XPath-based Concurrency Control for XML Data

Eun HYE CHOI† and Tatsunori KANAI†

† Corporate Research & Development Center, Toshiba Corporation
1, Komukai Toshiba-cho, Saiwai-ku, Kawasaki-shi, 212–8582 Japan
E-mail: {eunhye.choi,tatsunori.kanai}@toshiba.co.jp

Abstract For concurrency control of XML data, this paper proposes a new locking method based on XPath with a new data management model. Although increasing concurrency for XML data becomes an important issue in XML data management, few effective methods providing high concurrency of transactions working on the same XML document have been proposed so far. To overcome this problem, we propose a locking method that can achieve high concurrency while guaranteeing the serializability of transactions to the same XML document. In the proposed method, logical locks are set on XPath expressions used in transaction accesses, and conflicts that violate serializability are detected each time a read or a write access is requested by concurrent transactions. Efficient algorithms for detecting conflicts are also proposed in this paper. Since the proposed locking is at the level of precise data in XML documents, high concurrency for XML data can be achieved assuring serializability.
Key words Concurrency control, XML data management, XPath, Locking, Serializability, Phantom problem

1. Introduction

1.1 Backgrounds

As the eXtensible Markup Language (XML) [2] becomes a widely adopted standard for data representation in various application areas, the number of XML documents is rapidly growing and the subsequent need for sharing XML documents by different applications and multiple users is increasing. Concurrency control of transactions working on the same document is thus an important issue in XML data management. *Serializability*, which requires that concurrent transactions must produce the same result as the same transactions executed in a certain sequential order, has been considered as a fundamental problem for concurrency control, and locking is known as a robust technique to handle this problem. Locking to ensure the serializability has been extensively investigated for traditional RDBMSs, but not yet for XML data management systems. In the present XML data management, XML documents are usually stored either to relational databases as BLOBs or to native XML databases as XML format. However, in both cases, locking is often at the level of entire documents and thus no concurrency of transactions to the same documents is actually provided [1]. The aim of this paper is to overcome this problem and to achieve high concurrency for XML documents while ensuring serializability.

In order to ensure the serializability of transactions, locking must prevent the *phantom problem* [5]. A phantom indicates the data that was already disappeared from or will appear to a database, and thus locking on such phantoms is difficult since the phantoms do not physically exist at the time of locking. To solve the phantom problem, two well-known forms of locking have been proposed so far: physical index-based locking and logical predicate-based locking.

The index-based locking, such as key range locking [9], prevents the phantom problem by locking on the index entry to the phantom. So far, because of low locking cost and useful index structure such as the B-Tree [7], the index-based locking have been widely adopted for RDBMSs. However, the index-based locking is not applicable for XML data management in the absence of an efficient index structure for an XML document model.

The predicate-based locking, such as predicate locking [5] and precision locking [8], prevents the phantom problem by locking on the predicate that identifies the phantom. In the predicate locking, any conflict to violate serializability is detected by checking if two predicates used in concurrent transaction accesses are mutually satisfiable. The predicate locking is more general than the index-based locking, but is more expensive since such the satisfiability problem between arbitrary two predicates is known to be NP-complete. To overcome this shortcoming, the precision locking performs the conflict check between predicates and updates, instead of that between two predicates, in the following way: As a transaction performs a read access and a write access, the predicate used in the read access and the update by the write access are posted in a predicate list and an update list, respectively. In order to detect a conflict between predicates and updates by different transactions, each predicate (update) posted is checked against updates (predicates) by other transactions in the update list (the predicate list). The precision locking performs the conflict check against only actual predicates and updates, and thus provides high concurrency and a relatively lower cost than the predicate locking. Inherently, we conjecture that the concept of the precision locking could be helpful to ensure serializability in XML data management. However, the problem to perform practical conflict checks that can handle XML documents still remains.

1.2 New Results

Our contribution is a new locking method that guarantees serializability and provides high concurrency of transactions to the same XML documents. To our best knowledge, the only one method by Grabs et al. [6] tackled the same concurrency control problem for XML documents so far. They proposed a combination of well-known granularity locking and predicate locking which provides high concurrency, but their locking is applicable to only restricted XML documents with simple XPath query for transaction accesses. Against [6], our method allows general XML documents and full XPath query. In the proposed method, each time an access is requested, conflict checks to ensure serializability are performed in the following way: As for each read access, conflicts against all

```
<?xml version="1.0" ?>
< flowers >
    < flower >
        <name> Tulip </name>
        <color> Yellow </color>
        <price unit="JPY"> 150 </price>
    < /flower >
    < flower >
        <name> Rose </name>
        <color> Red </color>
        <price unit="JPY"> 500 </price>
    </ flower >
</ flowers >
```

図 1 An example XML document.

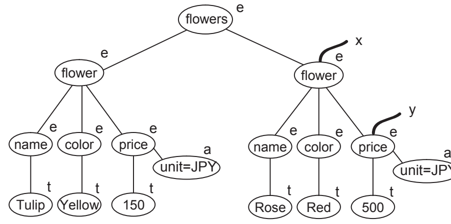

図 2 A document tree.

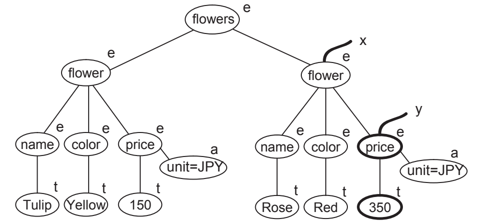

図 3 An updated document.

previous write accesses by distinct transactions are checked. As for each write access, conflicts against all previous read accesses by distinct transactions are checked. In the following, the former is called a read-write check; the latter is called a write-read check.

The basic ideas of the proposed method are as follows. First, we introduce a new data management model, which handles the three kinds of documents: (1) an original document in the database, denoted by D_{st}, (2) a copy of D_{st} for each transaction T_i, denoted by D_i and (3) a copy of D_{st} for detecting conflicts, denoted by D_{all}. Each transaction T_i has accesses to document D_i instead of original document D_{st}, and write accesses by T_i update both D_i and D_{all}. Document D_{all} thus represents the document state updated by all transactions. Only when T_i commits, the updates by T_i are reflected in D_{st}.

Next, we propose a logical locking approach in which locks are set on the XPath expressions used read accesses and conflicts between the XPath expressions and the updates by write accesses are checked. To detect conflicts, checking the satisfiability of an update with an XPath expression is necessary but is difficult since XPath is a path-based and semistructured query language, not a generic boolean predicate. In order to handle this deficiency, we present a sufficient condition such that any conflict between the XPath expression and the update can be detected by checking the condition when evaluating the expression on two documents such that one contains the update and the other does not. The proposed locking thus resolves the phantom problem by regarding XPath expressions as predicates.

Based on our data management model and XPath-based locking, conflicts between concurrent transaction accesses are checked as follows: The read-write check is performed by comparing nodes in documents D_i and D_{all} which are reachable when evaluating XPath expressions. Since the updates of all previous write accesses are reflected in D_{all}, the read-write check can be completed by just one comparison using D_{all}. On the other hand, dealing with the write access is typically complicated much more than dealing with the read access. The write-read check is performed based on the idea of using previous states of D_{all}. We also present an algorithm to dynamically determine whether the current state of D_{all} provides a large effect on subsequent write-read checks. In addition, we show that the proposed scheduling of saving previous states of D_{all} is always optimal in terms of the effects obtained.

2. Preliminaries

This section describes the XML document model and the transaction access model used in our research.

