Regular Paper

QoS-oriented Computation in Multimedia Objects

Tetsuo Kanezuka,† Katsuya Tanaka,† Hiroaki Higaki†
and Makoto Takizawa†

A multimedia object supports methods for manipulating the multimedia data it contains. A method changes not only the state of the object but also the QoS of the state. We discuss new equivalent and conflicting relations among methods with respect to QoS. We also discuss a locking scheme for objects.

1. Introduction

Distributed applications are composed of various kinds of multimedia objects. An object is realized by an encapsulation of data and methods for manipulating the data. CORBA [11] is becoming a general framework for realizing distributed applications. The service supported by the object is characterized by parameters showing the *quality of service* (*QoS*), such as the frame rate. Takizawa, et al. [13] model a *movement* of a mobile object as a change in the QoS supported by the object.

To support applications with service, an object uses methods. A method may change not only the state of the object but also the QoS supported by the object. Relations among the methods have hitherto been discussed with respect to the states of the objects. For example, a pair of methods are *compatible* if the states obtained by performing the methods in any order are the same [1]. In this paper, we discuss types of relations among the methods with respect to the QoS. For example, suppose that a state s_2 is obtained by dropping some frames in a state s_1 of a multimedia object. If the state s_2 satisfies the applications' requirements, s_2 is equivalent with s_1. In addition, there are two aspects of QoS, namely *state QoS* and *view QoS*. The state QoS means the QoS that the state of the object intrinsically supports. On the other hand, the applications can view the QoS of the object only through the methods.

It takes a long time to perform methods on multimedia objects. The throughput of the system is decreased if each method is mutually exclusively performed. We discuss a new *serialization lock* and *mutually exclusive lock* for each method based on QoS-based conflicting relations among the methods. The serialization locks are used to serialize conflicting methods, while the mutually exclusive locks are used to perform methods mutually exclusively.

The effects obtained by performing methods have to be removed if they do not satisfy applications' requirements. This can be done by *compensation*[8),12)] of the methods performed. It takes time to restore a large volume of multimedia data such as high-resolution video data. We can reduce time taken to recover the system if data with lower resolution but satisfying the applications' requirements are restored instead of restoring the state. In this paper, we discuss a compensation method whereby an object is surely rolled back to a state supporting a QoS that satisfies the requirements, not the previous state.

In Section 2, we present a system model. In Sections 3 and 4, we discuss conflicting relations among the methods based on QoS and how to lock objects.

2. System Model

2.1 Objects

A system is composed of multiple objects. An object o is an encapsulation of data and a collection of abstract methods $op_1, \ldots, op_l$ through which alone o can be manipulated. There are two types of object, *class* and *instance*. A class gives a set of attributes and collection of methods. An instance is a tuple of values each of which is given to each attribute of the class. From now on, an "object" means an instance in this paper. Each object is uniquely identified by an object identifier (*oid*). The *states*, that is, the values of the object, can be changed by the methods, but its *oid* is never changed.

On receipt of a request message with a method op_t, op_t is performed in an object o.

† Department of Computers and Systems Engineering, Tokyo Denki University

$op_t(s)$ shows a state obtained by performing op_t on a state s of the object o, and $[op_t(s)]$ is the response. For example, $[display(s)]$ shows an image displayed on a monitor or printer from a state s of a multimedia object. $op_t \circ op_u$ means that op_u is performed after op_t is terminated. Here, a method op_t *conflicts* with a method op_u if $op_t \circ op_u(s) \neq op_u \circ op_t(s)$ for some state s of the object o [8]. For example, a method *record* conflicts with *delete* in an object *movie*. op_t is *compatible* with op_u unless op_t conflicts with op_u. We assume that the conflicting relation is symmetric.

An object is composed of other objects. For example, a *scene* object is composed of objects showing a person, car, road, and background. In MPEG-4, multimedia data are composed of multiple objects such as audio/video objects (AVOs) and sound objects.

2.2 Quality of Service (QoS)

Applications obtain service supported by an object o through the methods of the object o. Each service is characterized by parameters such as the level of resolution. The a quality of service (QoS) supported by the object o is given by the parameters.

