データベースシステム 92－3
（1993. 3. 22）

データの状態に基づく並行処理制御の正当性

徐 海燕[†] 古川 哲也[††] 史 一華[†††]

[†]福岡工業大学 [††]九州大学 [†††]福岡工業短期大学

直列可能性に基づく従来の並行処理制御方式は，高水準データベース(DB)には適用できなくなっていることが広く認識されてきた．この問題に対処するため，本論文では並行処理制御の正当性基準という問題の根本から見直しを行う．従来のDBにおいて，各々直列に実行される処理単位がDBの一貫性を保持すると仮定する方法で一貫性制御を行ってきたため，直列可能である並行実行のみが正当となっていた．本論文では，設計DBにおいて，データの状態を利用することにより，統一的な判定基準でDBの一貫性制御を行えることを指摘する．さらに，設計DBにおける並行処理制御の正当性基準を定め，直列可能性を要求しなくてもよいことを示す．

Correctness of Concurrency Control Utilizing Data Status

Haiyan XU[†], Tetsuya FURUKAWA[††], Yihua SHI [†††]

[†]Fukuoka Institute of Technology, [††]Kyushu University, [†††] Fukuoka Junior College of Technology

The concept of serializability is central to all the concurrency control mechanisms of conventional database systems. Since they are inadequate for advanced database systems, we reconsider the correctness of concurrency control essentially. By associating a status with each data, we show the consistency of design databases can be validated by the integrity constraints. After handling the consistency problem by the new method, we establish the correctness of concurrency control for design database systems, which do permit nonserializable executions.

1 まえがき

並行処理制御の研究は，並行処理の正当性はスケジュールが直列可能であるということを前提にしてなされてきた．しかし，計算機技術の進歩により，様々な分野でデータベース (DB) が利用されるようになり，応用分野によっては，長時間の処理を含むような場合などに直列可能性を満足しなければならないことによる問題点が指摘されている[1]．本論文では，DB の一貫性を統一的な判定基準で管理することにより、直列可能でない並行処理スケジュールも正当であると考えてよいことを，設計 DB における並行処理制御を例にして示す．並行処理の正当な範囲を直列可能性から広げることにより，処理の並行性を増し，指摘されている問題点を解決することができる．

変更操作を含む処理単位は，DB の一貫性を保持しなければならない．DB が一貫しているかどうかを判断するのに，通常，次の 2 つの方法が考えられる．

(1) 統一的な判定基準を定める方法．

(2) 処理単位が正しいものと仮定する方法．即ち，一貫した DB から単独実行された処理単位は，DB を他の一貫した状態に写像すると仮定する．

従来の事務 DB においては，統一的な判定基準を定めにくい[1]ことと，処理単位が短いという性質により方法 (1) によるオーバヘッドをなるべく避ける必要のある[4]ことから，方法 (2) が用いられてきた．即ち，各々直列に実行された処理単位が DB の一貫性を保持できるなら，直列可能である (実行結果がある順序での直列実行と等価となる) 並行実行も必ず DB の一貫性を保持できる．

しかし，長大な処理単位を対象とする設計 DB[2, 5, 9] に対しては，直列可能性という並行処理制御の正当性基準を適用することは困難である．これは，長大な処理単位に対しても直列可能である並行実行しか許可できなければ，並行性の低下もしくは後退復帰の増加を招くことになる．共同作業や対話方式などの特徴を持つ設計作業においては，これらのどちらの結果とも利用者にとっては受け入れにくいである．この問題は広く認識されており，様々な角度から研究されてきている[1, 3, 6, 7, 8]．しかし，並行処理制御の正当性とは何か，直列可能性を要求しなくても正当な並行処理制御を行えるかどうかというレベルまで研究されているものはまだ知られていない．

本論文では，設計作業の特徴を分析することにより，設計 DB における一貫性制約を，設計規則や設計手順によって定義される状態の変化に対する状態制約とすべきであることを指摘する．次に，このように定義された状態制約が設計 DB が一貫しているかどうかを判断するための統一的な基準となる特徴を利用することにより，方法 (1) を用いた場合における並行処理制御の正当性基準を定める．方法 (1) を用いて DB が一貫しているかどうかを判断できれば，直列可能性を要求しなくても正当な並行処理制御を行えることを示す．さらに，正当な並行処理制御を行える具体的な非直列可能制御方式を提案し，DB の一貫性を保持するためのオーバヘッドについても議論する．

本論文は，次のように構成される．2 節では，設計作業の特徴について議論し，状態制約という設計 DB における新たな一貫性制約の概念を提案する．さらに，状態制約は設計 DB が一貫しているかどうかを判断するための統一的な基準であることを指摘する．3 節では，状態制約という統一的な判定基準を用いた場合における並行処理制御の正当性基準を定め，正当なスケジュールは必ずしも直列可能とは限らない結果を示す．4 節では，直列実行における状態制約の保持方法を検討する．5 節では，4 節の状態制約の保持方法を利用することにより，正当な並行処理制御を行える非直列可能制御方式を提案する．6 節では，従来の研究との比較である．従来の研究により許可される非直列可能である並行実行は，正当ならば必ず本論文で提案する方式によっても許可できることを示す．7 節では，全体のまとめである．