2.1 XML Document Model

An XML document is modeled as a tree, which details the parent-child relationship between various elements of the document. Each node in the tree has a type such as element, text and attribute. (For more detail of XML, refer to [2].) In the following, an XML document is referred to as a *document* for short. Fig. 1 and Fig. 2 respectively show an example XML document and its representation as a tree. In Fig. 2, "e", "t" and "a" attached to each node represent that the node type is element, text and attribute, respectively.

2.2 Transaction Model

We consider a set of concurrent transactions $\mathcal{T} = \{T_1, \cdots, T_n\}$ $(n \geqq 2)$ which accesses the same document where n denotes the number of transactions. Hereafter, the term *transactions* means concurrent active transactions otherwise indicated. In this paper, the case where a transaction accesses to more than one document is not considered since such accesses could be classified into the accesses to each document and only accesses to the same document is important when considering concurrency control.

Each transaction sequentially performs several read accesses (*R-accesses*, for short) and write accesses (*W-accesses*, for short) to the document. The time-ordered sequence of the accesses by transaction T_i is referred to as an *access sequence* of T_i and is denoted by AS_i. The access sequence of T_i then consists of *read sequences* (*R-sequences*, for short) and *write sequences* (*W-sequences*, for short), and is modeled as illustrated in Fig. 4. An read sequence (write sequence) indicates a sequence of consecutive read accesses (write accesses) in the access sequence. We also define the following notations:

- $WS_i(k)$ $(k > 0)$: the k-th write sequence of T_i.
- $|WS_i(k)|$: the number of write accesses in $WS_i(k)$.
- wn_i : the current number of write sequences of T_i.
- $RS_i(k)$ $(k \geqq 0)$: if $k = 0$, the first read sequence; otherwise, the read sequence right after write sequence $WS_i(k)$.

Fig. 4 shows an example access sequence of a transaction T_i. In the figure, the box and the vertical line represent a write access and a read sequence which contains one or more read accesses, respectively. Write sequences $WS_i(1)$ and $WS_i(2)$ are the first and the second write sequences respectively, and $|WS_i(1)| = 1$ and $|WS_i(2)| = 2$. Read sequence $RS_i(0)$ denotes the read sequence before the first write access, and read sequences $RS_i(1)$ and $RS_i(2)$ follow right after $WS_i(1)$ and $WS_i(2)$, respectively.

2.3 XPath Access Model

We assume that transactions have accesses to the document using the XPath [4], which is the language to address parts of the XML document and is being used as a base in a number of other XML standards such as XQuery [3]. We assume that the reader is familiar with XPath and basic notations such as an XPath expression, a location path, and an axis. (For the detail of XPath, refer to [4].)

An R-access is performed to the document by specifying an XPath expression that identifies a node (or nodes) in the document. A read operation is represented by *Read*($path$) with XPath expres-

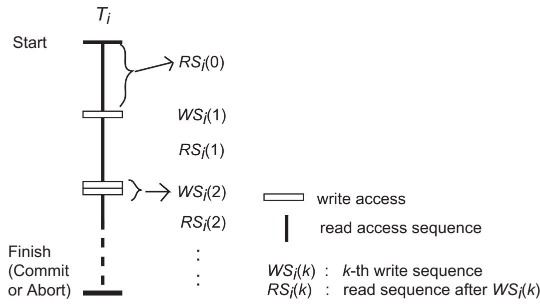

図 4 An access sequence of transaction T_i.

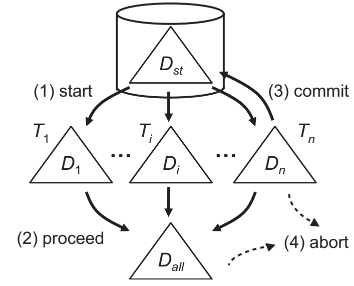

図 5 Data management model.

		Previous access by other transactions	
		Read	Write
Requested access	Read	a) −	b) √
	Write	c) √	d) −

√ means that the detection of a conflict is needed.

図 6 Conflicts to be checked.

sion *path*.

A W-access is performed to the document by specifying a write operation and the target node (or nodes) of the write operation. The target node is selected by the R-access to the document. We consider three kinds of write operations on the document: insert, delete, and replace. The insert operation and the delete operation could be to insert and delete a node (or nodes) in the document. The replace operation could be to replace the value of a node, such as an element node or an attribute node, with a new value. Note that the proposed locking does not depend on the type of write operation.

〔Example 1〕 Consider the document shown in Fig. 2. First, suppose that an R-access $Read(path)$ with $path$="flower[name=Rose] /price" is performed to the document. XPath expression $path$ is evaluated on the document and then element node "price" contained in "flower" element whose "name" element has the value of "Rose" is selected. The selected node is node y in Fig. 2. The selected node is marked with bold lines in Fig. 2. Note that the value of the selected node is "500", that is the value of its child text node. Next, suppose that a W-access that replaces the value of the selected node by the R-access to "350" is performed to the document. The document is then updated by the W-access as shown in Fig. 3. The updated parts are marked with bold lines in Fig. 3.

3. Proposed Locking Method

In order to ensure the serializability, a locking method must prevent any conflict between R-accesses and W-accesses executed concurrently by different transactions. The proposed locking method detects conflicts between XPath expressions used in R-accesses and updates by W-accesses by different transactions under our data management model. This section describes a new data management model and the overview of conflict check.

3.1 Data Management Model

The data management model is illustrated in Fig. 5 where each triangle represents a document state and the following three kinds of document states are contained:

- D_{st} : the committed document state in the database.
- D_i for T_i : the document state containing the updates of T_i.
- D_{all} : the document state containing the updates of all active transactions.

The state of D_{st} is copied to D_{all} initially, and as a new transaction T_i is issued, documents D_{st}, D_i and D_{all} are handled and updated in the following manner:

(1) When T_i starts, generate D_i as a copy of D_{st}.

(2) While T_i proceeds, T_i accesses to D_i instead of D_{st}. When D_i is updated by each W-access of T_i, the update is also reflected in D_{all}.

(3) When T_i commits, reflect the updates by T_i in D_{st}.

(4) When T_i aborts, D_i is just deleted and D_{all} is regenerated for taking away the updates by T_i reflected in D_{all}.

In our data management model, the abort or rollback of a transaction can be easily handled since the updates by each transaction T_i are reflected in document D_i but D_{st}. Document D_{all} containing the updates of all transactions is used for the proposed locking method to efficiently detect conflicts to violate serializability.

Here we define the *equivalence of nodes* in distinct documents. Nodes are *equivalent* to each other in the following cases: (1) one is copied from the other, (2) nodes are copied from the same node, (3) nodes are the same updates contained in different documents. For two sets of nodes, $N = \{v_1, \cdots, v_m\}$ and $N' = \{v'_1, \cdots, v'_m\}$, in distinct documents, N is equivalent to N' iff $v_j \equiv v'_j$ for any $1 \leqq j \leqq m$. For example, consider the documents shown in Fig. 2 and Fig. 3 again. Nodes x's in the two documents are equivalent but nodes y's are not equivalent to each other.

In addition, we define *merging* two documents D_i and D_j as follows: the updated part of T_i contained in D_i, the updated part of T_j contained in D_j and the equivalent part are reflected in the result document. Note that the updated part, i.e. nodes, in D_i and the updated part in D_j are mutually disjoint. Suppose that there are such nodes, then there are previous W-accesses by T_i and by T_j that conflict with each other. As will be shown later, such a conflict is detected when the W-access is requested. In the following, the term $Merge(D_i, D_j)$ denotes the document merging D_i and D_j.

