マルチメディア通信と分散処理　52－24
（1991．9．26）

# 階層型トランザクションの補償時に生じるデッドロックの二つの解法

滝沢 誠, 兵頭 章子

東京電機大学理工学部経営工学科

埼玉県比企郡鳩山町石坂

**あらまし**　階層型トランザクションの同時実行により生じるデッドロック問題について考える。トランザクションのアボート方法としては、更新された物理的なページ状態をログに記憶し、これにより更新前の状態に戻す方法がこれまでのシステムでは用いられてきている。しかし、*CAD*等のトランザクションは、従来のトランザクションとは異なり、実行時間が長くかつ多くのデータオブジェクトを利用する。こうしたシステムでは、ページの状態をログに記憶する方法では、ログが大きくなってしまう問題点がある。このために、本論文では、ページ状態ではなく実行された演算をログに記憶し、トランザクションをアボートするためには、ログ内の演算の補償演算を実行する方法を考える。補償演算自体もトランザクションであることから、トランザクションのアボート時にデッドロックが生じる場合がある。本論文では、ある補償演算の実行により解除できないデッドロックが存在することを示すとともに、これの解決方法として、退避点を用いるものと、用いないものを示す。

# Two Methods to Resolve Deadlock to Compensate Nested Transactions

Makoto Takizawa and Akiko Hyoudou

Dept. of Information and Systems Engineering
Tokyo Denki University

Ishizaka, Hatoyama, Hiki, Saitama 350-03, Japan
TEL. 0492-96-2911 ext.2406 FAX. 0492-96-6185
{taki, hyo}@takilab.k.dendai.ac.jp

Abstract　Since transactions in new applications like *CAD* require more objects for longer time than the conventional ones, there is higher possibility that deadlock occurs, and also it is important to reduce data stored in the log to abort and restart transactions. In the conventional database systems, when some deadlock occurs, one deadlocked transaction $T$ is selected and the whole part is aborted by restoring the old states in the log. Another way to abort $T$ is to execute a *compensate operation* $op^{\sim}$ of each operation $op$ executed in $T$. Each $op^{\sim}$ is a transaction which restores the old state on which $op$ is applied. By this method, we can reduce time for aborting and restarting $T$ since only a part of $T$ is aborted, and can reduce the log size since operations are stored in the log in stead of storing the state. The *compensate operations* may cause further deadlock, since the *compensate operations* are also transactions and require locks on the objects. In this paper, we show that there exists *unresolvable deadlock* which cannot be resolved by some *compensate operations*. We show that there is some system where no *unresolvable deadlock* occurs. Also, we show two methods for resolving the *unresolvable deadlock*, i.e. stack and save-point based ones.

## 1. Introduction

Transactions in conventional database systems are flat sequences of *read/write* operations on physical page objects. In order to realize complex applications, transactions have to be composed of transactions as modules, i.e. *nested* [1, 15-17, 21-23]. Since these transactions require more objects and take longer time than the conventional ones, there is higher probability that deadlock [11, 18] occurs. In this paper, we discuss how to resolve deadlock in the interleaved execution of nested transactions. The deadlock is detected if a wait-for graph [10, 11] includes some deadlock cycle. One transaction $T$ in the cycle is selected and is aborted by restoring the old state from the state log [2, 9]. Since nested transactions hold more objects for longer time, it is important to reduce cost for aborting the transactions and storing recovery data in the log. In our method, only a part of $T$ which is necessary to resolve the deadlock is aborted instead of aborting the whole part to reduce cost for aborting and restarting $T$. Also, operations are stored in the log in stead of storing the page image to reduce the log size. In order to abort the operations in $T$, the compensate operations [6, 13] are executed. We assume that every operation $op$ has some compensate operation $op^{\sim}$ such that for every system state $s$, $op^{\sim}(op(s)) = s$. Since the compensate operations are also transactions, they require locks of objects. Another deadlock may occur by the compensate operations. We show that there exists deadlock which cannot be resolved by some compensate operations. Also, we present two methods, i.e. stack and save point based ones to resolve the unresolvable deadlock when it is detected.

In section 2, we describe a system model. In section 3, we define the deadlock of nested transactions. In section 4, the compensate operations are discussed. In section 5, we discuss the unresolvable deadlock. In section 6, we present one method to resolve the unresolvable deadlock.

## 2. System Model

### 2.1 System Structure

A system $M$ is composed of objects. Each object $o$ is an abstract data type [14], i.e. a pair of data structure and two kinds of operations for manipulating, i.e. *primitive* and *non-primitive* operations. Users can manipulate $o$ by using only the non-primitive operations. The primitive operations directly manipulate $o$ without using any lock [5] and do not call another operations. Each non-primitive operation of $o$ requests a lock on $o$ in a mode $mode(op)$ and can call primitive operations of $o$ and non-primitive operations.

