補償不可能デッドロックの解除法

安沢伸二[†] 滝沢誠[†] S. Misbah Deen[††]

本論文では，階層型トランザクションのインタリープ実行において生じるデッドロックの解除法について論じる．*CAD* のような新しい応用のトランザクションでは，従来の応用に比べて実行時間が長く，多くのオブジェクトを要求する．このために，デッドロックの生じる確率が従来のシステムより高い．デッドロックの解除方法として，デッドロックしたあるトランザクションを選択し，アボートする方法がある．本論文では，デッドロックの解除に必要な部分のみを，実行された演算の補償演算を実行することによりアボートする方法を述べる．ログにシステム状態のかわりに演算を記憶することにより，ログを減少できる．補償演算は，オブジェクトのロックを要求するため，これの実行により，デッドロックが起き得る．本論文では，補償演算によって解除できない補償不可能なデッドロックの存在を示す．また，基本演算の補償演算を実行することによって，補償不可能なデッドロックを解除する方法を述べる．

Resolution of Uncompensatable Deadlock

Shinji Yasuzawa[†], Makoto Takizawa[†] and S. Misbah Deen[††]

In this paper, we discuss how to resolve deadlock occurred in interleaved execution of nested transactions. Since transactions in new applications like groupware systems and *CAD* require more objects for longer time than conventional ones, there is higher possibility that deadlock occurs, and more data has to be stored in the log. We discuss a method for resolving the deadlock where only a part of a deadlocked transaction T is aborted by executing the compensate operations. This method can reduce time for aborting and restarting T, and can reduce the log size. The compensate operations may cause further deadlock, since they require locks on the objects. We show that there exists *uncompensatable deadlock* which cannot be resolved by the compensate operations. Also, we show a method for resolving the uncompensatable deadlock by the compensate operations at the lowest level.

1. はじめに

銀行業務等の従来のトランザクションはデータベース内の小量のデータを短時間操作するものであった．これに対して，グループウェア[16]，*CAD*[13] といった新しい応用では，複雑な構造のデータが，大量に，かつ長時間操作される．従来のトランザクションに対して，こうした応用のトランザクションでは，より多くのデータが長時間操作されることから，デッドロックの発生する確率が増大する．デッドロックを解決する方法として，デッドロック状態のトランザクションの一つを選択し，アボートさせることがある[2]．トランザクションが大量データを長時間操作することから，トランザクションをアボートし，再開するために多くの計算資源が費やされる．このために，まず，トランザクション内で，デッドロックの解決のために必要な部分のみをアボートすることを考える．トランザクション全体をアボートするのに対して，その一部がアボートされることから，アボートと再開のために操作されるデータ量と時間を減少できる．

次の問題は，トランザクションをアボートするために必要となる情報量である．トランザクションが操作するデータ量が増加するにつれて，ログに記憶するデータ量も増大する．したがって，大量データを操作するトランザクションでは，ログの大きさをどのように小さくするかが問題となる．本論文では，更新されたデータではなく，実行された演算をログに記録する方式を用いる．一般に，更新されるページ等の状態情報よりも更新演算を記録する方が，ログの記憶量を減少できる．このため，ここでは実行された演算をログに記録する方式を用いる．トランザクションをアボー

† 東京電機大学理工学部経営工学科
Department of Computer and Systems Engineering, Tokyo Denki University
†† Dake Centre and Dept. of Computer Science, University of Keele

トするためには，実行された演算に対して，その補償演算[5),12),20)]を実行する．補償演算は，演算の実行前の状態に戻すトランザクションである．

補償演算がトランザクションであることから，データのロックを要求することにより，デッドロックが生じる可能性がある．本論文では，この問題について議論し，ある種の補償演算の実行によっては解決できないデッドロックが存在することを示す．これを補償不可能デッドロックとする．さらに，補償不可能デッドロックを起こさないような補償演算の定義方法を示す．

第2章では，システムモデルを述べる．第3章では，補償演算について議論する．第4章では，補償不可能デッドロックを定義する．第5章では，補償不可能デッドロックが起こらない安全システムを示す．第3章では，非安全システムにおける補償不可能デッドロックの解除法を示す．

2. システムモデル

2.1 システム構造

システム M は，通信網によって結合された複数のオブジェクトから構成される．各オブジェクト o は，データ構造と操作演算により与えられる抽象データ型[14)]である．o の属性 a を $o.a$ と示す．操作演算には，基本演算と公開演算がある．ロックによる排他制御なしに，o を直接操作し，他の演算を呼び出さない演算を基本演算とする．これに対して，公開演算は，o の基本演算と他のオブジェクトの公開演算を呼び出すことにより実現される演算である．利用者は，o を公開演算を通してのみ操作できる．また，公開演算は，o のロック[4)]を要求する．ここで，M の各状態 s に対して，$op(s)$ は，演算 op を s に実行して得られた状態を示す．

