Hybrid Concurrency Control for Multimedia Objects

Naokazu Nemoto, Katsuya Tanaka, and Makoto Takizawa

Tokyo Denki University
E-mail {nemoto, katsu, taki}@takilab.k.dendai.ac.jp

Abstract

It is critical for applications to obtain enough quality of service (QoS) from multimedia objects. Not only states but also QoS of objects are changed by performing methods. The objects are required to be consistent in presence of concurrent manipulations by multiple transactions. We discuss new types of consistency of multimedia objects with respect to QoS. Since it takes a longer time to perform a method on multimedia objects, the throughput of the system is decreased if objects are exclusively locked in traditional concurrency control. We propose a hybrid concurrency control with two orthogonal mechanisms, locking one for serializing transactions and the timestamp ordering one for mutually exclusive access to objects.

マルチメディアオブジェクトにおける同期制御手法

根本 直一　田中 勝也　滝沢 誠

東京電機大学理工学部情報システム工学科
E-mail {nemoto, katsu, taki}@takilab.k.dendai.ac.jp

分散アプリケーションでは複数のマルチメディアオブジェクトが操作される。マルチメディアオブジェクトでは、メソッドの実行により、オブジェトの状態とともにオブジェクトの提供する QoS も変化する。本論文では、新たに QoS を考慮した演算間の競合関係を定義する。また、オブジェクトの同時実行制御方式として、直列可能性を保障するための時刻印順序方式と排他制御を行うためのロック方式を複合した方式を提案する。これにより、システムのスループット向上をはかる。

1 Introduction

Distributed applications, are now realized in an object-based framework [10,12]. A system is composed of objects including multimedia objects. An object is an encapsulation of data and methods for manipulating the data. Service supported by a multimedia object is characterized by parameters showing *quality of service* (*QoS*) [3, 4, 13] like frame rate and number of colours. Not only the state but also QoS of the object are changed if methods are performed on the object. For example, suppose a *movie* object supports a pair of methods *add-car* and *grayscale*. A *car* object is *added* to the *movie* object by *add-car*. A *movie* is changed to a *monochromatic* version by *grayscale*. If *grayscale* is performed after *add-car*, every object is monochromatic in the *movie*. On the other hand, if the coloured *car* is added by *add-car* after *grayscale*, only the car is coloured but the others are monochromatic in the *movie*. If the application is not interested in the colour, both the *movies* are considered to be consistent from the QoS point of view. A pair of methods *conflict* iff the result obtained by performing the methods depends on the computation order of the methods. A pair of methods are compatible iff the methods do not conflict. Hence, *add-car* and *grayscale* do not conflict even if the states obtained are different.

Multiple transactions simultaneously manipulate multimedia objects in applications like cooperating authoring systems. According to the traditional concurrency control theories [1], a pair of conflicting methods on an object are serially performed by using locking protocols. It takes a longer time to perform a method on a multimedia object because a larger amount of structured data are manipulated. Hence, the throughput of the system is decreased if objects are exclusively locked. A pair of compatible methods like *read* methods can be performed in any order and concurrently. However, a pair of compatible methods *add-car* and *grayscale* cannot be concurrently performed on a *counter* object. Furthermore, even some pair of conflicting methods can be concurrently performed while it is critical to decide which methods to be started before the other. Thus, we have to realize the serializability of conflicting methods and mutually exclusive computation of methods. We adopt two different types of concurrency control mechanisms; timestamp ordering (TO) scheduler [1] and the locking protocol [1]. The TO scheduler is used to serialize methods which conflict with each other with respect to the QoS-based conflicting relations. The locking protocol is used to exclusively perform methods. We discuss types of lock modes based on the QoS-based relations.

In section 2, we present a system model. In section 3, we discuss the QoS-based conflicting relations among methods. In section 4, we discuss concurrent manipulation of objects. In section 5, we discuss the TO scheduler and locking protocol.

2 System Model

A system is composed of *classes* and *objects*. A class c is composed of *attributes* and *methods*. An object o is created from the class c. A tuple of attribute values is a *state* of the object o. Each object has one state at a time. A *state* of a class also means a state of an object of the class. An object has a unique invariant identifier while its state is variant.

A new class c_2 can be derived from an existing

class c_1. In addition, a class c can be composed of *component* classes c_1, ..., c_n. Let $c.c_i$ show a component class c_i of the class c. Let $c_i(s)$ denote a projection of a state s of the class c to a component class c_i. For example, a class *karaoke* is composed of three component classes, *music*, *words*, and *background* [Figure 1]. *background*(k) shows a state of *background* in a state k of a *karaoke* object. The *background* class is furthermore composed of *car*, *tree*, and *cloud* classes.