3.2 Conflict Check

In the proposed locking, each time a transaction requests an R-access or a W-access, checking conflicts of the access against previous accesses by other transactions is performed. Among the four cases shown in Fig. 6, checking the read-write conflict (Case b) and the write-read conflict (Case c) is sufficient to ensure the serializability. There is no conflict between R-accesses. On the other hand, the write-write conflict can be detected by the read-write conflict or the write-read conflict since, before a write operation on some nodes that leads to a conflict, a read operation on the same nodes is performed.

The proposed locking method consists of the following two phases for checking conflicts: read-write check (R-W check, for short) and write-read check (W-R check, for short).

- Read-Write Check : Each time a transaction requests an R-access, check whether the R-access leads to the read-write conflict with previous W-accesses of other transactions.
- Write-Read Check : Each time a transaction requests a W-access, check whether the W-access leads to the write-read conflict

with previous R-accesses of other transactions.

When a conflict is detected by the R-W check or the W-R check, a requested access is delayed until the other transaction that the access conflicts finishes. Actually which transaction is blocked can be determined by transaction priority. Deadlock which occurs by blocking transactions could be detected by using well-known solutions such as the wait-for graph [7]. The proposed algorithms for the R-W check and the W-R check are given in Section 4 and Section 5, respectively.

4. Read-Write Check

This section describes the proposed method for the R-W check. Each time an R-access is requested, the R-W check is performed for detecting the R-W conflict caused by the R-access. The R-W conflict is detected based on the XPath evaluation under our data management model. A conflict of an R-access with previous W-accesses means that the result of the R-access to a document is different from that to the document updated by the W-accesses. Hereafter we define the boolean function $Conf(R, \mathcal{W})$ to be $true$ (1) iff an R-access R conflicts a sequence, $\mathcal{W}$, of W-accesses.

Consider that a transaction T_i requests an R-access $R_i = Read(path)$. In our data management, R_i is then performed to document D_i and some nodes on D_i are selected by evaluating XPath expression $path$ on D_i as a result. In the following, the function $GetN(D_i, path)$ is defined as the nodes on document D_i selected by XPath expression $path$. Let $\mathcal{W}_j$ denote a sequence of previous W-accesses by transaction T_j. Iff $Conf(R_i, \mathcal{W}_j) = true$ with $R_i = Read(path)$, then

$$GetN(D_i, path) \not\equiv GetN(Merge(D_i, D_j), path). \qquad (4.1)$$

(Since the previous W-accesses of T_j is reflected in D_j, $Merge(D_i, D_j)$ is equal to the state of D_i updated by $\mathcal{W}_j$.) The conflict of R_i with previous W-accesses by any other transaction T_j is then detected by checking formula (4.1).

Here we note that an R-access could conflict with W-accesses of more than one transactions even when the R-access does not conflict with those of each of the transactions. For transactions T_j and T_h in $(\mathcal{T} - \{T_i\})$, there is a case where $Conf(R_i, \mathcal{W}_j + \mathcal{W}_h) = true$ but $Conf(R_i, \mathcal{W}_j) \neq true$ and $Conf(R_i, \mathcal{W}_h) \neq true$. The conflict is then detected by checking if $GetN(D_i, R_i) \equiv GetN(D'_i, R_i)$ where D'_i is the state of D_i merging both D_j and D_h. On the other hand, there is also a case where $Conf(R_i, \mathcal{W}_j) = true$ or $Conf(R_i, \mathcal{W}_h) = true$ but $Conf(R_i, \mathcal{W}_j + \mathcal{W}_h) \neq true$. Hence the R-W conflict of R_i with previous W-accesses needs to be checked against all combinations of transactions in $(\mathcal{T} - \{T_i\})$, but such a conflict check against all the combinations is extremely time consuming. (The number of all combinations to be considered for each R-access is $2^{n-1} - 1$.)

To handle this problem, we present a sufficient condition for the R-W conflict of R_i such that a conflict caused by R_i with previous W-accesses by any set of other transactions is detected by checking if the condition holds in the XPath evaluation on the state of document D_i updated by the W-accesses of all transactions. In other words, if R_i causes a conflict with previous W-accesses by any set of transactions in $\mathcal{T} - \{T_i\}$, then the condition holds with the state of document D_i updated by $\mathcal{W}_1 + \cdots + \mathcal{W}_j + \cdots + \mathcal{W}_n (= \mathcal{W})$.

In the following, we define the function $GetD(D_i, \mathcal{W})$ as the state of document D_i updated by W-accesses $\mathcal{W}$. Let N_j in $GetD(D_i, \mathcal{W})$ be the part, i.e. nodes, updated by $\mathcal{W}_j$.

$$\text{For any } N_j \text{ and } N_h\ (j \neq h),\ N_j \cap N_h \neq \emptyset \qquad (4.2)$$

since there is no conflict between previous W-accesses $\mathcal{W}_j$ and $\mathcal{W}_h$. (Such the conflict was previously detected.) In addition, for each node in N_j, all nodes in the subtree whose root is the node are also contained in N_j.

In the following explanation for the conflict check, we refer to the semantics of XPath presented in [10], which is shown in Appendix A. The proposed method however is a general approach that can be applicable to any semantics for XPath. To save space, we omit the precise explanation for the semantics of XPath, so refer to [10] for more detail. The semantics of XPath is specified by three functions $\mathcal{S}$, $\mathcal{Q}$, and $\mathcal{E}$ with two parameters: an axis a and a context node x. Among the three functions, both functions $\mathcal{Q}$ and $\mathcal{E}$ recursively call function $\mathcal{S}$ and the results of them (boolean and numerical value, respectively) are determined from the nodes returned by $\mathcal{S}$. Here we thus concentrate on function $\mathcal{S}$. Function $\mathcal{S}^a[\![p]\!]x$ denotes the nodes selected by location path p with parameters a and x. When XPath expression $path$ is evaluated on document D_i, function $\mathcal{S}$ could be called several times for specifying the resulting nodes on D_i. We use the term $\mathcal{S}^a[\![p]\!]x(D_i)$ to indicate the result of $\mathcal{S}$ on D_i. Lemma 1 and Theorem 1 then hold.

[Lemma 1] If $GetN(D_i, path) \not\equiv GetN(D'_i, path)$ with XPath expression $path$ and two documents D_i and D'_i, function $\mathcal{S}$ such that $\mathcal{S}^a[\![p]\!]x(D_i) \not\equiv \mathcal{S}^a[\![p]\!]x(D'_i)$ is called in the evaluation of $path$ on D_i and D'_i.

proof: Suppose that such function $\mathcal{S}$ is not called in the evaluation of $path$, that is, $\mathcal{S}$ always returns the equivalent results for D'_i and D'_i. Then, obviously, $GetN(D_i, path) \equiv GetN(D'_i, path)$. This is a contradiction. □

[Theorem 1] Consider two sets, $\mathcal{T}'$ and $\mathcal{T}''$, of transactions such that $\mathcal{T}' \cap \mathcal{T}'' = \emptyset$ and $\mathcal{T}' + \mathcal{T}'' \subseteq \mathcal{T} - \{T_i\}$. Let $\mathcal{W}'$ and $\mathcal{W}''$ be previous W-accesses of $\mathcal{T}'$ and $\mathcal{T}''$, respectively. If $R_i (= Read(path))$ conflicts with $\mathcal{W}'$, function $\mathcal{S}$ such that

$$\mathcal{S}^a[\![p]\!]x(D_i) \not\equiv \mathcal{S}^a[\![p]\!]x(GetD(D_i, \mathcal{W}' + \mathcal{W}'')) \qquad (4.3)$$

is called in the evaluation of $path$ on D_i and $GetD(D_i, \mathcal{W}' + \mathcal{W}'')$.