2 統一的な判定基準：状態制約

DB で大量のデータを統一的に管理するために，実世界の事物の値を正しく表現していることを保証するための条件を一貫性制約として表す．ただし，設計 DB における一貫性制約は，設計作業の次のような特徴を考慮したものでなければならない．

(1) 異なる設計段階で生成される様々な中間結果．

(2) 設計作業の従っている設計規則や手順．

本論文では，設計データを実体とその状態で記述することにより特徴 (1)，設計規則や設計手順を設計 DB における一貫性制約とすることにより特徴 (2) に対処する．

定義 1 設計 DB は $< U, R, M >$ の 3 元組である．

- 実体の集合: $U = \{o_i \mid o_i$ は実体である $\}$
- 関連の集合:

$$R = \left\{ r_m(o_i, o_j) \;\middle|\; \begin{array}{l} o_i から o_j への種類 r_m \\ の関連が存在する \end{array} \right\}$$

- 状態の集合:

$$M = \left\{ test_k(o_i) \;\middle|\; \begin{array}{l} o_i が k 番目のテストを通過 \\ ただし,\ k \in \{1, \cdots, n\} \end{array} \right\}$$

□

議論を簡単にするため，テストの種類を n 以内としている．また，否定情報の取り扱いは閉世界仮説を利用する．即ち，$test_k(o_i) \notin M$ ならば，o_i は k 番目のテストを通過していない．さらに，全ての関連を2項関連で表す．n 項関連も変数を導入することにより，2項関連で表せるため，一般性は失われない．

本論文では状態を明記しているので，設計規則や手順は，状態と関連を素論理式とする関数記号なしのホーン節によって表現できる．

定義 2 関数記号なしのホーン節とは，次の形をした表現である．

$$B \leftarrow A_1, \cdots, A_n$$

ここで，$B, A_1, \cdots, A_n$ は素論理式 (atomic fomula) であり，$n \geq 0$ である．素論理式 $A_1, \cdots, A_n$ はこのホーン節のボディといい，B をヘッドという．素論理式とは，次の形をした表現である．

$$p(t_1, \cdots, t_m)$$

ここで，p が m 変数の述語記号，$t_1, \cdots, t_m$ は変数，あるいは定数記号である． □

以下では，関数記号なしのホーン節を簡単にホーン節と呼ぶ．ホーン節中のボディもヘッドも持つ節を，規則という．また，$B \leftarrow$ のような節を簡単に B で書き，事実といい，$\leftarrow A_1, \cdots, A_n$ $(n > 0)$ のような節を NG 節 (nogood clause) という．

例えば，$\leftarrow test_i(x), test_j(x)$ という NG 節は，$test_i$ と $test_j$ が同時に通過できないテストであることを表現している．また，実体 x が単体テストを通過しなければ，x を含む実体集合 z が同種類の複合テストを通過できない制約は，$test_k(x) \leftarrow test_k(z), in(x, z)(k = 1, 2, \cdots, n)$ という規則で表すことができる (z が x を含む関連を $in(x, z)$ とする)．1番のテストを通過しなければ，2,3番目のテストを通過できない制約は，$test_1(x) \leftarrow test_2(x)$; $test_1(x) \leftarrow test_3(x)$ のような規則で表現できる．

定義 3 設計 DB $< U, R, M >$ における状態制約は，次のような実体と状態，状態と状態間の一貫性を要求する制約である．

- 実体・状態間制約：各 $test_k(o_i) \in M$ が現時点での $o_i (o_i \in U)$ の状態を記録しているものであることを要求する．
- 状態間制約：M 内の状態間の満たすべき性質を要求する．NG 節と規則の集合を用いて定義される． □

即ち，設計手順のような動的な状態間制約は規則，静的な状態間制約は NG 節によって表現される．ただし，規則の集合は，既存の事実から新しい事実を推論するのではなく，既存の事実間の満たすべき性質を要求している．設計 DB $< U, R, M >$ が状態間制約 P を満たすとは，$M \cup R$ の下で P が真であることをいう．これを $M \cup R \models P$ で表す．設計作業に必要な設計規則や手順を状態制約として定義することにより，設計 DB $< U, R, M >$ が一貫していることを，状態制約を満たしていることで表現できる．

状態間制約は，関連する実体の状態間の満たすべき性質と，同一実体の異なる種類のテスト結果間の満たすべき性質を要求するものに分類できる．このような考えに基づいて，本論文では，状態間制約に用いられる NG 節と規則の形式をそれぞれ次のように限定する．

- NG 節：$\leftarrow test_i(x), test_j(y), r_m(x, y)$.
  $\leftarrow test_i(x), test_j(x)$.
- 規則：$test_i(x) \leftarrow test_j(y), r_m(x, y)$.
  $test_i(x) \leftarrow test_j(x)$.

述語集合を，DB で定義された状態間制約 P で関連する部分集合に分類するため，次のような記号を定義しておく．

$pred(A)$: 素論理式 A の述語記号．

$PS(P)$: P に現われている述語記号の全体の集合．

Rel: $PS(P)$ 上の次のような二項関係．

$$Rel = \left\{ (q_i, p) \;\middle|\; \begin{array}{l} B \leftarrow A_1, \cdots, A_m \in P, p = pred(B), \\ q_i = pred(A_i), 1 \leq i \leq m \end{array} \right\}$$

$$\cup \left\{ (q_i, q_j) \;\middle|\; \begin{array}{l} \leftarrow A_1, \cdots, A_m \in P, q_i = pred(A_i), \\ q_j = pred(A_j), 1 \leq i, j \leq m \end{array} \right\}$$

$C_b(q)$: 述語記号 q がボディに現れる P 中のホーン節の集合.

