An optimistic multi-level concurrency control for nested typed objects

M. Cart  J. Ferrié  H. Richy

Centre de Recherche en Informatique de Montpellier
Université de Montpellier, CNRS
860 rue de St Priest, 34090 Montpellier (France)
Tel: (33) 67 63 04 60, E-mail: CRIMEFRMOP1.bitnet

ABSTRACT

In object-oriented environments typed objects and their operations are constructed by considering different levels of abstraction. Various multi-level concurrency control methods, well suited to these environments, have been proposed in the literature in order to serialize transactions; all these are based on locking. In this paper we propose an optimistic multi-level method that also exploits commutativity of typed operations in order to enhance concurrency. One of the main advantages of this method is that it allows to take into account commutativity which depends on return values; other methods, proposed until now, have not been able to exploit this aspect.

1. INTRODUCTION

One of the main goals of computing systems, in particular operating systems and database systems, is to achieve a high degree of concurrency while preserving consistency of information manipulated by concurrent activities.

Concurrency control study when considering shared abstract data types is particularly attractive: it enables exploitation of their semantics and moreover is well suited to object-oriented environments. Typed objects are generally constructed by considering consecutive levels of abstraction, thus resulting in an object hierarchy. This paper considers an execution scheme based on both nested transactions and the object hierarchy. Then we propose an optimistic multi-level concurrency control method well suited to this environment.

Traditional concurrency control methods [4] assume a sequential transaction model in which only read and write operations are considered. In [15] and [18], the authors showed that using typed objects through specific operations when exploiting their semantics increases concurrency. On this basis a number of methods using typed locks and, recently, optimistic methods [10] have been proposed. However, these methods, although they use commutativity of typed operations, do not exploit the object hierarchy, that is, all operations are supposed to be at the same level.

On the other hand, several researchers suggested expression of the hierarchical construction of typed objects by using the nested transaction model; in order to do this, they developed a concurrency control theory dedicated to this model [11], [2], [3] without in fact proposing any specific method. Some multi-level methods, adapted to nested transactions and taking the object hierarchy into account have been proposed in [14], [12]. In [14], a method based on “two-phase” locking at each level and that uses typed locks is described. Recently, in [12], a locking algorithm at the leaf level that allows more parallelism than the former is presented. Indeed, instead of blocking a transaction when invoking a conflicting high-level operation, blocking is delayed until invocation of some leaf operation in conflict. However both of these pessimistic methods based on locking are unable to use return value of operations in determining whether or not two operations commute. The optimistic multi-level scheme that we describe here does not suffer these shortcomings. The example below illustrates this idea.

Let us consider a bank transaction, transfer (A,B,C,v), which credits the bank account A with
some value \( v \) from a bank account \( B \) if sufficiently supplied or from \( C \) otherwise. An object \( X \), of the bank account type, can be used through operations \( \text{credit}(X,v) \) and \( \text{debit}(X,v,r) \) with boolean \( r \) specifying whether debit has been successful or not. The \( \text{credit} \) operation increments the account balance. The \( \text{debit} \) operation attempts to decrement the balance; a debit is successful only if the balance of the account exceeds the amount to be debited. Otherwise the operation returns \( r = 'false' \), leaving the account balance unchanged. Account \( X \) is implemented by an integer \( x \), accessed through \( \text{read} \) and \( \text{write} \) operations.

A concurrent execution of two bank transfers, \( \text{transfer}(A,B,C,v_1) \) and \( \text{transfer}(B,A,D,v_2) \), is represented by the nested transactions \( T_1 \) and \( T_2 \). The execution order of \( \text{read} (R) / \text{write} (W) \) operations is numbered as shown in figure 1.

![Figure 1. Concurrent execution of transfer transactions.](image)

This execution is correct (\( T_1 \) and \( T_2 \) are serializable) when using \( \text{credit} / \text{debit} \) commutativity. In particular we will see later on that \( \text{credit}(B,v_2) \) and \( \text{debit}(B,v_1,r) \) commute when considering that \( r \) value is false (cf figure 2). By letting operations run without blocking and by exploiting commutativity at their termination, our method can use return value knowledge and thus accept the execution above. On the other hand, control proposed by [14] and [12] would be unable to produce this schedule because of deadlock. More precisely, with the algorithm in [14], \( T_1 \) would be blocked when invoking \( \text{debit}(B) \) and \( T_2 \) when invoking \( \text{debit}(A) \). With the algorithm in [12], \( T_1 \) would be blocked further when reading \( b \) (at 5) and \( T_2 \) when reading \( a \) (at 6).

