マルチメディア通信と分散処理　50－5
（1991. 5. 16）

階層型トランザクションの並行実行による デッドロックの解除法

大内　拓磨　　安沢　伸二
東京電機大学大学院
理工学研究科システム工学専攻
ouchi@takilab.k.dendai.ac.jp
yasu@takilab.k.dendai.ac.jp

滝沢　誠
東京電機大学理工学部
経営工学科
taki@takilab.k.dendai.ac.jp

要旨

分散型システムは、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システムである。トランザクションは、オブジェクトを操作する複数の演算の実行系列であり、利用者の原子的な実行単位である。トランザクションは、いくつかの副トランザクションから構成され、副トランザクションも同様に副トランザクションから構成される。この様な形のトランザクションを階層型トランザクションとする。階層型トランザクションでは、ある副トランザクション aによって呼び出される副トランザクション bと cの間に、実行順序についての依存関係が存在しないならば、bと cを並行に実行することができる。本論文では、副トランザクションを並行に実行した場合、従来の直列実行時に生じるデッドロックに加えて、並行実行に生じるデッドロックについて考える。また、デッドロックを解除するために、トランザクションをアボートする必要がある。ここでは、実行された演算の補償演算を用いてアボートする方法を示す。また、トランザクション全体をアボートするのではなく、デッドロックに関係している部分のみをアボートする方法を示す。

Deadlock Resolution in Parallel Execution of Nested Transactions

Takuma OHUCHI, Shinji YASUZAWA, Makoto Takizawa
Dept. of Information and Systems Engineering
Tokyo Denki University
Ishizaka, Hatoyama, Hiki, Saitama 350-03, Japan

Abstruct

A distuributed system is composed of multiple objects interconnected by communication networks. Each object is an abstract data type. That is, each object provides operations for manipulating it. Users write transactions to manipulate objects. Transactions are composed of operations and also atomic units of work for users. Each operation can call operations on another object. Suppose that one operation a on an object o calls two operations b on p and c on q. if b and c are independent, they can be called in parallel. This means that a can be executed in parallel. In this sense, nested transactions can be executed in parallel in the distributed systems. In this paper, we would like to discuss deadlock problems occurred when transactions are executed in parallel. In this paper, we define what kind of deadlock occurs when the transactions are concurrently executed. When the deadlock is detected, one transaction is selected to be aborted. We present how to abort the deadlocked transaction by executing the compensate operations.

1. はじめに

本論文では、抽象データ型[8]としてのオブジェクトが、*SPO*通信網と呼ばれる高信頼な放送通信網[11, 12]によって相互に接続される分散型データベース管理システム*DDBMS/ADT*を考える。利用者は、オブジェクトを、これが提供する操作演算*OP*を用いてのみ操作できる。トランザクション[4]は1つ以上の操作演算の実行系列であり、利用者からみた原子的実行単位である。オブジェクトの演算には、そのオブジェクト固有の演算であるアクションと、いくつかのアクションや他のオブジェクトの演算を呼び出すような演算がある。例えば、オブジェクトoの演算aが、それぞれ別のオブジェクトpとqの演算bとcを呼び出すことが可能である。このために、トランザクションは、階層的[10]に構成される。階層型トランザクションでは、演算を並行実行することで、さらに同時実行性を高めることができる。もし、bとc間に実行順序についての関係が無いならば、bとcを並行に実行することが可能となる。これまでに、階層型トランザクションにおける補償演算を用いた部分アボート[3, 4, 7, 13]および、解除不能なデッドロック[5, 13]の問題について議論されている。本論文では、階層型トランザクション[9, 10]を構成する副トランザクションを並行実行することによって生じるデッドロック問題について考える。

2章では、システムのモデルと補償演算の概念について説明する。3章では、階層型トランザクションを定義する。4章では、並行実行によるデッドロックについて従来の逐次実行との違いについて考察する。5章では、補償演算を用いたデッドロックの解除方法について述べる。6章では、解除不能なデッドロックについて、その解除策について示す。