$C_h(q)$: 述語記号 q がヘッドに現れる P 中のホーン節の集合.

定義 4 P が状態間制約とする. $p \in PS(P)$ に対して, $IS(p)$ と $IS^r(p)$ を, 次のように定義する.

- $IS(p) = \{q \mid (p,q) \in Rel^*\}$: p が影響する述語集合.
- $IS^r(p) = \{q \mid (p,q) \in (Rel^*)^r\}$: p を影響する述語集合.

ただし, Rel^* は関係 Rel の反射的推移閉包で, $(Rel^*)^r$ が Rel^* の逆方向の関連である. □

例 1 次のような状態間制約 P

$$P = \left\{ \begin{array}{l} test_k(x) \leftarrow test_k(z), in(x,z) (k = 1, 2, \cdots, n) \\ test_1(x) \leftarrow test_2(x), \\ test_1(x) \leftarrow test_3(x) \end{array} \right\}$$

に対して, IS と IS^r 関数が次のようになる.

$IS(test_1) = \{test_1\}$,
$IS(in) = \{test_1, test_2, test_3, in\}$,
$IS(test_2) = \{test_1, test_2\}$,
$IS(test_3) = \{test_1, test_3\}$,
$IS^r(test_1) = \{test_1, test_2, test_3, in\}$,
$IS^r(in) = \{in\}$,
$IS^r(test_2) = \{test_2, in\}$,
$IS^r(test_3) = \{test_3, in\}$ □

事実 l_k が追加／削除される場合に, 状態間制約 P 中に恒真性が影響されうるホーン節の集合は $C_b(u)$ / $C_h(v)$ である. ただし, $u \in IS(pred(l_k))$ /$v \in IS^r(pred(l_k))$. 以下では, 設計 DB $< U, R, M >$ に対して, 次の 3 つの関数を用いて状態制約による関連を表す (o_j が o_i を含む関連を $in(o_i, o_j) \in R$ で表すとする).

- $g(l_k) = \{l_j \mid pred(l_j) \in IS(pred(l_k)) \cup IS^r(pred(l_k))\}$: M から M の部分集合の集合への関数であり, $l_k \in M$ が状態間制約で関連する M 内の事実を表す.

  $IS \cup IS^r$ は $Rel^* \cup (Rel^*)^r$ という反射・推移・対称律を満たす関係から定義されるため, この分類方法によって M が複数個の素である部分集合に分割される.

- $f_u(o_i) = \{test_k(o_j) \mid o_j = o_i \vee in(o_i, o_j), k \in \{1, \cdots, n\}\}$: U から M の部分集合の集合への関数であり, $o_i \in U$ と実体・状態間制約で関連する M 内の事実集合を表す.

- $f_m(test_k(o_j)) = \{o_i \mid in(o_i, o_j) \vee o_i = o_j\}$: M から U の部分集合の集合への関数であり, $test_k(o_j) \in M$ と実体・状態間制約で関連する U 内の実体集合を表す.

従って, 事実 l_k が追加／削除された場合に, l_k / $\neg l_k$ が $g(l_k)$ 内の事実と状態間制約を満たすならば, 変更された DB も状態間制約を満たす. また, 実体・状態間制約の定義により, l_k / $\neg l_k$ が $f_m(l_k)$ 内の実体や実体集合と実体・状態間制約を満たせば, 変更された DB も実体・状態間制約を満たす. 同様に, o_i が変更される場合に, o_i が $f_u(o_i)$ 内の事実と実体・状態間制約を満たせば, 変更された DB も実体・状態間制約を満たす.

3 統一的な判定基準の下での並行処理制御の正当性

本節では, 状態制約という設計 DB が一貫しているどうかを判断できる統一的な基準を用いた場合に, 並行処理制御の正当性基準を定める. さらに, その正当性の性質を分析する.

定義 5 設計 DB $< U, R, M >$ に対する基本操作を, 次のものとする (関連 $r_m(o_i, o_j)$ に関する基本操作を明確に検討しないことにする).

- $r(o_i)$: 実体 $o_i \in U$ を読み出す.
- $w(o_i)$: 実体 $o_i \in U$ を書き変える.
- $r(l_k)$: 事実 $l_k \in M$ を読み出す.
- $ins(l_k)$: 事実 l_k を M に追加する.
- $del(l_k)$: 事実 l_k を M から削除する. □

本論文では，$r(o_i)$ と $w(o_i)$，$w(o_i)$ と $w(o_i)$，$r(l_k)$ と $ins(l_k)$，$r(l_k)$ と $del(l_k)$，$ins(l_k)$ と $del(l_k)$ を競合する基本操作とする．$ins(l_k)$ と $del(l_k)$ とも l_k を変更する操作であるが，意味論によりそれらは競合しないことが分かる．