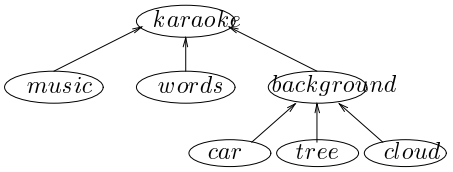

Figure 1: Karaoke class.

On receipt of a request of a method *op*, *op* is performed on an object o. Let $op(s)$ denote a state obtained by performing *op* on a state s of the object o. Let $[op(s)]$ show output obtained by performing *op* on s. Here, $op_1 \circ op_2$ and $op_1 \oplus op_2$ show that a pair of methods op_1 and op_2 are serially and concurrently performed, respectively.

Applications obtain service from a multimedia object only through methods supported by the object. Each service is characterized by *quality of service* (QoS) like level of resolution. Each state s of an object o supports QoS denoted by $Q(s)$. For example, $Q(play(s))$ shows QoS of music, sound, and background image played on a state s of a $Karaoke$ object. $Q(s)$ indicates QoS of the state s of the $Karaoke$ object. $Q(s_1)$ *dominates* $Q(s_2)$ ($Q(s_1) \succeq Q(s_2)$) iff a state s_1 supports better QoS than another state s_2. For example, $\langle 30[\text{fps}], 16[\text{colours}]\rangle \preceq \langle 40, 64\rangle$. The formal definition of the dominant relation $\preceq$ is discussed in the papers [6]. Since "$\preceq$" is a partially ordered relation, a *least upper bound* (*lub*) $q_1 \cup q_2$ of QoS q_1 and q_2 is some QoS q_3 in S such that 1) $q_1 \preceq q_3$ and $q_2 \preceq q_3$, and 2) no QoS q_4 where $q_1 \preceq q_4 \preceq q_3$ and $q_2 \preceq q_4 \preceq q_3$. For example, $\langle 30, 1024\rangle \cup \langle 40, 512\rangle = \langle 40, 1024\rangle$. Not only the state but also QoS of the object are changed by methods. An application requires an object to support some QoS which is referred to as *requirement* QoS (*RoS*). Let r be RoS of an application. If $Q(play(s)) \succeq r$, the application can accept the $Karaoke$ service supported thought the method *play*.

3 QoS Based Compatible Relations

3.1 Consistent relations

An object k_1 created from the *karaoke* class is also composed of a *music* object m_1, *words* object w_1, and *background* object b_1. Another *karaoke* object k_2 is same as k_1 except that the *background* object of k_2 is b_2 ($\neq b_1$). An application considers k_1 and k_2 to be consistent since the application is interested in only *words* and *music*. A class of application's interest is referred to as *mandatory*. The classes *words* and *music* are mandatory. On the other hand, a class like *background* is *optional*, in which applications are not interested. The state k_1 is *semantically* consistent with k_2 since the mandatory objects are the same, i.e. $m_1 = m_2$ and $w_1 = w_2$. An application specifies whether each component class c_i is *mandatory* or *optional*. Every object o of the class c is required to include an object o_i of a mandatory class c_i. If c_i is optional, the object o may include no object of c_i and the application does not care QoS of c_i even if the object of c_i is included in o.

There are following types of consistent relations between a pair of states s_t and s_u of a class c:

- s_t is *state-consistent* with s_u ($s_t - s_u$) iff $s_t = s_u$.
- s_t is *semantically consistent* with s_u ($s_t \equiv s_u$) iff $s_t - s_u$ or $c_i(s_t) \equiv c_i(s_u)$ for every mandatory component class c_i of the class c.
- s_t is *QoS-consistent* with s_u ($s_t \approx s_u$) iff $s_t - s_u$ or s_t and s_u are obtained by degrading QoS of some state s of c.
- s_t is *semantically QoS-consistent* with s_u ($s_t \simeq s_u$) iff $s_t \approx s_u$ or $c(s_t) \simeq c(s_u)$.
- s_t is *r-consistent* with s_u on RoS r (s_t is *r-equivalent* with s_u) ($s_t \approx_r s_u$) iff $s_t \approx s_u$ and $Q(s_t) \cap Q(s_u) \succeq r$.
- s_t is *semantically r-consistent* with s_u ($s_t \equiv_r s_u$) iff $s_t -_r s_u$ or $c_i(s_t) \equiv_r c_i(s_u)$.