The *scheme* of QoS is a tuple of attributes $\langle a_1, \ldots, a_m \rangle$ $(m \geq 1)$. Let $\mathrm{dom}(a_i)$ be a *domain* of an attribute a_i, that is, a set of possible values to be taken by a_i $(i = 1, \ldots, m)$. A QoS *instance* q of the scheme $\langle a_1, \ldots, a_m \rangle$ is given in a tuple of values $\langle v_1, \ldots, v_m \rangle \in \mathrm{dom}(a_1) \times \ldots \times \mathrm{dom}(a_m)$. Let $a_i(q)$ show value v_i of an attribute a_i in the QoS instance q. Let S be a set of QoS instances. A QoS value v_1 *precedes* another one v_2 $(v_1 \succeq v_2)$ in $\mathrm{dom}(a_i)$ if v_1 shows a better QoS than v_2. For example, $120 \times 100 \preceq 160 \times 120$ [pixels] for the attribute *resolution*. q_1 *totally dominates* q_2 iff q_1 and q_2 have the same scheme $\langle a_1, \ldots, a_m \rangle$ and $a_i(q_1) \succeq a_i(q_2)$ for every attribute a_i. Let A be a subset $\langle b_1, \ldots, b_k \rangle$ of the QoS scheme $\langle a_1, \ldots, a_m \rangle$ where $b_j \in \{a_1, \ldots, a_m\}$ $(j = 1, \ldots, k)$ and $k \leq m$. A QoS *instance* q_1 of a scheme A_1 *partially dominates* q_2 of A_2 iff $a(q_1) \succeq a(q_2)$ for every attribute a in $A_1 \cap A_2$. q_1 *dominates* q_2 $(q_1 \succeq q_2)$ iff q_1 partially dominates q_2 and $A_1 \supseteq A_2$. Let S be a set of QoS *instances* whose schemes may be different. A QoS instance q_1 is *minimal* in S iff there is no QoS instance q_2 in S such that $q_2 \preceq q_1$. q_1 is *minimum* iff $q_1 \preceq q_2$ for every q_2 in S. q_1 is *maximal* iff there is no q_2 in S such that $q_1 \preceq q_2$. q_1 is *maximum* iff $q_2 \preceq q_1$ for every q_2 in S. A *least upper bound* (*lub*) $q_1 \cup q_2$ is some QoS instance q_3 in S such that 1) $q_1 \preceq q_3$ and $q_2 \preceq q_3$, and 2) there is no instance q_4 in S where $q_1 \preceq q_4 \preceq q_3$ and $q_2 \preceq q_4 \preceq q_3$.

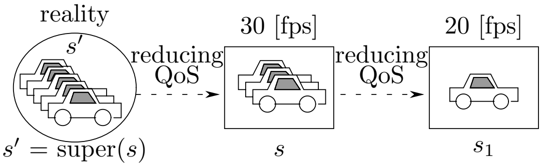

Fig. 1 State-equivalent states ($s \approx s_1$).

Applications require an object o to support a QoS which is referred to as its *requirement* QoS (*RoS*). Let r be a RoS instance. Here, suppose an object o supports a QoS instance q. If $q \succeq r$, the applications can get enough service from q. Here, q is referred to as *satisfy* r. Otherwise, q is *less qualified* than r.

2.3 QoS of an Object

In this paper, we assume that objects support the same type of media. Real objects in the real world support the maximum, possibly infinite, level of QoS. A real object is realized in a computer by reducing the QoS of the object. Thus, each state s' of the real object is realized by mapping the maximum level of QoS to the limited level depending on the capabilities of the computer, that is, $Q(s') \succeq Q(s)$. The state of the real object is referred to as the *super state*. Let $\mathrm{super}(s)$ denote the super state of a state s of an object o that is realized in the computer. We assume that there exists exactly one super state for each state s.

[**Definition**] A state s_1 is *state-equivalent* with s_2 of an object o $(s_1 \approx s_2)$ iff $\mathrm{super}(s_1) = \mathrm{super}(s_2)$. □

For example, suppose that a state s_1 of the object *video* supports video data with a frame rate of 30 fps (**Fig. 1**). Suppose that a new state s_2 with a frame rate 20 fps is obtained by dropping some frames in the state s_1. If $s_2 \approx s_1$, s_1 and s_2 are derived from the same super state s' by reducing the QoS, but they support different levels of QoS. Let $SE(s)$ be an equivalent set $\{s_1 \mid s_1 \approx s\}$ for a state s. Every state in $SE(s)$ has the same super state. Practically speaking, $s_1 \approx s_2$ if s_1 could be obtained from another state s_2 by changing the QoS of s_2.

The QoS of an object o has two aspects: *state QoS*, which shows the QoS of a state of o, and *view QoS*, which is obtained by performing a method of o. For example, let us consider an object *video* with a method *display* as

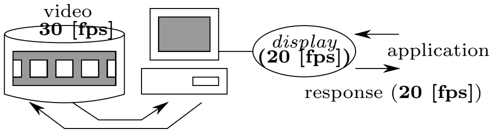

Fig. 2 QoS of video object.

shown in **Fig. 2**. A state s of the object *video* supports video data with a frame rate 30 [fps] and 32 [colors]; that is, its state QoS $Q(s) = \langle 30\,[\text{fps}],\ 32\,[\text{colors}]\rangle$. The method *display* can display the view $[display(s)]$ of the video data on the monitor from the state s only at a rate of 20 fps; that is its view QoS $Q([display(s)]) = \langle 20\,[\text{fps}],\ 32\,[\text{colors}]\rangle$. Here, there is a constraint "$Q([display(s)]) \preceq Q(s)$" for every state s of an object o. The object o cannot support applications with a higher QoS than the methods can support. If $Q([op_t(s)]) \prec Q(s)$ for some state s of the object o, op_t is *less qualified*. op_t is *fully qualified* if $Q([op_t(s)]) = Q(s)$ for every state s of o. Here, suppose *movie* supports two versions *old-display* and *new-display* of *display*. *new-display* can display video data at a faster rate than *old-display*. *new-display* is considered to be the same as *old-display* because the methods output the same image data and do not change the state of *movie*. However, they support different levels of QoS, that is, $Q([\textit{old-display}(s)]) \preceq Q([\textit{new-display}(s)])$ for every state s of *movie*.