定義 6 設計 DB $< U, R, M >$ における処理単位 T_i は，次にような部分順序 $<_i$ を持つ操作系列である．

(1) $T_i \subseteq \{r_i(o_j), w_i(o_j), r_i(l_k), ins_i(l_k), del_i(l_k) \mid o_j \in U, l_k \in M\}$,

(2) 任意の $s,t \in T_i$(s と t が競合する基本操作) に対して，$s <_i t$ または $t <_i s$ が成り立つ．

ここで，(1) は処理単位内の操作を定義しており，(2) は部分順序 $<_i$ の条件である． □

本論文では，回復制御 (Recovery control) について考慮しないため，T_i に対する abort/commit 操作の議論を省略している．複数個の処理単位が並行に実行されるスケジュールとは，各処理単位内の実行順序を保持した基本操作の系列である．

定義 7 $T = \{T_1, T_2, \cdots, T_n\}$ を処理単位の集合とすると，T 上のスケジュール H は，下記のような部分順序 $<_H$ を持つ操作系列である．

(1) $H = \cup_{i=1}^{n} T_i$;

(2) $<_H \supseteq \cup_{i=1}^{n} <_i$;

(3) 任意の $s,t \in H$(s と t が競合する基本操作) に対して，$s <_H t$ または $t <_H s$ が成り立つ． □

並行実行にされる処理単位が，互いに影響し合わないことを保証するために，論理性，検索一貫性と結果一貫性という概念が知られている．従来の DB においては，処理単位が DB の一貫性を保持すると仮定する方法で一貫性制御を行っていたため，直列可能に実行される処理単位が検索一貫性と結果一貫性を満たすものとなる．また，論理性は処理単位内の初期操作結果が同一処理単位内のその後の操作に利用できることを要求する (即ち，T_i が複数回に同一のデータを操作する間に，そのデータが他の処理単位によって変更されないことを要求する) ので，直列可能に実行される処理単位は，必ず論理性を満たす．しかし，上記のように，設計 DB においては統一的な判断基準で DB の一貫性を制御できるので，必ずしも直列可能性に頼る必要はなくなる．また，論理性も状態制約を利用することによって，より厳密な (厳しい) 解釈を与えることができる．

定義 8 $T = \{T_1, T_2, \cdots, T_n\}$ 上のスケジュール H において，各 $T_j \in T$ の論理性，検索一貫性と結果一貫性を満たすとは，次のようなことを意味する．

- 論理性：T_j が状態制約で関連する実体や状態を操作している間に，その間の一貫性を破壊する他の処理単位による変更操作が禁止される．即ち，
  - (a) $\{r_j(o_i), w_j(o_i), r_j(l_k), ins_j(l_k) \mid l_k \in f_u(o_i)\}$ 内の任意の 1 つの操作が始まってから，その中の全ての操作が終了するまで，o_i は他の処理単位によって変更されてはならない．
  - (b) $l_k \in M$ が T_j によって操作されてから，T_j が $\forall l_s \in g(l_k)$ を操作終了するまで，l_k は他の処理単位によって変更されてはならない．
- 検索一貫性：$\{o_i \mid r_j(o_i)\} \cup \{l_k \mid r_j(l_k)\}$ は状態制約を満たす．
- 結果一貫性：状態制約で関連する各々の部分集合において，T_j の操作結果は状態制約を満たす：
  - (a) $\forall w_j(o_i)$ に対して，o_i と $ins_j(l_k) >_j w_j(o_i) \wedge del_j(l_k) >_j w_j(o_i)$ の $l_k \in f_u(o_i)$ が実体・状態間制約を満たす．
  - (b) $\forall ins_j(l_k)$ に対して，l_k と $w_j(o_i) >_j ins_j(l_k)$ の $o_i \in f_m(l_k))$ が実体・状態間制約を満たす．
  - (c) $\forall ins_j(l_k)/del_j(l_k)$ に対して，l_k と $ins_j(l_p) >_j ins_j(l_k) \wedge del_j(l_p) >_j ins_j(l_k)$ の $l_p \in g(l_k)$ が状態間制約を満たす． □

次は，その 3 つの性質を分析したものである．

定理 1 $T = \{T_1, T_2, \cdots, T_n\}$ 上のスケジュール H に対して，その中の各 $T_i \in T$ が論理性，検索一貫性と結果一貫性を満たすならば，スケジュール H が終了される時点の DB は必ず一貫する．

証明：スケジュール H が終了される時点の DB が一貫していないとすると，一貫していない 2 つのデータともスケジュール H の操作結果または，その中の 1 つはスケジュール H の操作結果であるということになる．後者の場合は，その操作結果を生成した処理単位 $T_i \in T$ が結果一貫性を満たさないことになるので，定理の条件と矛盾する．前者の場合は，さらに，以下のように細分できる．

(1) その 2 つの操作結果は，ある 1 つの処理単位 T_i によって生成された場合．

(2) その2つの操作結果は，2つの異なる処理単位T_iとT_jによって生成された場合．

場合(1)は，T_iが結果一貫性を満たさないことになるので，定理の条件と矛盾する．場合(2)は，T_iとT_jとも論理性と結果一貫性を満たしているため，そのようなことはありえない． □