公開演算 op は，オブジェクト o をモード $mode(op)$ でロック[4)]する．op と op' を o の二つの公開演算とする．op が操作している o を op' が操作できるとき，$mode(op')$ は $mode(op)$ と伴立[11)]である．

[例 2.1] 車の設計を行う CAD システム C を考える．C は *car*, *body*, *chassis*, *seat* の4種類のオブジェクトから構成される．*car* は属性として車高 *height* と車幅 *width* を持つ．また，*car* を拡大する公開演算 *Enlarge* が提供される．*body* は属性として，車高 *height* と車幅 *width* を持ち，公開演算 *Extend*(E), *Shorten*(S), *Up*(U), *Down*(D) を提供する．これらの演算は，基本演算 *Write*(wr) により実現される．wr はオブジェクトの属性値の更新を行う．*body* の E と S は，wr を用いて *width* の拡大あるいは縮小を行う．*body* の U と D は，*height* を増加あるいは減少させる．*chassis* は属性としてシャーシ幅 *width* を持ち，公開演算 E と S を提供する．E と S は，wr により *width* を操作する．車幅の拡大（縮小）と車高の増加（減少）は二つの属性に対する操作であり，車幅を拡大（縮小）しているときに車高を増加（減少）できるので，E と S のモードは D と U に対して伴立である．しかし，同じ属性を操作する2つの E，2つの S，および E と S は伴立でない．□

2.2 トランザクション

トランザクション[4)] T は，オブジェクトの公開演算の実行系列であり，原子的な実行単位である．T の各公開演算 op はあるオブジェクトの演算であり，他の演算 $op_1, \cdots, op_n$ を順に実行する．これを $\langle[op, op_1, \cdots, op_n, op]\rangle$ と書く，$[op$ と $op]$ は，op の開始とコミットを示す．このとき，op は op_i を呼び出し，op を op_i の親とする．さらに，各 op_i が公開演算ならば，他の演算 $op_{i1}, \cdots, op_{im_i}$ を呼び出すことになる．さらに，op_{ij} が他の演算を呼び出すといったように，節点を演算とし，各節点の子の順序が実行順序を示す順序木として T を示せる．これをトランザクション木とし，T を階層型トランザクション[1),4),15),17),18),21)~23)]とする．

[定義] トランザクション T に対するトランザクション木を，以下のように定義される順序木とする．

(1) 根は，トランザクション全体を示す．これを T とする．

(2) 葉節点は，基本演算を示す．

(3) 根と葉以外の各節点 op は，公開演算を示す．op が，他の演算 $op_1, \cdots, op_n$ をこの順序で呼び出し実行するとき，op を親，$op_1, \cdots, op_n$ を子として，この順に並べる．□

ここで，T 内の二つの演算 op_1 と op_2 に対して，$lca(op_1, op_2)$ を，T のトランザクション木内の op_1 と op_2 の最小共通祖先とする．

[例 2.2] 例 2.1 で，$En(=Enlarge)$ は，*car* オブジェクトを拡大するトランザクションである（**図 1**）．b は *body*, c は *chassis*, w は *width* を示す．En は根である．葉の $wr(o, w)$ はオブジェクト o の属性 *width* を操作する基本本演算を示す．$\langle[En, [E(c),$

$wc(c.w)$, $E(c)$], [$U(b)$, $wr(b.h)$, $U(b)$], [$E(b)$, $wr(b.w)$, $E(b)$], En]〉は En の演算系列を示す．次に，car の車幅を広げるトランザクション Wd は，$body$ と $chassis$ の $Extend$ を起動する（**図 2**）．〈[Wd, [$E(b)$, $wr(b.w)$, $E(b)$], [$E(c)$, $wr(c.w)$, $E(c)$], Wd]〉は Wd の演算系列を示す． □

2.3 同期方法

トランザクション T 内の各演算 op は，実行前に op のオブジェクト o をロックする．T によって得られたすべてのロックは，T がコミットしたとき解放されるので，T は2相ロック形式[4]である．記法 op_T は T の演算を示す．[op は，o のロック演算である．ここで，通常の関数呼び出しの機構と同じように，op のローカル変数は，スタック ST_T に割り当てられる．op] は，op の終了を示し，ST_T からローカル変数を解放する．しかし，ロックは解放されない．T] は，T が獲得した全ロックを解放する．

オブジェクト o の2つのモード m_1 と m_2 において，m_2 が m_1 より排他的である（$m_1 \subseteq m_2$）とは，任意のモード m_3 に対して，以下が成り立つことである[11]．