**[Example 2.1]** A banking system $B$ is composed of two objects, *Bank* and *File*. The accounts of individuals are stored in *File*. *Bank* is a view of *File*. *Bank* provides non-primitive operations *Withdrawal*, *Deposit*, *Check*, *Open*, and *Close*. *File* provides non-primitive operations *Read* and *Write*, and primitive ones *Fread* and *Fwrite*. *Withdrawal*($x$) withdrawals money from an account $x$ and calls *Read* and *Write* on *File*. *Deposit*($x$) calls *Read* and *Write* on *File* to deposit money in $x$. *Check*($x$) reads $x$ and calls *Read* on *File*. *Open*($x$) and *Close*($x$) opens and closes $x$, respectively, which call *Write* on *File*. *Read* and *Write* are realized by primitive *Fread*, and *Fread* and *Fwrite*, respectively. *Fread*($x$) and *Fwrite*($x$) read and write $x$ in *File* without locking $x$, respectively. For example, *Write*($x$) is realized by a lock, *Fread*, and *Fwrite* on $x$. In *Bank*, *Withdrawal* and *Deposit* are compatible, but they are not compatible with *Check*, *Open*, and *Close*. *Check*, *Open*, and *Close* are not compatible. In *File*, *Read* and *Write*, and two *Write* are not compatible. □

### 2.2 Transactions

A transaction $T$ is an atomic sequence of operations [5, 8]. Each operation $op$ calls another operations $op_1, ..., op_n$, which is written as $<[op, op_1, ..., op_n, op]>$. $[op$ and $op]$ denote *begin* and *commit* of $op$, respectively. Each $op_i$ may call another operations $op_{i_1}, ..., op_{i_{l_i}}$. Here, $op_i$, $op_{i_j}$, ... are said to be *called* by $op$. Thus, $T$ is structured in an ordered tree, i.e. *nested* [17]. The depth-first ordering of nodes in the tree denotes the execution sequence of operations. $op_1$ *precedes* $op_2$ if $op_1$ is executed before $op_2$ in $T$. The immediate successor and predecessor are defined as usual. $lca(op_1, op_2)$ denotes a least common ancestor of $op_1$ and $op_2$ in $T$. Here, let $op_T$ denote an operation $op$ in $T$.

**[Example 2.2]** A transaction $V$ to transfer money from an account $x$ to $y$, i.e. withdrawal the money from $x$ and deposit it to $y$, is represented as shown in Fig.1. Here, let $W$, $D$, $C$, $r$, and $w$ denote *Withdrawal*, *Deposit*, *Close*, *Read*, and *Write*, respectively. Also, let $fr$ and $fw$ be *Fread* and *Fwrite*, respectively. $V$ denotes the root transaction. The leaves like $fr(x)$ are primitive operations. $<[V, [W(x), [w(x), fr(x), fw(x), w(x)], W(x)], [D(y), [w(y), fr(y), fw(y), w(y)], D(y)], V]>$ denotes the execution sequence of operations in $V$. Another transaction $U$ transfers all the money from $x$ to $y$, and then closes $x$. □

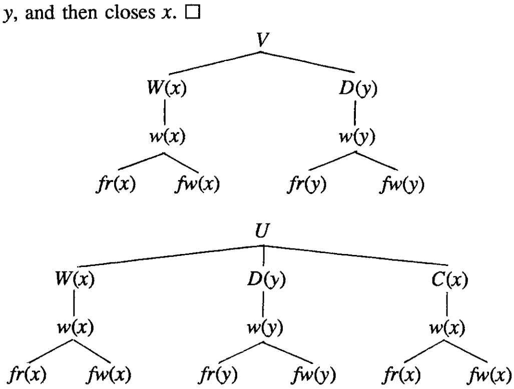


**Fig.1** Transaction Trees

On *Bank* of Fig.1, any interleaved sequence like $L_1 = <W_V(x), W_U(x), D_U(y), D_V(y), C_U(x)>$ and $L_2 = <W_V(x), W_U(x), D_U(y), C_U(x), D_V(y)>$ is serializable with respect to the compatibility relation among the operations of *Bank*. From [15-17, 21], $<w_V(x), w_U(x), w_U(y), w_V(y), w_U(x)>$ obtained from $L_1$ is *semantically* correct although it is not serializable from the serializability theories [2, 5].

$T$ has a local state, i.e. the local variables. The total state of the system $M$ is a pair of the system state and the collection of the transaction states. The system state is a set of object states. The transaction state of $T$ is stored in a transaction stack $ST_T$. The local variables of $op_j$ are allocated in $ST_T$ when $op$ (directly) calls $op_j$, and are popped up when $op_j$ commits.

