) we give an alternative method for treating text and attributes.

We now turn to an example to illustrate the xp-tree concept.

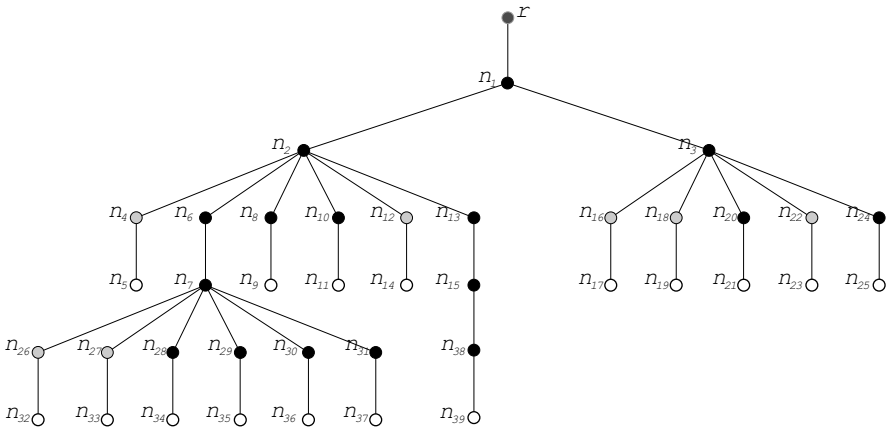

Figure 2. An xp-tree representation of XML document D.

EXAMPLE 2. *Figure 2 shows an xp-tree representation d of document D given in Figure 1. Note that for purely didactical reasons we have made a distinction in the manner in which nodes of different types (elements, attributes, and text) are presented. Thus, black circles denote element nodes, while white circles denote text nodes and gray circles*

represent attribute nodes. The names of the nodes are given in Table II. Note also that the document order is not preserved.

Table II. Node names of the xp-tree shown in Figure 2.

$\nu(n_1) =$ `doc`	$\nu(n_{11}) =$ `Park17`	$\nu(n_{21}) =$ `Mary`	$\nu(n_{31}) =$ `hobby`
$\nu(n_2) =$ `person`	$\nu(n_{12}) =$ `@age`	$\nu(n_{22}) =$ `@spouse`	$\nu(n_{32}) =$ `3`
$\nu(n_3) =$ `person`	$\nu(n_{13}) =$ `child`	$\nu(n_{23}) =$ `1`	$\nu(n_{33}) =$ `22`
$\nu(n_4) =$ `@id`	$\nu(n_{14}) =$ `55`	$\nu(n_{24}) =$ `hobby`	$\nu(n_{34}) =$ `John`
$\nu(n_5) =$ `1`	$\nu(n_{15}) =$ `person`	$\nu(n_{25}) =$ `paint`	$\nu(n_{35}) =$ `Unistr1`
$\nu(n_6) =$ `child`	$\nu(n_{16}) =$ `@id`	$\nu(n_{26}) =$ `@id`	$\nu(n_{36}) =$ `swim`
$\nu(n_7) =$ `person`	$\nu(n_{17}) =$ `2`	$\nu(n_{27}) =$ `@age`	$\nu(n_{37}) =$ `cycling`
$\nu(n_8) =$ `name`	$\nu(n_{18}) =$ `@age`	$\nu(n_{28}) =$ `name`	$\nu(n_{38}) =$ `name`
$\nu(n_9) =$ `Peter`	$\nu(n_{19}) =$ `43`	$\nu(n_{29}) =$ `addr`	$\nu(n_{39}) =$ `David`
$\nu(n_{10}) =$ `addr`	$\nu(n_{20}) =$ `name`	$\nu(n_{30}) =$ `hobby`	

We now define the notion of a *label path*.

DEFINITION 2 (Label Path). *The* label path *from node n_1 to node n_m in an xp-tree d, denoted $\overline{\lambda}_d(n_1, n_m)$ and abbreviated to $\overline{\lambda}(n_1, n_m)$ when it is clear to which xp-tree the label path is associated, is the string "$l_1/\ldots/l_{m-1}$" if there is in d a path from n_1 to n_m such that for all i, $1 \leq i \leq m-1$, the i-th edge in this path ends in a node with label l_i. If $n_1 = n_m$, then the label path is the empty string. If no sequence of edges exists between n_1 and n_m, and $n_1 \neq n_m$, then the label path $\overline{\lambda}_d(n_1, n_m)$ is not defined.*

Now that we have defined the data model, we turn our attention to the data manipulation language. We will start with the query language.

4.1.2. *Query Language*

The user accesses an XML document through the use of a query operation. This operation in turn makes use of XPath-like path expressions. Before turning to the precise semantics of the query operation for our data model, we give the grammar and semantics of our path expressions. We use a limited version of the surface syntax of XPath which is described by the following grammar

$$\begin{aligned} \mathcal{P} &::= \mathcal{F} \mid \mathcal{P}/\mathcal{F} \mid \mathcal{P}//\mathcal{F} \\ \mathcal{F} &::= E \mid * \end{aligned}$$

where E is the universal set of strings representing the names of elements.

Note that in the full XPath surface syntax it is possible to start a path expression with a single or a double slash, the former denoting an absolute path, and the latter denoting the descendant-of axis. In this paper, an absolute path can be easily simulated in the query operation defined later in this section. An XPath expression starting with a double slash can be simulated by using two of our path expressions. For example, an XPath expression `//person/child` can be simulated by the path expressions `person/child` and `*//person/child`.

We now turn to the semantics of path expressions. These will be useful in Section 5.2 in the description of conflict rules in the Path Lock Satisfiability scheme. In the following definition, the dot $\cdot$ denotes the concatenation of sets of strings. Also, we use the Kleene-star * to denote the zero or more recurrences of a substring.

DEFINITION 3 (Semantics of path expressions). *Let* $\mathbf{L}(p)$ *be the set of label paths selected by path expression* p*. We define* $\mathbf{L}$ *recursively as follows.*

$$\begin{aligned} \mathbf{L}(*) &= E \\ \mathbf{L}(e) &= \{e\} \text{ with } e \in E. \\ \mathbf{L}(p/f) &= \mathbf{L}(p) \cdot \{/\} \cdot \mathbf{L}(f) \\ \mathbf{L}(p//f) &= \mathbf{L}(p) \cdot \{/\} \cdot (E \cdot \{/\})^* \cdot \mathbf{L}(f) \end{aligned}$$

Thus, $\mathbf{L}(p)$ is the language of which the strings are the label paths represented by the path expression p.

Query Operation Path expressions are used in query operations over a certain xp-tree d. Informally, the semantics of a query operation $\mathbf{Q}(n, p)$ are the same as for XPath expressions: the result of $\mathbf{Q}(n, p)$ is the set of nodes selected by the path expression p started from the *context node* n. Note that $\mathbf{Q}$ denotes "query".