proof: Let $\mathcal{W} = \mathcal{W}' + \mathcal{W}''$, $D'_i = GetD(D_i, \mathcal{W}')$, and $D''_i = GetD(D_i, \mathcal{W})$. By Lemma 1, the theorem holds if $GetN(D_i, path) \not\equiv GetN(D''_i, path)$. Otherwise, $GetN(D_i, path) \not\equiv GetN(D'_i, path)$ and thus function $\mathcal{S}$ such that $\mathcal{S}^a[\![p]\!]x(D_i) \not\equiv \mathcal{S}^a[\![p]\!]x(D'_i)$ is called in the evaluation of $path$ on D_i and D'_i. Then, there are a node $y \in \mathcal{S}^a[\![p]\!]x(D_i)$ and a node $y' \in \mathcal{S}^a[\![p]\!]x(D'_i)$ such that $y \not\equiv y'$. Moreover, by formula (4.2), there is a node $y'' \in \mathcal{S}^a[\![p]\!]x(D''_i)$ such that $y'' \equiv y'$. Since $y \not\equiv y''$, $\mathcal{S}^a[\![p]\!]x(D_i) \not\equiv \mathcal{S}^a[\![p]\!]x(D''_i)$. □

By Theorem 1, using document D_i and a document updated by transactions, the conflict between R-access R_i and previous W-accesses of any subset of transactions can be detected. In our data management, the updates by all previous W-accesses are reflected in D_{all}. Thus, if an R-access $R_i = Read(path)$ causes the R-W conflict with previous W-accesses of any set of transactions, the evaluation of $path$ on D_i and D_{all} must call a function $\mathcal{S}$ such that

$$\mathcal{S}^a[\![p]\!]x(D_i) \not\equiv \mathcal{S}^a[\![p]\!]x(D_{all}). \qquad (4.4)$$

Hence, the R-W conflict of R_i is detected by checking formula (4.4) when function $\mathcal{S}$ is called for the XPath evaluation. Note that this is a sufficient condition for a conflict, i.e., the calling such function $\mathcal{S}$ whose results are not equivalent does not necessarily lead to a

conflict. A more efficient sufficient condition for the conflict check is omitted here (although we have some idea) but will be addressed in future work. In addition, the equivalence check for function $\mathcal{S}$ is reduced in the following way. Consider the case where function $\mathcal{S}^a[\![p]\!]x(D_i)$ recursively calls $\mathcal{S}'^{a'}[\![p']\!]x'(D_i)$ with $x' \in \mathcal{S}$. As mentioned previously, if a node x' in document D_i (D_{all}) is updated, nodes in the subtree whose root is x' are also updated. Hence, if axis(注1) a' is for searching nodes in the subtree of context node x' and formula (4.4) holds, then $\mathcal{S}'^{a'}[\![p']\!]x'(D_i) \not\equiv \mathcal{S}'^{a'}[\![p']\!]x'(D_{all})$. Therefore, when function $\mathcal{S}$ is recursively called by function $\mathcal{S}'$ with such an axis, the equivalence check for $\mathcal{S}'$ can be omitted since the conflict can be detected by the equivalence check for $\mathcal{S}$.

[Example 2] Consider that T_i requests an R-access $Read(path)$ with $path$="flower[price<400]/name". Suppose that D_i equals to the document shown in Fig. 2 and D_{all} equals to the document shown in Fig. 3, which was updated by a previous W-access of the other transaction. When $path$ is evaluated on D_i and D_{all}, function $\mathcal{S}^a[\![price]\!]x$ with $a = child$ is called. The resulting node, y, of $\mathcal{S}$ on D_i is not equivalent to that on D_{all}, that is, $\mathcal{S}^a[\![price]\!]x(D_i) \not\equiv \mathcal{S}^a[\![price]\!]x(D_{all})$. Therefore, the conflict of the R-access with the W-access is detected.

Here we define the boolean function $Check(D, D', R)$ with two documents D and D' and R-access $R = Read(path)$ as follows: $Check = ok$ (1) if function $\mathcal{S}$ is called in the evaluation of $path$ such that the resulting nodes on D are not equivalent to the resulting nodes on D'; $Check \neq ok$, otherwise. The following theorem then holds.

[Theorem 2] If an R-access R_i by a transaction T_i leads to the R-W conflict, $Check(D_i, D_{all}, R_i) \neq ok$. □

The proposed algorithm for the R-W check is then as follows: When a transaction T_i requests an R-access R_i,

(1) Check if $Check(D_i, D_{all}, R_i) = ok$.

(2) If it holds, then R_i causes no R-W conflict and thus T_i proceeds.

(3) Otherwise, R_i causes a R-W conflict with other transactions. Find transaction $T_j (i \neq j, 1 \leqq j \leqq n)$ that causes the conflict with R_i, and block T_i until T_j finishes. When T_j finishes, T_i restarts.

In the R-W check, XPath evaluation is processed in both D_i and D_{all}. The execution cost needed for the proposed algorithm is then the sum of the cost for evaluating an XPath expression and the cost for equivalence checks in the XPath evaluation. Equivalence checks can be easily implemented by, for instance, sharing a pointer between equivalent nodes in distinct documents.

5. Write-Read Check

This section describes the proposed method for the W-R check. Each time a W-access is requested, the W-R check is performed for detecting the W-R conflict caused by the W-access, using D_{all} under our data management model.

Consider that a transaction T_i requests a W-access W_i. In our data management, W_i is performed to document D_i, and then D_i is updated by W_i as a result. Let $D'_i = GetD(D_i, W_i)$. Let R_j be a previous R-access by any other transaction T_j. By theorem 2, if W_i causes a conflict with R_j, then $Check(D'_j, Merge(D'_j, D'_i), R_j) \neq ok$ where D'_j denotes the previous state of D_j to that R_j was performed. When W_i is requested,

(注1): Such axes in XPath are child, descendant, self, attribute, and namespace.

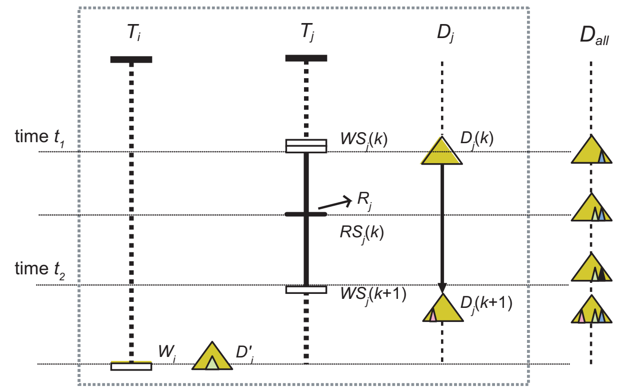

図 7 Example two transactions.

this conflict check for W_i is needed against each of all previous R-accesses by any transaction $T_j (\in \mathcal{T} - \{T_i\})$. Thus all the previous states of D_j's for each T_j are needed for the W-R check when T_i requests a W-access.

Two simple ways to obtain the previous states of D_j's are considered: one is storing all the previous states and the other is regenerating the previous states by performing previous updates again. However both of them are impractical if there are a number of concurrent transactions and updates due to memory and time consuming, respectively. To overcome this deficiency, we present the method for an efficient W-R check using previous states of D_{all} instead of using previous states of D_j for every transaction T_j.