### 2.3 Synchronization Method

For a nested transaction $T$, the following synchronization mechanism is used to execute an operation $op$ on an object $o$ [16]. Here, $Lock(op)$ is a set of objects held by $op$ or by the operations called by $op$.

**[Locking scheme]** (1) If *op* is the root, nothing is done and *Lock(op)* = $\phi$.

(2) [*op* is not the root] If *o* is already locked, (2-1) if *mode(op)* is compatible with every mode of operation which locks *o*, *op* can be applied to *o*, (2-2) otherwise, *op* waits until (2-1) holds.

(3) When $op_1$, ..., $op_m$ called by *op* commit, all the locks in *Lock(op)* are released if *op* is the *root* of *T*. Otherwise, no lock in *Lock(op)* is released, and $Lock(op) = \{o\} \cup Lock(op_1) \cup ... \cup Lock(op_m)$. □

The locking scheme is two-phase locked [5]. Here, *o* is *held* and *obtained* by *op iff op* locks *o* and *iff o* is held by *op* or by operations called by *op*, respectively. *[op* is a lock on *o* and allocates area for storing the local state in $ST_T$. *op]* releases $ST_T$ and locks obtained in *op* if *op* is the root of *T*. If not, *op]* denotes only the end of *op*, and releases the local state from $ST_T$.

Among two modes $m_1$ and $m_2$ of *o*, $m_1$ is *less exclusive* than $m_2$ ($m_1 \subseteq m_2$) [12] *iff* for every mode $m_3$, (1) if $m_1$ is compatible with $m_3$, then $m_2$ is compatible with $m_3$, and (2) if $m_3$ is compatible with $m_2$, then $m_3$ is compatible with $m_1$. For example, *mode(Withdrawal)* $\subseteq$ *mode(Close)*. $m_1$ and $m_2$ are *equivalent iff* $m_1 \subseteq m_2$ and $m_2 \subseteq m_1$. Let $\cup$ be a *least upper bound* (*lub*) on $\subseteq$. If *T* uses the same object *o* by multiple operations, the lock mode on *o* has to be converted [12]. For example, suppose that *Bank* is locked by *Withdrawal* and then by *Close* in Example 2.1. Here, the mode on *Bank* has to be converted from *mode(Withdrawal)* to *mode(Close)* when *Close* is applied. Since *T* is two- phase locked, the mode cannot be converted to a less exclusive mode. Suppose that *o* is locked in a mode $m_1$ by *T* already, and $op_2$ in *T* tries to lock *o* in a mode $m_2$. If $m_1 \subseteq m_2$, the mode is converted to $m_2$ if $m_2$ is compatible with every mode of the holders of *o*. If $m_2 \subseteq m_1$, $op_2$ uses *o* without converting the modes. If neither $m_1 \subseteq m_2$ nor $m_2 \subseteq m_1$, the mode is converted to $m_1 \cup m_2$ if $m_1 \cup m_2$ is compatible with every mode of the holders of *o*.

## 3. Deadlock

A state *s* of the system *M* is represented in a well-known *wait-for* (*WF*) graph[10, 11]. We extend the *WF* graph to include the precedence relation among the operations. First, an operation $op_1$ *depends* on $op_2$ ($op_1 \rightarrow op_2$) *iff* (1) $op_1$ waits on an object held by $op_2$ in an incompatible mode, (2) $op_1$ precedes $op_2$ for some transaction *T*, or (3) for some $op_3$, $op_1 \rightarrow op_3 \rightarrow op_2$. An *extended wait-for* (*EWF*) graph *G* is a directed graph whose nodes are operations, and whose edges denote the dependency relation $\rightarrow$. An operation *op* is *deadlocked iff* it is included in a directed cycle of *G*, i.e. $op \rightarrow op$.

Let *Current(T)* denote the current operation being executed in *T*. Let *FirstD(T)* be a directly deadlocked operation in *T* such that there is no directly deadlocked operation which precedes it. In Fig.2, *Current(V)* is *[w(y)*, and *FirstD(V)* is *[W(x)*.

**[Example 3.1]** In Fig.1, when $[w_V(y)$ requires *y*, *y* is already obtained by $[w_U(y)$. Thus, $[w_V(y)$ waits for $[w_U(y)$. Also, $[C_U(x)$ waits on *x* obtained by $[W_V(x)$ because *C* and *W* on the same object are incompatible. The *EWF* is represented in Fig.2. Here, $[w_V(y) \rightarrow [w_U(y) \rightarrow [C_U(x) \rightarrow [W_V(x) \rightarrow [w_V(y)$. Thus, *V* and *U* are deadlocked. □