DEFINITION 4 (Semantics of the query operation). *The* result *of a query operation* $\mathbf{Q}(n, p)$ *over xp-tree* d *is defined as the set of all nodes* n' *in* d *such that* $\overline{\lambda}_d(n, n') \in \mathbf{L}(p)$*.*

A query operation never fails; if the context node n *is not a node in the xp-tree* d*, we define the result of the query operation to be empty.*

We assume that during a transaction the user has a set of variables $\mathcal{X} = \{\mathtt{x}_1, \mathtt{x}_2, \ldots\}$ into which he can store the results of the queries. The contents of these variables may be manipulated, in a generic or non-generic manner, by the user as long as they always contain sets of nodes that were previously retrieved in the same transaction.

When the user first makes a connection to the XML database with the intention to query or update a document represented by the xp-tree d, he receives the root node r of d. The node r can then be used as the context node in a query operation of which the result is stored in a variable x_i of $\mathcal{X}$. One or more of the nodes in x_i can then be used as the context node in subsequent query operations.

4.1.3. *Update Language*
Next to the query operation we also define update operations that can be called by the user to change the document. The parameters to our operations include nodes which were extracted from the result of the query operations that the user posed before requesting an update. Thus, writing (i.e., updating the document) always implies reading. This will be formalized in Definition 7 later in this section.

We now turn to the formal presentation of the addition and deletion operations.

DEFINITION 5 (Update operations on xp-trees). *The following two update operations operate on a certain xp-tree d.*

$\mathbf{A}(n, a)$ *This operation creates a new node n' with $\nu(n') = a$ and a new edge (n, n') in the xp-tree d. The operation fails if the resulting document is not an xp-tree. If successful, the* result *of the addition operation is $\{n'\}$.*

$\mathbf{D}(n)$ *This operation removes the node n and the edge incident to n from the xp-tree d. The operation fails if the resulting document is not an xp-tree. If successful, the* result *of the deletion operation is the empty set.*

Note that **A** is an abbreviation of "addition", while **D** denotes "deletion". We proceed with a small example to clarify these operations.

EXAMPLE 3. *Consider the xp-tree d given in Figure 2. The update operation $\mathbf{A}(n_3, \texttt{child})$ is successful; its result is the new node n_{40} with $\nu(n_{40}) = \texttt{child}$.*

The operation $\mathbf{D}(n_2)$ fails because the resulting document is not an xp-tree (the graph becomes unconnected).

We are now ready to formally define transactions as seen from the user's perspective. In addition to the three operations defined earlier — $\mathbf{Q}(n, p)$, $\mathbf{A}(n, a)$, and $\mathbf{D}(n)$ — the following definition will mention the *commit* operation $\mathbf{C}()$ of which the semantics, for now, is taken to be empty. We will revisit this operation in the next section.

DEFINITION 6 (User Transaction). *A* user transaction *is a finite list of operations* $\mathbf{Q}(n,p)$, $\mathbf{A}(n,a)$, $\mathbf{D}(n)$, *and* $\mathbf{C}()$. *This list contains exactly one* $\mathbf{C}()$ *operation; occurring at the end.*

The following definition formalizes our earlier remark that in our transaction model writing implies reading.

DEFINITION 7 (Node-correct Transaction). *A transaction is said to be* node-correct *if for every operation that uses a certain node identifier other than the root* r *as a parameter, there is an earlier operation (an addition or a query) in the transaction that had this node in its result.*

We have now defined the data model and transactions of data manipulation language operations over the data model that can be used by a user to alter a document. We will now turn to the internal representation of the document by the scheduler, and review the semantics of the data manipulation operations. Stated differently, we now turn to the scheduler's perspective on transactions.

4.2. The Scheduler's Perspective

In this and the next sections, we will detail the internal workings of the concurrency control mechanism we propose for XML databases. The prime notion in this context is that of the *scheduler*. While we will define and discuss two schedulers in detail in Section 6, we will already start using the term; it is sufficient for now to recall the traditional meaning of the scheduler.

In this section we will extend the simple data model and transaction model presented in the previous section. The extension is necessary for the scheduler to manage concurrent manipulations of a document by multiple users.

We proceed with extending the xp-tree data model.

DEFINITION 8 (Instance graph and actual instance). *The* instance graph (N, B, r, ν, δ) *is a rooted acyclic graph with vertices* N, *edges* $B \subseteq N \times N$, *the root* r, *nodes labeled with element names by* $\nu : N \to E$ *and with sets of transaction identifiers by* $\delta : N \to 2^T$. *The sets of transaction identifiers indicate that the node has been deleted by these transactions. The subgraph defined exactly by the nodes that are labeled by* δ *with the empty set is called the* actual instance *and is presumed to be always an xp-tree extended with the* δ *function.*

An xp-tree is itself a special case of an instance graph which forms a tree in which all nodes are labeled by δ with the empty set of transactions. We give a small example next.

EXAMPLE 4. *Consider the xp-tree d given in Figure 2. Let d also be an instance graph I. Assume that we extend I with the following δ-labelings: $\delta(n_{15}) = \{t_1\}$, $\delta(n_{38}) = \{t_1\}$, and $\delta(n_{39}) = \{t_1, t_2\}$, to I'.*

In I', the actual instance is the graph d without the nodes n_{15}, n_{38} and n_{39} and the edges (n_{13}, n_{15}), (n_{15}, n_{38}) and (n_{38}, n_{39}).

As we detailed in the previous section, users can send data manipulation operations to the document server consecutively; the operations that he can send are the ones given in Definition 6. However, these operations were defined on xp-trees, because for users this is the intended semantics. Since the scheduler makes use of an instance graph (to keep track of 'dirty' updates) rather than an xp-tree, we need to redefine the operations slightly. The users need not be aware of these new semantics, since ultimately the scheduler must make sure that the semantics are the same as the intended semantics.

DEFINITION 9 (Operations on the instance graph). *The four following operations are defined on an instance graph.*

A(n, a) *The semantics of this update operation are largely unchanged from Definition 5, except that now the operation fails if in the resulting instance graph the actual instance is not an xp-tree*[1] *with root r.*

D(n) *The semantics of this update operation have changed completely. No nodes are removed from the instance graph; instead this operation adds the transaction identifier of the executing transaction to the set that the node n is δ-labeled with. The operation fails if in the resulting instance graph the actual instance is not an xp-tree*[1] *with root r.*

Q(n, p) *This query operation works* only *on the actual instance of the instance graph. As such, its semantics are unchanged.*

C() *The* commit *operation removes nodes from the instance graph that are labeled by δ with* at least *the identifier of the executing transaction.*

[1] We mean an xp-tree extended with the δ function that maps each of its nodes to the empty set of transactions.