5.1 Write-Read Check using previous D_{all}'s

Consider a W-access W_i requested by T_i and a previous R-access R_j in R-sequence $RS_j(k)$ $(0 \leqq k \leqq wn_j)$ of T_j. (Recall that wn_j is the current number of W-sequences of T_j.) Now we define the notation $D_j(k)$ with $(0 \leqq k \leqq wn_j)$ as follows: if $k = 0$, $D_j(k) = D_{st}$; otherwise, $D_j(k) = GetD(D_{st}, \mathcal{W})$ where $\mathcal{W} = WS_j(1) + \cdots + WS_j(k)$. Then $D_j(k)$ is equivalent to the previous state of D_j to that R-access R_j in $RS_j(k)$ is performed. Hence, a conflict with R_j caused by W_i is detected by checking if

$$Check(D_j(k), Merge(D'_i, D_j(k)), R_j) \neq ok. \quad (5.1)$$

In the following explanation, we refer to Fig. 7, which illustrates an example of two transactions T_i and T_j. Suppose that W-sequence $WS_j(k)$ finished and D_j and D_{all} were updated by the last W-access in $WS_j(k)$ at time t_1. Suppose also that the next W-sequence $WS_j(k+1)$ started right after time t_2. Document $D_j(k)$ is then the state of D_j at any time t with $t_1 \leqq t \leqq t_2$. ($D_j(k)$ equals to the state of D_j in which updates until $WS_j(k)$ are reflected.) Let D'_{all} be the state of D_{all} at any time $t(t_1 \leqq t \leqq t_2)$. Document D'_{all} is the document state updated by all transactions by time t, that is, D'_{all} is the document merging $D_j(k)$ and all D_l's$(l \neq j, 1 \leqq l \leqq n)$ at time t. By Theorem 2,

$$Check(D_j(k), D'_{all}, R_j) = ok \quad (5.2)$$

since R_j was checked not to lead to the R-W conflict when it is requested. By formula (5.1) and equation (5.2), we obtain the following theorem.

[Theorem 3] If a W-access W_i by a transaction T_i leads to the W-R conflict, there is an R-access R_j in $RS_j(k)(0 \leqq k \leqq wn_j)$ by any other transaction T_j such that

$$Check(D'_{all}, Merge(D'_i, D'_{all}), R_j) \neq ok$$

where $D'_i = GetD(D_i, W_i)$ and D'_{all} is a previous state of D_{all} at any time after $WS_j(k)$ and before $WS_j(k+1)$. □

By Theorem 3, the W-R conflict caused by a requested W-access with any R-accesses in $RS_j(wn'_j)$ can be detected by using the previous state of D_{all} when wn'_j was the current number of W-sequences of T_j.)

For the W-R check using previous states of D_{all}, the states of D_{all} then need either to be saved before updating or to be regenerated when checking the conflict. If all the previous states of D_{all} can be held, no write operations for regenerating the states of D_{all} is needed for the W-R check. However, it is hardly a practical solution due to the limitation of memory resources. In the following, the term sn_{max} denotes the maximum number of states of D_{all} that can be held in the system. We assume that the value of sn_{max} can be changed depending on the available size of memory. An algorithm that determines sn_{max} states of D_{all} providing large effects on subsequent W-R checks is proposed in Section 5. 2.

In the following, the proposed method for the W-R check using previous states of D_{all} is described. We define several notations for the following explanation. The term *s-point* is used to indicate the point of time that the state of D_{all} is saved.

- $sn(\leqq sn_{max})$: the current number of s-points.
- $sp(h)$ $(1 \leqq h \leqq sn)$: the h-th s-point.
- $D_s(h)$: the state of D_{all} saved at s-point $sp(h)$.
- $wn_i(h)$ for each T_i : the value of wn_i at s-point $sp(h)$.

We say that s-point $sp(h)$ is set in write sequence $WS_i(k)$ for each transaction $T_i(\in \mathcal{T})$ if $k = wn_i(h)$.

For each R-sequence $RS_j(k)(1 \leqq k \leqq wn_j)$, if there is s-point $sp(h)$ such that $wn_i(h) = k$, then the W-R check against R-accesses in $RS_j(k)$ can be performed using $D_s(h)$. Suppose that there is no s-point $sp(h)$ such that $wn_j(h) = k$. There are two cases to be considered. The first case is that there is s-point $sp(h')$ such that $wn_j(h') = k'$ where k' is the maximum value with $k' < k$. The second case is that there is no such s-point $sp(h')$. In the first case, the updates until W-sequence $WS_j(k')$ are reflected in $D_s(h')$. Let D'_j be the state of $D_s(h')$ in which the updates by T_j from $WS_j(k'+1)$ to $WS_j(k)$ are also reflected. In the second case, let D'_j be the state of D_{st} in which the updates by T_j until W-sequence $WS_j(k)$ are reflected. Then, in the both cases, the W-R check against R-accesses in $RS_j(k)$ can be performed using D'_j.

The proposed algorithm for the W-R check is then as follows: When a transaction T_i requests a W-access W_i,

(1) Prepare D which is a copy of document D_{st}, and prepare D'_i which is the state of D_i updated by W_i.

(2) For each transaction $T_j(\in \mathcal{T} - \{T_i\})$, execute accesses from the first R-sequence to the R-sequence just before $WS_j(wn_j(1))$, i.e, the W-sequence in which the first s-point is set.

(2.1) If the access is W-access W_j, reflect the update by W_j in document D.

(2.2) If the access is R-access R_j, then check if $Check(D, Merge(D'_i, D), R_j) = ok$. If it holds, then W_i does not conflict with R_j, and thus go (2) and execute the next access. Otherwise, W_i conflicts with R_j, and thus block T_i until T_j finishes. When T_j finishes, T_i restarts.

(3) For each s-point $sp(h)$ with $1 \leqq h \leqq sn$,

(3.1) Prepare D which is a copy of document $D_s(h)$.

(3.2) For every transaction T_j and every R-access R_j in $RS_j(wn_j(h))$, check if $Check(D, Merge(D'_i, D), R_j) = ok$. If not so, W_i conflicts with R_j and thus block T_i until T_j finishes. When T_j finishes, T_i restarts.

(3.3) For each transaction T_j, if an s-point is not set in W-sequence $WS_j(wn_j(h)+1)$, execute accesses from $WS_j(wn_j(h)+1)$ to the R-sequence just before the W-sequence in which s-point $sp(h+1)$ is set or to the latest R-sequence.

(3.3.1) If the access is W-access W_j, reflect the update by W_j in document D.

(3.3.2) If the access is R-access R_j, then check if $Check(D, Merge(D'_i, D), R_j) = ok$. If it holds, then W_i does not conflict with R_j, and thus go (3.3) and execute the next access. Otherwise, W_i conflicts with R_j, and thus block T_i until T_j finishes. When T_j finishes, T_i restarts.

The pseudo code of the proposed algorithm for the W-R check based on s-points is shown in Appendix B. Steps 1, 2, and 3 in Appendix correspond to parts (1), (2), and (3) of the above explanation, respectively.