2. システムモデル

*DDBMS/ADT*は、高信頼な放送通信網*SPO*通信網により個々のオブジェクトが結合された分散データベース管理システムである。*SPO*通信網によって、各オブジェクトはいくつかの宛先のオブジェクトに、送信順序を保ちかつ紛失することがなくメッセージを送ることができる。各オブジェクトoは、それぞれ固有のデータ構造D_oとD_oを操作するための操作演算P_oおよびD_o上の制約I_oを持つ。操作演算には、私用演算と公開演算がある。私用演算は、アクションとも呼ばれ、オブジェクトのデータ構造を直接操作することができる原子演算でありかつ、他のオブジェクトの演算が利用することはない。利用者は、オブジェクトを操作するために、公開演算を用いる。公開演算は、アクションまたは他のオブジェクトの公開演算を呼び出すことが可能である。

オブジェクトのデータ構造Dは、演算OPの実行によって別の状態D'に変化する。各オブジェクトは、個々の演算の結果を補償する補償演算を備えている。オブジェクトoのデータ構造の状態をDとする。Dの状態で演算OPを実行すると、別の状態D'となる。oの状態集合をEとする。E内のDにおいて、$OP(D)$を状態Dで演算OPを適用した状態とする。

[定義2.1]演算OP^{-1}がOPの補償演算であるとは、これらの演算が、同一のオブジェクトoによって提供され、E内の各状態Dについて、$OP^{-1}(OP(D)) = D$である[図1]。□

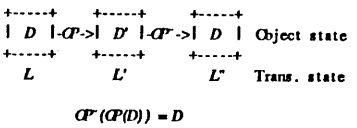

図1 補償演算

例えば、*B*-木オブジェクトにおいて、追加演算*append*で、あるオブジェクトkを追加し、続けて削除演算*delete*でオブジェクトkを削除すれば、追加前と削除後で*B*-木の状態は変わらない。このように、*B*-木オブジェクトでは追加(削除)演算は削除(追加)演算の補償演算である。本論文では、各演算に対して、その補償演算が定義されているものとする。各補償演算はそれぞれの演算の意味によって定義される。

[例1]分散型システムの例として、管理者*manager*、従業員*employee*、秘書*secretary*、会議室*room*、ビデオ*video*などのオブジェクトから構成されるオフィスシステムを考える。各部には、管理者と従業員がいて、管理者には秘書がついている。管理者と従業員はそれぞれ予定表*mgr_sched*と*emp_sched*をデータ構造として持ち、操作演算として予定の予約*Book*と解約*Cancel*、検索*Retrieve*をもつ。会議室とビデオ

には、それぞれ予約表 *room_book* と *video_book* があり、操作演算として予約と解約、検索を備えているものとする。秘書オブジェクトは、管理者と従業員に会議予約演算 *BooK_meeting* と会議解約演算 *Cancel_meeting* を提供し、部員に対して会議の予約を行う。□

3. トランザクション

3.1 トランザクションモデル

階層型トランザクション T はオブジェクトの演算より構成される。各々の演算は、他の演算を呼び出すことが可能であるために、トランザクションは木構造をしている。ここで、根を根トランザクション、葉でない節を副トランザクション、葉をアクションとする。枝は、親の演算が子の演算を呼び出すことを示している。

オブジェクト o の演算 a が、m 個の演算 $\Sigma_a = \{a_1, ..., a_m\}$ を呼び出すとする。ここで、a_i は o のアクションか副トランザクションである ($i = 1, ..., m$)。演算 a によって呼び出される演算間の先行関係を表すために関係 $\rightarrow_a$ を用いる。$a_i \rightarrow_a a_j$ であるならば、演算 a_i がコミットした後に a_j が実行され、a_j は a_i に内部依存しているという。もし、a_i と a_j 間に先行関係が存在しないならば、a_i と a_j を並行に実行できる。