Note that the definitions of the four operations are such that given an instance graph in which the actual instance is indeed an xp-tree (as it is presumed to be), then the resulting actual instance is also an xp-tree. This is true since the addition and the deletion fail if the actual instance in the result would not be an xp-tree, and the query and the commit do not change the actual instance. In addition, we can prove the following lemma.

LEMMA 1. *The application of an operation on an instance graph does not introduce cycles (directed or undirected) in the resulting instance graph.*

Proof. It is clear that only the addition operation generates new edges. Consider the addition $\mathbf{A}(n, a)$ which creates a new edge (n, n'). Since n' is always a new node, no cycles can be constructed. □

Now we are ready to formally define actions, transactions (as seen from the scheduler's perspective) and schedules.

DEFINITION 10 (Action, Transaction, and Schedule). *An* action *is a pair $a(o, t)$ where o is one of the operations given in Definition 9 and t is a transaction identifier. A* transaction *is a finite list of actions that all have the same transaction identifier and in which there is exactly one commit operation which is in the last action. A* schedule *is an interleaving of a set of transactions.*

EXAMPLE 5. *Consider the xp-tree d given in Figure 2 which acts as the initial instance graph. Consider also the schedule*

$$S = \langle a_1(\mathbf{D}(n_{39}), t_1), a_2(\mathbf{D}(n_{38}), t_2), \\ a_3(\mathbf{C}(), t_2), a_4(\mathbf{C}(), t_1)\rangle.$$

The result, after action a_3 but before action a_4, is an instance graph that is unconnected. The actual instance, however, remains an xp-tree.

DEFINITION 11 (Node-correct Schedule). *A schedule is said to be* node-correct *if all its transactions are node-correct (see Definition 7).*

Thus, the schedule S presented in Example 5 is not a node-correct schedule. Indeed, transactions t_1 and t_2 have used the nodes n_{38} and n_{39} without having obtained them through an earlier query or update operation.

In the remainder of this paper, we will always assume schedules to be node-correct.

We now turn to the discussion of path locks which will be used by the scheduler to ensure serializability of schedules.

5. Path Lock Schemes

In this section we introduce read and write locks that are associated to the instance graph which represents an XML document. We will present two path lock schemes: "Path Lock Satisfiability"(abbreviated to SAT) and "Path Lock Propagation" (abbreviated to PROP) (Dekeyser and Hidders, 2002b). The latter scheme causes a multitude of read locks to be obtained but makes checking for conflicts trivial, as it only checks lock equality locally (i.e. within one node). The former scheme, in contrast, sets very few locks but requires more work when checking for conflicting locks. We will formally compare the two schemes in Section 5.3.

We first introduce the path lock propagation scheme.

5.1. Path Lock Propagation Scheme

We will start by defining read and write locks, and discuss which locks must be obtained by which operations. We then proceed with giving the conflict rules. Finally, we also characterize the complexity of checking for lock conflicts in the propagation scheme.

5.1.1. *Read Locks and Write Locks*

We start with the definition of the read locks.

DEFINITION 12 (Read Lock). *A* read lock *is a tuple* $rl(t, n, p)$ *where* t *is the transaction which* owns *the lock,* n *is the node identifier in the instance graph for which the lock holds and* p *is a path expression in* $\mathcal{P}$*.*

The informal meaning of such a lock is that the transaction has issued a query p starting from node n.

The *initial read lock* for a given query operation $\mathbf{Q}(n, p)$ that is issued by transaction t is simply $\mathrm{rl}(t, n, p)$. From the initial read lock we can derive other read locks by a process called *read-lock propagation.*

The process of read-lock propagation causes read locks on a node to be propagated to nodes just below this node in the instance graph. This is done with the rules shown in Table III.

The process of read-lock propagation is applied until no more new read locks are added; this process ends since the instance graph is both finite and acyclic. The result is a set R_q^* of read locks.

Table III. Propagation Rules for Read Locks.

1.	$\mathrm{rl}(t, n, a/p)$	$\rightarrow$	$\mathrm{rl}(t, n', p)$	if $(n, n') \in B$ and $\nu(n') = a$.
2.	$\mathrm{rl}(t, n, */p)$	$\rightarrow$	$\mathrm{rl}(t, n', p)$	if $(n, n') \in B$.
3.	$\mathrm{rl}(t, n, a//p)$	$\rightarrow$	$\mathrm{rl}(t, n', p)$	if $(n, n') \in B$ and $\nu(n') = a$.
4.	$\mathrm{rl}(t, n, a//p)$	$\rightarrow$	$\mathrm{rl}(t, n', *//p)$	if $(n, n') \in B$ and $\nu(n') = a$.
5.	$\mathrm{rl}(t, n, *//p)$	$\rightarrow$	$\mathrm{rl}(t, n', p)$	if $(n, n') \in B$.
6.	$\mathrm{rl}(t, n, *//p)$	$\rightarrow$	$\mathrm{rl}(t, n', *//p)$	if $(n, n') \in B$.

The following defines which read locks need to be *obtained* for the query operation $\mathbf{Q}(n, p)$ issued by transaction t:

$\mathbf{Q}(n, p)$: The lock $\mathrm{rl}(t, n, p)$ and those that are derived from it through propagation.

Note that this set of locks depends upon the instance graph, it has to be recomputed every time the instance graph is updated. This recomputation after an update can be done by propagating only the locks of the parent that a node is created or deleted under. Thus, it can be done relatively efficiently.

We proceed with the definition of the write locks.

DEFINITION 13 (Write Lock). *A* write lock *is a tuple $wl(t, n, f)$ for which t is the transaction which owns the lock, n is the node identifier for which the lock holds, and f is an expression over $\mathcal{F}$.*

The following defines which write locks must be *obtained* for which update operation issued by transaction t:

$\mathbf{A}(n, a)$: A write lock $\mathrm{wl}(t, n, a)$ on node n for transaction t.

$\mathbf{D}(n)$: Write locks $\mathrm{wl}(t, n, *)$ and $\mathrm{wl}(t, n', a)$ where n' is the parent of n in the instance graph and a is the label of n. If n or n' does not exist, then the corresponding write lock does not need to be obtained.

We now turn to an example to clarify the definitions in this section.