Let R be a set of RoS. Let $\equiv_R$ and $\approx_R$ show sets $\{ \equiv_r \mid r \in R \}$ and $\{ \approx_r \mid r \in R \}$ which are referred semantically R- and R-consistent relations, respectively. Here, let *State*, *Sem*, *QoS*, *R*, *Sem-QoS*, and *Sem-R* show sets of possible state-, semantically, *QoS*-, *R*-, semantically QoS-, and semantically *R*-consistent relations of a class c. Let C be a family {*State*, *Sem*, *QoS*, *R*, *Sem-QoS*, *Sem-R*} of the sets of the consistent relations. For a set α in C, let $\square_\alpha$ show an α-consistent relation. For example, $\square_{Sem}$ (or $\square_\equiv$) stands for the semantically consistent relation $\equiv$. For a pair of methods op_1 and op_2 of a class c, "op_1 $\square_\alpha$ op_2" shows that $op_1(s)$ $\square_\alpha$ $op_2(s)$ for every state s of c. For a pair of sets α and β in C, α *dominates* β ($\alpha \rightarrow \beta$) means $\alpha \subseteq \beta$, showing that s_1 $\square_\beta$ s_2 if s_1 $\square_\alpha$ s_2 for every pair of states s_1 and s_2. Figure 2 shows a Hasse diagram where a node shows a set of consistent relations in C and a directed edge from α to β shows a dominant relation "$\alpha \rightarrow \beta$". For example, "*State* $\rightarrow$ *Sem*" means that $s_1 \equiv s_2$ if $s_1 - s_2$ for every pair of states s_1 and s_2.

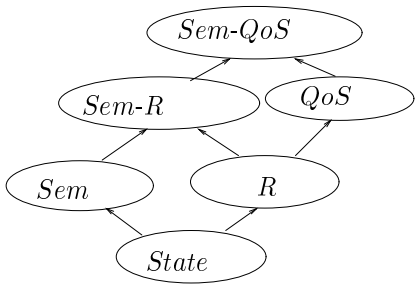

Figure 2: Hasse diagram.

3.2 Compatible relations

According to the traditional theories [1, 7], a method op_t *conflicts* with another method op_u on an object o iff the result obtained by performing the methods op_t and op_u depends on the computation order the methods. The result means not

only the state of the object but also the outputs of op_t and op_u. Multimedia objects are characterized by QoS in addition to the states. The compatible relation between a pair of methods op_t and op_u of a class c can be extended by considering QoS as follows:

- op_t is *state-compatible* with op_u ($op_t \mid op_u$) iff $(op_t \circ op_u) - (op_u \circ op_t)$.
- op_t is *QoS-compatible* with op_u ($op_t \wr\!\wr\ op_u$) iff $(op_t \circ op_u) \approx (op_u \circ op_t)$.
- op_t is *r-compatible* with op_u on RoS r ($op_t \wr\!\wr_r op_u$) iff $(op_t \circ op_u) \approx_r (op_u \circ op_t)$.
- op_t is *semantically compatible* with op_u ($op_t \mid\mid\mid op_u$) iff $(op_t \circ op_u) \equiv (op_u \circ op_t)$.
- op_t is *semantically QoS-compatible* with op_u ($op_t \mid\wr op_u$) iff $(op_t \circ op_u) \simeq (op_u \circ op_t)$.
- op_t is *semantically r-compatible* with op_u ($op_t \mid\mid\mid_r op_u$) iff $(op_t \circ op_u) \equiv_r (op_u \circ op_t)$.

Let "α-*compatible* relation ($\diamond_\alpha$)" show some of the compatible relations where $\alpha \in C$ (=$\{$*State*, *Sem* (*semantically*), *QoS*, *R*, *Sem-QoS*, *Sem-R*$\}$). $op_1 \diamond_\alpha op_2$ iff $(op_1 \circ op_2) \ \square_\alpha \ (op_2 \circ op_1)$. For example, $\diamond_{QoS}$ shows a *QoS*-compatible relation ($\wr\!\wr$) and $\diamond_r$ shows an *r*-compatible relation ($\mid_r$) on some RoS r in *R*. op_t *α-conflicts* with op_u ($op_t \not\diamond_\alpha op_u$) unless $op_t \diamond_\alpha op_u$. $\diamond_\alpha$ and $\not\diamond_\alpha$ are symmetric and transitive.