By using the states of D_{all} that are saved at s-points, the write operations for generating previous states of D_{all} in subsequent W-R checks are reduced. The number of write operations can be reduced by each s-point $sp(h)$ is referred as $E(h)$ in the following section. (The definition of $E(h)$ and the determination of s-point $sp(h)$ providing a large value of $E(h)$ are given in Section 5. 2.) Let nr_j and nw_j be the number of R-accesses and W-accesses in AS_j, respectively. In addition, let cr_j and cw_j be the cost of performing an R-access and evaluating function $Check$, and the cost of performing a W-access, respectively. When a W-access W_i is requested, the execution cost needed for the proposed W-R check on s-points is then equal to

$$\sum_{1 \leqq j \leqq n, i \neq j} (nr_j \times cr_j + nw_j \times cw_j)$$
$$-\Big(\sum_{1 \leqq h \leqq sn_{max}} E(h) - \sum_{1 \leqq g \leqq wn_i, wn_i(g) \neq wn_i(g-1)} |WS_i(g)| \Big) \times cw_j.$$

5. 2 Determination of S-Points

This section presents the method to dynamically determine the point of time to save the state of D_{all}. Since D_{all} is continuously updated, we need to check if the current state of D_{all} has a large effect on subsequent W-R checks and to determine if setting an s-point. In the following, the way to measure the effect of each s-point, i.e. the effect obtained by D_{all} saved at the s-point is first described and the algorithm to determine if setting an s-point or not is next given.

Whenever a W-access is requested by transaction T_i after s-point $sp(h)$, the use of $D_s(h)$ can reduce $\sum_{1 \leqq j \leqq n, j \neq i} |WS_j(wn_j(h))|$ write operations in the W-R check for the W-access. As a consequence, it is considered that the effect obtained by saving the state of D_{all} at s-point $sp(h)$ is proportional to the sum of the numbers of W-accesses in $WS_i(wn_i(h))$'s with $1 \leqq i \leqq n$. We define $E(h)$ of a s-point $sp(h)$ as follows:

$$\begin{cases} \text{If } h = 1, & E(h) = \sum_{1 \leqq i \leqq n} |WS_i(wn_i(h))|. \\ \text{Otherwise,} & E(h) = \sum_{1 \leqq i \leqq n, wn_i(h) \neq wn_i(h-1)} |WS_i(wn_i(h))|. \end{cases}$$

When $h = 1$, i.e., the s-point is the first one, effect $E(h)$ is the sum of the numbers of W-accesses in the recent W-sequences of all transactions. On the other hand, when $h > 1$, the number of W-accesses in $WS_i(wn_i(h))$ is not added to $E(h)$ if $wn_i(h) = wn_i(h-1)$. In that case, the W-R check against $RS_i(wn_i(h))$ can be performed using either $D_s(h)$ or $D_s(h-1)$. Thus we add

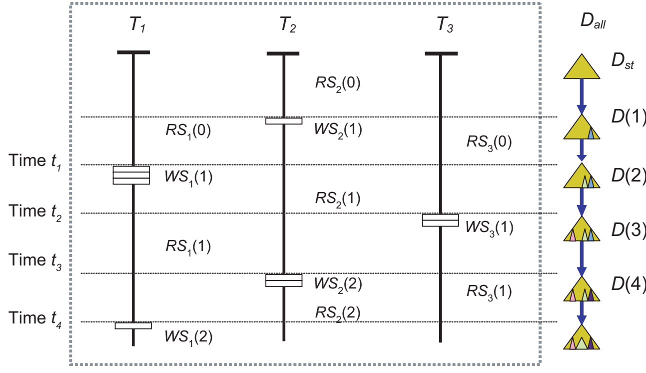

図 8 Example concurrent transactions.

$|WS_i(wn_i(h))|$ to the effect of the previous s-point, and not add to $E(h)$ to prevent redundancy.

In addition, we also define the reduced effect, $E^-(h)$, by deleting s-point $sp(h)$. For each s-point $sp(h)$, $E^-(h)$ is computed as follows: Initially, $E^-(h) = E(h)$. For each T_i,

(1) If $sp(h)$ is not the latest, $(wn_i(h) = wn_i(h+1))$, and $((h = 1)$ or $(wn_i(h) \neq wn_i(h-1)))$, then reduce $|WS_i(wn_i(h))|$ from $E^-(h)$. When deleting $sp(h)$, $|WS_i(wn_i(h))|$ is then added to $E(h+1)$.

(2) If $sp(h)$ is the latest s-point and $(wn_i(h) = wn_i)$, then reduce $|WS_i(wn_i(h))|$ from $E^-(h)$. When deleting $sp(h)$, $|WS_i(wn_i(h))|$ is then added to the effect of a new s-point.

Using effects $E(h)$ and $E^-(h)$ as the criteria, a new s-point is set if its effect is larger than the previous s-point having the minimum value, E^-_{min}, of $E^-(h)(1 \leqq h \leqq sn_{max})$. In the following, sp_{min} denotes the previous s-point having E^-_{min}.

We present the proposed method that determines sn_{max} s-points that provide large effects. Right before D_{all} is updated by W-accesses in a new W-sequence, the proposed determination of an s-point is performed as follows:

(1) Calculate the increased effect, E^+, by setting a new s-point for saving the current state of D_{all}.

(2) If sn, i.e. , the current number of s-points is less than sn_{max}, set a new s-point and its effect E^+.

(3) Otherwise, compare E^+ with E^-_{min}. Iff E^+ is larger than E^-_{min}, delete s-point sp_{min} and set a new s-point $sp(sn_{max})$. If sp_{min} is not the latest s-point, effect $E(sn_{max})$ of the new s-point is equal to E^+. Otherwise, $E(sn_{max}) = E^+ + e(= E(sp_{min}) - E^-_{min})$ where e is the effect added to the next s-point when s-point sp_{min} ie deleted.

The pseudo code of the proposed algorithm to determine s-points is shown in Appendix C. Steps 1, 2, and 3 in Appendix C correspond to parts (1), (2), and (3) of the above explanation, respectively.

Since a new s-point is always set only when the increased effect by the new s-point is larger than the minimum effect decreased by deleting a previous s-point, the s-point schedule by the proposed algorithm is optimal under the given time-ordered write sequences of transactions.

［Example 3］ We illustrated the determination of s-points for the example transactions shown in Fig. 8. In Fig. 8, each times t_1, t_2, t_3, and t_4 denote the times right before $WS_1(1)$, $WS_3(1)$, $WS_2(2)$, and $WS_1(2)$, respectively. Document $D(i)$ with $1 \leqq i \leqq 4$ denotes the state of D_{all} at time t_i. Assume sn_{max}=2. In Fig. 9, we show the scheduling of s-points at each time $t_i(1 \leqq i \leqq 4)$ and the information of s-points, i.e. $wn_i(h)$ for each transaction $T_i(1 \leqq i \leqq 3)$,

(1) at time t_1

$sp(h)$	$wn_1(h)$	$wn_2(h)$	$wn_3(h)$	$E(h)$	$E^-(h)$	$D_s(h)$
$sp(1)$	0	1	0	1	-	$D(1)$

(2) at time t_2

$sp(h)$	$wn_1(h)$	$wn_2(h)$	$wn_3(h)$	$E(h)$	$E^-(h)$	$D_s(h)$
$sp(1)$	0	1	0	1	0	$D(1)$
$sp(2)$	1	1	0	3	-	$D(2)$

(3) at time t_3

$sp(h)$	$wn_1(h)$	$wn_2(h)$	$wn_3(h)$	$E(h)$	$E^-(h)$	$D_s(h)$
~~$sp(1)$~~	~~0~~	~~1~~	~~0~~	~~1~~	~~0~~	~~$D(1)$~~
$sp(1)$	1	1	0	4	0	$D(2)$
$sp(2)$	1	1	1	2	-	$D(3)$

(4) at time t_4

$sp(h)$	$wn_1(h)$	$wn_2(h)$	$wn_3(h)$	$E(h)$	$E^-(h)$	$D_s(h)$
~~$sp(1)$~~	~~1~~	~~1~~	~~0~~	~~4~~	~~0~~	~~$D(2)$~~
$sp(1)$	1	1	1	6	1	$D(3)$
$sp(2)$	1	2	1	2	-	$D(4)$

— means that the reduced effect is not determined yet.