EXAMPLE 6. *Consider the xp-tree d given in Figure 2 and the first two actions of schedule S given below. The following table presents all the locks associated to d', which is the instance graph obtained after a_1 and a_2 are applied to xp-tree d.*

$$S = \langle a_1(\mathbf{Q}(r, \texttt{doc/person/child/person/name}), t_1), a_2(\mathbf{D}(n_{25}), t_2), \ldots \rangle$$

$rl(t_1, r,$ `doc/person/child/person/name`$)$
$rl(t_1, n_1,$ `person/child/person/name`$)$
$rl(t_1, n_2,$ `child/person/name`$)$
$rl(t_1, n_3,$ `child/person/name`$)$
$rl(t_1, n_6,$ `person/name`$)$
$rl(t_1, n_{13},$ `person/name`$)$
$rl(t_1, n_7,$ `name`$)$
$rl(t_1, n_{15},$ `name`$)$
$wl(t_2, n_{24},$ `paint`$)$
$wl(t_2, n_{25}, *)$

As will be explained in Section 6, locks are kept until a transaction commits. This amounts to *two-phase locking* (2PL); together with correct locking behavior, this will ensure serializability.

5.1.2. *Lock Compatibility*
We have established which read locks are to be obtained by queries and which write locks are to be obtained by update operations. What remains to be defined is when exactly such locks can be obtained when other transactions have also obtained locks upon the same document. For this purpose we define the notion of *conflicting locks*.

DEFINITION 14 (Conflicting Path Locks in PROP). *A read lock such as $rl(t, n, a)$ or $rl(t, n, *)$ conflicts with a write lock $wl(t', n, a)$ and a write lock $wl(t', n, *)$ if $t \neq t'$. Note that a is a single symbol in $\mathcal{F}$. All other locks do not conflict.*

According to the definition, two write locks do not conflict. Indeed, the serializability proof in Section 7 shows that the conflict rules given above are sufficient to ensure serializability. The intuition behind this is as follows.

1. In general, two update operations from different transactions logically commute. Indeed, since we work with unordered documents, it does not matter for example which one of two additions comes first.

2. For those cases which do not logically commute — for instance, if a transaction deletes a node created by another transaction — the node-correctness property of transactions ensures that a read-write conflict arrises prior to the update.

EXAMPLE 7. *The read lock* $rl(t_1, n_{38}, \texttt{name})$ *conflicts with the write lock* $wl(t_2, n_{38}, *)$*; but* $rl(t_1, n_{25}, \texttt{*//child//hobby})$ *and* $wl(t_2, n_{25}, *)$ *do* not *conflict. The set of locks associated to the instance graph* d' *of Example 6 did not contain conflicting locks.*

5.1.3. *Complexity*

The complexity of the Path Lock Propagation scheme is as follows. Checking for a read- and write lock conflict is clearly relatively simple in this system. Consider the path locks $\mathrm{rl}(t, n, a)$ and $\mathrm{wl}(t', n, a')$, where a is an expresion over $\mathcal{F}$. The two locks are associated to the same node n, and only the equality of a and a' needs to be checked. Thus, the time complexity of *checking for conflicts* in the propagation system is $O(1)$.

The setting of read locks can occur at the time of query evaluation; this causes only a small increase in execution time.

Thus, while the time complexity of checking for conflicts in this method is low, the space and time complexity of *setting the locks* is a more serious issue. Specifically, the complexity is $O(nm)$, where n is the size of the tree, and m is the length of the lock expression.

5.2. Path Lock Satisfiability Scheme

We now turn to an alternate path lock scheme. In contrast to the Path Lock Propagation scheme, the satisfiability scheme requires fewer locks to be obtained but is more complex with regard to testing for conflicts.

5.2.1. *Read Locks and Write Locks*

Read Locks Read locks in SAT are defined as in PROP. However, it is sufficient to obtain for a given query operation only the initial read lock. Thus, the lock propagation process is not applied in this case.

Write Locks Write locks in SAT are defined exactly as in PROP. Also, the update operations are to obtain the same write locks as defined earlier.

As in PROP, all locks owned by a transaction are retained until that transaction commits.

5.2.2. *Lock Compatibility*

As in the path lock propagation scheme, we must define when path locks in the satisfiability scheme conflict.

DEFINITION 15 (Conflicting Path Locks in SAT). *Read lock* $rl(t, n, p)$ *conflicts with write lock* $wl(t', n', f)$ *iff (1)* $t \neq t'$*, (2)* n *is an ancestor of* n'*, and (3)* $\overline{\lambda}(n, n')/f \in \mathbf{L}(p)$*. All other locks do not conflict.*

EXAMPLE 8. *Consider the same schedule S and instance graph d' as in Example 6. In the* SAT *scheme, the following are the only path locks that need to be obtained.*

$$rl(t_1, r, \texttt{doc/person/child/person/name})$$
$$wl(t_2, n_{24}, \texttt{paint})$$
$$wl(t_2, n_{25}, *)$$

There are no conflicting locks, as

$$\overline{\lambda}_{d'}(r, n_{24})/\texttt{paint} \notin \mathbf{L}(\texttt{doc/person/child/person/name}).$$

Likewise, $\overline{\lambda}_{d'}(r, n_{25})/* \notin \mathbf{L}(\texttt{doc/person/child/person/name})$.

5.2.3. *Complexity*
Clearly the space complexity in this scheme is not an issue; we have already shown that fewer locks need to be obtained than in PROP. Relative to the size of the instance graph, the space complexity is $O(1)$. However, the time complexity is more important here than in the alternative locking scheme. Indeed, we need to check the satisfiability of a path by a more general path expression. Such a test can be likened to checking whether a string is accepted by a non-deterministic automaton. The path expression p can be written as an NFA while a pathname t can be seen as a string. If the string is accepted by the NFA, the pathname is equal to one of the pathnames represented by p. From Automata Theory, we know that this problem is of time complexity $O(n^4)$.

5.3. Equivalence of the Path Lock Schemes

In the remainder of this paper, we will work primarily with the path lock propagation scheme. However, the results also apply to the path lock satisfiability scheme since the two are equivalent with respect to the conflicts they detect. This is shown in the following theorem.

THEOREM 1. *A conflict occurs in the instance graph I according to the path lock propagation scheme if and only if a conflict occurs in I according to the path lock satisfiability scheme.*

The proof can be found in (Dekeyser, 2003).

6. Schedulers for XML Databases

In this section we present two different schedulers for XML databases that work on instance graphs and make use of the path locks. It is the scheduler's task to guarantee that the schedules it accepts are serializable.

DEFINITION 16 (Equivalent and Serializable Schedules). *Two schedules are* equivalent *if (1) one is a permutation (preserving the order of actions within a transaction) of the other, (2) the resulting instance graph is in both cases the same, and (3) all the queries in one schedule return the same result as the corresponding queries in the other schedule. A schedule is said to be* serializable *if it is equivalent with a serial schedule.*

Depending on the definition of the scheduler, it can accept more or fewer schedules. We will first present the commit scheduler (Dekeyser and Hidders, 2003) and then proceed with the conflict scheduler.