[Example 1] Let us consider the *background* object b in the *karaoke* object k [Figure 1]. The object b is a *video* object composed of component objects, a *car* object c, *tree* t, and *cloud* c. The object b supports methods *add*, *grayscale*, *mediascale*, and *reduce*. Suppose that a *car* object is newly added in the *background* object by the method *add-car*. Here, the state of the *background* object is changed by *add-car*. A coloured video is degraded to a white-black gradation video by the method *grayscale*, a frame rate is reduced to half of the original one by *mediascale*, and a number of colours is decreased to 16 colours by *reduce*. QoS of the *background* object is manipulated by these methods. Suppose an application obtains a state b_1 by performing *grayscale* on the object b after *add-car*, i.e. $b_1 =$ *add-car* $\circ$ *grayscale*(b). Here, b_1 shows a white-black gradation video with *car*. On the other hand, another *background* state b_2 is obtained by *grayscale* $\circ$ *add-car*(b), b_2 shows a coloured *car* with the white-black *tree* and *cloud*. Here, states obtained by *add-car* $\circ$ *grayscale* and *grayscale* $\circ$ *add-car* support different levels of QoS, $Q(b_2) \neq Q(b_1)$. Hence, *add-car* QoS-conflicts with *grayscale* (*add-car* $\not\wr\!\wr$ *grayscale*). Suppose that the application is only interested in a coloured *car*. Let r show the RoS. The response data obtained by *grayscale* $\circ$ *add-car* satisfies the application requirement RoS r. However, a response data obtained by *add-car* $\circ$ *grayscale* does not satisfy RoS r. That is, *add-car* *Sem*-r-conflicts with *grayscale* (*add-car* $\not\mid\mid_r$ *grayscale*). Here, suppose there is another RoS r' showing that an application is not interested in colour of a *background* object. Since a result obtained by *add-car* $\circ$ *grayscale* is considered to be r'-consistent with *grayscale* $\circ$ *add-car* with respect to RoS r'. Thus, *add-car* is r'-compatible with *grayscale* (*add-car* $\wr\!\wr_{r'}$ *grayscale*). $\square$

Let CF be a set $\{\ \not\diamond_\alpha \mid \alpha \in C\}$ of conflicting relations. For a pair of consistent relations α and β in C where $\alpha \rightarrow \beta$, $op_t \not\diamond_\alpha op_u$ if $op_t \not\diamond_\alpha op_u$.

4 Concurrent Manipulation of Conflicting Methods

4.1 Traditional locking protocol

Suppose a pair of transactions T_i and T_j issue methods op_t and op_u to an object o of a class c, respectively. Here, T_i *precedes* T_j ($T_i \rightarrow T_j$) iff op_t and op_u conflict, i.e. op_t and op_u *State*-conflict ($op_t \not| op_u$) and op_t is performed before op_u. A collection of transactions $T_1, \ldots, T_m$ are *serializable* iff both $T_i \rightarrow T_j$ and $T_j \rightarrow T_i$ do not hold for every pair of transactions T_i and T_j, i.e. the relation "$\rightarrow$" is acyclic [1]. In order to serialize the transactions, each object o is locked before a method is performed on the object o. Suppose a method op_t is issued to the object o. If the object o is locked for a method conflicting with the method op_t, op_t blocks. It is well known that a collection of transactions are *serializable* if every transaction is locked in a two-phase mode [1]. Every pair of compatible methods can be performed on the object o in any order.

Multiple conflicting methods cannot be concurrently performed on an object. It is still question whether or not compatible methods can be concurrently performed. Some compatible methods like a pair of *read* methods can be concurrently performed on the *file* object. On the other hand, methods *add* and *subtract* cannot be concurrently performed on a *counter* object while the methods are compatible. A pair of methods *add-car* and *grayscale* are r'-compatible (*add-car* $\wr\!\wr_{r'}$ *grayscale*) where RoS r' shows "application is not interested in colour" but cannot be concurrently performed on the *movie* object as shown in Example 1. Thus, a pair of orthogonal synchronization mechanisms are adopted, one for serialization of conflicting methods and the other for mutually exclusive computation methods.

4.2 α-conflicting relations

If a pair of methods op_t and op_u α-conflict on an object o ($op_t \not\diamond_\alpha op_u$), the result obtained by performing the methods op_t and op_u depends on the computation order of the methods. Figures 3 (1) and (2) show serial computations of *reduce* and *mediascale*. Figure 3 (3) shows a concurrent computation *reduce* $\oplus$ *mediascale*. Here, the states obtained by the computations (1) and (2) are r-consistent but *reduce* and *mediascale* cannot be concurrently performed. On the other hand, a pair of *display* methods can be performed in any order since *display* is *state*-compatible with itself. In addition, multiple *display* methods can be concurrently performed because multiple transactions can view the *background* object b at the same time. The traditional concurrency control theories in database systems [1] assume every pair of conflicting methods like *write* and *read* are mutually exclusive while other compatible methods like a pair of *read* methods can be concurrently performed. However, some pair of compatible methods cannot be necessarily concurrently performed on a multimedia object as shown in Figure 3. Furthermore, some pair of conflicting methods can be concurrently performed on the multimedia object

while it is critical to consider which method to be started before the other.

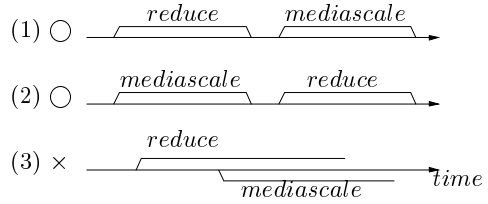

Figure 3: Concurrent computation.