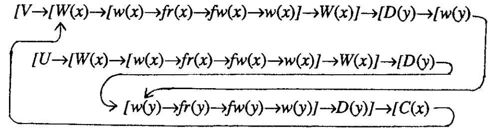


**Fig.2** *EWF* Graph and Deadlock

## 4. Compensation

### 4.1 Stack Based (STB) Compensation

A *compensate* $op^{\sim}$ of *op* is an operation such that for every system state *s*, $op^{\sim}(op(s)) = s$. The meaning and properties of the compensate operations are discussed in [13]. It is noted that compensate operations are also transactions, i.e. they require locks on objects. In this paper, we assume that for every *op*, at least one compensate operation $op^{\sim}$ is defined based on the application semantics. $\langle op_1, ..., op_m\rangle^{\sim}$ is $\langle op_m^{\sim}, ..., op_1^{\sim}\rangle$. For example, *Withdrawal*$^{\sim}$ is *Deposit*, and *Deposit*$^{\sim}$ is *Withdrawal* in Example 2.1.

Another point to be considered in the compensation is that each operation *op* in *T* has a local state. Suppose that *op* calls $op_1$, ..., $op_m$. Each time when $op_j$ is called, the state is changed from $\langle S_{j-1}, L_{j-1}\rangle$ to $\langle S_j, L_j\rangle$ where $S_j$ is a system state and $L_j$ is a local state of *op* ($j = 1, ..., m$). Then, when $op_m$ commits, *op* commits and the local state is released from $ST_T$. Finally, *op* changes the system state from $S_0$ to $S_m$. Here, suppose that some failure occurs just after $op_j$ commits, i.e. the state is $\langle S_j, L_j\rangle$ and $op_j$ has to be aborted. $S_{j-1}$ is restored by $op_j^{\sim}$. Problem is whether $op_j$ can be restarted after $op_j$ is compensated by $op_j^{\sim}$. If $op_j$ is executed on $\langle S_{j-1}, L_j\rangle$, the different result might be obtained from the first execution of $op_j$. However, if *op* is restarted after all $op_1$, ..., $op_j$ called by *op* are aborted and the local state of *op* is released from $ST_T$, the problem mentioned here does not occur since the local state of *op* is newly created when *op* is called again.

**[Example 4.1]** Let us consider *Move* which transfers all the money in an account *x* to *y* and closes *x* which is shown as follows. Here, suppose that the operations from (2) to (5) have to be aborted after (1) to (5) complete. By the compensate operations of (5) and (2), the update effect on *x* and *y* are removed and (2) is restarted. Although the states of *x* and *y* are restored by the compensate operations, the local state of *t* is not restored, i.e. still one obtained by (4). If (2) is restarted, *t*'s value obtained at (4) is used to withdrawal money from *x* by (2). Here, (2) fails because *x* is over- withdrawn. Therefore, in order to restart the operation aborted, the local state of *Transfer* is also restored. After compensating (5) and (2), and releasing the area for the local state from the stack $ST_{Move}$, if *Transfer* is called again, (2) and (5) can be restarted because the local states of *t* and *u* are recreated in $ST_{Move}$. □

*Move(x, y) account x, y;*
{ *int t, u;* (1) $t = Read(x)$; (2) *Withdrawal(x, t)*;
(3) $u = Read(y)$; (4) $t = t + u$; (5) *Deposit(y, t)*;
(6) *Close(x)*;}

In order to restart the operations, both the old system and local states must be restored as explained in Example 4.1. Suppose that $op_1$ is *Current(T)* and $op_2$ is *FirstD(T)*. If operations from $op_1$ to $op_2$ in *T* are aborted, the deadlock can be resolved. Also, we assume that the local state of each *op* is created in $ST_T$ when *op* is called. Let $op_0$ be $lca(op_1, op_2)$. Let $op_{00}$ be an operation which calls $op_0$. The local state of $op_{00}$ is stored in $ST_T$. Therefore, if all the operations in $op_0$ are aborted by the compensate operations, $op_0$ can be restarted from $ST_T$.

**[Example 4.2]** In Example 3.1, the operations from *[W(x)* to *[r(y)* are deadlocked [Fig.3]. Since *lca(W(x), r(y))* is the root *V*, *V* has to be aborted. One method is to abort all the operations from *[V* to *[r(y)* by $<([r(y)))^{\sim}, [D(y)^{\sim}, (W(x)])^{\sim}, w(x)]^{\sim}, fw(x)^{\sim}, ([w(x)))^{\sim}, (r(x)])^{\sim}, fr(x)^{\sim}, ([r(x)))^{\sim}, ([W(x)))^{\sim}, ([V)^{\sim}>$. Here, for *op*, $([op)^{\sim}$ releases the locks obtained in *op*. $(op])^{\sim}$ does nothing. Here, $W^{\sim}$, $D^{\sim}$, $w^{\sim}$, $fw^{\sim}$, and $fr^{\sim}$ are *D*, *W*, *w*, *fr*, and *null*, respectively. It corresponds to the conventional abortion of *V*. On the other hand, *T* can be aborted by another $<[V, W(x), [D(y), [r(y)>^{\sim} = <([r(y)))^{\sim}, ([D(y)))^{\sim}, W(x)^{\sim}, ([V)^{\sim}>$. Thus, the latter includes higher level operations than the former ones. □