[a と *a]* をそれぞれ副トランザクションの開始と終了とする。*[a* は、o を a のモードでロックする演算である。*a]* は、a が呼び出したすべての演算が終了するのを待ちコミットする演算である (即ち、*AND*-並列である)。ただし、コミットによって a のロックしたオブジェクトは解放されない。*T]* により T 内のすべてのオブジェクトが解放される。a を演算とした時、a によって呼び出されるすべての演算系列 $<[a, a_1, ..., a_m, a]>$ を a の最大拡張系列 $a^*(GRS(a)$ とも書く)、a を系列 $<[a, a_1, ..., a_m, a]>$ の最大縮退系列 $a^{-*}(TRS(a)$ とも書く) とする。また、*[[a]]* によってアクション a の実行を示す。

[例 2] 階層型トランザクションの例として例 1 のオフィスシステムにおいて、管理者 *mgr3* が管理者 *mgr* と *mgr2* と部長会議を行うプログラム $T_{Manager}$ を考える [図 2]。管理者 *mgr3* は自分の予定表に会議の予定を書き込み (ステップ c)、秘書 *sec* に会議の参加者と会議室 *room* の予約を依頼する (ステップ f)。また、a, b と d, e の各ステップはプログラム内でのローカルな処理である。秘書オブジェクトは、管理者 *mgr* と *mgr2* および会議室 *room* に会議予約を行う副トランザクション *Book(mgr)*、*Book(mgr2)*、*Book(room)* を実行する。これらの演算間には実行順序についての先行関係 $\rightarrow_{Book\_meeting}$ が存在しないために並行実行することが出来る。なぜなら、これらは別々のオブジェクトであり、トランザクションのローカル状態を変化させないからである。しかし、*Book(mgr3)* $\rightarrow_{T_{Manager}}$ *Book_meeting(sec)* であるためにこれらの演算には実行順序が存在する。図 3 に部長会議予約トランザクションを順序木によって表す。ここで、r と w はそれぞれ、予定表または予約表の読み込みと書き込みをするアクションである。□

```
T_Manager()
{
    schedule sched;      member_list *mem;
    item_list *item;

    sched->date = "May.16 1991 10:00-12:00";
    sched->theme = "Database Systems";
    mgr3. Book(sched);
    mem = {mgr, mgr2};
    item = {video, room};
    sec. Book_meeting(sec);
}
```

(a) 部長会議予約プログラム

```
class manager
{
    private:
        schedule mgr_sched;
        r(shedule mgr_sched){ read(mgr_sched); }
        w(shedule mgr_sched){ write(mgr_sched); }
    public:
        Boolean  Book(shedule mgr_sched)
        {
            if (retrieve(mgr_sched) != Free)
                    return(False);
            w(mgr_sched);
            return(True);
        }
        ...
};
class secretary
{
    ...
    public:
        Boolean  Book_meeting(schedule sched,
                    member_list member, item_list item)
        {
            Boolean state;

            // member,item のオブジェクトにメッセージを送る。
            // [[ ... ]] 内の演算を並行実行する。
            [[member.Book(sched); item.Book(sched)]]
            wait(state);      // 並行実行した演算の終了待ち。
            if (state != True) return(false)
                else return(True);
        }
        ...
};
```

(b) オフィスシステムのオブジェクト

図 2 会議予約トランザクション

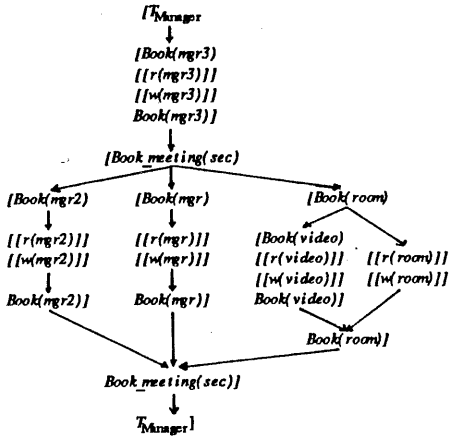

図3 階層型トランザクション

3.2 同期機構

複数のトランザクションがオブジェクトを操作するとき、システムの一致性を保つための同期機構が必要となる。本論文では、オブジェクトの演算間の伴立性をもとにした同期機構を用いる。各演算 aは、演算を実行する前に oをモード $mode(a)$でロックする。