（1） m_1 が m_3 と伴立ならば，m_2 は m_3 と伴立であり，

（2） m_3 が m_2 と伴立ならば，m_3 は m_1 と伴立である．

モード集合は $\subseteq$ について半順序集合となり，$\cup$ は $\subseteq$ 上の最小上界（lub）とする．T の複数の演算が o を操作するとき，o のロックモードを転換[11]する必要がある．例えば，T が，あるオブジェクト x を $read$ した後に，$write$ する場合を考える．$read$ から $write$ にモードを転換するときに，モードがより排他的となるためにデッドロックが起きる可能性がある．$read$ するときに $write$ モードでロックすることも一つの方法であるが，同時実行性が減少してしまう．このために，文献11）には，$read$ より排他的であるが $write$ より排他的でないモード U^{write}_{read} を定め，非対称なモード転換が論じられている．

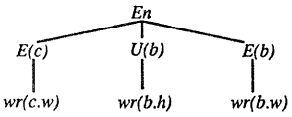


図 1 トランザクション木 En
Fig. 1 Transaction tree of En.

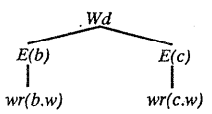


図 2 トランザクション木 Wd
Fig. 2 Transaction tree of Wd.

2.4 デッドロック

演算 op_1 が op_2 に依存する（$op_1 \rightarrow op_2$）ことを，以下のいずれかが成り立つことと定義する．

（1） op_2 が獲得しているオブジェクト o を，op_1 が op_2 と非伴立なモードで待っている．

（2） あるトランザクションで，op_1 は op_2 に先行する．

（3） ある演算 op_3 に対して，$op_1 \rightarrow op_3 \rightarrow oap_2$ である．

節点を演算，op_1 から op_2 の有向辺が $op_1 \rightarrow op_2$ を示す有向グラフを，拡張待ち（EWF）グラフとする．演算 op が EWF グラフ内の有向サイクルに含まれるとき，op はデッドロックしている．本論文では，ある機構[10),19)]により，各システム状態に対する EWF グラフが構成されるとして，デッドロックの解除方法を検討する．

[例 2.3] 図1と図2の2つのトランザクション En と Wd を考える．[$E_{Wd}(b)$ が既に獲得している b(=$body$) を，[$E_{En}(b)$ がロックを要求したとする．このとき，[$E_{En}(b)$ は [$E_{Wd}(b)$ を待ち，[$E_{En}(b) \rightarrow$[$E_{Wd}(b)$ である．同様に，[$E_{Wd}(c) \rightarrow$[$E_{En}(c)$ である．このときの EWF グラフを**図 3** に示す．ここで，[$E_{En}(b) \rightarrow$[$E_{Wd}(b) \rightarrow$[$E_{Wd}(c) \rightarrow$[$E_{En}(c) \rightarrow$[$E_{En}(b)$ であり，En と Wd はデッドロックしている． □

トランザクション T 内で，$Current(T)$ は実行中の演算，$FirstD(T)$ はデッドロックしている最先頭の演算を示すとする．図3で，$Current(En)$ は [$E(b)$, $FirstD(En)$ は [$E(c)$ である．

3. 補　　償

デッドロックが検出されたとき，デッドロックしているあるトランザクション T がアボートされる．ここでは，T をどのようにアボートするかについて考える．

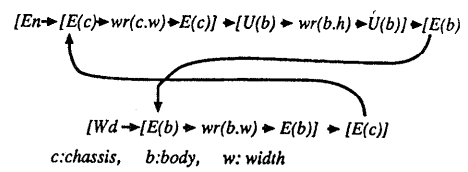


図 3 EWF グラフとデッドロック
Fig. 3 EWF graph and deadlock.

3.1 補償演算

op を演算，s をシステム状態とすると，$op^{\sim}(op(s))=s$ となる演算 $op^{\sim}$ を op の補償演算とする．補償演算の性質は，文献 5)，12)で論じられている．本論文では，各 op に対して，少なくとも 1 つの補償演算 $op^{\sim}$ が応用の意味に基づいて定義されているとする．例えば，例 2.1 で，$E(xtend)^{\sim}$ は $S(horten)$ である．また，$\langle op_1,\cdots,op_m\rangle^{\sim}$ は $\langle op_m{}^{\sim},\cdots,op_1{}^{\sim}\rangle$ である．補償演算が，公開演算ならば，オブジェクトのロックを要求する．あるトランザクションで，op の後に $op^{\sim}$ が実行されたときは，op と $op^{\sim}$ によりロックされたオブジェクトは解放される．このとき，op は $op^{\sim}$ によりアボートされたことになる．補償演算によるアボートを補償とする．基本演算 op の $op^{\sim}$ は，op と同じオブジェクトの基本演算とする．