The paper is organized as follows. In section 2 we define commutativity predicates as a means for expressing commutativity between typed operations. Section 3 presents the theoretical framework using the nested transactions model and allowing exploitation of different kinds of commutativity. Section 4 proposes an optimistic multi-level method based on timestamps' intervals that we introduced in [7].

2. COMMUTATIVITY

Several studies [2], [15] and [18], have shown that taking typed operations and their semantics into account increases concurrency, thanks to the properties of commutativity. So concurrent transactions may appear to be serializable whereas they would not be when only considering \( \text{read} / \text{write} \) operations. For example in the execution shown by figure 1, \( T_1 \) and \( T_2 \) are serializable in terms of \( \text{credit} \) and \( \text{debit} \) but they are not in terms of \( \text{read} / \text{write} \).

In this section, we define commutativity using the operations' effects [8]; we identify different kinds of commutativity to be exploited by a concurrency control method.

2.1. Operation effects

Let \( \text{op} \) be a typed operation defined on object \( x \). We note \( T.\text{op}(x,a,r) \), for short \( (\text{op}) \), the execution of operation \( \text{op} \) by transaction \( T \), \( a \) and \( r \) indicating call and return parameters. Effects of this execution may be observed according to two points of view:
- the effect on the object \( x \), noted \( \text{effect}_x \),
- the effect on the transaction \( T \) that invoked the operation \( \text{op} \), noted \( \text{effect}_T \).

In determining these effects, we assume the operations are \textit{indivisible}. We define \( x_i \) [resp. \( x_f \)] as the state of the object \( x \) before [resp. after] execution of operation. Obviously, effects are relative to the initial state \( x_i \) of object \( x \).

**Definition 1.**

Effect on object \( x \) of operation \( \text{op}(x,a,r) \) invoked by transaction \( T \) is given by:

\[
\text{effect}_x (x_i, (\text{op})) = x_f.
\]

**Definition 2.**

Effect on transaction \( T \) of operation \( \text{op}(x,a,r) \) is defined by:

\[
\text{effect}_T (x_i, (\text{op})) = r,
\]

where \( r = 0 \) if \( \text{op} \) has no return parameter.

Let \( T_1.\text{op}_1(x,a_1,r_1) \); \( T_2.\text{op}_2(x,a_2,r_2) \) be the sequential execution of operations \( \text{op}_1 \) and \( \text{op}_2 \) respectively by \( T_1 \) and \( T_2 \). We assume that \( \text{op}_2 \) immediately follows \( \text{op}_1 \) and we note \( (\text{op}_1;\text{op}_2) \) this execution.

**Definition 3.**

Effects of \( \text{op}_1;\text{op}_2 \) (noted \( \text{EFFECT} \)) both on object \( x \) and on transactions \( T_1 \) and \( T_2 \) are defined by:

\[
\text{EFFECT}_x (x_i, (\text{op}_1;\text{op}_2)) = \text{effect}_x (\text{effect}_x (x_i, \text{op}_1), \text{op}_2)
\]

\[
\text{EFFECT}_T (x_i, (\text{op}_1;\text{op}_2)) = \text{effect}_T (\text{effect}_T (x_i, \text{op}_1), \text{op}_2).
\]
2.2. Different kinds of commutativity

We refer to commutativity as a property related to a pair of ordered operations \((op_1; op_2)\) such that execution in the order \(T_1.op_1(x,a_1,r_1); T_2.op_2(x,a_2,r_2)\) has the same effects (on the object and on the transactions) as the execution in the reverse order \(T_2.op_2(x,a_2,r_2); T_1.op_1(x,a_1,r_1)\).

**Definition 4.**
In the execution \(T_1.op_1(x,a_1,r_1); T_2.op_2(x,a_2,r_2),\) starting from initial state \(x_i\), the pair \((op_1; op_2)\) of operations has the commutativity property if:

\[
\forall x_i, \forall a_1, \forall a_2 \text{ then } \text{COMMUT}(op_1; op_2).
\]

From now on, we use a predicate \(\text{COMMUT}(op_1; op_2)\) to denote commutativity of the pair \((op_1; op_2)\) of operations. Commutativity may depend on the initial state of the object as well as call and return values of the operations. In the following, we are only interested in commutativity conditions that are independent of the initial state of the object. According to whether the commutativity is independent or not of parameters we refer to it either as general or conditional commutativity.