A method op_t is α-*exclusive* with another method op_u on a class c with respect to a consistent relation α in C iff some concurrent computation $op_t \oplus op_u$ is neither α-consistent with ($op_t \circ op_u$) nor ($op_t \circ op_u$). Unless op_t is α-exclusive with op_u, every result obtained by concurrently performing op_t and op_u is α-consistent with some result obtained by serially performing op_t and op_u. The α-exclusive relation is symmetric and transitive.

[**Definition**] Let op_t and op_u be methods of a class c. For every consistent relation α in C ;

1. op_t *strongly* α-*conflicts* with op_u on c iff op_t α-conflicts with op_u and op_t is α-exclusive with op_u.
2. op_t *weakly* α-*conflicts* with op_u on c iff op_t α-conflicts with op_u and op_t is not α exclusive with op_u.
3. op_t is *strongly* α-*compatible* with op_u iff op_t is α-*compatible* with op_t and op_t and op_u are not α-exclusive.
4. A op_t is *weakly* α-*compatible* with op_u iff op_t is α-*compatible* with op_u and op_t and op_u are not α-exclusive. □

The traditional conflicting relation shows the strong conflicting relation. A pair of methods *reduce* and *mediascale* are weakly *r-compatible* on a *background* object because *reduce* and *mediascale* are *r*-compatible but cannot be concurrently performed.

4.3 Modes of methods

A pair of consistent relations α_1 and α_2 in C are *uncomparable* ($\alpha_1 \parallel \alpha_2$) if neither $\alpha_1 \rightarrow \alpha_2$ nor $\alpha_2 \rightarrow \alpha_1$. In Figure 2, *Sem* || *R*, *Sem* || *QoS*, and *Sem-R* || *QoS*. For every pair of consistent relations α_1 and α_2 in C, $\alpha_1 \cap \alpha_2$ shows a greatest lower bound (*glb*) of α_1 and α_2 in the *dominant* relation "→" shown in Figure 2. For example, r = *Sem-r* ∩ *QoS* since $r \rightarrow$ *Sem-r* and $r \rightarrow$ *QoS* hold and are not transitive. Each transaction T takes some type α of consistent relation specified by an application. Let *type*(T) show the consistent type α of a transaction T. The transaction T issues a method *op* to an object o. A method issued by a transaction has a same consistency type as the transaction. Here, let *type*(*op*) show the consistent type of a method *op* which is *type*(T) of a transaction T which issues up. If a pair of transactions T_1 and T_2 issue a method *op* on the object o, each instance of *op* has a different type depending on a type of the transaction issuing *op*. $\mu_\alpha(op)$ is referred to as *mode* of a method *op* on a consistent relation α. Here, a conflicting relation among modes is defined as follows:

> $\mu_{\alpha_1}(op_1)$ α-*conflicts* with $\mu_{\alpha_2}(op_2)$ with respect to a consistent relation α in C if op_1 α-conflicts with op_2 ($op_1 \not\diamond_\alpha op_2$) where $\alpha = \alpha_1 \cup \alpha_2$.

A mode τ is α-compatible with another mode τ' iff τ does not α-conflict with τ'. Let $C_\alpha(\tau)$ be a set { τ' | τ α'-conflicts with τ' and $\alpha \rightarrow \alpha$'} of modes which conflict with a mode τ for a consistent relation α. If an object is locked with some mode in $C_\alpha(\tau)$, a method of a mode τ with a consistent relation α blocks. The following property holds for the dominant relation "→" on the consistent relations' sets.

[**Theorem**] For every mode τ and every pair of consistent relations α_1 and α_2 in C, $C_{\alpha_1}(\tau) \subseteq C_{\alpha_2}(\tau)$ if $\alpha_1 \rightarrow \alpha_2$. □

[**Definition**] Let τ_1 and τ_2 be a pair of modes on consistent relations α_1 and α_2 in C, respectively. τ_1 is *stronger* than τ_2 ($\tau_1 \succ \tau_2$) iff $C_{\alpha_1}(\tau_1) \subseteq C_{\alpha_2}(\tau_2)$. □

Suppose that a pair of RoS r_1 and r_2 show monochromatic and coloured movies, respectively, in Example 1. Here, $r_2 \succ r_1$. A method *grayscale* r_1-conflicts with a method *add* but is r_2-compatible with *add*. The method *grayscale* is stronger than *mediascale* ($\tau_1 \succ \tau_2$) since $C_{r_1}(\tau_1)$ (= {*mediascale*, *reduce*, *add*, *grayscale*}) ⊃ $C_{r_2}(\tau_2)$ (= {*mediascale*, *reduce*, *grayscale*}). The relation "$\tau_1 \succ \tau_2$" means that a mode τ_1 conflicts with more number of modes than another mode τ_2. The mode τ_1 is more restricted than τ_2. For example, *write* ≻ *read*.