従来のDBにおいては，処理単位がDBの一貫性を保持すると仮定する方法で一貫性制御を行っていたため，直列可能である並行実行のみが正当なスケジュールとなる．しかし，設計DBにおいては統一的な判断基準でDBの一貫性を制御できるので，正当なスケジュールを以下のように定義できる．

定義 9 $T = \{T_1, T_2, \cdots, T_n\}$上のスケジュール$H$において，各$T_i \in T$が論理性，検索一貫性と結果一貫性を満たすならば，$H$が正当なスケジュールという．

次の定理は，正当なスケジュールの実現方法に関連するものである．

定理 2 検索一貫性及び結果一貫性を満たした複数個の子処理単位上のスケジュールである処理単位T_iが論理性を満たすならば，T_iは必ず検索一貫性と結果一貫性を満たす．

証明：T_iが論理性を満たすため，T_iが状態制約で関連する実体や状態を操作している間に，他の処理単位によるそれらの間の一貫性を破壊する変更操作が禁止される．従って，T_iが検索一貫性を満たした複数個の子処理単位上のスケジュールであれば，T_iが検索一貫性を満たす．同様に，T_iが結果一貫性を満たした複数個の子処理単位上のスケジュールであれば，T_iが結果一貫性を満たす． □

従って，$T = \{T_1, T_2, \cdots, T_n\}$上のスケジュール$H$に対して，その中の各$T_i \in T$が論理性を満たし，かつ$T_i$が検索一貫性及び結果一貫性を満たした複数個の子処理単位から構成されるならば，スケジュールHは必ず正当である．ただし，正当なスケジュールは必ず直列可能であるとは限らない．

例 2 次の並行実行においては，T_iとT_jは共に論理性，検索一貫性及び結果一貫性を満たす(T_sとT_tはMの異なる部分を変更とする)が，互いに相手の操作結果を読み出しているため，直列可能ではない．

	T_s	T_t
t_1	$w_s(o_i)$;	$w_t(o_j)$;
	状態制約保持のため，	状態制約保持のため，
	$M_1 \in M$を変更；	$M_2 \in M$ $(M_1 \cap M_2 = \phi)$を変更；
t_2	$r_s(o_j)$;	$r_t(o_i)$;
t_3	$r_s(test_1(o_i))$	$r_t(test_1(o_j))$ □

4 直列実行における状態制約の保持

3節でまとめた並行処理制御の正当性に基づいて並行処理の非直列可能制御方式を提案するためには，本節では，状態制約の保持方法について検討する．

設計DBが変更される時に，実体・状態間制約の保持に対しては，次のような管理方法しか用いられない．

定義 10 実体・状態間制約の1つの保持方法：

(1) $w(o_i)$操作の後，$\forall l_k \in f_u(o_i)$に対して，必ず$del(l_k)$操作を行う．

(2) $ins(test_k(o_i))$操作は，必ずo_iがk番目のテストを通過できたテスト操作の後に行う． □

補題 1 定義10の方法に従えば，任意の直列実行において実体・状態間制約は保持される．

証明：実体・状態間制約を破壊する可能性があるのは，変更操作中の$w(o_i)$と$ins(test_k(o_i))$操作のみである．その2つの操作が以上のように実行されれば，明らかに直列実行において実体・状態間制約が保持される． □

状態間制約の保持方法として，真理保全と検証という2つの方法がある．

定義 11 $< U, R, M >$を設計DBとし，Pをその状態間制約とする．ある変更操作によって，設計DBが$< U, R, M' >$に変更される時に，検証とは，$M' \cup R \models P$を検査することである．式が成り立つなら，検証が通過できたといい，そうでなければ，検証が通過できていないという．$Q \subseteq P$を状態間制約内の規則部分とすると，真理保全とは，$M' \cup M'' \cup R \models Q$(削除する場合，$(M' - M'') \cup R \models Q$)が恒真になるような極小な$M''$を求めることである．$M' \cup M'' \cup R \models P - Q / (M' - M'') \cup R \models P - Q$が成り立つ場合，設計DBを$< U, R, M' \cup M'' > / < U, R, M' - M'' >$に変更し，真理保全が完成できたという．そうでなければ，真理保全が完成できないといい，変更を拒否する． □

検証と真理保全を行う手続きは，付録に示されている．検証のコストの上限は$|U|$である．真理保全のコストの上限は$n \times |U|$になる．

定義 12 直列実行におけるDB操作とは，次の条件を満たした基本操作の系列から構成されるものである．

(1) $w(o_i)$ と $ins(l_k)$ は，定義10に従って行われなければならない．

(2) $ins(l_k)$ と $del(l_k)$ に対しては，事前の検証通過もしくは事後の真理保全完成が要求される． □

例 3 例1の時点 t_1 における操作が定義12に従って行われたとすると，T_i と T_j は以下のようなDB操作から構成されることになる (状態間制約を例1内の P とするため，真理保全を必要としなくなる)．

	T_s	T_t
t_1	$w_s(o_i)$;	$w_t(o_j)$;
	For $k = 1$ to n *do*	*For* $k = 1$ to n *do*
	$del_s(test_k(o_i))$;	$del_t(test_k(o_j))$;
	For $\forall o_p(in(o_i, o_p) \in R)$ *do*	*For* $\forall o_q(in(o_j, o_q) \in R)$ *do*
	For $k = 1$ to n *do*	*For* $k = 1$ to n *do*
	$del_s(test_k(o_p))$;	$del_t(test_k(o_q))$;
t_2	$r_s(o_j)$;	$r_t(o_i)$;
t_3	$r_s(test_1(o_i))$	$r_t(test_1(o_j))$ □