図 9 An example s-point schedule.

effect $E(h)$, and $D_s(h)$ saved at s-point $sp(h)$.

At times t_1 and t_2, s-points are set because two($=sn_{max}$) s-points can be set. When the first s-point $sp(1)$ is set, $D(1)$ is saved as $D_s(1)$ and effect $E(1)$ is 1($=wn_1(1)+wn_2(1)+wn_3(1)$), as shown in Fig. 9(1). As shown in Fig. 9(2), when the second s-point $sp(2)$ is set, $D(2)$ is saved as $D_s(2)$. Effect $E(2)$ equals to 3($=wn_1(2)$) since $wn_2(1) = wn_2(2)$ and $wn_3(1) = wn_3(2)$.

At time t_3, the number of s-points equals to sn_{max}, and thus it is necessary to determine whether a new s-point is set and s-point $sp_{min}(= sp(1))$ is deleted. Effect $E^-(1)$ reduced by deleting $sp(1)$ equals to 0 since $wn_2(1)$ is added to effect $E(2)$ of s-point $sp(2)$ when deleting $sp(1)$. Effect $E^+(= 2)$ increased by setting a new s-point is larger than $E^-_{min}(= E^-(1))$, and thus a new s-point is set as shown in Fig. 9(3). At time t_4, s-point sp_{min} is $sp(1)$ and effect $E^-(1)$ equals to 0. Effect $E^+(= 2)$ increased by setting a new s-point is larger than $E^-(1)$, and thus a new s-point is set as shown in Fig. 9(4).

6. Conclusion

In this paper, we proposed a new locking method guaranteeing serializability which supports general XML documents and full XPath Query. The proposed method resolves the phantom problem by adopting a logical locking approach and achieves high concurrency by producing locking at the level of precise data in XML documents. In the proposed method, locks are set on XPath expressions used in transaction accesses and conflicts between the XPath expressions and the updates by different transactions are checked to ensure serializability. To detect conflicts efficiently, we introduced a new data management model, which handles two versions of a document: document D_i in which updates by each transaction T_i are reflected and document D_{all} in which updates by all transactions are reflected. Under our data management model, conflict checks are performed based on XPath evaluation. As for the read-write check when a transaction T_i requests a read access, XPath expression used in the read access is evaluated on documents D_i and D_{all} and the equivalence of nodes in both documents which are reachable by the expression is checked. As for the write-read check, previous states of D_{all} saved at s-points are used. A dynamic algorithm to determine s-points providing large effects on subsequent write-read checks was also proposed and the proposed s-point schedule was shown to be always optimal in terms of the effect. To evaluate the proposed algorithms for the read-write check and the write-read check, we performed the qualitative analysis of computing costs and

confirmed the effectiveness. The implementation of the proposed method and the quantitative analysis of execution times and memory sizes needed for the proposed locking should be included in future works.

Reference

[1] R. Bourret, "XML and databases," Internet Document, February 2002, http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[2] T. Bray, J. Paoli and C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0.," *W3C Recommendation*, February 1998, http://www.w3.org/XML.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie and J. Simeon, "XQuery 1.0: An XML Query Language," *W3C Working Draft*, August 2002, http://www.w3.org/XML/Query.

[4] J. Clark and S. DeRose, "XML Path Language (XPath) 1.0.," *W3C Recommendation*, November 1999, http://www.w3.org/TR/xpath.

[5] K. P. Eswaran, J. Gray, R. Lorie and I. Traiger, "The notions of consistency and predicate locks in a database systems," *Comm. of ACM*, Vol. 19, No. 11, pp. 624–633, November 1976.

[6] T. Grabs, K. Böhmd and H. Schek, "XMLTM: efficient transaction management for XML documents," *Proc. of the 19th CIKM Conference*, pp. 142–152, 2002.

[7] P. Gray and A. Reuter, "Transaction processing: concepts and technology," *Morgan Kaufmann*, 1993.

[8] J. R. Jordan, J. Banerjee and R. B. Batman, "Precision locks," *Proc. of ACM SIGMOD International Conference on Management of Data*, pp. 143–147, April 1981.

[9] D. B. Lomet, "Key range locking strategies for improved concurrency," *Proc. of the 19th VLDB Conference*, pp. 655–664, 1993.

[10] P. Wadler, "Two semantics for XPath," *Technical report*, January 2000, http://www.research.avayalabs.com/user/wadler/papers/xpath-semantics.

Appendix

A. Semantics of XPath [10]

$$\begin{array}{lcl}
\mathcal{S} : Axis \rightarrow Pattern \rightarrow Node \rightarrow Set(Node) & & \\
\mathcal{S}^a[\![p_1 | p_2]\!]x & = & \mathcal{S}^a[\![p_1]\!]x \cup \mathcal{S}^a[\![p_2]\!]x \\
\mathcal{S}^a[\![/p]\!]x & = & \mathcal{S}^a[\![p]\!](root(x)) \\
\mathcal{S}^a[\![p_1/p_2]\!]x & = & \{x_2 | x_1 \in \mathcal{S}^a[\![p_1]\!]x, x_2 \in \mathcal{S}^a[\![p_2]\!]x_1\} \\
\mathcal{S}^a[\![a_1 :: p_1]\!]x & = & \mathcal{S}^{a_1}[\![p_1]\!]x \\
\mathcal{S}^a[\![n]\!]x & = & \{x_1 | x_1 \in \mathcal{A}[\![a]\!]x, nodetype(x_1) = \mathcal{P}[\![a]\!], \\
 & & \quad name(x_1) = n\} name(x_1) = n\} \\
\mathcal{S}^a[\![*]\!]x & = & \{x_1 | x_1 \in \mathcal{A}[\![a]\!]x, nodetype(x_1) = \mathcal{P}[\![a]\!]\} \\
\mathcal{S}^a[\![\texttt{text()}]\!]x & = & \{x_1 | x_1 \in \mathcal{A}[\![a]\!]x, nodetype(x_1) = \texttt{Text}\} \\
\mathcal{S}^a[\![p\texttt{[}q\texttt{]}]\!]x & = & \text{let } S_1 = \mathcal{S}^a[\![p]\!]x \text{ in} \\
 & & \text{let } n = size(S_1) \text{ in} \\
 & & \{x_1 | \\
 & & \quad x_1 \in S_1 \\
 & & \quad \text{let } j = size(\{x_2 | x_2 \in S_1, x_2 \leqq_{doc} x_1\}) \text{ in} \\
 & & \quad \text{let } k = (\text{if } \mathcal{D}[\![a]\!] = \texttt{forward} \text{ then } j \text{ else } n + 1 - j) \\
 & & \quad \text{in } \mathcal{Q}[\![q]\!](x_1, k, n)\}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{Q} : Qualifier \rightarrow (Node, Number, Number) \rightarrow Boolean & & \\
\mathcal{Q}[\![q_1 \texttt{ and } q_2]\!] & = & \mathcal{Q}[\![q_1]\!](x, k, n) \wedge \mathcal{Q}[\![q_2]\!](x, k, n) \\
\mathcal{Q}[\![q_1 \texttt{ or } q_2]\!] & = & \mathcal{Q}[\![q_1]\!](x, k, n) \vee \mathcal{Q}[\![q_2]\!](x, k, n) \\
\mathcal{Q}[\![\texttt{not(}q\texttt{)}]\!] & = & \neg \mathcal{Q}[\![q]\!](x, k, n) \\
\mathcal{Q}[\![p]\!](x, k, n) & = & \mathcal{S}^{\text{child}}[\![p]\!]x \neq \emptyset \\
\mathcal{Q}[\![e_1\texttt{=}e_2]\!](x, k, n) & = & \mathcal{E}[\![e_1]\!](x, k, n) = \mathcal{E}[\![e_2]\!](x, k, n)
\end{array}$$