[定義3.1]a、bを演算、oをオブジェクトとする。m_a、m_bを $mode(a)$、$mode(b)$とする。m_aが m_bと伴立であるとは、bが oを m_bでロックしているときに、aが m_aで oをロックできることである。また、bを所有者、aを使用者の演算と呼ぶ。m_aが m_bと非伴立であるとは、m_aが m_bと伴立でないことである。このとき aは要求者である。□

$MODE_o$をオブジェクト o上の演算によって要求されるロックのモードの集合とする。$MODE_o$は、モード間の半順序関係 $\subseteq$について束となる。以下の二つの条件を満足するときモード m_aは m_bよりも排他的ではない($m_a \subseteq m_b$)[6]とする。

(1) m_bが m_cと伴立であるならば、m_aは m_cと伴立である。

(2) m_cが m_bと伴立であるならば、m_cは m_aと伴立である。また、$\cup$を $\subseteq$上の最小上界とする。

次に、階層型トランザクション Tの実行方式を示す。$Lock(a)$を、演算 aがコミットした後に aがロックしているオブジェクトの集合、$Omode(o)$を oロックしている演算のロックモードの集合とする。

[実行スキーマ]

(1) aが根トランザクションであるならば、$Lock(a) := \phi$とする。

(2) aが副トランザクションで、かつ oが他の演算にロックされていないならば、oを $mode(a)$でロックし、$Omode(o) := \{mode(a)\}$とする。また、トランザクション Tの現在のローカル状態 L_aをトランザクションスタック $TS(T)$に積む。

(3) aが副トランザクションで、かつ oが既に他の演算によってロックされているならば、

(3-1) $Omode(o)$内のすべてのモードと伴立であるならば、aは oのロックを獲得し、$Omode(o) := Omode(o) \cup \{mode(a)\}$とする。また、$T$の現在のローカル状態 L_aを $TS(T)$に積む。

(3-2) $Omode(o)$内のあるモードと伴立でないならば、(3-1)が満たされるまで、ブロックする。

(4) aが直接呼び出した $a_1,...,a_m$がコミットした時、

(4-1) aが根トランザクションならば、$Lock(a)$内のすべてのロックを解除する。

(4-2) aが根トランザクションでないならば、$Lock(a) := Lock(a) \cup Lock(a_1) \cup ... \cup Lock(a_m)$、$TS(T)$からローカル状態 L_aを取り出す。□

すべてのロックは、根トランザクションがコミットした時に解放されるので、すべてのロックは2相ロック方式[2]である。よって、これから作られるログは、直列可能で、連鎖的アボート[1]は起きない。

4. デッドロック

トランザクションの同期機構としてロック機構を用いる場合、デッドロックが発生することがある。ある演算がいくつかの演算を並行に実行する様な場合、従来とは異なった状態においてデッドロックが生じることがある。本章では、トランザクションが副トランザクションを並行実行した場合に生じるデッドロックについて論じる。

4.1 従来のデッドロック

はじめに、従来のデッドロックとして、トランザクションを逐次実行している場合に生じるデッドロックを考える。デッドロックを定義するために、依存関係を定義する。

[定義4.1] a、bをオブジェクトoの演算とし、bはoをロックしているとする。このとき、aがbに直接的に依存するとは、$mode(a)$が$mode(b)$と非併立であることである。□

[定義4.2] 次のいずれかの条件を満足するならば、aは、bに依存している ($a \Rightarrow b$) という。
(1) aが、bに直接的に依存している。
(2) あるトランザクションT内においてbがaに先行 ($b \rightarrow_T a$) して実行されている。
(3) $a \Rightarrow c \Rightarrow b$なる演算$c$が存在する。□