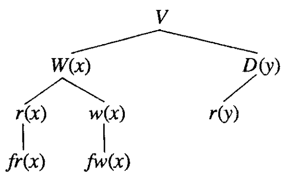


**Fig.3** Transaction Tree

In Example 4.2, $<[V, W(x), [D(y), [r(y)>$ is called a *greatest committed sequence* (*GCS*) of *V* from *[V* to the current *[r(y)*, where it does not include operations called by operations committed. A *GCS* of *T* from $op_0$ to $op_1$ is defined to be an operation sequence obtained by the following *GCS* procedure. Here, for a sequence $L = <a_1, ..., a_m>$, let $L_i$ denote the *i*th element $a_i$. For a sequence $I = <i_1, ..., i_k>$, $<L, I>$ denotes $<a_1, ..., a_m, i_1, ..., i_k>$.

**[GCS Procedure** *GCS(T, $op_0$, $op_1$)*] (1) Let *L* be a subsequence of *T* from $op_0$ to $op_1$. Let *M* be ϕ. Let *m* be the number of the elements in *L*.

(2) For *i* = *m*, *m* -1, ..., 1,
  if $L_i$ = *op]*, then { *j* = *i*; do *j* = *j* -1 while $L_j$ is not *[op* and *j* > 0;
    if *j* > 0 /* *[op* is found */, then $M = <<op>, M>$; *i* = *j*;}
    else /* not found */ $M = <<L_i>, M>$;
  else $M = <L_i, M>$;

(3) *M* is the greatest committed sequence. □

The *GCS* includes only *begin*/*commit* operations and committed operations. Let *GCS(T)* denote *GCS(T, lca(FirstD(T), Current(T)), Current(T))*. On the other hand, a subsequence from *lca(FirstD(T), Current(T))* to *Current(T)* is named a *transaction sequence* (*TRS*) of *T*, *TRS(T)*. For example, $<[V, W(x), [D(y), [r(y)>$ is *GCS(V)* and *TRS(V)* is $<[V, [W(x), [w(x), fr(x), fw(x), w(x)], W(x)], [D(y), [r(y)>$ in Fig.1.

### 4.2 Save Point Based (SPB) Compensation

In a save point of *T*, the local state of *T*, i.e. $ST_T$ is saved in the transaction log $L_T$ by an operation *Save*.

**[Example 4.3]** For *V* in Example 3.1, suppose that operations from *[w(x)* to *[r(y)* are directly deadlocked, and *V* takes a save point *s* after *r(y)* in *W(x)* as shown in Fig.4. Unless *s* is taken in *V*, the *GCS* from *[V* to *[r(y)*, i.e. $<[V, W(x), [D(y), [r(y)>$ is compensated and then *V* is restarted. However, since *s* is taken just before *[W(x)*, after compensating the *GCS* $<w(x), W(x)], [D(y), [r(y)>$ from *[w(x)* to *[r(y)*, the local state of *V* is restored from $L_V$ saved by *s*, and the operations are restarted from *w(x)*. □

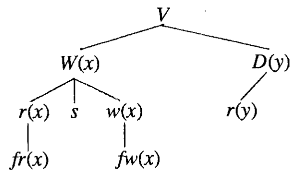


**Fig.4** Save Point

Suppose that *T* is deadlocked, $op_1$ = *Current(T)*, and $op_2$ = *FirstD(T)*. Let *sp* be a save point preceding and nearest to $op_2$ in *T*. The procedure to abort and restart *T* is shown as follows.

**[Save Point Based (SPB) Compensation]** (1) A *GCS G* from *sp* to $op_1$ is obtained by *GCS(T, sp, $op_1$)*.

(2) The compensate of *G* obtained by (1) is executed.

(3) The local state is restored from $L_T$ obtained at *sp*.

(4) *T* is restarted from *sp*. □

Unless any save point is taken before $op_2$, all the operations on *T* have to be aborted, i.e. totally aborted.