6.1. The Commit Scheduler

In this section we detail the working of the commit scheduler. The term is based on the theoretical notion of *commit serializability* (Weikum and Vossen, 2002).

User processes send actions to the document server in a sequence that can (at least *a posteriori*) be seen as a transaction. When an action is sent, the process waits for a reply from the scheduler. This reply can be positive, in which case the result (if any) of the action is obtained. It can also be negative, if the action caused a conflict or if the action failed. Whatever the reply from the scheduler, the sending process may use the answer to decide whether or not to subsequently send another action. If it decides to send another action, it may use the result from any previous action as a parameter in the next action.

In our theoretical model we will assume that the scheduler replies to every request before it processes the next request. Strictly speaking dead lock cannot occur then, but it replaces the problem with live lock if transactions repeat requests that caused conflicts. It is easy to see how the model could be extended such that the scheduler keeps requests that cause conflicts in a waiting queue until they no longer conflict, in which case dead lock can be detected in the usual way with a wait-for graph.

We now turn to the formal definition of the commit scheduler.

DEFINITION 17 (Commit Scheduler). *The* commit scheduler *is the automaton whose state consists of a schedule S of actions that it has previously accepted and processed, a set of locks L and an instance graph I. Its transition function γ maps S, L, I and a newly requested action $a(o,t)$ to a schedule S', a set of locks L' and an instance graph I' as follows:*

1. *The new instance graph I' is obtained by applying operation o to instance graph I. If the operation fails, then γ is not defined*[2].
2. *For update and query operations, the set of locks L' is obtained by adding to L the locks required by the operation o and the locks that are caused by propagation in I'. For the commit operation, L' is obtained by removing all locks from L which are owned by the transaction that commits, plus those locks on the nodes that are now deleted from the instance graph.*

   If some of the locks required by o conflict with those in L, then γ is not defined[2].
3. *The schedule S' is S augmented with $a(o,t)$ provided that γ did not become undefined due to the previous points.*
4. *The sending process receives the result of o, if any.*

The execution of the commit scheduler on a given instance graph I starts with the empty schedule S, the empty set of locks L, and the instance graph I. It receives the actions of S sequentially, and its result is either (1) the output schedule *S, the set of locks L, and the instance graph I transformed according to each iteration of the commit scheduler, or (2) undefined.*

We will illustrate the definition in Example 9 below.

As can be deduced from the definition, in its first phase the scheduler effectively does two things. First, it does a test-processing of the newly arrived operation to see if o does not fail. Second, if the operation does not fail, it attempts to obtain all the locks that are necessary for o. If it can obtain all the locks, then the scheduler proceeds with the second phase. If one of the two conditions is not met, then the action is rejected and the sending process is notified. In case there is no failure, nor a conflict, the operation o is effectively processed and appended to the output schedule.

[2] If γ is undefined, the sending process is notified that its action is not accepted, and the scheduler waits for a new action. Thus deadlocks cannot occur.

A commit of a transaction t causes all read and write locks of t to be removed from the instance graph. Additionally, all nodes δ-labeled with at least t are removed from the instance graph, together will all locks on those nodes.

A serial schedule equivalent to the output schedule is obtained by sorting the transactions according to the commit-points in the output schedule.

We proceed with characterizing the output schedule of the commit scheduler.

DEFINITION 18 (Legal and Fail-free Schedules). *A schedule is said to be* fail-free *if all its operations can be executed without any of them failing.*

A schedule is said to be a legal schedule *if (1) it is node correct, (2) fail-free and (3) for all L_i^S it holds that they only contain compatible locks.*

LEMMA 2. *Any output schedule of the commit scheduler is a legal schedule.*

The proof is straightforward.

At this point we have a complete transaction model plus scheduling algorithm for an XML database. Using the path lock propagation scheme and the commit scheduler, we can now solve use case 1. Note that use case 2 can also be solved at this point, but we will do this in the next section.

EXAMPLE 9. *Consider use case 1 and the xp-tree d given in Figure 2. The actions described in the use case translate to the schedule S_{in} given next*[3]*. Let $p_1 =$* `doc//child//hobby`*, $p_2 =$* `doc/person/hobby/paint`*, and $p_3 =$* `doc/person/hobby`*.*

$$\begin{aligned} S_{\text{in}} \;=\; \langle & a_1(\mathbf{Q}(r,p_1),t_1), a_2(\mathbf{Q}(r,p_2),t_2), a_3(\mathbf{Q}(r,p_3),t_2), a_4(\mathbf{D}(n_{25}),t_2), \\ & a_5(\mathbf{A}(n_{24},\texttt{painting}),t_2), a_6(\mathbf{C}(),t_2), a_7(\mathbf{C}(),t_1)\rangle \end{aligned}$$

Let the state of the commit scheduler consist of the empty schedule S and the instance graph I_0 which is equal to d extended with the δ-function which labels all non-root nodes with the empty set. The commit

[3] As discussed in Section 4.1, the manner in which we perform a text-value update is a simulation. The result is defined up to isomorphism on the nodes. For other transactions, this is sufficient as node identifiers are incidental.

scheduler accepts individual actions from S_{in} in the sequence given by S_{in}.

The schedule S_{in} is itself already a legal schedule, thus the output schedule of the commit scheduler is the same as S_{in}.

A serial schedule equivalent to S_{in} is the schedule obtained by first taking all actions of transaction t_2 and then all actions of t_1.

6.2. The Conflict Scheduler

In this section we detail the working of the conflict scheduler. As with the commit scheduler, user processes send actions to the document server in a sequence that can (at least *a posteriori*) be seen as a transaction. In contrast to the commit scheduler which effectively lets transactions wait if their request cannot be processed, the conflict scheduler keeps accepting and processing actions until it fails.

6.2.1. *Schedules Without Commits*

The commit operation as defined for the Commit Scheduler is not suitable for the Conflict Scheduler. Therefore, we first present a scheduler which does not accept commit operations. In the subsequent section we will add a new kind of commits.

DEFINITION 19. *The* conflict scheduler *is the automaton whose state consists of a schedule S of actions that it has previously accepted and processed, a set of locks L, a dependency graph G which is a directed graph whose nodes are transaction identifiers, and an instance graph I. Its transition function γ maps S, L, G, I and a newly requested action $a(o,t)$ to a schedule S', a set of locks L', a dependency graph G' and an instance graph I' as follows:*

1. *The new instance graph I' is obtained by applying operation o to instance graph I. If the operation fails, then γ is not defined*[2]*.*
2. *The new set of locks L' is obtained by adding to L those locks that are required by the operation o. If one of these locks conflicts with a lock in L of transaction t' then G' is equal to G plus the edge (t', t), otherwise G' is equal to G.*
3. *If G' contains cycles, then γ is not defined*[2]*.*
4. *The schedule S' is S augmented with $a(o,t)$ provided that γ did not become undefined due to the previous points.*
5. *The sending process receives the result of o, if any.*

The execution of the conflict scheduler on a given instance graph I starts with the empty schedule S, the empty set of locks L, an empty graph G and the instance graph I. It receives the actions of S sequentially, and its result is either (1) the output schedule *S, the set of locks L, the dependency graph G and the instance graph I transformed according to each iteration of the commit scheduler, or (2) undefined.*

The output schedule of the conflict scheduler is always node-correct and fail-free, but is not always legal.