**General commutativity**
The pair \((op_1; op_2)\) of operations has the general commutativity property if:

\[
\forall x_i, \forall a_1, \forall a_2 \text{ then } \text{COMMUT}(op_1; op_2).
\]

**Conditional commutativity**
This kind of commutativity depends on a predicate \(\text{COND}\) using operation parameters (call and return value). Thus, a pair \((op_1; op_2)\) of operations has the conditional commutativity property if:

\[
\forall x_i, \text{ if } \text{COND}(a_1,r_1,a_2,r_2) \text{ then } \text{COMMUT}(op_1; op_2).
\]

Commutativity properties may be indicated within a commutativity table, as shown by figure 2. An entry \(\rightarrow\) means that operations do not commute. When the entry is \(\text{COND}\), involved operations commute depending on whether predicate \(\text{COND}\) is satisfied. General commutativity entry indicates that operations always commute.

**Example.**
Let us consider the bank account \(X\) (whose balance is \(x\)) that was previously described. It may be accessed through the \(\text{credit}(X,v)\), \(\text{debit}(X,v,r)\) and \(\text{consult}(X,v)\) operations. Commutativity of the different pairs of operations (cf figure 2) is issued from the functions \(\text{effect}\); as an example, \(\text{debit}\) effects are the following:

\[
\text{effect}_x(x, \text{debit}(X,v,r)) = \begin{cases} 
\text{if } v \leq x \text{ then } x := x - v \\
\text{else } r := \text{true} 
\end{cases}
\]

We can notice that commutativity is not a symmetrical property and therefore commutativity of \((op_1; op_2)\) does not imply commutativity of \((op_2; op_1)\). Commutativity that has been established for ordered pairs of indivisible operations nevertheless persists when considering atomic (serializable) operations. In this case, the order in which operations are considered has to be the serialization order. Moreover ability of exploiting conditional commutativity depends heavily on which concurrency control method is used. Thus, pessimistic methods using locking can benefit from its use as long as that predicate \(\text{COND}\) is independent of \(r_2\) (i.e. \(\text{COND}(a_1,r_1,a_2)\)). Only optimistic methods can exploit conditional commutativity depending on the return parameter \(r_2\) of the second operation (i.e. \(\text{COND}(a_1,r_1,a_2,r_2)\)).

3. NESTED TRANSACTIONS MODEL INTEGRATING COMMUTATIVITY

3.1. Levels of abstraction and hierarchy of objects

Here, we consider that objects are built according to different levels of abstraction. Thus, from primitive
preexisting objects, being accessed through elementary operations, more elaborated typed objects may be constructed by degrees. This construction leads to a hierarchy of levels such that:
- level 0 contains primitive objects usable through elementary operations (read/write) assumed to be indivisible,
- level i contains objects \( (X, Y, Z, \ldots) \) usable through operations \( OP \). Each object \( X \), at this level, is represented by a set of objects \( (x, y, \ldots) \) at level \((i-1)\). In the same way, each operation \( OP \), defined on the object \( X \), is expressed in terms of operations \( op \) defined on objects \( (x, y, \ldots) \) at level \((i-1)\).

In the following, we suppose that objects' representations are strictly nested and that the system is composed of \( n \) abstraction levels numbered from 0 to \((n-1)\).

3.2 Execution structure: the decomposition tree

The model

In this object-oriented environment, execution of a transaction \( T \) corresponds to invocation of operations on \((n-1)\)th level objects. Taking the objects' hierarchy into account, each invocation induces in turn a succession of invocations of nested operations at lower levels.

In our model each invocation of an operation \( OP \) on a \( i \)th level object \( X \) results in creating a transaction \( T^i \) in order to perform the operations \( op \) on \((i-1)\)th level objects \( x, y, \ldots \) that implement \( X \). We note \( OP(X, T^{i+1}) \) the invocation by transaction \( T^{i+1} \) of operation \( OP \) on the \( i \)th level object \( X \). We say that transaction \( T^{i+1} \) invokes the operation \( OP \) and \( T^i \) performs operation \( OP \) (noted \( T^i = OP \)). It results that execution of transaction \( T \) leads to creation of a set of transactions linked together by a tree structure which is called a decomposition tree. In the following we refer to usual terminology on trees e.g. parent, child, ancestor, descendant.