## 5. Unresolvable Deadlock

### 5.1 Definition

**[Example 5.1]** Suppose that $T_1 = <[T_1, [a, b, [c, d, [e>$ where a deadlock cycle exists from *[c* to *[e*, is selected to be aborted. $<[c, d, [e>$ in $T_1$ is aborted by $<([e)^{\sim}, d^{\sim}, ([c)^{\sim}>$ where $d^{\sim}$ calls a sequence $<d_1, d_2, d_3, ...>$. $([e)^{\sim}$ means that $T_1$ stops requesting the object of *e*. Then, the system is in a state *s*, $T_1 = <[T_1, [a, [b, .., b], [c, [d, ..., d], [d^{\sim}, d_1, d_2, [d_3>$ and the *GCS* $<[T_1, [a, b, [c, d, [e, ([e)^{\sim}, [d^{\sim}, d_1, d_2, [d_3>$. Also, $<[T_2, [k, l, m, [m^{\sim}, m_1, [m_2>$ where $m^{\sim}$ calls $<m_1, m_2, ...>$. Suppose that $[m_2$ requests a lock on *x* held by *[a*, and $[d_3$ requests a lock on *y* held by *[k*. In order to resolve the deadlock, one operation, say *[a*, is selected to be aborted. First, $<[d^{\sim}, d_1, d_2, [d_3>^{\sim}$ is executed. One problem is that another deadlock may occur while it is being aborted. Here, suppose that the abortion is successful. In order to abort *d*, $d^{\sim}$ has to be executed again. However, $d^{\sim} = <[d^{\sim}, d_1, d_2, d_3, ...>$ implies the same deadlock as shown in Fig.5. Even if $T_2$ is tried to be partially aborted, the deadlock cannot be resolved in the same way. Such deadlock is *unresolvable*. □

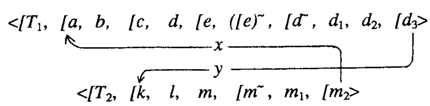


**Fig.5** Unresolvable Deadlock

Unresolvable deadlock is deadlock which may not be resolved by the compensate of the *GCS*. Now, we define formally what is the unresolvable deadlock. The following notations are defined for *T*. Let *I(T)* be a compensate operation appeared lastly in *T*, and *L(T)* be a set of operations which precede *I(T)* in *T*. *T* is said to *unsafely depend* on a transaction *U* *iff* there is some operation *op* in *L(U)* such that *Current(T)* depends on ($\Rightarrow_L$) *op*.

**[Definition]**[19] *T* is said to be *unresolvable deadlocked* in a state *iff T* unsafely depends on *T*. □

### 5.2 Safe System

In Example 4.2, $W(x)^{\sim}$ (= *D(x)*) uses the same *Bank* in the mode equivalent to *W(x)*, and *File* is used by *Read* and

*Write* called by $W(x)^{\sim}$. Since *Bank* and *File* are held by $W(x)$, $W(x)^{\sim}$ can use them without waiting. Like this, even if the compensate is executed, no unresolvable deadlock occurs. For each operation *op* in *T*, $op^{\sim}$ is said to be *safe* if $op^{\sim}$ and every operation called by $op^{\sim}$ can use the objects without waiting on them to abort *T*.

Based on the conversion concept [12], a modified conversion $C_x^y$ which has the following compatibility relation is introduced.

**[Compatibility of $C_x^y$]** (1) $z$ is compatible with $C_x^y$ if $z$ is compatible with $x$, and $y$ is compatible with $z$.
(2) $C_x^y$ is compatible with $z$ if $y$ is compatible with $z$.
(3) $C_x^y$ is compatible with $C_v^w$ if $x$ is compatible with $v$, and $y$ and $w$ are compatible with each other. □

Here, (2) and (3) are the same as $U_x^y$ [12]. Let *op* be an operation of *T*, and $x$ and $y$ be modes of *op* and $op^{\sim}$, respectively. Suppose that after *op* in *T* locks $o$ in $C_x^y$, another transaction *U* locks $o$ in $z$ according to (1). When $op^{\sim}$ would like to lock $o$, $op^{\sim}$ can use $o$ since $y$ is compatible with both $x$ and $z$. Here, it is noted that *op* precedes $op^{\sim}$ in *T*. If every operation *op* obeys this conversion rule, $op^{\sim}$ can use $o$ without causing further deadlock.

**[Conversion rule]** (1) If $y \subseteq x$, *op* locks $o$ in $x$ and then $op^{\sim}$ can use $o$ in $x$.
(2) If $x \subseteq y$, *op* holds $o$ in $C_x^y$, and $op^{\sim}$ converts the mode to $y$.
(3) If neither $x \subseteq y$ nor $y \subseteq x$, *op* holds $o$ in $C_x^{x \cup y}$ and then $op^{\sim}$ converts the mode to $x \cup y$. □

Suppose that $op_1$ called by $op^{\sim}$ tries to hold an object $o_1$ in a mode $m_1$, and $o_1$ is already held in $m_2$ in *T*. If $m_2 \subseteq m_1$, $op_1$ can use $o_1$ in $m_2$ without waiting. For example, suppose that *Read* tries to use *record* locked in a *Write* mode by the same transaction. In this case, *Read* can use *record*. Unless $m_2 \subseteq m_1$, since $op_1$ has to require more exclusive lock than $m_2$, $op_1$ may wait. This may cause further deadlock.