[**Definition**] A method op_t is *more qualified* than another method op_u of an object o iff $Q([op_t(s)]) \succeq Q([op_u(s)])$ and $op_t(s)$ is state-equivalent with $op_u(s)(op_t(s) \approx op_u(s))$ for every state s of the object o. □

The applications cannot differentiate states s_1 and s_2 if $[op_t(s_1)] = [op_t(s_2)]$ in the object o, because the applications view s_1 and s_2 as the same through op_t. A state s_1 is op_t*-equivalent* with a state s_2 in an object o iff $[op_t(s_1)] = [op_t(s_2)]$ for a method op_t.

[**Definition**] A state s_1 is *method-equivalent* with a state s_2 of an object o iff $[op_t(s_1)] = [op_t(s_2)]$ for every method op_t of o. □

Since there are two aspects of objects, namely states and QoS, each object supports two types of primitive methods, a *state* method for manipulating the state of the object and a *QoS* method for manipulating the QoS of the object. *drop* is a QoS method because *drop* only changes the QoS of the object *video*. For a QoS method op_Q, $op_Q(s)$ is state-equivalent with every state s of an object $o\,(op_Q(s) \approx s)$, but $Q(op_Q(s)) \neq Q(s)$. For a pair of QoS methods op_t and op_u, $op_t(s) \approx op_u(s)$ and $[op_t(s)] \approx [op_u(s)]$ for every state s of an object o because they only change the QoS of the object o. On the other hand, for a state method op_S, $Q(op_S(s)) = Q(s)$ while $s \neq op_S(s)$. For example, a method *add* appends some image data in *video* but does not change the QoS. **Figure 3** shows a transition diagram where a node shows a state and a directed edge indicates a state transition. A horizontally directed edge "$s \rightarrow s_1$" indicates that a state s is transitted to a state s_1 by a state method. Here, $Q(s_1) = Q(s)$. A vertically directed edge "$s \rightarrow s_2$" shows that s_2 is obtained from s by changing the QoS of s through a QoS method. Here, $s \approx s_2$. A public method changes not only the state but also QoS of the state. In Fig. 3, an edge $s \rightarrow s_3$ denotes that op changes both the state and QoS.

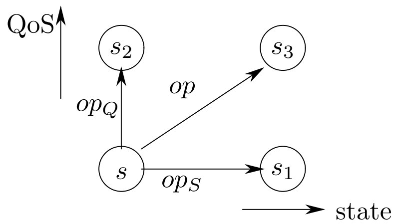

Fig. 3 Transition diagram.

3. QoS Relations among Methods

3.1 Equivalency

We discuss how methods $op_1, \ldots, op_l$ supported by an object o are related with respect to the QoS. A method op_t is *equivalent* with another method op_u in an object o iff $op_t(s) = op_u(s)$ and $[op_t(s)] = [op_u(s)]$ for every state s of o. That is, op_t and op_u not only output the same response data but also change the state of the object o to the same state.

An object *movie* is composed of two subobjects: *advertisement* and *content* objects. The *advertisement* object is removed from *movie* by a method *delete*. The movie obtained from the original version m_1 is the updated version m_2. An application does not take only account of the difference between the versions m_1 and m_2 of *movie*, since it is interested only in the content of *movie*. m_2 is considered to be *equivalent* with m_1 from the application's point of view. m_1 and m_2 support the same QoS.

[**Definition**] A state s_1 is *semantically equiva-*

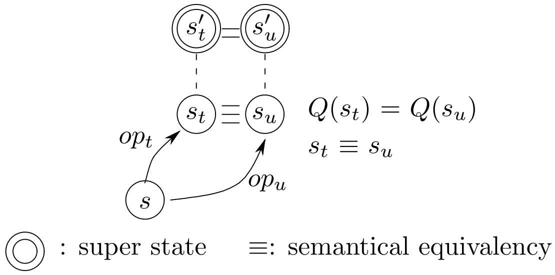

Fig. 4 Semantically equivalent methods.

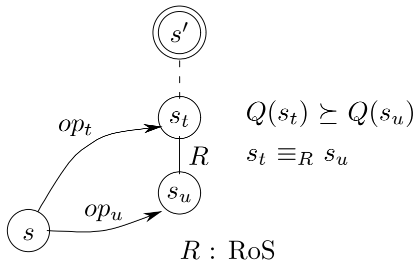

Fig. 6 RoS-equivalent methods.

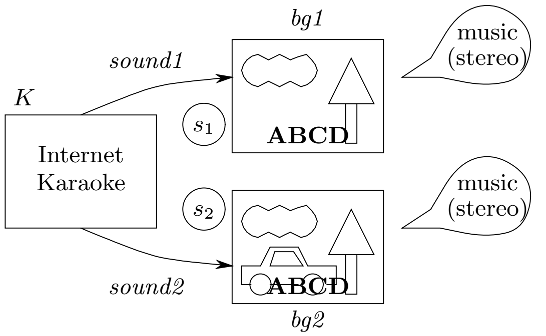

Fig. 5 Internet karaoke object.

lent with s_2 of an object $o\,(s_1 \equiv s_2)$ iff super(s_1) and super(s_2) are considered to be the same by the application. □

Suppose that an application considers a pair of super states s'_t and s'_u of an object o to be the same. Suppose $s_t = op_t(s)$ and $s_u = op_u(s)$ for a state s of the object o. States s_t and s_u are obtained by reducing QoS of s'_t and s'_u, respectively, and $Q(s_t) = Q(s_u)$ (**Fig. 4**). Here, s_t is semantically equivalent with s_u ($s_t \equiv s_u$).