5 Concurrently Control

5.1 Timestamp ordering(TO) scheduler

Each object is provided with two types of concurrency control mechanisms; a timestamp ordering (TO) scheduler [1] and locking protocol [1] [Figure 4]. The TO scheduler is used to serialize every pair of α-conflicting methods issued to an object, which are issued by different transactions. The object is locked by the locking mechanism in order to realize mutual exclusion among methods. Each transaction T is assigned a timestamp $ts(T)$ which shows a local time when T is initiated. Transactions are initiated in client computers. Some kind of logical clock like linear clock [8] can be used as the timestamp. If the logical times of T_1 and T_2 are uncomparable or equal, the timestamp of T_1 is smaller than T_2 ($ts(T_1) < ts(T_2)$) if the address of the client computer of T_1 is smaller than T_2. The timestamp can be realized by a concatenation of the local time and the computer identifier [1]. For every pair of different transactions T_1 and T_2, either $ts(T_1) < ts(T_2)$ or $ts(T_1) > ts(T_2)$. Every method *op* issued by a transaction T carries the timestamp $ts(T)$, i.e. $ts(op) = ts(T)$. We assume each transaction issues a method by using a synchronous remote procedure call. That is, each transaction issues at most one method at a time and waits for a response of the method.

Each transaction T manipulates objects according to some consistent relation α in C ($type(T) = \alpha$). T issues a method *op*, where $type(op) = type(T)$. Transactions issue requests of methods to the TO scheduler of an object o. The methods are buffered and are ordered in the TO scheduler according to the following timestamp or-

dering (TO) rule:

[**Timestamp ordering (TO) rule**] For every pair of methods op_1 and op_2 on an object o, op_1 *precedes* op_2 in the TO scheduler ($op_1 \Rightarrow_o op_2$) if op_1 α-conflicts with op_2 ($op_1 \not\diamond_\alpha op_2$), $ts(op_1) < ts(op_2)$, and $\alpha = type(op_1) \cap type(op_2)$. □

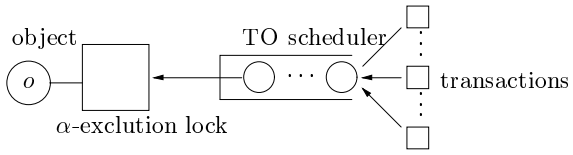

Figure 4: TO scheduler and locking.

Suppose there are a pair of transactions T_1 and T_2 where $ts(T_1) < ts(T_2)$. The transaction T_1 issues a method *grayscale* and the other transaction T_2 issues another method *add-car* to the *background* object b. Suppose $type(T_1) = type(T_2) = QoS$. Since *grayscale* *QoS*-conflicts with *add-car*, T_1 *QoS*-conflicts with T_2. *grayscale* has to precede *add-car* in the TO scheduler (*grayscale* $\Rightarrow_b$ *add-car*) since ts(*grayscale*) $<$ ts(*add-car*). Next, suppose $type(T_1) = QoS$ and $type(T_2) = r$ where RoS r shows "application not interested in the colour of a car." $QoS \cap r = r$ since *grayscale* and *add-car* are r-compatible (*grayscale* $\wr\wr_r$ *add*). Hence, *add-car* $\Rightarrow_b$ *grayscale* even if ts(*grayscale*) $< ts$(*add-car*).

First, suppose a transaction T issues a method op to the TO scheduler of an object o. There is a variable $mts(op)$ showing the timestamp of a method op which is most recently started on the object o. The variable $mts(op)$ is initially 0. The method op is stored in the TO scheduler according to the ordering rule as "op' precedes op ($op' \Rightarrow_o op$)" if $mts(op') < ts(op)$ for every method op' α-conflicting with the method op. Otherwise, op is rejected and then the transaction T is aborted. The number of transactions can be reduced to be aborted if a top method op of the TO scheduler is delayed as discussed in the paper [1]. In the TO scheduler, every pair of α-conflicting methods are ordered in the timestamp order. If some pair of methods op_1 and op_2 are α-compatible, op_1 may precede op_2 ($op_1 \Rightarrow_o op_2$) even if $ts(op_1) > ts(op_2)$.

5.2 Serializability

Let T_i and T_j be a pair of transactions issuing methods op_i and op_j to an object o, respectively. The transaction T_i α-*precedes* T_j with respect to a consistent relation α ($T_i \xrightarrow{\alpha} T_j$) iff op_i α-*conflicts* with op_j ($op_i \not\diamond_\alpha op_j$) where $\alpha = type(T_i) \cap type(T_j)$ and op_i is started before op_j on the object o. In addition, "$\xrightarrow{\alpha}$" is transitive.