3.2 補償演算によるアボート

T の各演算 op が呼び出されると，これらのローカル変数の領域が，スタック ST_T に割り当てられる．op が演算 $op_1,\cdots,op_m$ を呼び出すとする．op_j の実行により，状態は $\langle S_{j-1},L_{j-1}\rangle$ から $\langle S_j,L_j\rangle$ に変更される $(j=1,\cdots,m)$．ここで，S_j と L_j は，各々オブジェクトの状態と op のローカル変数の状態である．op_j のコミット後の状態は，$\langle S_j,L_j\rangle$ であり，ここで，障害が起きたとする．$op_j{}^{\sim}$ により，オブジェクト状態は S_{j-1} に復旧されるが，ローカル状態は L_j のままである．ここで，op_j が $\langle S_{j-1},L_{j-1}\rangle$ で再実行されると，$\langle S_{j-1},L_{j-1}\rangle$ で op_j を実行した結果と異なる場合がある．しかし，$op_1,\cdots,op_j$ が $op_j{}^{\sim},\cdots,op_1{}^{\sim}$ によりアボートされ，ST_T から op のローカル変数の解放後に，すなわち，$([op)^{\sim}$ の実行後に，あらためて，op を呼び出すとする．op のローカル変数は，新しく割り当てられるので，ここで述べた問題は起きない．このように，トランザクションを部分的にアボートし，再実行するためには，オブジェクト状態とともにローカル状態の復旧が必要となる．

[例 3.1] *car* オブジェクトを拡大するトランザクション *Enlarge* は**図 4** のように書ける．ここで，*const* はある定数とする．*Enlarge* の演算(4)が終了した後に，(2)までがアボートされるとする．補償演算 $\langle E^{\sim}(c),U^{\sim}(b)\rangle$ により，*chassis* の *width* と *body* の *height* に対する更新結果が除去される．しかし，変数 u の値は復旧されず，(3)で得られた値である．ここで，(2)から再実行されると，(3)で u が再度更新されるために，(4)では，以前の更新と異なった値により *chassis* の *width* が更新される．したがって，再実行するために，u の値も復旧されねばならない．全演算を補償し，スタックから u の領域を解放した後，再び *Enlarge* を呼び出すなら，スタック内に u が新しく割り当てられ，再実行できる． □

Enlarge
(1) 変数 t と u に *car* の *width* と *height* を *read* する．
(2) *body* の *height* を t に変更 $(U(b))$.
(3) $u=u+const$.
(4) *chassis* の *width* を u に変更 $(E(c))$.

図 4 *Enlarge* トランザクション
Fig. 4 Enlarge transaction.

デッドロックしたトランザクション T の全体をアボートするかわりに，デッドロック閉路内の演算をアボートすることにより，デッドロックを解除することを試みる．このようにトランザクションの一部の演算をアボートすることを部分アボートとする．**図 5** で op_1 を $Current(T)$，op_2 を $FirstD(t)$，op_0 を op_1 と op_2 の最小共通祖先 $lca(op_1,op_2)$ とする．また，op_{00} を op_0 の親とする．op_2 が op_1 まで補償しても，例 3.1 に示したように再実行できない．このために，op_0 をアボートする．すなわち，op_{00} が呼び出している演算を補償し，op_{00} のローカル変数を解放する．この後に，op_0 を再実行する．

[例 3.2] 例 2.3 で，*En* の $[E(b)$ から $[E(c)$ までの演算は，デッドロックしている．この場合，$lca(E(b),E(c))$ は根 *En* である．$[En$ から $[E(b)$ までの全演算 $\langle[En,[E(c),wr(w),E(c)],[U(b),wr(h),U(b)],[E(b)\rangle$ を，補償演算系列 $\langle([E(b)^{\sim},(U(b)])^{\sim},wr^{\sim}(w),$

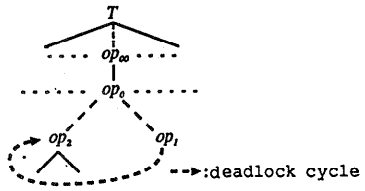


図 5 トランザクション木
Fig. 5 Transaction tree.

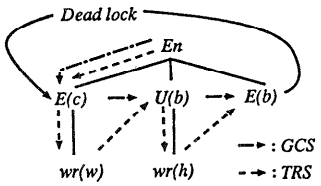


図 6 最大コミット系列
Fig. 6 Greatest committed sequence (*GCS*).