**[Safe condition]** For every operation $op_1$ called by $op^{\sim}$, there exists $op_2$ called by *op* such that $mode(op_1) \subseteq mode(op_2)$. □

If (1) $op^{\sim}$ satisfies the conversion rule and (2) every operation called by $op^{\sim}$ satisfies the safe condition, $op^{\sim}$ is said to be *safe*. A system is *safe* if every compensate operation is safe. In Example 4.2, $D^{\sim}$ and $W^{\sim}$ are safe.

**[Theorem 5.1]** If the system is safe, no unresolvable deadlock occurs.

**[Proof]** According to the definition, every operation called by $op^{\sim}$ can always use the object since they are already locked by *op*. Therefore, no deadlock occurs while the compensate operations are executed. ■

## 5.3 Logging

$op^{\sim}$ is executed to abort *op* after *op* commits in *T*. Hence, when *op* commits, the operations called by *op* can be removed from the transaction log $L_T$. $L_T$ can be realized as a stack. In the *STB* compensation, each *op* on $o$ is logged by the following procedure.

**[Stack Based (STB) Logging]** (1) If *op* is non-primitive, before *op* is executed on $o$, <*[op*, {$o$}> is pushed down into $L_T$.
(2) If *op* is primitive, before *op* is executed, <*[[op]]*, { }> is pushed down into $L_T$.
(3) After *op* commits, i.e. *op]* is executed, operations are popped up from $L_T$ until *[op*. Here, each time when <*op'*, *O'*> is popped up, $O = O \cup O'$ where *O* is initially $\phi$. <*op*, *O*> is pushed down into $L_T$. □

Here, *O* denotes a set of objects obtained by *op*. The *EWF* graph *G* is constructed from the transaction logs. Suppose that a current operation $op_1$ of $T_1$ waits on an object $o_1$. If there is $op_2$ of $T_2$ such that <$op_2$, $O_2$> is in the stack and $o_1$ is in $O_2$, $op_1 \rightarrow op_2$ in *G*. Suppose that *T* is aborted from $op_2$ to the current $op_1$. The abortion is done as follows.

**[Stack Based (STB) Compensation]** (1) <*op*, *O*> is popped up from $L_T$.
(2) $op^{\sim}$ is executed. The objects in *O* are released. Furthermore, all the objects obtained by $op^{\sim}$ are released when *op* commits. The local state is popped up from $ST_T$.
(3) If *op* is $op_2$, *T* is restarted from $op_2$, else go to (1). □

Next, let us consider save points. Each operation *op* is logged and then *T* is partially aborted from $op_2$ to the current $op_1$ by the following methods.

**[Save Point Based (SPB) Logging]** (1) and (2) are the same as the stack based logging method.
(3) When *Save* is executed, the local state of *T* is pushed into $L_T$.
(4) After *op* commits, the operations are searched from the top of $L_T$. If *Save* is found before *[op* is found, *op]* is pushed down into $L_T$. If *[op* is found, the operations are popped up from $L_T$ until *[op*. <*op*, *O*> is pushed down into $L_T$ where *O* is a set of objects held in *op*, which can be obtained by the same way as the *STB* logging. □

**[Save Point Based (SPB) Compensation]** (1), (2), (3) are the same as the stack based one.
(4) When *Save* is popped up before $op_2$, it is ignored. If *Save* is popped up after $op_2$, the local state of *T* is replaced by one saved in $L_T$. Then, *T* is restarted from the operation immediately preceded by the *Save*. □

## 6. Unsafe System

In an unsafe system, the compensate operations of *T* may require new locks which have not been obtained yet in *T*. The compensate of the *TRS* requires no other locks than ones obtained already since the *TRS* includes only primitive operations and no locks obtained in *T* are released. This is a point to resolve the unresolvable deadlock.

After deadlock is detected, one directly deadlocked transaction *T* is selected. Then, it is checked whether the deadlock is unresolvable or not. If not, *T* is aborted by the compensate of the *GCS*. If the deadlock is unresolvable, *T* is aborted by the compensate of the *TRS*. Here, every operation is saved in $L_T$. First, $op_0$ = *lca(FirstD(T), Current(T))* for the *STB* system or $op_0$ = save point nearest to *FirstD(T)* for the *SPB* system is found in $L_T$. The following is executed for unresolvable deadlock.

**[Unresolvable Deadlock Compensation (URC)]** For each *op* in $L_T$ (in the backward direction), if $op \neq op_0$, then execute $op^{\sim}$ and remove *op* from $L_T$, otherwise terminate. □

By the *URC*, all the operations called by $op_0$ are aborted and the local state of $op_0$ is released in the *STB* system. On the other hand, the local state of *T* is restored from $L_T$ for the *SPB* system. Then, *T* is restarted from $op_0$.