[**Theorem**] A transaction T_i α'-precedes another transaction T_j ($T_i \xrightarrow{\alpha'} T_j$) with respect to a consistent relation α' if $T_i \xrightarrow{\alpha} T_j$ and $\alpha' \rightarrow \alpha$. □

[**α-Serializability**] A collection $\boldsymbol{T}$ of transactions $T_1, \cdots, T_m$ are α-*serializable* with respect to a consistent relation α if both $T_i \xrightarrow{\alpha} T_j$ and $T_j \xrightarrow{\alpha} T_i$ do not hold for every pair of transactions T_i and T_j where $\alpha = type(T_i) \cap type(T_j)$. □

Let *State*, *Sem*, *QoS*, *R*, *Sem-QoS*, and *Sem-R* be sets of possible transactions which are *State*, *Sem*, *QoS*, *R*, *Sem-QoS*, and *Sem-R* serializable, respectively. Let SR be a family {*State*, *Sem*, *QoS*, *R*, *Sem-QoS*, and *Sem-R*} of the transaction sets. For a pair of transaction sets α_1 and α_2 in SR, $\alpha_1 \rightarrow \alpha_2$ shows $\alpha_1 \subseteq \alpha_2$. Let $\boldsymbol{T}$ be a set of $\{T_1, \ldots, T_n\}$ of transactions. Suppose $\alpha_1 \rightarrow \alpha_2$ for $\alpha_1, \alpha_2 \in SR$. $\boldsymbol{T}$ is α_2-serializable if $\boldsymbol{T}$ is α_1-serializable. For example, $\boldsymbol{T}$ is *QoS*-serializable if $\boldsymbol{T}$ is *State*-serializable. The Hasse diagram for SR and $\rightarrow$ is isomorphic with Figure 2.

[**Serializability**] A set $\boldsymbol{T}$ of transactions is *serializable* iff $\xrightarrow{\alpha}$ is acyclic for every consistent relation α. □

5.3 Locking protocol

Methods in the TO scheduler are ordered according to the TO rule. A top method op in the TO scheduler is first taken. We have to decide whether or not op can be performed on the object o. A variable R_o shows a set of methods which are being performed on the object o. If op satisfies the following execution rule, op is removed from the TO scheduler and is performed on the object o:

[**Execution rule**] If one of the following rules is satisfied, op is performed on an object o,

1. R_o is empty.
2. If R_o is not empty, op is not α-exclusive with every method op' in R_o where $\alpha = type(op) \cap \{type(op') \mid op' \in R_o\}$. □

If the method op satisfies the execution rule, op is started and is added to R_o. If op completes, op is removed from R_o.

In order to realize the execution rule, the locking mechanism is adopted. For a top method op in the TO scheduler, a lock request of a mode $\mu_\alpha(op)$ is issued to the object o where $\alpha = type(op)$. For every method op' in R_o, if $\mu_\alpha(op')$ is not α''-exclusive with μ_α are $\alpha'' = \alpha' \cap \alpha$, the object o is locked in the mode $\mu_\alpha(op)$. If succeed in locking the object o, the method op is started performed and op is added to R_o. Here, $mts(op)$ is a maximum one of $ts(op)$ and $mts(op)$, $mts(op) :=$ $\max(ts(op), mts(op))$. Otherwise, the method op is kept waited in the TO scheduler.

Suppose the top method op in the TO scheduler does not satisfy the execution rule. The method op has to stay in the TO scheduler. Until the top method satisfies the execution rule, i.e. the object is locked, every method has to wait in the TO scheduler. In order to increase the throughput, another methods than the top method is tried to be performed. A method which is α-compatible with op and preceded by op in the TO scheduler can be performed on the object o.

[**Definition**] A method op is α-*ready* in the TO scheduler of an object o with respect to a consistent relation α iff every method op' preceding op is α-compatible with op and $\alpha = type(op) \cap type(op')$. □

An α-ready method op is referred to as *top* α-*ready* method in the TO scheduler iff op precedes every other α-*ready* methods in the TO scheduler. If the top α-*ready* method op satisfies the execution rule, op is removed from the TO scheduler and then is performed on the object o. This is repeated until there is no α-*ready* method in the

TO scheduler.

If a method *op* completes and the lock of *op* is released, the procedure presented here is applied from the top method in the TO scheduler.

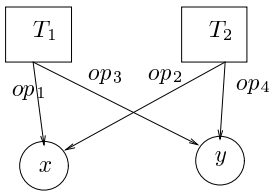

Figure 5: Concurrent access.