Construction of an operation \( OP \), from operations \( op \) belonging to the next lower level, implies a control structure expressed by means of some programming language and that enables specification of sequentiality and/or parallelism between operations \( op \). The model we propose is independent of synchronization induced by the control structure. However we suppose that operation \( OP \) terminates only when all the operations \( op \) invoked by it, are themselves terminated. Termination of operation \( OP \) triggers commitment of transaction \( T^i \) that performed it, if it is serializable. \( T^0 \) is said to be active so long as it is not committed.

The decomposition tree, whose depth is \((n+1)\), that represents execution of transaction \( T \) is rooted at \( T \). \( T \) is also called a top level transaction (TL transaction for short). The edges of this tree express both child relationship between transactions from adjacent levels and nesting of operations.

More formally, a transaction at level \( i \) in this tree is defined as follows:

\[
\begin{align*}
\forall i \in \{0, \ldots, n\} & \quad T^i = \{ T^{i+1}_k, \ k \in \{0, \ldots, \lceil \log_2 n \rceil \}, k \in \text{child}(T^i) \} \\
& \quad T^0 = \text{elementary operation on an object at level 0} \\
& \quad T^n = T
\end{align*}
\]

An intermediate vertex \( T^i \) at level \( i \) may be considered from two points of view:
- either as an operation \( OP \) (on a \( i \)th level object) invoked by transaction \( T^{i+1} \) (where \( T^i \in \text{child}(T^{i+1}) \)),
- or as a transaction that in turn invokes operations \( op \) on \((i-1)\)th level objects.

Concurrent execution of a set of TL transactions with the corresponding decomposition trees may be represented by a unique tree \( t \), whose depth is \((n+2)\), that is obtained by hanging all those trees to a fictitious root.

Nested transactions theory using abstraction levels has been developed by [3], [14]. It has shown that a concurrent execution of TL transactions is correct if, at each level \( i \), transactions \( T^i \) are serializable and \( i \)th level serialization order preserves that of \((i-1)\)th level.

Transactions properties

When executing a set of TL transactions, the system has to ensure several properties for the transactions.
In the classical scheme of flat transactions, these properties are [16]:

A1. **failure atomicity** that guarantees that either all or none of transaction’s effects are visible;

A2. **serializability** that ensures, for each transaction, its isolation from effects due to concurrency;

A3. **persistence** of a transaction’s effects once it is committed.

As for nested transactions, properties attached to a transaction have to be considered relatively to concurrent transactions running at the same level as it. So, serializability means that transactions belonging to a same level are serializable.

Moreover, these properties differ according to whether we consider a transaction \( T_n \) or a transaction \( T_i \) at level \( i (\neq n) \) [9]. For a transaction \( T_n \), the system must guarantee all properties of a flat transaction; by contrast for a transaction \( T_i \) (\( i \neq n \)) only properties A1 and A2 have to be fulfilled. Indeed persistence of the effects of a transaction \( T_i \) (\( i \neq n \)) depends on the commitment of all transactions \( T_j \), ancestors of \( T_i \) in the decomposition tree. So commitment of a transaction \( T_0 \) becomes permanent only after commitment of the TL transaction upon which it depends. In order to distinguish the two types of commitment, we refer to the commitment of a TL transaction as a permanent commitment. In the following we say that a transaction (or the operation it performs) is **atomic** if it fulfills properties A1 and A2.

### 3.3 Dependency graphs

The aim of this section is to take the commutativity of operations into account in the nested transactions model. Consider concurrent execution of a set of transactions \( T^n \). We assume that these transactions are completed. Their nesting is described by the tree \( \tau \). In order to control their serializability while taking the properties of commutativity into account we will use, in addition to \( \tau \), dependency graphs that enable expressing dependencies between transactions. These graphs are associated with the different levels of the tree \( \tau \); they are built layer by layer from conflicts due to elementary operations.

#### Conflicts

Two atomic operations \( op_j(x) \) and \( op_k(x) \) **conflict** if they do not commute when they are considered in the order of access to object \( x \). When operations are indivisible (i.e. operations at level 0), the access order to the object is the chronological order of execution; this order is noted \( \prec \). So \( op_j \prec op_k \) means that \( op_j \) has been executed before \( op_k \). In the general case where \( op_j \) and \( op_k \) are atomic, access order to the object corresponds to serialization order of \( op_j \) and \( op_k \).

#### Dependency relations

In order to express the conflicts we define, at each level \( i \), two dependency relations between transactions \( T_i \): \( \prec_i \) and \( \prec^*_i \).