補題 2 定義12の任意のDB操作の直列実行の結果は，状態制約を満たす．

証明：実体・状態間制約は補題1により，状態間制約は完成できた真理保全あるいは通過できた検証により保証される． □

5 並行処理の非直列可能制御方式

本節では，3節で示した並行処理制御の正当性に基づいて，DB操作を利用することにより，並行処理の非直列可能制御方式を示す．

定義 13 並行実行におけるDB操作は，定義12の直列実行におけるDB操作に次のように施錠し，最後に一括解錠するものである．

(a) $r(o_i)/w(o_i)(o_i \in U)$ 操作の前に，$rl(o_i)$ / $wl(o_i)$ を行う．

(b) $r(l_k)/ins(l_k)/del(l_k)$ 操作の前に，$rl(l_k)$ / $il(l_k)$ / $dl(l_k)$ を行い，さらに，$r(l_k)$ / $ins(l_k)$ 操作の前に，実体 $o_i(o_i \in f_m(l_k))$ に対しても $rl(o_i)$ を行う．

ただし，異なるDB操作内の $rl(o_i)$ と $wl(o_i)$，$wl(o_i)$ と $wl(o_i)$，$rl(l_k)$ と $il(l_k)/dl(l_k)$，$il(l_k)$ と $dl(l_k)$ は競合する． □

例 4 例3中の T_s の時点 t_1 での直列実行におけるDB操作を，並行実行におけるDB操作に拡張すると，次のように施錠，解錠されることになる．

	DB操作 $\in T_s$
t_1	$wl_s(o_i)$; $w_s(o_i)$;
	For $k = 1$ to n *do*
	begin
	$dl_s(test_k(o_i))$;
	$del_s(test_k(o_i))$;
	end;
	For $\forall o_p(in(o_i, o_p) \in R)$ *do*
	For $k = 1$ to n *do*
	begin
	$dl_s(test_k(o_p))$;
	$del_s(test_k(o_p))$;
	end;
	$wu_s(o_i)$;
	For $k = 1$ to n *do*
	$du_s(test_k(o_i))$
	For $\forall o_p(in(o_i, o_p) \in R)$ *do*
	For $k = 1$ to n *do*
	$du_s(test_k(o_p))$ □

定理 3 任意の並列実行において，*DB* 操作の結果が状態制約を満たす．

証明： 並行実行におけるDB操作は，狭義二相ロック方式に従って施錠・解錠しているため，任意の並行実行において必ず直列可能である．従って，補題2により，定理が成り立つ． □

次に，処理単位をDB操作から構成させる．

定義 14 処理単位 T_i は，次にような部分順序 $<_i$ を持つDB操作の系列である．

(1) $T_i \subseteq \{p_i \mid p_i$ は並行実行におけるDB操作 $\}$，

(2) $<_i$：T_i 内の並行実行におけるDB操作によって決められる部分順序である． □

定義 15 統一的な判断基準に基づく並行処理制御方式とは，各 T_i に対して次のように施錠・解錠するものである．ただし，$trl_i(o_j)$ と $wl_s(o_j)$，$trl_i(l_k)$ と $il_s(l_k)/dl_s(l_k)$ は競合する．

(1) $trl_i(o_j)$ を任意の $rl_i(o_j)$ $wl_i(o_j)$ より前に，$tru_i(o_j)$ を任意の $ru_i(o_j)/wu_i(o_j)$ より後に行う．

(2) $trl_i(l_k)$ を任意の $rl_i(l_k)/il_i(l_k)/dl_i(l_k)$ より前に，$tru_i(l_k)$ を任意の $ru_i(l_j)/iu_i(l_j)/du_i(l_j)$ $(l_j \in g(l_k))$ より後に行う． □

例 5 例3で示している処理単位 T_s に対しては，次のように $trl_s(o_j)/tru_s(o_j)$，$trl_s(l_k)/tru_s(l_k)$ が行われる．

T_s

t_1 $trl_s(o_i)$; $wl_s(o_i)$; $w_s(o_i)$;
For $k = 1$ to n *do*
begin
$trl_s(test_k(o_i))$; $dl_s(test_k(o_i)$;
$del_s(test_k(o_s))$;
end;
For $\forall o_p(in(o_i, o_p) \in R)$ *do*
For $k = 1$ to n *do*
begin
$trl_s(test_k(o_p))$;$dl_s(test_k(o_p))$;
$del_s(test_k(o_p))$;
end;
$wu_s(o_i)$;
For $k = 1$ to n *do*
$du_s(test_k(o_i))$;
For $k = 2$ to n *do*
$tru_s(test_k(o_i))$;
For $\forall o_p(in(o_i, o_p) \in R)$ *do*
For $k = 1$ to n *do*
begin
$du_s(test_k(o_p))$;$tru_s(test_k(o_p))$;
end

t_2 $trl_s(o_j)$;$rl_s(o_j)$; $r_s(o_j)$;
$ru_s(o_j)$; $tru_s(o_j)$;

t_3 $rl_s(o_i)$;$rl_s(test_1(o_i))$; $r_s(test_1(o_i))$;
$ru_s(test_1(o_i))$; $tru_s(test_1(o_i))$; $tru_s(o_i)$ □