$([U(b)]^{\sim}, (E(c)])^{\sim}, wr^{\sim}(h), ([E(b)]^{\sim}, ([En)^{\sim}\rangle$ により補償することが一つの方法である．ここで，演算 op に対して，$([op)^{\sim}$ は，op によって得られたロックの解放を行う．$(op])^{\sim}$ は，空演算である．一方，各演算は応用の意味にしたがって定義された補償演算を持ち，$E(c)$, $U(b)$ はコミットしている．したがって，$\langle[En, E(c), U(b), [E(b)\rangle^{\sim} = \langle([E(b))^{\sim}, U^{\sim}(b), E^{\sim}(c), ([En)^{\sim}\rangle$ によっても En を補償できる．これは，$[E(b)$ のロックを解放した後，$U^{\sim}(b)$, $E^{\sim}(c)$, $([En)^{\sim}$ が実行されることを示す．ここでは，トランザクション木内のコミットした上位の演算の補償演算が実行される． □

トランザクション T で，$lca(FirstD(T), Current(T))$ から $Current(T)$ までの演算系列をトランザクション系列 $TRS(T)$ とする．$TRS(T)$ は，ロック演算と基本演算の系列である．$TRS(T)$ が，各演算 op について，部分系列 $\langle[op, \cdots, op]\rangle$ を含めば，すなわち，op がコミットしていれば，これを op で置き換えていき，これ以上置き換えできなくなったときに得られる系列を最大コミット系列 $GCS(T)$ とする．$GCS(T)$ は，コミットした演算を含み，コミットした演算が呼び出す演算を含まない．例 3.2 で，$GCS(En) = \langle[En, E(c), U(b), [E(b)\rangle$, $TRS(En) = \langle[En, [E(c), wr(w), E(c)], [U(b), wr(h), E(b)], [E(b)\rangle$ である．$TRS(T)$ は，最下位のレベルで実行された演算の系列である．これに対し，$GCS(T)$ は，コミットした演算の系列であり，上位のオブジェクトの演算から構成されている．したがって，$GCS(T)^{\sim}$ により，より意味的なレベルで T の補償を行える．さらに，$GCS(T)$ は，$TRS(T)$ よりも少ない演算を含むので，ログのサイズを小さくできる．

4. 補償不可能デッドロック

補償演算は，オブジェクトのロックを要求するので，次の例に示すデッドロックが生じる場合がある．

[例 4.1] $T_1 = \langle[T_1, [a, b, [c, d, [e\rangle$ において，$[c$ から $[e$ にデッドロック閉路が存在し，T_1 をアボートするとする．T_1 で，$\langle[c, d, [e\rangle$ は，$\langle([e)^{\sim}, d^{\sim}, ([c)^{\sim}\rangle$ により補償できる．ここで，$d^{\sim}$ は $\langle d_1, d_3, d_3, \cdots\rangle$ を呼び出すとする．この実行により，**図7**に示すように，T_1 と T_2 がデッドロックするとする．最大コミット系列 $GCS(T_1)$ は，$\langle[T_1, [a, b, [c, d, [d^{\sim}, d_1, d_2, [d_3\rangle$ である．$[m_2$ は，$[a$ が獲得しているオブジェクト x のロックを要求し，$[d_3$ は，$[k$ が獲得している y のロックを要求している．このデッドロックを解除するために，T_1 を補償し，$\langle[d^{\sim}, d_1, d_2, [d_3\rangle^{\sim}$ が実行される．その後，d をアボートするために $d^{\sim}$ が再度実行される．この結果，図7と同じデッドロック状態になる．T_2 を補償しても同様である．これを補償不可能デッドロックとする． □

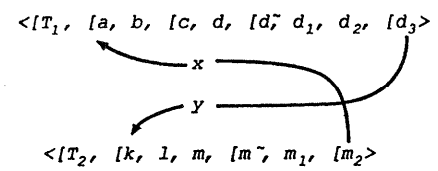


図 7 補償不可能デッドロック
Fig. 7 Uncompensatable deadlock.

補償不可能デッドロックは，トランザクション T の $GCS(T)^{\sim}$ を実行することによって解除できないデッドロックである．補償不可能デッドロックを定義する．まず，$I(T)$ を T 内の補償演算で最後に実行されたものとし，$L(T)$ を T において $I(T)$ に先行する演算の集合とする．ここで，T がトランザクション U に非安全に依存する $(T \Rightarrow U)$ とは，$Current(T)$ が依存する演算が $L(U)$ 内の演算に存在すること，または，$T \Rightarrow V \Rightarrow U$ なるトランザクション V が存在することとする．

[定義] T が補償不可能にデッドロックしているとは，T が T に非安全に依存する $(T \Rightarrow T)$ ことである． □

[例 4.2] 例 4.1 で，$I(T_1) = [d_3, L(T_1) = \{[a, b, [c, d\}$ である．また，$I(T_2) = [m_2, L(T_2) = \{[k, l, m\}$ である．$[d_3$ は $[k$ を待っているので，$T_1 \Rightarrow T_2$．また，$[m_2$ は $[a$ を待っているので，$T_2 \Rightarrow T_1$．したがって，T_1 と T_2 は，補償不可能にデッドロックしている．例 4.1 で示したように，このデッドロックは，最大コミット系列の補償によって解除できない． □