(1) relation \( \prec_i \) allows control of the serializability of \( i^{th} \) level transactions:  
\[
\forall i \in [1,n] \quad T_i^j \prec_i T_i^k \quad \text{iff} \quad \exists T_i^{j} \in \text{child}(T_i) \quad \text{and} \quad \exists T_i^{k} \in \text{child}(T_i^k) \quad \text{such as} \quad T_i^{j} \prec_i T_i^k
\]

(2) relation \( \prec^*_i \) allows to exploit commutativity of operations:  
\[
\forall i \in [0,n] \quad T_i^j \prec^*_i T_i^k \quad \text{iff} \quad (T_i^j \prec T_i^k \text{ and not COMMUT (} op_j ; op_k ))
\]

Different relations that can bring identical results were proposed in [13].

#### Interpretation of dependency graphs

Correctness of a concurrent execution of top level transactions relies on a multi-level control. It uses the tree \( \tau \) and, for each level \( i \), two graphs \( G_i \) and \( G_{Pi} \), defined as follows:

\[
G_i = (\{ T_i \}, \prec_i) \quad \text{and} \quad G_{Pi} = (\{ T_i \}, \prec^*_i).
\]

This control consists, for each level \( i \) (with \( i = 0, 1, ...n \)):

- first, in verifying that all transactions \( T_i \) are serializable (i.e. \( G_i \) is acyclic);
- next, in exploiting commutativity of operations performed by transactions \( T_i \), so as to obtain \( G_{Pi} \).

Taking commutativity properties into account may suppress some conflicts between \( i^{th} \) level operations (i.e. \( G_i \supseteq G_{Pi} \)) and therefore dependencies between transactions \( T_{i+1} \). The following theorem gives the correctness criterion for a concurrent execution.

#### Theorem 1

A concurrent execution of a set of transactions \( T^n \) is correct if, for each \( i = 0, 1, ...n, G_i \) is acyclic.
The proof is shown by induction on the levels. \( G_0 \) expresses precedence (i.e. \( \cdot \cdot \cdot \) between elementary operations; as they are indivisible \( G_0 \) is acyclic. \( G_P_0 \) takes this precedence and also operations' commutativity into account: it expresses conflicts between operations (\( G_0 \geq G_P_0 \)). \( G_1 \) expresses dependencies between transactions \( T_i \), due to conflicts between the operations they performed at level 0. If \( G_1 \) is acyclic, then transactions \( T_i \) are serializable; consequently operations performed by transactions \( T_i \) are atomic and so commutativity may be exploited in order to build GP_1 and G_2.

More generally the transition from \( G_{i-1} \) to \( G_i \) is possible only if transactions \( T_{i-1} \) can be considered as atomic operations invoked by transactions \( T_i \). Dependencies between \( T_i \) result from conflicts between their operations \( T_{i-1}^{i-1} \) that are expressed by GP_{i-1}. If step by step, we finally succeed in building \( G_n \) and if \( G_n \) is acyclic then transactions \( T_n \) are serializable.

4. Optimistic Multi-level Concurrency Control

4.1. Introduction

The posteriori serialization control of nested transactions, as previously described, supposes that all the transactions are finished. In order to be suitable, the control has to be done on line, using either pessimistic (by locks, timestamps), or optimistic (by certification), or mixed (2PL and certification) methods. All multi-level control methods, known so far, are based on locking [14], [12].

The proposed multi-level method is an optimistic one; it extends to nested transactions the optimistic control by intervals of timestamps, presented in [7] and used for serializing flat transactions. It is also a "backward control" method: i.e. certification of a transaction is checked against the already committed transactions, at the same level. The interest of an optimistic strategy using a "backward control" is that all results are available during the certification of a transaction: those which have been returned by the operations it invoked, and those which have been returned by operations invoked by the transactions with which it must be checked. Therefore, it is possible to take any kind of commutativity among operations into account.

In order to describe this backward control, it is useful to distinguish, at each level \( i \), the graph \( G_i^* \) corresponding to the graph \( G_i \) restricted to all committed transactions \( T_i \) with their dependencies. Committed transactions are serializable by construction, thus \( G_i^* \) is acyclic. We denote \( G_i^*(T_i) \) the graph \( G_i \) extended to the active transaction \( T_i \). The principle of the backward control certification consists in allowing a transaction \( T_i \) to be committed if \( G_i^*(T_i) \) is acyclic; otherwise \( T_i \) must be rejected.

4.2 Principles

We consider a transaction \( T_i \) which has invoked some operations on objects at level \( (i-1) \); these operations have been performed by transactions \( T_{i-1}^{i-1} \). The commitment of \( T_i \) depends on the issue of its certification which takes place when all its \( T_{i-1}^{i-1} \) are committed. The certification is built on the following principles.