システムの状態を拡張待ち (EWF) グラフGによって表す。EWFグラフの各節点は演算を、各有向辺が演算間の依存関係を示す有向グラフである。
EWFグラフGが巡回閉路を含むならば、システムはデッドロック状態である。

[定義4.3] 演算aが直接デッドロックであるとは、演算aが自分自身に依存する ($a \Rightarrow a$) ことである。また、演算aがデッドロック状態であるとは次のいずれかのときである。
(1) 演算aが直接デッドロックしている。
(2) 演算aがデッドロックしている演算bに依存している。□

[例3] トランザクションT_1の演算aをa^{T1}で表す。図4の2つのトランザクションの実行状態で、演算の依存関係 ($[b^{T1} \Rightarrow [b^{T2} \Rightarrow [a^{T2} \Rightarrow [a^{T1} \Rightarrow [b^{T1}$) を表す$EWF$グラフが巡回閉路を含んでいる。よって、図4はデッドロック状態である。□

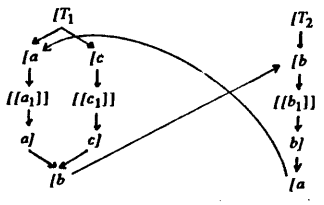

図4 デッドロック状態のEWFグラフ

4.2 並行実行でのデッドロック

[例4] 図5のある部の会議予約トランザクション$T_{Meeting}$と、図2の部長会議予約トランザクション$T_{Manager}$を考える。この2つが同時に実行され、図5の状態にあると仮定する。演算間の依存関係$\Rightarrow$は、$[Book(mgr)^{Manager} \Rightarrow [Book(mgr)^{Meeting} \Rightarrow [Book(room)^{Meeting} \Rightarrow [Book(room)^{Manager}$となり、$[Book(mgr)^{Manager}$から$[Book(room)^{Manager}$には依存関係が存在する。しかし、$[Book(room)^{Manager}$から$[Book(mgr)^{Manager}$には依存関係が存在しないために、$EWF$グラフに巡回閉路が生じない。定義4.3のデッドロックの定義ではデッドロックでは無いことになる。しかしながら、この2つのトランザクションはデッドロック状態である。$[Book(room)^{Manager}$と$[Book(mgr)^{Manager}$は$[BooK_meeting(sec)$によって並行に呼び出された演算であり、AND並行の性質から$[BooK_meeting(sec)$によって並行に呼び出されたすべての演算が終了しなければ、この演算はコミットすることができない。よって、デッドロック状態である。□

例4で示したデッドロックを定義するために、定義4.2に次の条件を加えることにする。

[定義4.4] aとbを、それぞれ別々のトランザクションの演算とする。aはa_1とa_2の先祖の演算でかつa_1とa_2の間には$\rightarrow_a$関係が無いものとする (即ち、a_1とa_2は並行に実行できる)。この時、a_1、a_2、bの間に、実行によって、依存関係$a_1 \Rightarrow b \Rightarrow a_2$が存在する時、、$a_2$は$a_1$に依存 ($a_2 \Rightarrow a_1$) する。□

定義4.4から、$T_{Manager}$は実行により$[Book(mgr) \Rightarrow [Book(room)$の依存関係を生じる。これらの演算は$[Book_meeting(sec)$により並行に呼ばれた演算であるので、$[Book(room) \Rightarrow [Book\ (mgr)$の依存関係を生じ、$EWF$グラフに$[Book(mgr) \Rightarrow [Book(mgr)$の巡回閉路を持つ。即ち、図5はデッドロック状態である。

$Current(T)$をトランザクションTの現在実行中である演算集合、$Waited(T)$をT内の演算で直接デッドロック状態である演算集合とする。例えば、図5の会議予約トランザクションでは、$Current(T_{Meeting})$ $= \{Book\_meeting(sec2)], [Book(room)\}$、$Waited($