5.4 Commitment

We discuss how a transaction terminates. A transaction issues special types of methods $commit(c)$ and $abort(a)$ to objects in addition to methods to manipulate the objects. Suppose a transaction T issues methods op_1, $\cdots$, op_m to an object o. After the methods op_1, $\cdots$, op_m are performed on the object o, the transaction T issues a commit method c to the object o, and the commit method c is performed on the object o. Here, the locks held by the methods op_1, $\cdots$, op_m are released. Similarly, the locks are released of $abort(a)$ is performed. That is, a strict two-phase locking protocol [1] is adopted. Each of commit and abort methods is timestamped as well as the other methods.

[**Definition**] Let e_1 and e_2 be commit or abort methods of an object o issued by transactions T_1 and T_2, respectively. e_1 precedes e_2 in the TO scheduler (e_1 $Rightarrow_o$ e_2) if $ts(e_1) < ts(e_2)$ and T_1 α-conflicts with T_2 when $\alpha = type(T_1) \cap type(T_2)$. □

Suppose a top method is a commit method c of a transaction T in the TO scheduler of an object o. If the commit c satisfies the execution rule, c is removed. The object o is physically updated and the lock of the object o is released. If the top method is an abort method a, the lock of the object o is just released.

5.5 Implementation of lock modes

In this paper, we assume each component class of a class is defined to be either mandatory or optional by a designer. Based on the types of the component classes, a semantically conflicting relation among methods is defined for each class.

Each transaction T has its own *RoS* $type(T)$. It consumes plenty of computation power to compare arbitrary RoS instances. Hence, we assume some limited number of RoS instances are specified when the objects are defined in order to reduce the computation overhead. Let RP be set of *RoS* profiles. Each transaction T takes one profile in RP which satisfies *RoS*, i.e. $type(T) \in RP$. The profiles in RP are ordered in the dominant relation $\prec$. The ordering relation is predefined in the table when RP is defined. For every pair of profiles r_1 and r_2, it is easily decided whether $r_1 \prec r_2$, $r_2 \prec r_1$, or $r_1 \parallel r_2$ by searching the table. Each object maintains a table showing the conflicting relations among the methods. By using the conflicting table, it is decided if the method issued to the object can be performed on the object.

6 Concluding Remarks

We discussed novel types of conflicting and exclusive relations among methods on the basis of QoS and the state of an object, i.e. state-, semantically, *QoS*-, *R*-, and semantically *QoS*- and *R*-conflicting and -exclusive relations of methods in the object-based multimedia system. We presented the TO scheduler and the locking protocol to realize these new types of conflicting and exclusive relations. The TO scheduler is used to serialize the transactions based on the α-conflicting relations. By using the TO scheduler and α-exclusive locks, we can increase the performance of the system.

References

[1] Bernstein, P. A., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, 1987.

[2] Cambell, A., Coulson, G., Garcfa, F., Hutchison, D., and Leopold, H., "Integrated Quality of Service for Multimedia Communication," *Proc. of IEEE InfoCom*, 1993, pp.732–793.

[3] Campbell, A., Coulson, G., and Hutchison, D., "A Quality of Service Architecture," *ACM SIGCOMM Comp. Comm. Review*, *Vol.* 24, 1994, pp.6–27.

[4] Gall, D., "MPEG: A Video Compression Standard for Multimedia Applications," *Comm. ACM*, Vol.34, No.4, 1991, pp.46–58.

[5] Grosling, J. and McGilton, H., "The Java Language Environment," *Sun Microsystems, Inc.*, 1996.

[6] Kanezuka, T. and Takizawa, M., "Quality-based Flexibility in Distributed Objects," *Proc. of 1st IEEE Int'l Symp. on Object-oriented Real-time Distributed Computing* (*ISORC'98*), 1998, pp.350–357.

[7] Korth, H. F., Levy, E., and Silberschalz, A., "A Formal Approach to Recovery by Compensating Transactions," *Proc. of VLDB*, 1990, pp.95–106.

[8] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM*, Vol.21, No.7, 1978, pp.558–565.

[9] Nemoto, N., Tanaka, K., and Takizawa, M., "QoS-based Synchronization of Multimedia Objects," *Proc. of the 11th Int'l Conf. on Database and Expert Systems Applications* (*DEXA'00*) (*Lecture Notes in Computer Science, Springer-Verlag*, *No.* 1873), 151–160, 2000.

[10] Object Management Group Inc., "The Common Object Request Broker: Architecture and Specification, Rev2.0," 1995.

[11] Owen, C. B. and Makedon, F., "Computed Synchronization for Multimedia Applications," *Kluwer Academic*, 1999.

[12] Tari, Z. and Bukhres, O., "Fundamentals of Distributed Object Systems," *Wiley*, 2001.

[13] Wang, Z., "Internet QoS", *Morgan Kaufmann*, 2001.