**P1.** With each committed transaction \( T_i \) is associated a timestamp \( t_i \) expressing the place of \( T_i \) in the serialization order at level \( i \); this order is compatible with \( G_i^* \).

**P2.** With each active transaction \( T_i \) is associated an interval of timestamps \( [T_i, T_i'] \) expressing dependencies between \( T_i \) and already committed transactions \( T_{i-1}^{i-1} \).

Let us consider \( I(T_i/T_{i-1}^{i-1}) = [a, b] \) (with \( a, b \in \mathbb{R} / a < b \)); therefore:

- if \( T_i < T_i' \) then \( b < t_i' \)
- if \( T_i' < T_i \) then \( t_i < a \)

The set of dependencies \( C \) between \( T_i \) and all the \( T_{i-1}^{i-1} \) is given by:

\[
I(T_i) = \bigcup_{T_{i-1}^{i-1}} I(T_i/T_{i-1}^{i-1})
\]

We will see further how to calculate \( I(T_i/T_{i-1}^{i-1}) \) from dependencies at level \( (i-1) \). Serializability of \( T_i \) is checked, when \( T_i \) certifies, thanks to the following theorem, proved in [7].

**Theorem 2.**

If \( I(T_i) \neq 0 \) then \( G_i^*(T_i) \) is acyclic.

Thus, if \( I(T_i) \neq 0 \), \( T_i \) may be committed because it does not introduce any cycle into \( G_i^* \). The timestamp \( t_i \) which is then associated with it, is chosen in \( I(T_i) \).
P3. Calculation of $I(T_i/T_j)$

Suppose that:

$T_{c_i} \in \text{child}(T_i)$, timestamped $t_{c_i}^1$, with $T_{c_i} = \text{op}_c$

and

$T_{c_j} \in \text{child}(T_j)$, timestamped $t_{c_j}^1$, with $T_{c_j} = \text{op}_j$.

The dependency $c_{ij}$ between $T_i$ and $T_j$, due to some conflicts induced by $T_{c_i}^1$ and $T_{c_j}^1$, is expressed by an interval denoted $I(T_i/T_{c_i}^1/T_{c_j}^1)$. It is built from the relation $r_{ij}$, as follows:

\[
\begin{align*}
\text{if } t_{c_i}^1 &< t_{c_j}^1 \text{ and not } \text{COMMUT}(\text{op}_c; \text{op}_j) \quad \text{then} \quad I(T_i/T_{c_i}^1/T_{c_j}^1) = [0, t_{c_i}^1]; \\
\text{if } t_{c_i}^1 &< t_{c_j}^1 \text{ and not } \text{COMMUT}(\text{op}_j; \text{op}_c) \quad \text{then} \quad I(T_i/T_{c_i}^1/T_{c_j}^1) = [t_{c_j}^1, +\infty].
\end{align*}
\]

Absence of any conflict between $T_{c_i}^1$ and $T_{c_j}^1$ is expressed by:

\[
I(T_i/T_{c_i}^1/T_{c_j}^1) = [0, +\infty].
\]

The set of dependencies between $T_i$ and $T_j$, due to conflicts induced by $T_{c_i}^1$ and all the $T_{c_k}^1 \in \text{child}(T_k)$, is given by $I(T_i/T_{c_i}^1/T_j)$ with:

\[
I(T_i/T_{c_i}^1/T_j) = \bigcap_{T_{c_k}^1 \in \text{child}(T_k)} I(T_i/T_{c_i}^1/T_k).
\]

Finally, the set of dependencies between $T_i$ and all the $T_{c_k}^1 \in \text{child}(T_k)$, due to conflicts induced by all the $T_{c_i}^1 \in \text{child}(T_i)$ and all the $T_{c_k}^1 \in \text{child}(T_k)$, is given by $I(T_i/T_j)$ with:

\[
I(T_i/T_j) = \bigcap_{T_{c_i}^1 \in \text{child}(T_i)} I(T_i/T_{c_i}^1/T_j).
\]

Since the method is adapted to an object-oriented environment, the calculation of the interval $I(T_i)$ which is associated with each certifying $T_i$ is rewritten in terms of objects. This rewriting is possible on one hand thanks to mathematical properties of intervals’intersection, on the other hand because any dependency between two transactions results from a conflict on one object.