A serial schedule equivalent to the output schedule is obtained by sorting the transactions according to the topological sort of the dependency graph.

At this point we have an alternative transaction model for an XML database. Using the path lock propagation scheme and the conflict scheduler, we can now solve use case 2. Note that use case 1 can also be solved with this model.

EXAMPLE 10. *Consider use case 2 and the xp-tree d given in Figure 2. The actions described in the use case translate to the schedule S_{in} given next.*

$$\begin{aligned} S_{\text{in}} \;=\; \langle & a_1(\mathbf{Q}(r, \texttt{doc/person//hobby}), t_1), a_2(\mathbf{Q}(r, \texttt{doc}), t_2), \\ & a_3(\mathbf{A}(n_1, \texttt{person}), t_2), a_4(\mathbf{A}(n_{40}, \texttt{hobby}), t_2)\rangle \end{aligned}$$

Let the state of the conflict scheduler consist of the empty schedule S, the instance graph I_0 which is equal to d extended with the δ-function which labels all non-root nodes with the empty set, and the edgeless dependency graph G_0. The conflict scheduler accepts individual actions from S_{in} in the sequence given by S_{in}.

After the first three actions, no conflicts have appeared and the dependency graph G_3 is still without edges. However, action a_4 causes a conflict between locks needed for a_1 and a_4. Thus, G_4 contains an edge from t_1 to t_2. After a_4, no further conflicts appear, so the conflict scheduler finishes and accepts S_{in} as its output schedule.

A serial schedule equivalent to S_{in} is the schedule obtained by first taking all actions of transaction t_1 and then all actions of t_2.

6.2.2. *Adding Commits*

The above Conflict Scheduler without commits is clearly somewhat impractical since nothing would ever be removed from the instance graph, nor from the dependency graph, nor from the set of locks in the scheduler's state. In this section we solve this issue.

Consider the following description of the commit operation which we now require to appear at the very end of each transaction.

C() If it does not fail, the commit operation does the following things:

1. It removes all the locks in L that are owned by the committing transaction.
2. It deletes from the instance graph I nodes with a non-empty δ function if there are no locks in L' for that node.
3. It deletes from the dependency graph G the node for the committing transaction.

The commit operation fails if in G there is an edge that arrives in the node of the committing transaction.

Failure of the commit operation means that the transaction has to resubmit its commit operation until it succeeds. In practice this would be cumbersome but the scheduler can be redefined such that it remembers every commit and processes it when all transactions that logically appear before[4] the respective transaction have committed.

6.3. Recovery

In the definitions of the Commit and the Conflict Schedulers we have explicitely mentioned a commit operation which must be used by a transaction to release its locks. We have not explicitely mentioned a *roll-back* or *abort* operation which is traditionally used for recovery from catastrophic events such as power outages. It is straightforward to extend our schedulers with such recovery methods. However, there are certain differences between the two.

Commit Scheduler Since the Commit Scheduler does not permit an action that causes a conflict to proceed, *dirty reads* cannot occur. A transaction is free to issue a roll-back, upon which the scheduler undoes the changes made to the instance graph by that transaction. The roll-back does not affect the other running transactions.

Conflict Scheduler The Conflict Scheduler can permit an action that causes a conflict to proceed, as long as there is no cycle in its dependency graph. Thus, dirty reads are possible and aborting a transaction can affect other transactions. The rule that states when a transaction may issue an abort operation or be rolled-back by the scheduler, is

[4] With respect to the topological sort of the dependency graph.

the same as for the commit operation in the Conflict Scheduler: a transaction may abort when all transactions occuring before it in the dependency graph have been rolled-back.

7. Theoretical Results

In this section, we give a number of technical results that show what the scheduling system we propose for XML databases can offer.

First, we characterize precisely the data structure needed for the instance graph in the state of the scheduler. Secondly, we will show that the output schedule of the commit scheduler is always serializable. We will then do the same for the conflict scheduler. Finally, we also show necessity for the conflict rules.

7.1. Instance Graph Connectedness

Since we define a scheduling system for XML databases, the scheduler, whether it is the commit or the conflict scheduler, always starts with an instance graph that is an xp-tree. We now show that during all iterations of the commit scheduler, its instance graph remains a tree. As a consequence, its actual instance is always a subtree of the instance graph, with the same root. This result does not hold for the conflict scheduler.

THEOREM 2 (Connectedness). *Let the state of the commit-scheduler contain the instance graph I. If I is a tree, then $\gamma(I)$ is also a tree.*

The proof can be found in (Dekeyser, 2003).

EXAMPLE 11. *Example 5 gave a schedule S which resulted in an instance graph that became unconnected. The schedule S was not node-correct, however. Still, it is possible to fix S such that it becomes node-correct. Schedule S' given next is such an improvement of S. Assume that* $p_1 =$ `doc/person/child/person/name/David` *and that* $p_2 =$ `doc/person/child/person/name/`.

$$S' = \langle a_1(\mathbf{Q}(r, p_1), t_1), a_2(\mathbf{Q}(r, p_2), t_2), a_3(\mathbf{D}(n_{39}), t_1), a_4(\mathbf{D}(n_{38}), t_2), a_5(\mathbf{C}(), t_2)\rangle.$$

Now we must show that S' is not a valid output schedule, otherwise we have given a counter example to Theorem 2.

Assume that we obtained locks according to the PROP *scheme. This means that at least two following two locks must have been obtained:*

$rl(t_1, n_{38},$ `David`$)$ and $wl(t_2, n_{38}, *)$. Clearly, these two locks conflict. Thus, S' is not a legal schedule.

7.2. Serializability

We give only a sketch of the serializability proofs for both schedulers. The full proofs can be found in (Dekeyser, 2003).