定理 4 統一的な判断基準に基づく並行処理制御方式によって生成されるスケジュール H は，必ず正当である．

証明： $trl_i(o_j)/tru_i(o_j)$ と $trl_i(l_j)/tru_i(l_j)$ の施錠／解錠の仕方により，各処理単位 $T_i \in T$ が必ず定義8で定義される論理性を満たす．従って，処理単位は状態制約を満たすDB操作から構成される性質を加えると，定理2により，$T_i \in T$ が必ず検索一貫性と結果一貫性を満たすことになる．論理性，検索一貫性と結果一貫性を満たす処理単位 $T_i \in T$ からなる T 上のスケジュール H は，正当である． □

ただし，例5により，非直列可能である並行実行は本方式より許可されることが分かる．

6 従来の研究との比較

設計DBなどを含む高水準DBにおける並行処理制御方式に関して，様々な研究がなれてきた．ここでは，その中のいくつかの研究との比較を行う [6, 7, 8]．

処理単位内の一部の結果を早く他の処理単位に見せられるために，処理単位 T を A，B という2つの直列可能な処理単位に分割する方法が提案されている [8]．その特徴は分割された処理単位 A を早く終了できることにより，並行性や回復処理能力を向上できることである．ただし，処理単位 T を A，B という順で直列可能な2つの処理単位に分けるために，次のようなよう条件が要求されている．

(1) $AWriteSet \cap BWriteSet \subseteq BWriteLast$

(2) $AReadSet \cap BWriteSet = \phi$

(3) $BReadSet \cap AWriteSet = ShareSet$

従って，二相ロック方式を採用した場合に，A を終了することにより，解錠もしくは緩められるロックは次となる．

(1) 解錠できる R ロック：
$AReadSet - AReadSet \cap BWriteSet - AReadSet \cap BReadSet = AReadSet - \phi - AReadSet \cap BReadSet = AReadSet - BReadSet$

(2) 解錠できる W ロック：
$AWriteSet - AWriteSet \cap BWriteSet - BReadSet \cap AWriteSet = AWriteSet - BWriteSet - BReadSet$

(3) W ロックを，R ロックに緩められるロック：
$(AWriteSet - AWriteSet \cap BWriteSet) \cap BReadSet = (AWriteSet - BWriteSet) \cap BReadSet$

場合 (2) と (3) に対して，$AWriteSet$ が一貫性制約を満たすと仮定されるので，明らかに本論文の方式でもそれらの解錠・緩め錠ができる．場合 (1) では，$AReadSet$ と $BReadSet$ が一貫性制約上で関連しないなら，本論文の方式でも，それらの R ロックを解錠できる．そうでなければ，分割された直列可能である処理単位 A と B によって検索されるデータが一貫するとは保証できないので，本論文で提案している方式とは比較できない．

補正子処理単位 (compensating subtransaction) で前向き回復 (forward recovery) を行う Sagas 方式も提案されている [6]. Sagas は相対独立である子処理単位 $T_1, T_2, \cdots, T_n$ の集合であり, $T_1, T_2, \cdots, T_n$ か, 或いは T_1, T_2, $\cdots$, T_j, C_j, $\cdots$, C_2, C_1 が実行されることを保証すれば良い. ここで, C_j は T_j の補正子処理単位である. しかし, Sagas において, 補正子処理単位は抽象的なもので, 利用者によって定義されなければならない, 本論文で提案している方式では, 真理保全という設計 DB による一貫性保持の具体的な手続きを提供している.

一方, 設計応用のための DB には, 直列可能を要求する必要はなく, 有限オートマトンで定義される規則を満たせば良いという並行処理制御基準の提案もある [7]. しかし, 有限オートマトンで定義される規則は, 必ず規則によって表現できる. さらに, 本論文で提案している統一的な判定基準に基づく制御方式は, 単なる直列可能性を放棄するレベルで留まっておらず, 直列可能性を要求しなくても, 正当な並行処理制御を行える方式というレベルまで保証している.

7 むすび

本論文では, 設計 DB に対して, 文献 [1] に指摘された高水準 DB における並行処理制御の新たな課題を取り組み, 設計 DB の特徴を利用した新たな並行処理制御の正当性基準を定めた. これによって, 従来の直列可能性を要求する方式から脱出する切り口を見つけていた.

本論文では, 設計 DB などの高水準 DB に適した並行処理制御の正当性を定めることを第一目的としているため, それに基づく実用的な並行処理制御方式の検討については, 今後の研究課題としておきたい. また, 統一的な判定基準に基づく並行処理制御方式は, 終了されていない処理単位の操作結果を他の処理単位に見せているため, 回復制御上で問題が生じる可能性がある. 我々は, それらの問題を後退復帰ではなく, 処理単位の構成を変更できるという方法を用いて対処していく予定である.

参考文献

[1] Barghouti, N. S. and Kaiser, G. E.: Concurrency Control in Advanced Database Applications, *ACM Computing Surveys*, Vol. 23, No. 3, pp. 269-317 (1991).