[**Definition**] A method op_t is *semantically equivalent* with op_u in an object $o\,(op_t \equiv op_u)$ iff $op_t(s) \equiv op_u(s)$ for every state s of o. □

Suppose that a class c is composed of subclasses $c_1, \ldots, c_m\,(m \geq 0)$. An application specifies whether each c_i is *mandatory* or *optional* for the class c. If c_i is mandatory, an object o of c is required to include an object o_i of c_i. If c_i is optional, the object o may not include o_i. For a pair of objects o_1 and o_2 for a class c, a state s_1 of o_1 is defined to be *semantically equivalent* with a state s_2 of o_2 iff the subobjects o_{1i} and o_{2i} for every mandatory subclass c_i have the same state in s_1 and s_2, respectively. The optional subclass c_k can take any state.

Let us consider an Internet karaoke object K (**Figure 5**) composed of multimedia subobjects, namely *music*, *words*, and *background* objects (AVOs), which are realized by using MPEG-4 [10]. K supports a pair of methods *sound1* and *sound2*. The method *sound1* plays sound data *music* while displaying a background *bg1* with *words*. *sound2* plays sound data *music* while displaying words and background video *bg2* that includes an object *car*. Here, let s_1 and s_2 be states obtained by performing *sound1* and *sound2* on K, respectively. Suppose an application is interested only in *music* and *words* but not in *background*. Here, *music* and *words* are mandatory, but *background* is optional in K. s_1 and s_2 are semantically equivalent ($s_1 \equiv s_2$), although $s_1 \neq s_2$ and $Q(s_1) = Q(s_2)$. Hence, *sound1* is *semantically equivalent* with *sound2* (*sound1* $\equiv$ *sound2*).

Let R be RoS. The application does not take account of the display speed of the object *movie*. Two methods *old-display* and *new-display* are considered to be equivalent with respect to R if they support a QoS satisfying R, even if $Q\left([\textit{old-display}(s_{movie})]\right) \neq Q\left([\textit{new-display}(s_{movie})]\right)$ for every state s_{movie} of *movie*.

[**Definition**] A state s_t is *RoS-equivalent* with a state s_u on RoS $R\,(s_t \equiv_R s_u)$ in an object o iff $Q(op_t(s)) \cap Q(op_u(s)) \succeq R$ and $op_t(s)$ is state-equivalent with $op_u(s)\,(op_t(s) \approx op_u(s))$ for every state s of o. □

[**Definition**] A method op_t is *RoS-equivalent* with a method op_u on RoS R in an object $o(op_t \equiv_R op_u)$ iff $op_t(s) \equiv_R op_u(s)$ for every state s of o. □

In **Fig. 6**, $s_t(= op_t(s)) \approx s_u(= op_u(s))$ because super(s_t) = super$(s) = s'$. If $Q(s_t)$ and $Q(s_u)$ satisfy RoS R, $op_t \equiv_R op_u$. In addition, op_t is more qualified than op_u since $Q(s_t) \succeq Q(s_u)$.

In the example of the object *movie*, suppose that *new-display* supports a higher level of QoS than *old-display*. The versions are not only semantically equivalent but also RoS-equivalent if *old-display* satisfies the application's require-

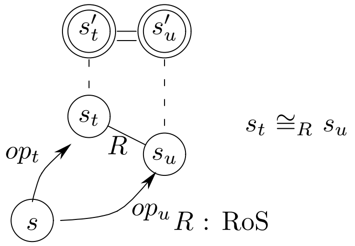

Fig. 7 Semantically RoS-equivalent methods.

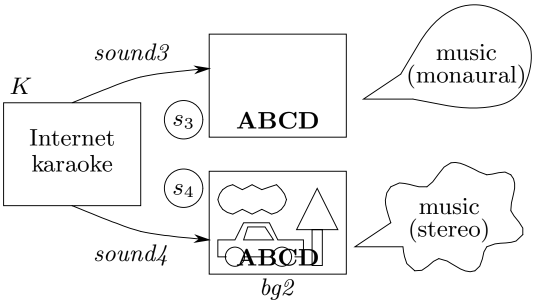

Fig. 8 Internet karaoke object.

ment.

[**Definition**] A state s_t is *semantically RoS-equivalent* with a state s_u on RoS $R\,(s_t \cong_R s_u)$ in an object o iff $\text{super}(op_t(s)) \equiv \text{super}(op_u(s))$ and $Q(op_t(s)) \cap Q(op_u(s)) \succeq R$ for every state s of o. □

[**Definition**] A method op_t is *semantically RoS-equivalent* with op_u of an object o on RoS $R\,(op_t \cong_R op_u)$ iff $op_t(s) \cong_R op_u(s)$ for every state s of o. □

In **Fig. 7**, $s_t = op_t(s)$ and $s_u = op_u(s)$. s'_t ($= \text{super}(s_t)$) $\equiv s'_u$ ($= \text{super}(s_u)$). $Q(s_t)$ and $Q(s_u)$ satisfy RoS R while $Q(s_t)$ may not be the same as $Q(s_u)$. Here, $op_t \cong_R op_u$ since $s_t \cong_R s_u$. It is straightforward for the following property to hold:

[**Proposition**] A method op_t is semantically RoS-equivalent with op_u on RoS R in an object o if $op_t \equiv_R op_u$. □

The Internet karaoke object K supports two types of methods, *sound3* and *sound4* (**Fig. 8**). *sound3* plays monaural sound obtained from *music* and displays *words* without *background*. *sound4* plays stereo *sound* while displaying *words* with the background object *bg2*. Here, let s_3 and s_4 be states obtained by performing *sound3* and *sound4*, respectively. The QoS obtained by performing *sound3* is different from *sound4* from an application point of view. That is, *sound4* is not semantically equivalent with *sound3* (*sound4* $\not\cong$ *sound3*) even if $\text{super}(s_4) \equiv \text{super}(s_3)$. Suppose a requirement QoS (RoS) R

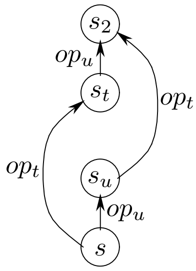

Fig. 9 QoS-compatible methods.

shows that the application does not care whatever the background is and how qualified the music is. The states obtained by performing *sound3* and *sound4* satisfy the application's requirement R. That is, *sound3* $\cong_R$ *sound4*.

3.2 Compatibility

We discuss in which order a pair of methods op_t and op_u supported by an object o can be performed in order to keep the object o consistent. In the traditional theory [1),8)], op_t *conflicts* with op_u iff the result obtained by performing op_u and op_t depends on the computation order. For example, *write* conflicts with *read*.

Suppose a multimedia object M displays MPEG-4 data composed of two objects showing a colored background and a car. A method *add* inserts an object *car* into the object M. A method *grayscale* changes a color video to a black-and-white gradation video. Suppose *grayscale* is performed on M after *add*. The MPEG-4 data obtained by performing *add* and *grayscale* is a black-and-white gradation video with the background and the car. However, the data obtained by performing *add* after *grayscale* is different from that obtained by performing the methods in the reverse order.

[**Definition**] A QoS method op_t is *QoS-compatible* with a QoS method op_u in an object o iff $s \approx op_u(s) \approx op_t(s) \approx op_t \circ op_u(s) \approx op_u \circ op_t(s)$ and $op_t \circ op_u(s) = op_u \circ op_t(s)$ for every state s of o (**Fig. 9**). □

If op_t is not QoS-compatible with op_u, op_t it QoS-conflicts with op_u. In the MPEG-4 example, *add* QoS-conflicts with *grayscale*.

Suppose MPEG-4 data is displayed from the multimedia object M, whose QoS is $\langle 30\,[\text{fps}], 256\,[\text{colors}]\rangle$. The method *mediascale* of the object M reduces the frame rate to half of the original one. The method *reduce* decreases the number of colors to 16. The application can obtain the same QoS by performing *mediascale* and *reduce* in any order. In any case, the application can get the MPEG-4 data with 15 fps

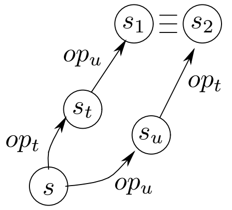

Fig. 10 Semantically compatible methods.

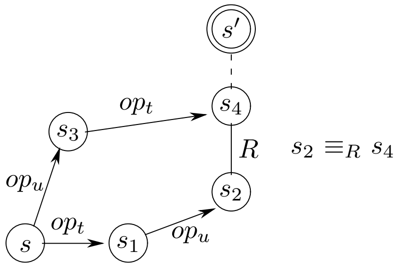

Fig. 11 RoS-compatible methods.

and 16 colors.

[**Definition**] A method op_t is *semantically compatible* with op_u in an object o iff $op_t \circ op_u(s) \equiv op_u \circ op_t(s)$ for every state s of o. □

op_t semantically conflicts with op_u unless op_t is semantically compatible with op_u. In **Fig. 10**, $s_1 = op_t \circ op_u(s)$ and $s_2 = op_u \circ op_t(s)$. $s_1 \equiv s_2$ if the super states of s_1 and s_2 are equivalent in the application. Here, op_t is semantically compatible with op_u.

[**Proposition**] A method op_t is semantically compatible with op_u in an object o if op_t is QoS-compatible with op_u. □

[**Definition**] A method op_t is *RoS-compatible* with op_u on RoS R in an object o iff $op_t \circ op_u(s)$ is RoS-equivalent with $op_u \circ op_t(s)$ on $R(op_t \circ op_u(s) \equiv_R op_u \circ op_t(s))$ for every state s of o. □

In **Fig. 11**, a state s_4 is state-equivalent with $s_2\,(s_4 \approx s_2)$ because super(s_2) = super(s_4). $Q(s_2) \neq Q(s_4)$ but $Q(s_2)$ and $Q(s_4)$ satisfy RoS R. Here, s_2 is RoS-equivalent with s_4 on $R\,(s_2 \equiv_R s_4)$.

Unless a method op_t is RoS-compatible with op_u, op_t *RoS-conflicts* with op_u. In the multimedia object M, the methods *reduce* and *mediascale* are RoS-compatible but *add* RoS-conflicts with *grayscale*.