7.2.1. *Commit Scheduler*

THEOREM 3 (Serializability). *Every output schedule of the commit scheduler is serializable.*

Sketch of the proof. We presume some ordering on the transaction identifiers used in S such that $t_i < t_j$ if the commit of t_i precedes the commit of t_j in S or there is a commit of t_i but not a commit of t_j in S. We serialize the schedule by repeatedly swapping two consecutive actions (t_i, o_i) and (t_{i+1}, o_{i+1}) if $t_i \neq t_j$ and $t_j < t_i$. It is easy to see that if there are no more such pairs then the schedule is serialized. It can also be shown that after a swap of such a pair the result will be an equivalent legal schedule if the schedule before the swap is a legal schedule. Assume that S is a legal schedule and we swap two consecutive actions (t_i, o_i) and (t_{i+1}, o_{i+1}) in S and $t_i \neq t_j$ and o_i is not a commit, then we prove that the following holds: (where L_i^S denotes the set of locks L in the state of the scheduler after processing the i-th step of S, and I_i^S denotes the resulting instance graph after this step)

1. the two swapped operations will not fail in S',
2. all locks in $L_i^{S'}$ are compatible,
3. $I_{i+1}^{S'} = I_{i+1}^S$,
4. if they exist the results of o_i and o_{i+1} remain the same, and
5. $L_{i+1}^{S'} \subseteq L_{i+1}^S$
6. S' is node correct,

It follows from these points that S' is equivalent with S, fail-free and in all sets $L_j^{S'}$ there are no incompatible locks, i.e., S' is legal.

Each of the six points is proven in (Dekeyser, 2003). □

7.2.2. *Conflict Scheduler*

We give only a sketch of the serializability proof. The full proof can be found in (Dekeyser, 2003). Note that the following proof only proceeds for schedules without commits; the subsequent theorem adds commits.

THEOREM 4 (Serializability). *Every output schedule of the conflict scheduler without commits is serializable.*

Sketch of the proof. We serialize the schedule by swapping consecutive operations. We assume some linear order on the transaction identifiers that respects the dependency graph at the end of the schedule, i.e., if the edge (t_j, t_i) is in this dependency graph then $t_j < t_i$. If there is a pair (t_i, o_i) and (t_{i+1}, o_{i+1}) in the schedule S and $t_i > t_{i+1}$ then we swap them. Note that since $t_i > t_{i+1}$ it follows that the locks of the two operations do not conflict because if they would then the edge (t_i, t_{i+1}) would be in the final dependency graph and therefore $t_i < t_{i+1}$. So we show that if we swap these two operations then it holds for the resulting schedule S' that: (where L_i^S and G_i^S denote the set of locks L and the dependency graph in the state of the scheduler after processing the i-th step of S, and I_i^S denotes the resulting instance graph after this step)

1. the two swapped operations will not fail in S',
2. $G_{i+1}^{S'} \subseteq G_{i+1}^S$,
3. $I_{i+1}^{S'} = I_{i+1}^S$,
4. $L_{i+1}^{S'} = L_{i+1}^S$, and
5. if they exist the results of o_i and o_{i+1} remain the same, and
6. S' is node correct.

Each of these points is proven in (Dekeyser, 2003). □

Note that this proof shows that the resulting *instance graphs* for S and S' are the same, not just the actual instances as required by the definition of equivalent schedules. Thus, this proof is stronger than strictly necessary.

THEOREM 5. *Every output schedule of the conflict scheduler which is enhanced with commits, is serializable.*

The proof can be found in (Dekeyser, 2003).

7.3. Necessity

We now turn to the issue of necessity of the conflict rules. We commence with showing that the rules are *locally necessary*, after which we show that *global necessity* does not hold. For simplicity, we will work with

the path lock satisfiability scheme (though the result can of course also be obtained for PL-PROP). For local necessity, in essence we need to show that if two consecutive actions in a schedule conflict, then they cannot commute.

THEOREM 6 (Local Necessity). *The following holds for all instance graphs I_i with conflict-free lock set L_i^S. Consider two fail-free actions $a_1(o_1, t)$ and $a_2(o_2, t')$ applied consecutively to I_i such that the instance graph I_{i+2} is yielded. If the lock set L_{i+2}^S contains conflicting locks, then the swapping of a_1 and a_2 yielding I'_{i+2} means that either (1) $I_{i+2} \neq I'_{i+2}$ or (2) a_1 or a_2 failed, or (3) the results of query operations have changed.*

Proof. Since L_{i+2}^S contains conflicting locks, a_1 (a_2) must be a query while a_2 (a_1) must be an update operation. Thus, possibilities (1) and (2) cannot occur. Indeed, a query operation does not change the instance, nor does it fail. Whether the update operation preceeds the query or not, does not change the instance nor the fail-free property. Consequently, we need to prove that the result of the query operation changes when a_1 and a_2 are swapped. We consider the following cases.

1. Assume that the conflicting locks are $\mathrm{rl}(t, n_q, p)$ and $\mathrm{wl}(t', n, a)$. Thus, $\overline{\lambda}_{I_{i+1}}(n_q, n)/a \in \mathbf{L}(p)$.
   - Assume that $a_1 = \mathbf{Q}(n_q, p)$.
     - Assume that $a_2 = \mathbf{D}(n')$. Since $\overline{\lambda}_{I_{i+1}}(n_q, n)/a \in \mathbf{L}(p)$ we know that n' is in the result of query a_1 at instance I_{i+1} (note that the path from n_q to n is part of the actual instance of instance graph I_{i+1}). Clearly, when the two operations are swapped, n' cannot be in the result of the query a_2.
     - Assume that $a_2 = \mathbf{A}(n, a)$. In instance I_{i+1} thenode n' does not exist, and $\overline{\lambda}_{I_{i+1}}(n_q, n) \notin \mathbf{L}(p)$. Thus, the result of query a_1 does not contain n nor n' which will become the child of n in instance I_{i+2}. Clearly, when the two operations are swapped, node n' will be in the result of the query a_2.
   - Assume that $a_2 = \mathbf{Q}(n_q, p)$. The proof is similar to the previous case.
2. Assume that the conflicting locks are $\mathrm{rl}(t, n_q, p)$ and $\mathrm{wl}(t', n', *)$. The proof is similar to the previous case. □

This result shows that the conflict rules, at least when considering consecutive actions, are not too restrictive. However, in general necessity does not hold. We illustrate this with the following example.

EXAMPLE 12. *Consider a document d represented by the instance graph I, with $N = \{r, n\}$, $B = \{(r, n)\}$, and $\nu(n) =$* `person`*. Assume the scheduler receives the following schedule (commits have been omitted for simplicity):*

$$S = \langle a_1(\mathbf{Q}(r, //\texttt{child/hobby}), t_1), a_2(\mathbf{A}(n, \texttt{child}), t_2), a_3(\mathbf{D}(n_1), t_2)\rangle.$$

Action a_2 has created the node n_1 as a child of n. Clearly there is no conflict between a_1 and a_2, but there is a conflict between a_1 and a_3. Thus, in this case the transaction that has created node n_1 may not delete that same node.