[定理] 補償不可能デッドロックしているトランザクションを最大コミット系列によって補償できない．

[証明] T と U を二つトランザクションとする．op_1 を $Curent(T)$, op_2 を $I(T)$ とする．また，op_2 は op_3 の補償演算とする．$Current(U)$ が依存する T 内の演算を op_4 とする．定義から，T 内の演算の実行系列は，$\langle[T, \cdots, op_4, \cdots, op_3, op_2, \cdots, op_1\rangle$ となる．op_1 から op_4 までを補償せねばならない．op_1 から op_2 までの補償が成功したとしても，op_3 を再度 op_2 で補償せねばならない．このために，はじめのデッドロック状態と同一の状態となってしまう． □

したがって，最大コミット系列によりトランザクシ

ョンを補償すると解除できないデッドロックが存在する．

5. 安全システム

トランザクション T が補償不可能にデッドロックしているとき，$Current(T)$ は，ある補償演算により呼び出されていて，他のトランザクションの演算を待っている．したがって，補償演算内の演算が他の演算を待たずに実行できれば，補償不可能デッドロックは生じない．この問題について考える．まず，演算 op と $op^{\sim}$ について考える．op がオブジェクトを獲得しているもとで，$op^{\sim}$ が実行される．このとき，$mode(op) \supseteq mode(op^{\sim})$ でないならば，ロックモードの転換が必要になる．文献 11)において，モード x から y への転換 U_x^y が論じられている．これをもとに，補償演算に対する転換 C_x^y を以下に定める．

[転換モード C_x^y]

(1) z が x に対して伴立で，y が z に対して伴立なら，z は C_x^y に伴立である．

(2) y が z に対して伴立なら，C_x^y は z に対して伴立である．

(3) x が v に対して伴立で，y と w が互いに伴立なら，C_x^y と C_x^w は伴立である． □

ここで，(2)と(3)は U_x^y と同じである．T の演算 op と $op^{\sim}$ のモードを x と y とする．op が C_x^y で o をロックした後，他のトランザクション S が，以上の(1)に従い，モード z で o をロックしたとする．$op^{\sim}$ が o のロックを要求したとき，y は，x と z に対して伴立なので，$op^{\sim}$ は o をロックできる．このことより，op と $op^{\sim}$ のロックモードの転換を以下のように定める．

[転換規則]

(1) $y \subseteq x$ ならば，op は x で o をロックし，$op^{\sim}$ は x で o を用いることができる

(2) $x \subseteq y$ ならば，op は C_x^y で o を獲得し，$op^{\sim}$ は y にモードを転換する．

(3) $x \subseteq y$ と $y \subseteq x$ のいずれでもない場合，op は $C_x^{x \cup y}$ で o を獲得し，$op^{\sim}$ は $x \cup y$ にモードを転換する． □

各演算 op が転換規則に従えば，$op^{\sim}$ は，待たずに o を操作できる．

次に，$op^{\sim}$ が呼び出す演算 op_1 について考える．T 内の演算によりモード m_2 でロックされているオブジェクト o_1 を，op_1 が m_1 でロックするとする．$m_1 \subseteq m_2$ ならば，op_1 は待たずに o_1 を利用できる．$m_2 \subseteq m_1$ ならば，op_1 は m_2 よりもさらに排他的なロックを要求せねばならないので，op_1 は o_1 を待つ場合がある．これが，補償演算により，デッドロックが起きる原因である．

[安全条件] $op^{\sim}$ が呼び出す任意の演算 op_1 に対して，$mode(op_1) \subseteq mode(op_2)$ である op_2 が，op により呼び出される． □

(1) $op^{\sim}$ が転換規則を満たし，(2) $op^{\sim}$ が呼び出す各演算 op_1 が安全条件を充足するとき，$op^{\sim}$ を安全とする．全補償演算が安全であるとき，システムを安全とする．安全システムにおいて，$op^{\sim}$ は待つことなくロックを行える．

[定理] システムが安全ならば，補償不可能デッドロックは起こらない．

[証明] 定義から，$op^{\sim}$ が呼び出すどの演算も，op によってロックされたオブジェクトのみを用いるので，待たずにロックを行える． □

安全システムでは，補償不可能デッドロックを起こさずに，最大コミット系列により，トランザクションをアボートできる．したがって，ログ内に，トランザクション T の全演算を記録せずに，$GCS(T)$ だけを記録すればよくなり，ログの大きさを縮小できる．即ち，演算 op がコミットしたとき，op が呼び出した演算をログから除き，op だけをログに記憶できる．

安全システムを構築するための転換規則と安全条件は，各補償演算 $op^{\sim}$ を定義するときの指針となる．例えば，$op^{\sim}$ の定義方法がいくつかあるときに，op が用いるオブジェクトのみを用いて定義することが一つの方法となる．このように定義できない場合には，システムは安全なものとは限らない．