Suppose an application is not interested in how colorful movies are. A method *update* changes an object *movie* from a color version to a monochrome one. The application *displays* the color *movie* m, i.e., $[display(m)]$. If *update* is performed on the state m, the monochrome

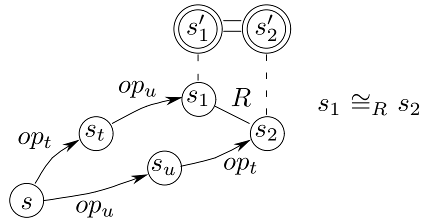

Fig. 12 Semantically RoS-compatible methods.

version of m is seen. Since the application is not interested in the color of the movie m, both versions are considered to satisfy RoS R. Hence, $Q([display(m)]) \cap Q([update \circ display(m)]) \succeq R$ and $Q(display \circ update(m)) = Q(update \circ display(m))$. *display* and *update* are RoS-compatible. However, they semantically conflict, because $Q([update \circ display(m)]) \neq Q([display(m)])$.

[**Definition**] A method op_t is *semantically RoS-compatible* with op_u on R in an object o iff $op_t \circ op_u(s)$ is semantically RoS-equivalent with $(\cong_R)\, op_u \circ op_t(s)$ on R for every state s of o. □

In **Fig. 12**, $s_1 = op_t \circ op_u(s)$ and $s_2 = op_u \circ op_t(s)$ for a state s of an object o where s'_1 (= super(s_1)) $\equiv s'_2$ (= super(s_2)). $Q(s_1)$ and $Q(s_2)$ satisfy RoS R. Hence, $s_1 \equiv_R s_2$ and op_t is RoS-compatible on R with op_u. Unless $op_t \cong_R op_u$, op_t *semantically RoS-conflicts* with op_u. For example, suppose *movie* is composed of subobjects *background* and *car*. *movie* supports a pair of methods *add*, which inserts *car* into the MPEG-4 data, and *grayscale*, which changes a color video object to a black-and-while gradation video. The response obtained by performing *add* after *grayscale* shows black-and-white gradation *background* and colored *car*. However, the white-black gradation video is obtained by performing the methods in the reverse order. Suppose the application is interested in a colored *car*. The response obtained by performing *add* after *grayscale* satisfies the applications' requirement R. However, the response obtained by performing the methods in the reverse order does not satisfy R. That is, *add* semantically RoS-conflicts with *grayscale*.

[**Proposition**] A method op_t is semantically RoS-compatible with op_u on RoS R in an object o if op_t is semantically or RoS-compatible on R with op_u. □

Every compatible and conflicting relation is assumed to be symmetric in this paper.

4. Synchronization

Multiple transactions concurrently manipulate a multimedia object o. According to the synchronization theory [1], every pair of conflicting methods issued by transactions have to be serializable. That is, every pair of conflicting methods issued by different transactions are required to be performed on every object in the same order. For this purpose, the object o is locked before every method op_t is performed on the object o. If the object o is already locked for a method op_u conflicting with op_t, op_t blocks until the lock of op_u is released. On the other hand, every pair of compatible methods such as *read* can be concurrently performed, i.e., interleaved.

In this section, let us consider that "conflict" is one of the kinds of conflicting relations discussed in this paper, namely semantically, RoS-, and semantically RoS- conflicting relations. In a conflicting relation, a pair of methods op_t and op_u conflict in an object o iff the result obtained by performing op_t and op_u depends on the computation order of op_t and op_u. Here, the object o is locked in order to serially perform op_t and op_u; that is, one of op_t and op_u is performed after the other one completes.

In the multimedia object M discussed in the preceding sections, the method *reduce* is RoS-compatible with *mediascale* on some RoS R. This means that *reduce* and *mediascale* can be performed in any order for a given RoS R. However, *reduce* and *mediascale* cannot be interleaved, that is, they cannot be mutually exclusive. A pair of *display* methods can be performed in any order, since *display* is compatible with itself. In addition, the methods can be interleaved. The traditional concurrency control theories [1] assume that every pair of conflicting methods are mutually exclusive, whereas compatible methods can be interleaved. However, some pairs of compatible methods cannot necessarily be interleaved in the multimedia objects. Hence, we introduce two new types of lock modes for a method op_t:

1. *serialization* lock mode $\sigma(op_t)$, and
2. *mutual exclusion* lock mode $\mu(op_t)$.

Serialization locks are used to serialize conflicting methods, while mutual exclusion locks are used to make methods performed mutually exclusively.

[**Definition**] For every pair of methods op_t and op_u,

1. $\sigma(op_t)$ *conflicts with* $\sigma(op_u)$ and $\mu(op_t)$ *conflicts with* $\sigma(op_u)$ iff op_t conflicts with op_u.
2. $\mu(op_t)$ *conflicts with* $\mu(op_u)$ iff op_t cannot be performed concurrently with op_u. □

The conflicting relation is assumed to be symmetric. Suppose that an object o is locked for a method op_t and another method op_u is issued to the object o. If $\sigma(op_u)$ conflicts with $\sigma(op_t)$, op_t blocks until op_u terminates. Suppose there are two objects x and y, where x has methods op_1 and op_2 and y has op_3 and op_4. A transaction T_1 issues op_1 to x and op_3 to y. Another transaction T_2 issues op_2 to x and op_4 to y. First, suppose that op_1 is compatible with op_2 and that op_3 is compatible with op_4. Here, the modes $\sigma(op_1)$ and $\sigma(op_3)$ are compatible with $\sigma(op_2)$ and $\sigma(op_4)$, respectively. op_1 and op_2 can be performed on x and op_3 and op_4 can be performed on y in any order. Suppose that op_1 is performed on x after op_2.