The example shows that in certain cases, our conflict rules are slightly too restrictive.

8. Conclusion and Future Work

We have shown that traditional concurrency control methods are insufficient to capture the complex update behavior which is possible in XML databases. We presented two perspectives on the transaction model which we introduce in this paper. We also introduced two equivalent locking schemes for XML databases. Finally, we have introduced two schedulers which may use either one of these path locks schemes. For both schedulers, we have shown that they guarantee serializability.

Regarding future work, we would like to extend the update language to include more expressive operations such as a *move*. For this, we need to investigate a model in which node identity plays a more important role. This would likely lead to new kinds of locks and compatibility rules. Such a model would also be more suitable to simulate ordered documents.

We are currently investigating a more theoretical approach where serializability of schedules can be determined independently from the instance. Preliminary results have been published in (Dekeyser et al., 2003).

Finally, we are working on a proof-of-concept implementation of both schedulers using the PL-PROP scheme. We would like to extend this implementation to obtain experimental results that show the performance gains possible when using our path lock transaction model compared to existing solutions.

References

Abiteboul, S.: 1997, 'Querying Semi-Structured Data'. In: *Proceedings of ICDT'97*. pp. 1–18.

Abiteboul, S., P. Buneman, and D. Suciu: 1999, *Data on the Web: From Relations to Semistructured Data and XML*. San Francisco: Morgan-Kaufmann.

Abiteboul, S., D. Quass, J. McHugh, J. Widom, and J. Wiener: 1997, 'The LOREL Query Language for Semistructured Data'. *The International Journal on Digital Libraries* **1(1)**, 68–88.

Barghouti, N. and G. Kaiser: 1991, 'Concurrency Control in Advanced Database Applications'. *ACM Computing Surveys* **23(3)**, 269–317.

Bernstein, P., V. Hadzilacos, and N. Goodman: 1987, *Concurrency Control and Recovery in Database Systems*. Reading, Mass.: Addison Wesley.

Bray, T., J. Paoli, et al.: 2000, 'Extensible Markup Language (XML) 1.0 (Second Edition)'. *W3C Recommendation*.

Buneman, P., M. Fernández, and D. Suciu: 2000, 'UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion'. *VLDB Journal* **9(1)**, 76–110.

Cederqvist, P. et al.: 1993, 'Version Management with CVS'. *WWW Manual*. http://www.cvshome.org/docs/manual/.

Cellary, W., E. Gelenbe, and T. Morzy: 1988, 'Concurrency Control in Distributed Database Systems'. *Studies in Computer Science and Artificial Science (3)*.

Cowan, J. and T. Richard: 2001, 'XML Information Set'. *W3C Recommendation*.

Date, C.: 2000, *An Introduction to Database Systems, 7th edition*. Addison Wesley Longman.

Dekeyser, S.: 2003, 'Multiple Query Environment Problems'. Ph.D. thesis, University of Antwerp. http://win-www.ruca.ua.ac.be/u/dekeyser/.

Dekeyser, S. and J. Hidders: 2002a, 'A Basic Locking Protocol for XML'. Technical Report 02-05, University of Antwerp.

Dekeyser, S. and J. Hidders: 2002b, 'Path Locks for XML Document Collaboration'. In: *Proceedings of the Third WISE Conference*.

Dekeyser, S. and J. Hidders: 2003, 'A Commit Scheduler for XML Databases'. In: *Proceedings of the Fifth Asia Pacific Web Conference*. Xi'an, China.

Dekeyser, S., J. Hidders, and J. Paredaens: 2003, 'Instance Independent Concurrency Control for Semistructured Databases'. In: *Proceedings of the Eleventh Italian Symposium on Advanced Database Systems (SEBD)*. Cetraro, Italy.

Deutsch, A., M. Fernández, and D. Suciu: 1999, 'Storing Semistructured Data with STORED'. In: *Proceedings ACM SIGMOD*. Philadephia.

Eswaran, K., J. Gray, R. Lorie, and I. Traiger: 1976, 'The notions of consistency and predicate locks in a database system'. *Communications of the ACM* **19:11**, 624–633.

Fernández, M., J. Marsh, and M. Nagy: 2002, 'XQuery 1.0 and XPath 2.0 Data Model'. *W3C Working Draft*.

Florescu, D. and D. Kossmann: 1999, 'Storing and Querying XML Data using an RDBMS'. *IEEE Data Engineering Bulletin* **22(3)**, 27–34.

Florescu, D., A. Levy, M. Fernández, and D. Suciu: 1997, 'A Query Language for a Web-Site Management System'. *SIGMOD Record* **26(3)**, 4–11.

Gray, J.: 1978, 'Notes on database operating systems'. In: *Operating Systems: an Advanced Course*. New York: Springer-Verlag.

Gray, J., A. Lorie, et al.: 1975, 'Granularity of Locks in a Large Shared Data Base'. In: *Proceedings of the International Conference on Very Large Data Bases*. Framingham, Massachusetts.

Gray, J., G. Putzolo, and I. Traiger: 1976, 'Granularity of locks and degrees of consistency in a shared data base'. In: *Modeling in Data Base Management Systems*. Amsterdam: North Holland.

Hunt, H. and D. Rosenkrantz: 1979, 'The Complexity of Testing Predicate Locks'. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. Boston, Massachusetts.

Khoshafian, S.: 1993, *Concurrency Control in Object-Oriented Databases*. John Wiley & Sons.

Lanin, V. and D. Shasha: 1986, 'Tree Locking on Changing Trees'. *FJCC* pp. 380–389.

Lehti, P.: 2001, 'Design and Implementation of a Data Manipulation Processor for an XML Query Language'. Technical report, Technische Universitat Darmstadt. http://www.lehti.de/beruf/diplomarbeit.pdf.

Loomis, M.: 1995, *Object Databases, The Essentials*. Addison Wesley.

McHugh, J., S. Abiteboul, R. Goldman, D. Quass, and J. Widom: 1997, 'Lore: A Database Management System for Semistructured Data'. *SIGMOD Record* **26(3)**, 54–66.

Papadimitriou, C.: 1986, *The Theory of Database Concurrency Control*. Rockville, MD: Computer Science Press.

Silberschatz, A. and Z. Kedem: 1980, 'Consistency in Hierarchical Database Systems'. *Journal of the ACM* **27(1)**, 72–80.

Tatarinov, I., Z. Ives, A. Halevy, and D. Weld: 2001, 'Updating XML'. In: *Proceedings of SIGMOD Conference*.

Weikum, G. and G. Vossen: 2002, *Transactional Information Systems*. Morgan Kaufmann.

Wood, L. et al.: 1998, 'Document Object Model (DOM) Level 1 Specification'. *W3C Recommendation*.