6. 非安全システム

次に安全でないシステムにおいて，補償不可能デッドロックの解除法を考える．トランザクション T の最大コミット系列 $GCS(T)$ の補償は，新たなロックを要求する場合がある．一方，トランザクション系列 $TRS(T)$ はロックと基本演算からなり，ロックは解放されない．したがって，$TRS(T)$ は，基本演算の補償を行い，新たなロックを要求しないので，デッドロックは生じない．

[定理] トランザクション T のトランザクション系列 $TRS(T)$ の補償は，デッドロックを起こさない．

[証明] $TRS(T)$ は，基本演算とロック演算の系列で

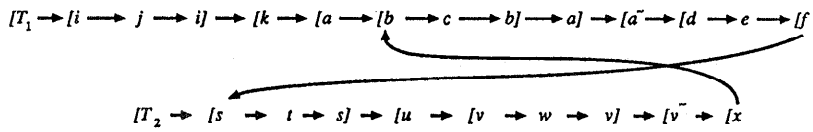


図 8 補償不可能なデッドロック
Fig. 8 Uncompensatable deadlock.

ある．基本演算はオブジェクトを直接操作する演算であり，これの補償演算は同じオブジェクトの基本演算である．トランザクションは 2PL なので，ロックは解放されない．したがって，オブジェクト o の公開演算 op が基本演算 op_1 を呼び出すとき，すなわち $\langle \cdots, [op, \cdots, op_1\rangle$，$o$ は $[op$ によりロックされているので，$op_1{}^{\sim}$ は待たずに実行できる．したがって，デッドロックが起きない． □

[例 6.1] 図 8 で，$\alpha^{\sim}$ が呼び出す $[f$ は $[s$ を待ち，$v^{\sim}$ が呼び出す $[x$ は $[b$ を待つ．したがって，T_1 と T_2 は補償不可能にデッドロックしている．T_1 をアボートする．$lca(b, f)=k$ より，トランザクション系列の補償 $\langle [k, [a, [b, c, b], a], [a^{\sim}, [d^{\sim}, e, [f^{\sim}\rangle = \langle [f^{\sim}, e^{\sim}, ([d^{\sim})^{\sim}, ([a^{\sim})^{\sim}, a]^{\sim}, b]^{\sim}, c^{\sim}, [b^{\sim}, [a^{\sim}, [k^{\sim}\rangle$ を実行し，$[k$ から $[f$ までがアボートされる．この系列には，ロック演算が存在しないので，デッドロックは生じない． □

以上から，どのようなデッドロックも，トランザクション T の $TRS(T)$ を補償することにより解決できることがわかる．$TRS(T)^{\sim}$ は，T が実行した演算を基本演算のレベルで実行と逆の順序で補償していくものである．これに対して，最大コミット系列は，コミットした演算 op に対して，これの $op^{\sim}$ が実行される．$op^{\sim}$ は，op により実行された演算をそのまま補償するとは限らず，意味的に決まるものである．さらに，最大コミット系列は，コミットした演算を含み，これが呼び出した演算を含まないのでトランザクション系列よりも演算が少ない．したがって，ログの記憶量を減少できる利点がある．

7. おわりに

本論文では，階層型トランザクションのデッドロック解除法について議論した．本論文で述べた方法では，従来のデータベースシステムのように，トランザクション全体のアボートではなく，その一部をアボートする．また，演算をアボートするために補償演算を用いる．システムの状態ではなく，演算をログに記憶するために，ログの大きさを減少できる．演算と同様に，補償演算はオブジェクトのロックを要求する．これは，補償演算が実行されたとき，デッドロックが発生する場合があることを意味する．安全システムにおいては，補償演算の実行による補償不可能デッドロックが起こらないことを示した．また，非安全システムにおいて，基本演算の補償演算を実行することによって，補償不可能デッドロックを起こさずにトランザクションを補償できることを示した．本論文で示した以外に，GCS により補償できないデッドロックがあるかどうかについては，今後の課題である．

参考文献

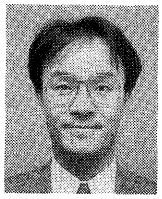
1) Beeri, C., Bernstein, P. A. and Goodman, N.: A Model for Concurrency in Nested Transaction Systems, *JACM*, Vol. 36, No. 2, pp. 230-269 (1989).
2) Bernstein, P. A., Hadzilacos, V. and Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison Wesley (1987).
3) Deen, S. M., Hamada, S. and Takizawa, M., Broad Path Decision in Vehicle System, *Proc. of International Conference on Database and Expert Systems Applications* (*DEXA*), pp. 8-13 (1992).
4) 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, pp. 624-637 (1976).
5) Garcia-Molina, H. and Salm, K.: SAGAS, *Proc. of the ACM SIGMOD Conf.*, pp. 249-259 (1987).
6) Garza, J. F. and Kim, W.: Transaction Management in an Object-Oriented Database System, *Proc. of the ACM SIGMOD Conf.*, pp. 37-45 (1988).
7) Gray, J.: The Transaction Concept: Virtues and Limitations, *Proc. of the 7th VLDB*, pp. 144-154 (1981).
8) Haerder, T. and Reuter, A.: Principles of Transaction-Oriented Database Recovery, *ACM Computing Surveys*, Vol. 5, No. 4, pp. 287-318 (1983).
9) Holt, R. C.: Some Deadlock Properties on Computer Systems, *ACM Computing Surveys*,