Next, suppose that $\mu(op_1)$ conflicts with $\mu(op_2)$ and that $\mu(op_3)$ is compatible with $\mu(op_4)$. The method op_2 can be started after op_1 completes. op_1 and op_2 cannot be interleaved. However, op_3 and op_4 can be interleaved on y.

[**Property**] A mode $\mu(op_t)$ conflicts with another mode $\mu(op_u)$ if $\sigma(op_t)$ conflicts with $\sigma(op_u)$. □

If op_t conflicts with op_u, op_t and op_u cannot be interleaved. For example, a pair of methods *reduce* and *mediascale* conflict and cannot be performed on the multimedia object M at the same time.

If a transaction T issues a method op_t to an object o, o is locked according to the following protocol:

[**Locking protocol**]

1. The transaction T issues a lock request $\sigma(op_t)$ to the object o.
2. If the object o is locked in a mode $\sigma(op_t)$, o is tried to be locked in a mode $\mu(op_t)$.
3. If o is locked in the mode $\mu(op_t)$, op_t is ready to be performed.
4. If o is not successfully locked in the mode $\mu(op_t)$, op_t blocks.
5. If o is not successfully locked in the mode $\sigma(op_t)$, op_t blocks. □

By the locking protocol, multiple compatible methods can be performed in the interleaved

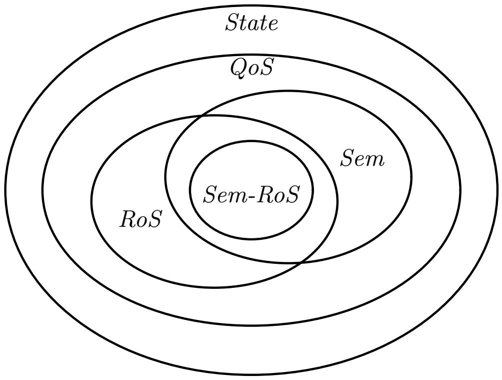

Fig. 13 Conflicting relations.

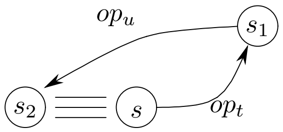

Fig. 14 Semantically compensating method.

manner.

Figure 13 shows that relations of four kinds of conflicting relations among methods. Here, QoS shows a set of possible QoS-conflicting relations. RoS, Sem, and Sem-RoS indicate sets of RoS-, semantically, and semantically RoS-conflicting relations, respectively. *State* shows a state-based conflicting relation. For example, a method op_t QoS-conflicts with op_u if op_t semantically conflicts with op_u.

5. Compensation

A method op_u is a *compensating* method of op_t if $op_t \circ op_u(s) = s$ for every state s of an object o [5),8)]. Let s_1 be a state obtained by performing op_t on a state s of o; that is $s_1 = op_t(s)$. Here, o can be rolled back to the initial state s from the state s_1 if the compensating method of op is performed on s_1. For example, *append* is a compensating method of *delete*.

[**Definition**] A method op_u *semantically compensates* op_t in an object o iff $op_t \circ op_u(s) \equiv s$ for every state s of o (**Fig. 14**). □

RoS-compensating methods are defined as follows on the basic of the RoS-equivalent relations.

[**Definition**] A method op_u *RoS-compensates* op_t on RoS R in an object o iff $op_t \circ op_u(s) \equiv_R s$ for every state s of o and R. □

In **Fig. 15**, it is noted that s_2 is state-equivalent with s ($s_2 \approx s$); that is, s and s_2 have the same super state s'. However, s and s_2 satisfy RoS R.

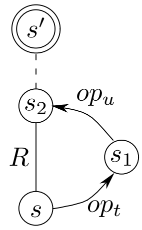

Fig. 15 RoS-compensating method.

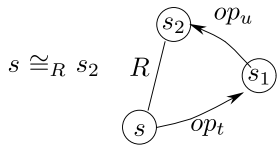

Fig. 16 Semantically RoS-compensating method.

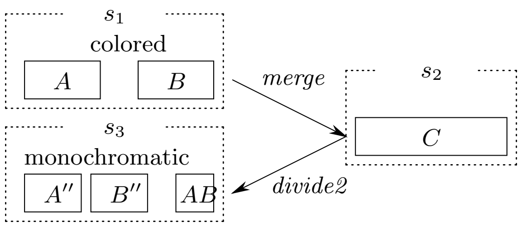

Fig. 17 Example of semantically RoS-compensating method.

[**Definition**] A method op_u *semantically RoS-compensates* a method op_t on RoS R in an object o iff $op_t \circ op_u(s) \equiv_R s$ for every state s of o. □

Figure 16 shows that op_u is a semantically RoS-compensating method of op_t on RoS R. A state $s_2\,(= op_t \circ op_u(s))$ is semantically RoS-equivalent on R with a state s of an object $o\,(s \cong_R s_2)$.