Thus, to each object $x$, invoked by transaction $T_i$ through operation $\text{op}(x)$ (with $\text{op}(x) = T_i^{1-1}$), corresponds the interval $I(T_i/T_{c_i}^1/x)$ given by:

\[
I(T_i/T_{c_i}^1/x) = \bigcap_{T_{c_i}^1 \in \text{child}(T_i)} I(T_i/T_{c_i}^1/T_{c_i}^1)
\]

and $T_{c_i}^1 = \text{op}(x)$.

The dependencies between $T_i$ and all the $T_{c_k}$, due to conflicts upon $x$, are expressed by:

\[
I(T_i/x) = \bigcap_{T_{c_k}^1 \in \text{child}(T_i)} I(T_i/T_{c_k}^1/x).
\]

Finally:

\[
I(T_i) = x \in \text{object}(T_i) \quad I(T_i/x)
\]

object($T_i$) being the set of objects, at level $(i-1)$, used by $T_i$.

4.3. Algorithms

They are given for an object $X$ at level $i$, usable through typed operations $\text{OP}$, invoked by some transactions $T_i^{1+1}$. In order to achieve concurrency control, the end of $\text{OP}(X,T_i^{1+1})$ is divided into two steps:

- certification and commitment of the $T_i$ which performed it,
- exploiting commutativity properties of the operations $\text{OP}$.

Besides, for each object $X$, operations $\text{PRE-COMMIT}(X,T_i^{1+1})$, $\text{COMMIT}(X,T_i^{1+1}, t_i^{1+1})$ and $\text{ABORT}(X,T_i^{1+1})$ are introduced in addition to operations $\text{OP}$ specific of its type. They are invoked at certification and commit time of $T_i^{1+1}$. Figure 4 illustrates a cascade of some invocations and returns resulting from the execution of $\text{OP}(X,T_i^{1+1})$.

Data structures

For each completed invocation $\text{OP}(X,T_i^{1+1}) = T_i$, we memorize:

$e_c = \{ \text{OP}_c(a,r), T_{c_i}^1, t_{c_i}^1 \}$ if $T_i^{1+1}$ is committed,

$e = \{ \text{OP}(a,r), T_i^{1+1}, t_i^{1+1} \}$ if $T_i^{1+1}$ is active,

$\text{OP}(a,r)$ being the operation name, with the call and return parameters, if they are needed to determine commutativity.

Let us denote $E_c$ (resp. $E$) the set of $e_c$ (resp. $e$) which is associated with the completed invocations on $X$. Moreover, with each active $T_i^{1+1}$ which invoked an operation upon $X$, at least once, is associated the interval $I(T_i^{1+1}/X)$. It is created at the first invocation, initialized as $[0, +\infty[$ and maintained up to the commitment of $T_i^{1+1}$.

Algorithms at level $i$

The $\text{OP}(X,T_i^{1+1})$ termination triggers the two following sequences.
The interval \( I(T^{i+1}/X) \) has to express, at the commit time of \( T^{i+1} \), the dependencies between the transaction \( T^{i+1} \) and all committed transactions \( T^{j+1} \) which used \( X \). As it is created during the execution of \( T^{i+1} \), it only takes transactions \( T^{j+1} \) committed at this moment into account; therefore, it has to be updated every time a new transaction \( T^{j+1} \) is committed, i.e. during COMMIT\((X,T^{j+1},t^{j+1})\).

Let us denote \( E(T^{i+1}) \) the set of invocations issued from \( T^{i+1} \) and \( \bar{E}(T^{i+1}) \) the complementary set of invocations issued from other transactions than \( T^{i+1} \).

### COMMIT\((X,T^{i+1},t^{i+1})\):

```plaintext
for all \( e \in E(T^{i+1}) \) do
begin
for all \( ec \in Ec \) do
begin
consider \( ec = \{ OP_{a}(a,r), T^{i+1}\_c, t^{i+1}, t^{i+1}\_c \} \)
if \( t^{i+1} < t^{i} \) and not COMMUT\((OP_{a}; OP)\) then
\( I(T^{i+1}/X) := I(T^{i+1}/X) \cap I(t^{i+1}, \infty) \);  
if \( t^{i+1} < t^{i} \) and not COMMUT\((OP; OP_{a})\) then
\( I(T^{i+1}/X) := I(T^{i+1}/X) \cap [0, t^{i+1}] \);  
end
end
end
```

### Figure 4. Execution of an operation \( OP(X,T^{i+1}) \).

- **PRE-COMMIT\((X,T^{i+1})\):**
  - return \( (I(T^{i+1}/X)) \).