Vol. 14, No. 3, pp. 179-196 (1972).
10) Knapp, E.: Deadlock Detection in Distributed Databases, *ACM Computing Surveys*, Vol. 19, No. 4, pp. 303-328 (1987).
11) Korth, H. F.: Locking Primitives in a Database System, *JACM*, Vol. 30, No. 1, pp. 55-79 (1983).
12) Korth, H. F., Levy, E. and Silberschatz, A.: A Formal Approach to Recovery by Compensating Transactions, *Proc. of the 16th VLDB*, pp. 96-106 (1990).
13) Kunii, T. L.: *Visual Database Systems*, Elesevier Science Publishers (North-Holland) (1989).
14) Liskov, B. et al.: Abstraction Mechanisms in CLU, *CACM*, Vol. 20, pp. 564-576 (1977).
15) Lynch, N. and Merritt, M.: Introduction to the Theory of Nested Transactions, *MIT/LCS/TR 367* (1986).
16) 松下 温：図解グループウェア入門，オーム社 (1991).
17) Moss, J. E.: *Nested Transactions: An Approach to Reliable Distributed Computing*, The MIT Press Series in Information Systems (1985).
18) 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).
19) Singhal, M.: Deadlock Detection in Distributed Database Systems, *IEEE Computer*, No. 11, pp. 37-48 (1989).
20) Takizawa, M. and Deen, S. M.: Lock Mode Based Resolution of Uncompensatable Deadlock in Compensating Nested Transaction, *Proc. of the 2nd Far-East Workshop on Future Database Systems*, pp. 168-175 (1992).
21) Weihl, W. E.: Commutativity-based Concurrency Control for Abstract Data Types, *Proc. of the 21th Annual Hawaii International Conf. on System Sciences*, pp. 205-214 (1988).
22) Weihl, W. E.: Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types, *ACM Trans. on Programming Language and Systems*, Vol. 11, No. 2, pp. 249-283 (1989).
23) Weikum, G.: Theoretical Foundation of Multilevel Concurrency Control, *Proc. of the 5th PODS*, pp. 31-42 (1986).
24) Yasuzawa, S., Takizawa, M. and Ouchi, T.: Resolution of Parallel Deadlock by Partial Abortion, *Proc. of the International Symposium on Communications (ISCOM)*, pp. 708-711 (1991).
25) Yasuzawa, S. and Takizawa, M.: Uncompensatable Deadlock in Distributed Object-Oriented Systems, *Proc. of the International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 150-157 (1992).

(平成4年5月11日受付)
(平成5年2月12日採録)

安沢 伸二 (学生会員)

1967年生. 1991年東京電機大学理工学部経営工学科卒業. 1993年東京電機大学大学院理工学研究科システム工学専攻修士課程修了. 同年4月三菱電機株式会社入社. 分散型データベースシステム，分散型システム等に興味を持つ.



滝沢 誠 (正会員)

1950年12月6日生. 1973年東北大学工学部応用物理学科卒業. 1975年東北大学大学院工学研究科応用物理学専攻修士課程修了. 同年(財)日本情報処理開発協会入社. 1986年東京電機大学理工学部経営工学科講師，1987年より同助教授. 工学博士. 1989年9月より1年間ドイツ国立情報処理研究所 (GMD) 客員教授. 1990年7月よりKeele大学 (英国) 客員教授. 分散型データベースシステム，通信網，分散型システム，知識ベースシステム等の研究に従事. 電子情報通信学会，人工知能学会，ACM, IEEE各会員.「知識工学基礎論」オーム社 (共著),「データベースシステム入門技術解説」ソフトリサーチセンタ,「分散システム入門」近代科学社(共著).

S. Misbah Deen

1938年バングラデシュ生まれ. ダッカ大学物理学科と同大学院修士課程 (物理学) を修了後，1985年 Imperial College (英国) より Ph. D (物理学). 1973年 Aberdeen 大学計算機科学科. 1986年より，Keele 大学の計算機科学科教授，および DAKE (Data and Knowledge Engineering) センタ所長. 1990年と1991年に，分散型データベースシステムと協調的知識データベースシステムの国際会議を主催し，データベースシステムと知識ベースシステムの統合化の研究を行っている. 著書, "Fundamentals of Database Systems", "Principles and Practice of Database Systems," 他. IEEE, ACM 各会員.