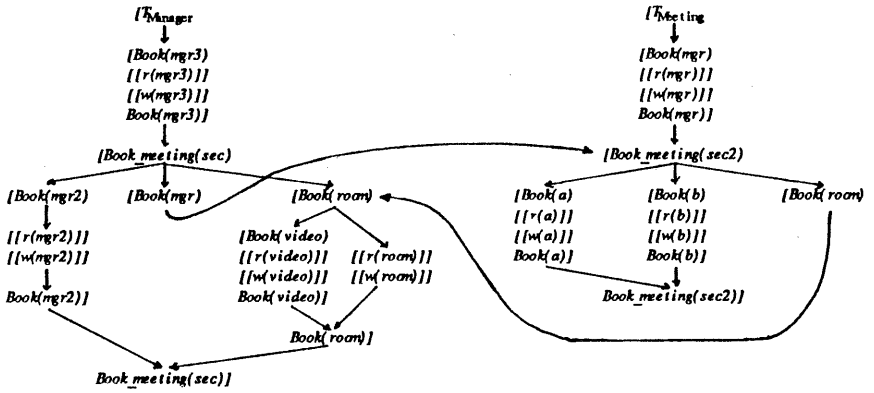

図5 並行実行におけるデッドロック

$T_{Meeting}$) = {<[Book(mgr), ..., [Book(room)>} である。部長会議予約トランザクションでは、*Current*($T_{Manager}$) = {Book_meeting(sec)], [Book(mgr)}、*Waited*($T_{Manager}$) = {<[Book(room), [Book(mgr)>} である。

5. 部分的アボート

デッドロックが生じると、デッドロック状態のトランザクションの中から1つを選択し、アボートすることで、デッドロックを解除する。従来の方法では、デッドロックを解除するために、コミットしたすべての演算をアボートしていたが、本論文では、アボートにかかるコストが大きくなるために、デッドロック解除に必要な演算のみを補償演算 [3, 13] を用いてアボートする部分的アボートを提案する。ある演算 aが m個の演算 $\{a_1, ..., a_m\}$ を呼んだとする。すべての演算が補償演算を備えていると仮定しているので、演算 aの実行した結果を補償するためには複数の方法が考えられる。1つの方法は、aの補償演算 a^{-1}を実行する (a^{-1}は n個の演算 $\{a_1, ..., a_n\}$ を実行する) 方法がある。別の方法として、演算 aが呼びだした m個の演算の補償演算 $\{a_1^{-1}, ..., a_m^{-1}\}$ を実行時とは逆の順序で実行する方法がある。前者を最大縮退 *TRS*系列による補償という。後者を最大拡張 *GRS*系列による補償という。 補償演算も通常の演算と同様で、実行のためにオブジェクトのロックを必要とする。副トランザクションの開始演算 $[a$の補償演算 $[a^{-1}$は、a^{-1}が呼び出したすべての補償演算を待ち合わせる。終了演算 $a]$の補償演算 $a]^{-1}$は、オブジェクトを $mode(a^{-1})$でロックする。補償演算 a^{-1}と演算 aがロックしたオブジェクトは、補償演算がすべて実行されたときに解放される。両者の補償演算の違いは、補償演算が使用するオブジェクトに違いがある。 *TRS*による補償を実行するためには、システムが安全 [5, 13] でなければならない。システムが安全でないならば補償演算を実行することによって、解除不能なデッドロックが生じることがある [5, 13]。本論文では、デッドロックはデッドロックマネージャによって検出されると仮定する。

5.1 並行アボート

デッドロックを解除するために、コミットした演算をアボートする。もし、補償演算間に先行関係が存在しないならば、通常の演算同様に補償演算も並行に実行することが可能である。補償演算は、オブジェクトの状態を復旧することができるが、トランザクションがオブジェクトを操作するために使用した作業領域の状態 (ローカル変数) を元に戻すことができない。アボートした後に再実行をするためには、トランザクションのローカル状態も復旧しなければならない。例えば、口座 aから bに金額 cを振り込むトランザクション T_1 を考える [図6]。ここで、a, ..., iは逐次実行され、ローカルな変数 xを使用するとする (aを実行する以前の xの状態を L_a で表わすことにする)。jを現在実行中の演算、eが他のトランザク