$$\begin{array}{lcl}
\mathcal{E} : Expr \rightarrow (Node, Number, Number) \rightarrow Number & & \\
\mathcal{E}[\![e_1 \texttt{ + } e_2]\!](x, k, n) & = & \mathcal{E}[\![e_1]\!](x, k, n) + \mathcal{E}[\![e_2]\!](x, k, n) \\
\mathcal{E}[\![e_1 \texttt{ * } e_2]\!](x, k, n) & = & \mathcal{E}[\![e_1]\!](x, k, n) \times \mathcal{E}[\![e_2]\!](x, k, n) \\
\mathcal{E}[\![\texttt{position()}]\!](x, k, n) & = & k \\
\mathcal{E}[\![\texttt{last()}]\!](x, k, n) & = & n \\
\mathcal{E}[\![i]\!](x, k, n) & = & i
\end{array}$$

B. Algorithm for the write-read check based on s-points

/* **Step 1**: initialization */
W_i := the requested W-access by T_i
$D := D_{st}$;
$D'_i := GetD(D_i, W_i)$;

/* **Step 2**: the W-R check for the first R-sequence */
For each T_j with $j \neq i$ and $1 \leqq j \leqq n$
  $l_{end} := 0$;
  IF $(sn = 0)$ Then $l_{end} := wn_j$
  Else IF $(wn_j(1) \neq 0)$ $l_{end} := wn_j(1) - 1$;
  For $(l := 0$ to $l_{end}; l := l + 1)$
    For each R-access R_j in $RS_j(l)$
      IF $Check(D, Merge(D'_i, D), R_j) \neq ok$ Then
        Add edge $(T_j \rightarrow T_i)$ to the wait-for graph;
        Check deadlock; Exit;
      End_IF
    End_For
    IF $(l \neq l_{end})$ Then
      For each W-access W_j in $WS_j(l + 1)$
        $D' := GetD(D', W_j)$;
      End_For
    End_IF
  End_For
End_For

/* **Step 3**: the W-R check based on s-points */
For $(h := 1$ to $sn; h := h + 1)$
  For each T_j with $j \neq i$ and $1 \leqq j \leqq n$
    IF $((h = 1)$ or $(wn_j(h) \neq wn_j(h - 1)))$ Then
      IF $(h = sn)$ Then $l_{end} := wn_j$
      Else IF $(wn_j(h) = wn_j(h + 1))$ Then $l_{end} := wn_j(h)$
      Else $l_{end} := wn_j(h + 1) - 1$;
      $D := D_s(h)$;
      For $(l := wn_j(h)$ to $l_{end}; l := l + 1)$
        For each R-access R_j in $RS_j(l)$
          IF $Check(D, Merge(D, D'_i), R_j) \neq ok$ Then
            Add edge $(T_j \rightarrow T_i)$ to the wait-for graph;
            Check deadlock, Exit;
          End_IF;
        End_For
        IF $(l \neq l_{end})$) Then
          For each W-access W_j in $WS_j(l + 1)$
            $D' := GetD(D', W_j)$;
          End_For
        End_IF;
      End_For
    End_IF
  End_For
End_For

C. Algorithm for the determination of s-points

$E^-{}_{min}$:= the minimum value of $E^-(h)$ with $1 \leqq h \leqq sn$;
$sp(m) := sp_{min}$, i.e. the s-point having $E^-{}_{min}$;
For each T_i with $1 \leqq i \leqq n$
  $wn_i(0) := 0; |WS(wn_i(0))| := 0$; /* initialization */

/* **Step 1**: computing effect E^+ increased by a new s-point
  and effect E^- reduced by deleting the latest s-point */
$E^+ := 0$;
IF $(sn > 0)$ $E^- := E(sn)$;
For each T_i with $1 \leqq i \leqq n$
  IF $(wn_i(sn) \neq wn_i)$ Then $E^+ := E^+ + |WS_i(wn_i)|$
  Else IF $((sn = 1)$ or $((sn > 1)$ and $(wn_i(sn) \neq wn_i(sn - 1)))$
    $E^- := E^- - |WS_i(wn_i)|$;
End_For

/* **Step 2**: setting a new s-point */
IF $(sn < sn_{max})$ Then
  $sn := sn + 1$;
  For each T_i with $1 \leqq i \leqq n$
    $wn_i(sn) := wn_i$;
  $E(sn) := E^+$; /* set the effect of new s-point $sp(sn)$ */
  $D_s(sn) := D_{all}$; /* save the current state of D_{all} */
  /* set the reduced effect by deleting the previous s-point */
  IF $(sn > 1)$ $E^-(sn - 1) := E^-$;
  IF $((sn = 1)$ or $(E^-(sn - 1) < E^-{}_{min}))$ Then /* determine sp_{min} */
    $E^-{}_{min} := E^-(sn - 1); m := sn - 1$;

/* **Step 3**: comparing a new s-point with sp_{min} */
Else
  /* determining sp_{min} */
  IF $(E^- < E^-{}_{min})$ Then
    $E^-{}_{min} := E^-; m := sn$;
  Else $E^-(sn) := E^-$;
  /* setting a new s-point and deleting sp_{min} */
  IF $(E^+ > E^-{}_{min})$ Then
    /* recalculate the effect */
    IF $(E(m) - E^-{}_{min} > 0)$
      IF $(sp_{min} = sp(sn))$ Then $E^+ := E^+ + (E(m) - E^-{}_{min})$
      Else $E(m + 1) := E(m + 1) + (E(m) - E^-{}_{min})$;
    End_IF
    /* delete sp_{min} */
    For $(h := m + 1$ to $sn_{max}; h := h + 1)$
      Replace $sp(h)$ to $sp(h - 1)$;
    /* set new s-point $sp(sn)$ */
    IF $(m < sn)$ Then $E^-(m) := E(m)$;
    IF $(m > 1)$ Then $E^-(m - 1) := E(m - 1)$;
    For each T_i with $1 \leqq i \leqq n$
      $wn_i(sn) := wn_i$;
      IF $((m < sn)$ and $(wn_i(m) = wn_i(m + 1)))$
        IF $((m = 1)$ or $(wn_i(m) \neq wn_i(m - 1)))$ Then
          $E^-(m) := E^-(m) - |WS_i(wn_i(m))|$;
      IF $((m > 1)$ and $(wn_i(m - 1) = wn_i(m)))$
        IF $((m > 2)$ and $(wn_i(m - 1) \neq wn_i(m - 2)))$ Then
          $E^-(m - 1) := E^-(m - 1) - |WS_i(wn_i(m - 1))|$;
    End_For
    $E(sn) := E^+$; /* set the effect of a new s-point */
    $D_s(sn) := D_{all}$; /* save the current state of D_{all} */
  End_IF
End_IF