Suppose that, in addition to the methods *merge* and *delete*, the multimedia object *ME* supports a method *divide2* that divides the movie C into three subobjects A'', B'', and AB (**Fig. 17**). A state s_1 of *ME* is composed of two subobjects A and B. A'' and B'' show the content parts of A and B, respectively, which are monochrome in state s_3. AB includes the advertisement objects A and B. Let s_3 denote a state where the objects A'', B'', and AB are obtained from A and B existing at state s_1. $s_1 \neq s_3$. Furthermore, $Q(s_3) \neq Q(s_1)$ because A and B are color but A'' and B'' are monochrome. That is, $Q(s_1) \succeq Q(s_2)$. Suppose an application would like to see the monochrome object. This is RoS R. Here, $Q(s_3) \succeq R$. Hence, *divide2* is a *semantically*

RoS-compensating method of *merge* on R. By performing *divide2*, ME can be restored from s_2 to s_3 instead of s_1.

[**Proposition**] A method op_u semantically RoS-compensates a method op_t on RoS R in an object o if op_u is a semantically compensating or RoS-compensating method of op_t on R. □

6. Concluding Remarks

We have discussed novel relations among methods on the basis of the QoS and state, that is, semantically, RoS-, and semantically RoS-equivalent and conflicting relations in object-based systems. We presented a locking protocol to realize QoS-conflicting relations, which introduces new lock modes, serialization, and mutually exclusive modes. By using serialization and mutually exclusive locks, we can increase the performance of a 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. IEEE InfoCom*, pp.732–793 (1993).
3) Campbell, A., Coulson, G. and Hutchison, D.: A Quality of Service Architecture, *ACM SIGCOMM Comp. Comm. Review*, Vol.24, pp.6–27 (1994).
4) Gall, D.: MPEG: A Video Compression Standard for Multimedia Applications, *Comm. ACM*, Vol.34, No.4, pp.46–58 (1991).
5) Garcia-Molina, H. and Salem, K.: Sagas, *Proc. ACM SIGMOD*, pp.249–259 (1987).
6) Kanezuka, T. and Takizawa, M.: QoS Oriented Flexibility in Distributed Objects, *Proc. Int'l Symp. on Comm.* (*ISCOM'97*), pp.144–148 (1997).
7) Kanezuka, T. and Takizawa, M.: Quality-based Flexibility in Distributed Objects, *Proc. 1st IEEE Int'l Symp. on Object-oriented Real-time Distributed Computing* (*ISORC'98*), pp.350–357 (1998).
8) Korth, H.F., Levy, E. and Silberschalz, A.: A Formal Approach to Recovery by Compensating Transactions, *Proc. VLDB*, pp.95–106 (1990).
9) MPEG Requirements Group: MPEG-4 Requirements, ISO/IEC JTC1/SC29/WG11 N2321 (1998).
10) MPEG Requirements Group: MPEG-4 Applications, ISO/IEC JTC1/SC29/WG11 N2322 (1998).
11) Object Management Group Inc.: The Common Object Request Broker: Architecture and Specification, Rev 2.0 (1995).
12) Takizawa, M. and Yasuzawa, S.: Uncompensatable Deadlock in Distributed Object-Oriented Systems, *Proc. IEEE ICPADS-92*, pp.150–157 (1992).
13) Yoshida, T. and Takizawa, M.: Model of Mobile Objects, *Proc. DEXA'96*, Vol.1134, pp.623–632, *Springer-Verlag* (1996).

(Received March 8, 1999)
(Accepted October 7, 1999)

Tetsuo Kanezuka was born in 1974. He received his B.E. degree in computers and systems engineering from Tokyo Denki Univ., Japan in 1997. He is now a graduate student of the master course in the Dept. of Computers and Systems Engineering, Tokyo Denki Univ. His research interests include distributed multimedia networks.

Katsuya Tanaka was born in 1971. He received his B.E. and M.E. degree in Computers and Systems Engineering from Tokyo Denki University, Japan in 1995 and 1997, respectively. From 1997 to 1999, he worked for NTT Data Corporation. Currently, he is a assistant in the Department of Computers and Systems Engineering, Tokyo Denki University. His research interests include distributed systems, transaction management, recovery protocols, and computer network protocols.

Hiroaki Higaki was born in Tokyo, Japan, on April 6, 1967. He received the B.E. degree from the Dept. of Mathematical Engineering and Information Physics, the Univ. of Tokyo in 1990. He is in the Dept. of Computers and Systems Engineering, Tokyo Denki Univ. He received the D.E. degree in 1997. His research interests include distributed algorithms and computer network protocols. He is a member of IEEE CS, ACM and IEICE.

Makoto Takizawa was born in 1950. He received his B.E. and M.E. degrees in Applied Physics from Tohoku Univ., Japan, in 1973 and 1975, respectively. He received his D.E. in Computer Science from Tohoku Univ. in 1983. From 1975 to 1986, he worked for Japan Information Processing Developing Center (JIPDEC) supported by the MITI. He is currently a Professor of the Dept. of Computers and Systems Engineering, Tokyo Denki Univ. since 1986. From 1989 to 1990, he was a visiting professor of the GMD-IPSI, Germany. He is also a regular visiting professor of Keele Univ., England since 1990. He was a program co-char of IEEE ICDCS-18, 1998 and serves on the program committees of many international conferences. His research interests include communication protocols, group communication, distributed database systems, transaction management, and security. He is a member of IEEE, ACM, IPSJ, and IEICE.