ションの演算と直接的に依存している演算とする。デッドロックを解除するために新たに補償トランザクション $T_1^{-1} = <j^{-1}, i^{-1}, ..., e^{-1}>$ が実行され、オブジェクトの状態が復旧される。Tを eから再実行するためには、ローカル状態が L_eでなければならない。しかし、すでに Tのローカル状態は L_eとなっているために、cから再実行することができない(xの状態が異なってしまったために、再実行すると矛盾が生じる)。この問題を解決するために、本論文では、関数コールに用いるスタックと同様の方法を用いる。即ち、トランザクションの実行過程として、副トランザクションを開始する以前のローカル状態をトランザクションスタックに退避し、副トランザクションが終了した時にローカル状態を取り出し実行を続けることにする。よって、演算 aが呼び出した演算のすべてを補償演算によってアボートし、トランザクションスタックから aを実行する以前のローカル状態 L_aを回復することで、aよりトランザクション Tを再実行することができる。

演算 aによって呼ばれる演算のすべてが並行に実行される副トランザクションであるとする[図6]。ここで、演算 bが他のトランザクションの演算と直接的に依存している演算であるとする。逐次実行システムではデッドロックを解除するためにすべての演算をアボートしなければならない。しかし、並行実行システムでは、各演算のローカル状態は各オブジェクト毎に保存される。このために、他のトランザクションの演算と直接的に依存している演算を含む部分木だけをアボートすることでオブジェクトの状態とトランザクションの状態を復旧することができる。よって、演算 bを含む部分木の演算を補償するだけでデッドロックを解除でき、bを再実行することができる。

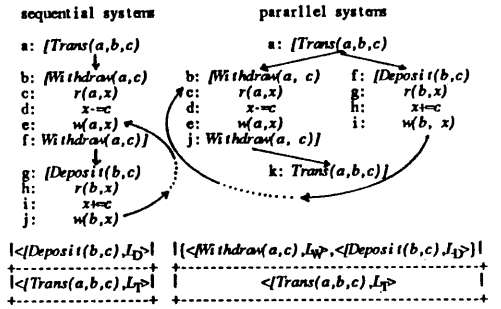

図6 デッドロックとローカル状態

[定義5.1]*Pabort(T)*を次のいずれかの条件により与えられる演算 pの集合とする。

(1) *Waited(T)*内の演算の依存関係が全順序関係 $\Rightarrow$ を持つならば、最初の演算を含む部分木の根の演算 pである。

(2) *Waited(T)*内の演算の依存関係が定義4.4によって定義されるならば、使用者である演算の中で他のトランザクションの演算と直接的に依存している演算を含む部分木の根の演算 pである。

□

デッドロックを解除するためには、*Pabort(T)*内の演算を根とする部分木を補償トランザクションによってアボートする。本システムでは、デッドロックマネージャがデッドロックを検出すると、デッドロックマネージャはアボートされるトランザクション Tを決定し、*Abort(Pabort(T))*を放送する。オブジェクトは、*Abort*を受信したとき次のように振る舞う。

(1) オブジェクト oは、ログ L_oを調べ *Pabort(T)*内の演算 *OP'*(オブジェクト *o'*の演算)によって推移的に呼ばれた演算 *OP*を持つかどうかを調べる。

(1-1) 現在実行中の演算 *OP*を L_o内に持つならば、*OP*の実行を中止して親の演算 *OP"*(オブジェクト *o"*の演算)に対して補償演算 *OP"*$^{-1}$を実行するように *Compensate(o", OP")*を *o"*に送る。

(1-2) 演算 *OP*が *Pabort(T)*内の演算であるならば、*OP*の実行を中止し、*Compensate*を自分自身に送る。

(1-3) 演算 *OP*が L_o内でコミットしているならば、*OP*$^{-1}$を実行して親の演算 *OP"*(オブジェクト *o"*の演算)に対して補償演算 *OP"*$^{-1}$を実行するように *Compensate(o", OP")*を *o"*に送る。