- **Exploiting commutativity of operation \( OP(X,T^{i+1}) \).**

- **CERTIFICATION and COMMIT of \( T^{i} = OP(X,T^{i+1}) \).**
For simplicity's sake, the sequences corresponding to an operation termination, PRE-COMMIT and COMMIT, are supposed to be indivisible. While relaxing this hypothesis, we are faced with the problem of concurrent certifications, studied in [6].

**Particular case: level 0**

Operations at level 0 (read/write) unlike those of other levels, are indivisible and performed on primitive objects. Timestamps \( t^0 \), associated with these operations, are given according to their execution (chronological) order. Reading is immediate and writing is delayed up to the commit of transaction \( T^i \), in order to avoid cascading aborts. For that reason, timestamp \( t^0 \) associated with a write operation is set to be \( t^0 = +\infty \), as long as the invoking transaction \( T^i \) has not been committed. When \( T^i \) is committed with timestamp \( t^1 \), then \( t^0 \) gets the same value as \( t^1 \). The algorithm of an operation termination, adapted to level 0 is given below.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If ( OP = \text{write} ) then ( t^0 := +\infty );</td>
</tr>
<tr>
<td>2</td>
<td>For all ( e_c \in E_c ) do</td>
</tr>
<tr>
<td></td>
<td>If not ( \text{COMMIT}(OP_c; OP) ) then</td>
</tr>
<tr>
<td></td>
<td>( I(T^i/X) := I(T^i/X) \cap [t^0, +\infty) );</td>
</tr>
</tbody>
</table>

**Termination of \( OP(X, T^i) = \text{(read / write)} \)**

The COMMIT\((X, T^i+1)\) procedure differs from the general case only because it modifies the object when writing and because it updates the timestamp \( t^0 \) associated with the write operation.

On the whole, our methods needs, for each object, to keep a trace \( E_c \) of operations invoked by committed transactions. Actually, for objects at level 0, it is easy to restrict the set \( E_c \) to the last read and write operations. For other levels \( i > 0 \), it is possible to keep the history of an object \( X \), only starting from timestamp \( t_c \); on the other hand, a drawback is the reject of operations \( OP = T^i \), with \( t^i < t_c \).

**Transaction reject**

This section will not investigate the problem of the reject in a multi-level environment, which is still research area. We only mention some aspects.

Reject of a non-serializable transaction \( T^i+1 \), needs by means of \( \text{ABORT}(X, T^i+1) \), to cancel its effects on each of the objects which it used; that consists in cancelling the effects of each invoked operation. Cancellation of an operation \( OP(X) \), performed by \( T^i \), may be achieved:

- either by cancelling in turn all invocations of \( T^i \),
- or by running an operation \( OP^{-1} \) which compensates the effects of \( OP \) [17].

Whereas cancellation recursively propagates and leads to reject transactions descendant from \( T^i+1 \), on the contrary running a compensatable operation stops this process.

This kind of reject which is internal to a top-level transaction is not only dedicated to our method but it does also exist in locking methods [12], [14] when there is a deadlock.

Another kind of reject, due to concurrency, may occur with our method insofar as reject of \( T^i \) can lead to the reject of some other transactions \( T_j \) so that: \( T^i < T_j \). However, this form of cascading rejects is restricted to those cases where neither commutativity properties nor recoverability properties [1] can be exploited between \( T^i \) and \( T_j \). For this reason, the permanent commitment of some transaction \( T^n \) only occurs after commitment of all transactions \( T^i \) such that they had a common period of activity with it and \( T_j < T^n \).

5. CONCLUSION

In this paper, we have presented a transactions model adapted to an object-oriented environment and capable of taking the semantics of objects and operations into account. Then we have described a multi-level concurrency control, fitted to this model.

The main advantage of this optimistic method using timestamps intervals, compared to locking methods, is that it exploits conditional commutativity, depending upon return values. Therefore, it is particularly well suited to situations where this kind of commutativity is frequent, especially when operations last long. Nevertheless, like any optimistic method, it is efficient only if conflicts are rare. In the cases where conflicts are frequent but essentially caused by some well identified operations, an idea would be to consider letting our method coexist, at any level, with a method using typed locks (mixed concurrency control). Furthermore, though having been presented for a centralized system, it could be of course distributed because an object, at any level, is located on only one site. Finally, transactions abort, system crashes and recovery are crucial problems which will have to be particularly investigated.

REFERENCES