**[Example 6.1]** Let us consider an *EWF* graph of an *STB* system as shown in Fig.6. $T_1$ and $T_2$ are unresolvably deadlocked since *[f* called by $a^{\sim}$ waits for $s$ and *[x* called by $v^{\sim}$ waits for $b$. Suppose that the unresolvable deadlock is detected and $T_1$ is selected to be aborted. By using the

*URC*, since *lca*(*b*, *f*) is *k*, $T_1$ is aborted from *[k* to *[f* by executing the compensate of *TRS*($T_1$), i.e. <*[k*, *[a*, *[b*, *c*, *b]*, *a]*, *[ã*, *[d*, *e*, *[f*>˜. Since no lock operations are executed in the sequence, no further deadlock occurs. □

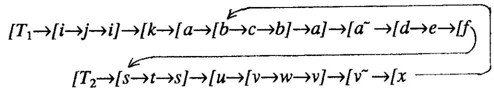


**Fig.6** Unresolvable Deadlock

## 7. Concluding Remarks

In this paper, we have discussed how to resolve deadlock in the interleaved execution of nested transactions. In our method, a transaction is partially aborted, although whole transaction is aborted in the conventional database system. The compensate operations are used to abort the operations. The log size can be reduced since the state like page image is not stored in the log. Since each operation can be considered to be a transaction, the execution of the compensate operations requires the locks on the objects. This means that another deadlock may occur when the compensate operations are executed. We have proved that no unresolvable deadlock occurs in *safe* systems. We have shown methods based on the stack and the save point to resolve deadlock by executing the compensate operations.

## References

[1] Beeri, C., Bernstein, P. A., and Goodman, N., "A Model for Concurrency in Nested Transaction Systems," *JACM*, Vol.36, No.2, 1989, pp.230-269.

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

[3] Chandy, K. M., Misra, J., and Haas, L. M., "Distributed Deadlock Detection," *ACM TODS*, Vol.1, No.2, 1983, pp.144-156.

[4] Chandy, K. M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. on Programming Language and Systems*, Vol.3, No.1, 1985, pp.63-75.

[5] Eswaren, K. P., Gray, J., Lorie, R. A., and Traiger, I. L., "The Notion of Consistency and Predicate Locks in Database Systems," *CACM*, Vol.19, No.11, 1976, pp.624-637.

[6] Garcia-Molina, H. and Salem, K., "Sagas," *Proc. of the ACM SIGMOD*, 1987, pp.249-259.

[7] Garza, J. F. and Kim, W., "Transaction Management in an Object-Oriented Database System," *Proc. of the ACM SIGMOD*, 1988, pp.37-45.

[8] Gray, J., "The Transaction Concept: Virtues and Limitations," *Proc. of VLDB*, 1981.

[9] Haerder, T. and Reuter, A., "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, Vol.5, No.4, 1983.

[10] Holt, R. C., "Some Deadlock Properties on Computer Systems," *ACM Computing Surveys*, Vol.14, No.3, 1972, pp.179-196.

[11] Knapp, E., "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, Vol.19, No.4, 1987, pp.303-328.

[12] Korth, H. F., "Locking Primitives in a Database System," *JACM*, Vol.30, No.1, 1983, pp.55-79.

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

[14] Liskov, B. and Zilles, S. N., "Specification Techniques for Data Abstractions, " *IEEE Trans. on SE*, Vol.1, 1975, pp.294-306.

[15] Lynch, N. and Merritt, M., "Introduction to the Theory of Nested Transactions," *MIT/LCS/TR* 367, 1986.

[16] Moss, J. E., "Nested Transactions: An Approach to Reliable Distributed Computing," *The MIT Press Series in Information Systems*, 1985.

[17] Moss, J, E., Griffeth, N. D., and Graham, M. H., "Abstraction in Concurrency Control and Recovery Management(revised)," *TR COINS* 86-20, *Univ. of Massachusetts*, 1986.

[18] Singhal, M., "Deadlock Detection in Distributed Database Systems," *IEEE Computer*, No.11, 1989, pp.37-48.

[19] Takizawa, M. and Deen, S. M., "Synchronization Problems of Compensate Operations in the Object-Model," *Proc. of International Conf. on Cooperating Knowledge Based Systems*, Keele, England, 1990.

[20] Traiger, I. L., "Trends in System Aspects of Database Management," *Proc. of the 2nd International Conf. on Database (ICOD-2)*, 1983, pp.1-21.

[21] Weihl, W. E., "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types," *ACM Trans. on Programming Language and Systems*, Vol.11, No.2, 1989, pp.249-283.

[22] Weikum, G., "Theoretical Foundation of Multi-level Concurrency Control," *Proc. of the PODS*, 1986, pp.31-42.

[23] Weikum, G., "Principles and Realization Strategies of Multilevel Transaction Management," *ACM TODS*, Vol. 16, No. 1, 1991, pp.132-180.