(2) *Compensate(o", OP")*を受け取ったオブジェクト *o"*は、演算 *OP"*を補償演算 *OP"*$^{-1}$を実行する。もし、演算 *OP"*が複数の並行実行された演算を持つならば、並行実行したすべての演算から *Compensate(o", OP")*を受けてから *OP"*$^{-1}$を実行する。さらに、親の演算に対して *Compensate*を送る。もし、*OP"*が *OP'*であるならば、即ち *Pabort(T)*内の演算であるならば、

補償演算 OP^{-1}を実行した後で、演算 OPと補償演算 OP^{-1}によってロックされたすべてのオブジェクトを解放する。□

[例5] 例4のデッドロック状態のトランザクションを再び考える。デッドロックマネージャによってデッドロックが検出されアボートするトランザクションとして部長会議予約トランザクション $T_{Manager}$ が選択され、$Pabort(T_{Manager}) = \{[Book(room)\}$ が放送されるとする。*room*は、*[Book(room)*がすでにコミットしているので、補償演算 $Book(room)^{-1}$を実行する。自分自身に *Compensate(room, Book(room))*を送り、ロックを解放する。□

6. 解除不能なデッドロック

補償トランザクションも通常のトランザクションと同様にオブジェクトをロックする必要がある。もし、補償トランザクションが今までに獲得したオブジェクト以外のオブジェクトにロックを要求するとデッドロックが新たに生じてしまう。この様に、補償演算を実行することによって発生するデッドロックを解除不能なデッドロック [5, 13] とする。また、解除不可能なデッドロックを起こすようなシステムを非安全なシステム [5, 13] という。解除不可能なデッドロックが生じた場合、通常の補償演算では解除することができないために、従来のログによる方法と同様な *TRS*による補償演算を実行することでデッドロックを解除しなければならない。*TRS*の補償演算は、新たなオブジェクトを要求しないので、デッドロックが生じることはない。

7. まとめ

本論文では、抽象データ型としてのオブジェクトから構成される分散システムにおいて、オブジェクトを操作するためのトランザクションを並行実行する時に生じる問題としてデッドロックを考え、その解除方法を示した。階層型トランザクションでは、部分木をアボートすることでデッドロックを解除することができ、従来のようなトランザクションすべてをアボートする方法と比べコストを減少できる。部分的なアボートを実現するために、本論文で補償演算を提案した。今後の課題として補償演算を実装したシステムを作り従来のアボートを用いた方法との比較検討を行うことである。

参考文献

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

[2] 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, 1967, pp.624-637.

[3] 大内 拓磨, 滝沢 誠, "入れ子型トランザクションの部分的アボート," データベースワークショップ論文集, 1991, pp.132-138.

[4] Garcia-Molina, H., and Salem, K., "SAGAS," *ACM SIGMOD*, 1987, pp.249-259.

[5] 兵頭 章子, 大内 拓磨, 滝沢誠, "階層型トランザクションの解除不能なデッドロック," データベース研究会 *82-2*, 1991.

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

[7] Korth, F. H., Levy, E., and Silberschatz, A., "A Formal Approach to Recovery by Compensating Transactions," *Porc. of the 16th VLDB Conf.* 1990, pp.95-106.

[8] Liskov, B. H. and Zilles, S. N., "Specification techniques for data abstractions," *IEEE Trans. on Software Engineering*, Vol.1, 1975, pp.294-306.

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

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

[11] Nakamura, A. and Takizawa, M., "Reliable Broadcast Protocol for Selectively Partially Ordering PDUs(SPO protocol)," *Proc. of the IEEE 11th ICDCS*, 1991.

[12] Nakamura, A. and Takizawa, M., "Design of Reliable Broadcast Protocol for Selectively Partially Ordering PDUs," *Proc. of IEEE COMPSAC91*, 1991.

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