[2] Bernstein, P. A.: Database Systems Support for Software Engineering - An Extended Abstract, *Proc. 9th Software Engineering Conf.*, pp. 160-178 (1987).

[3] Cellary, W., Gelenbe and E., Morzy, T.: Concurrency Control in Distributed Database Systems, *North-Holland* (1988).

[4] Elmagarmid, A. K.: Database Transaction Models for Advanced Applications, *Morgan Kaufmann Publishers* (1990).

[5] Katz, R. H.: Toward a Unified Framework for Version Modeling in Engineering Databases, *ACM Comput. Surv.*, Vol. 22, No. 4, pp. 375-408 (1990).

[6] Garcia-Molina, H. and Salem, K.: Sagas, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 249-259 (1987).

[7] Nodine, M. and Zdonik, S.: Cooperative Transaction Hierarchies: A Transaction Model to Support Design Application. *Proc. 16-th Int. Conf. on Very Large Data Bases*, pp. 83-94 (1990).

[8] Pu, C. Kaiser, G.E. and Hutchinson, N.: Split-Transactions for Open-Ended Activities, *Proc. 14-th Int. Conf. Very Large Data Bases*, pp. 26-37 (1988).

[9] 宇田川佳久：オブジェクト指向データベースの CAD への応用, 情報処理, Vol. 32 No. 5, pp. 585-592 (1991).

付録

アルゴリズム 1：状態間制約 P の下で, 設計 DB $< U, R, M >$ の M に事実 $test_j(\alpha)$ を追加するための検証手続き

procedure $check\_ins\_fact(test(\alpha), P, M, R)$
begin
  if $test_j(\alpha) \in M$ **then** return(1);
  for each rule $\in C_b(pred(test_j(\alpha)))$ **do**
    begin
      if $Rule =$ "$test_i(X) \leftarrow test_j(Y), r(X, Y)$"

```
    then
      for each β ∈ {x|R ⊢ r(x,α)} do
        if test_i(β) ∉ M then return(0)
    else/* Rule = "test_i(X) ← test_j(X)" */
        if test_i(α) ∉ M then return(0)
  end
  return(1)
end.
```

アルゴリズム 2：状態間制約 P の下で，設計 DB $< U, R, M >$ の M に事実 $test_j(\alpha)$ を追加するための保全手続き

```
procedure insert_fact(test_j(α), P, M, R, M')
begin
  if test_j(α) ∈ M ∪ M' then return;
  M' := M' ∪ {test_j(α)};
  for each rule ∈ C_b(pred(test_j(α))) do
    if Rule = "test_i(X) ← test_j(Y), r(X,Y)"
      then
        for each β ∈ {x|R ⊢ r(x,α)} do
          if test_i(β) ∉ M ∪ M' then
              call insert_fact(test_i(β), P, M, R, M');
      else /* Rule = "test_i(X) ← test_j(X)" (i ≠ j) */
        if test_i(α) ∉ M ∪ M' then
          call insert_fact(test_i(α), P, M, R, M');
end.
```

アルゴリズム 3：状態間制約 P の下で，設計 DB $< U, R, M >$ の M に事実 $test_j(\alpha)$ を削除するための検証手続き

```
procedure check_del_fact(test_i(α), P, M, R)
begin
  if test_i(α) ∉ M then return(1);
  for each rule ∈ C_h(pred(test_j(α))) do
    begin
      if Rule = "test_i(X) ← test_j(Y), r(X,Y)"
      then
        for each β ∈ {x|R ⊢ r(α,x)} do
          if test_j(β) ∈ M then return(0)
      else Rule = "test_i(X) ← test_j(X)" (i ≠ j) */
        if test_j(α) ∈ M then return(0)
    end
  return(1)
end.
```

アルゴリズム 4：状態間制約 P の下で，設計 DB $< U, R, M >$ の M に事実 $test_j(\alpha)$ を削除するための保全手続き

```
procedure delete_fact(test_i(α), P, M, R)
begin
  if test_i(α) ∉ M then return;
  M := M − {test_j(α)}
  for each rule ∈ C_h(pred(test_j(α))) do
    if Rule = "test_i(X) ← test_j(Y), r(X,Y)"
      then
        for each β ∈ {x|R ⊢ r(α,x)} do
          if test_j(β) ∈ M then
            call delete_fact(test_j(β), P, M, R);
      else /* Rule = "test_i(X) ← test_j(X)" */
        if test_j(α) ∈ M then
          call delete_fact(test_j(β), P, M, R);
end.
```

アルゴリズム 5：状態間制約 P の下で，設計 DB $< U, R, M >$ の M に事実 $test_j(\alpha)$ を追加するための NG 検証手続き

```
procedure check_ng(test(α), P, M, R)
begin
  for each goal ∈ C_b(pred(test_j(α))) do
    begin
      if Goal = " ← test_i(X), test_j(Y), r(X,Y)"
      then
        for each β ∈ {x|R ⊢ r(x,α)} do
          if test_i(β) ∈ M then return(0)
      else/* Goal = " ← test_i(X), test_j(X)" */
          if test_i(α) ∈ M then return(0)
    end
  return(1)
end.
```