BASEトランザクションにおけるデータ整合性

龍谷大学理工学部

西田紗知

目次

1. 研究背景
2. クラウドとBASEトランザクション
3. VDM++による整合性記述
4. CPNによるシステムモデリング
5. CPNによるデータ整合性検証
6. 結論

研究背景

クラウド
コンピューティング

利点

・高度に並列分散化された処理が可能
・可用性，拡張性に優れている

欠点

・整合性に関する問題
・トランザクションシステムの稼働は一般的に困難

稼働させるシステムが提供される機能に適合するかどうか、データ整合性の観点から評価

クラウドとBASEトランザクション

- 従来のトランザクションの管理＝トランザクション管理システム
  - ACID特性 ⇒ 整合性保証

- クラウドのトランザクション管理
  - BASE特性

⇒ 拡張性や並列性を優先

⇒ 整合性の保証に関する機能の一部が欠落

クラウド上で整合性を議論するにはデータ整合性とは何かを正確に議論する必要がある

クラウドとBASEトランザクション

今回の対象

データ整合性
- ・データベース整合性
  データベースレコードの一時点での値に成り立つべき制約
- ・トランザクション整合性
  トランザクション処理の前後間に成り立つべき制約

- 従来はトランザクション管理システムによって保証
- 十分な定式化はなされていない

データ整合性の定義が必要

- データ整合性の定義
- トランザクションシステムのモデル化

⇒シミュレーションによるデータ整合性の評価手法の提案

クラウドとBASEトランザクション

データベース整合性：

「データベース全体が、すべての時刻においてどのような制約を満たすべきか明示したもの」

分類

- データベースレコードの値に対する制約
- データベースレコードの集合に対する制約
- データベースレコードの存在に関する制約

↓

述語論理を用いる定義
形式仕様記述言語による定義

↓

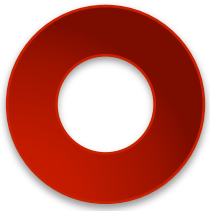
データベースやトランザクションの状態をどう捉え表現するかが重要

クラウドとBASEトランザクション

なぜ形式仕様記述言語による定義をするのか

述語倫理式 → CPN ML

あまりに違いすぎるため、直接記述するのが難しい



述語倫理式 → 形式仕様記述言語 → CPN ML

記述論理式とCPN MLの間に形式仕様記述言語をいれる

クラウドとBASEトランザクション

オブジェクト指向との類似性

- オブジェクトの状態
  - ⇒ インスタンス変数の値の組(または集合)とみなせる
    - ＊すべてのインスタンス変数が状態にかかわるわけではない

例

クラス名　Room

インスタンス変数　部屋番号、面積、室温、湿度

状態　「快適」、「蒸し暑い」、「肌寒い」、・・・など

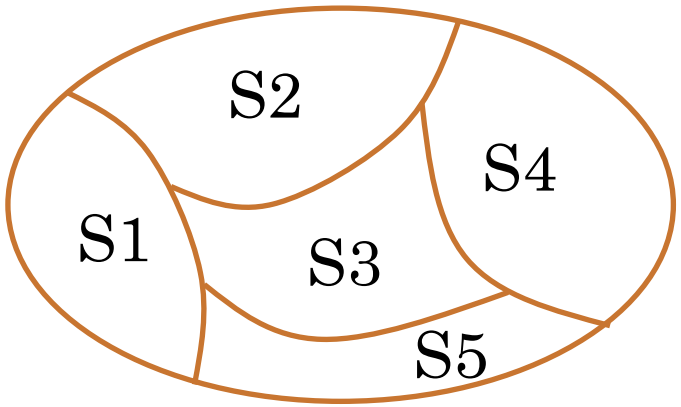
⇨ 室温と湿度だけで決まるなら、この二つの変数の値だけが状態を決定する

- データベースとオブジェクト
  - DBスキーマとクラスの類似性
  - DBレコードとオブジェクトの類似性

BASEトランザクションの状態

```
Class Room {
    int roomNumber;
    double floorSpace;
    double temperature;
    double humidity;
```

```
CREATE TABLE Room (
    roomNumber int,
    fllorSpace float,
    temperature float,
    humidity float,
```



変数の値(の組)が構成する空間の意味的な分割➔状態
分割できない場合各点が一つの状態

データ整合性

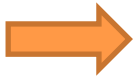
意味的に許される状態と許されない状態

temperature = 1000 ;
humidity = 120;

DBMS上は可能でもアプリケーション的に不可

許される状態➔整合状態
許されない状態➔不整合状態

整合状態を明確に定義するための規則



整合性制約

これを厳密に定義する
述語論理式➔VDM++　(DBとクラスの類似性より)

ACIDとBASEの違い

ACIDトランザクション

Isolation (排他制御 - ロック)により、未確定のDBレコードを外部から観測不可

↓

未確定レコードを除いた部分で整合性が保証できる

↓

すべての時刻で整合性を定義できる

排他制御により、システムの遷移先は決定的
新たなトランザクション投入がない場合、将来の状態は確定

BASEトランザクション

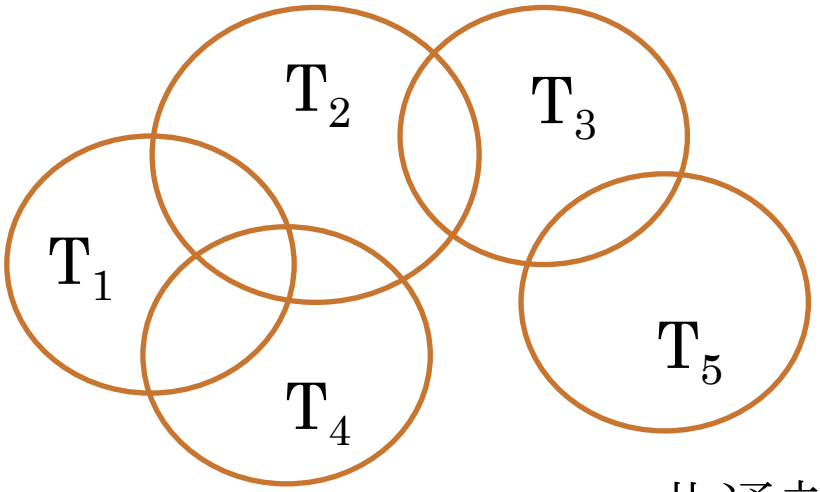
“Basically Available”原則によりすべてのレコードが観測可能

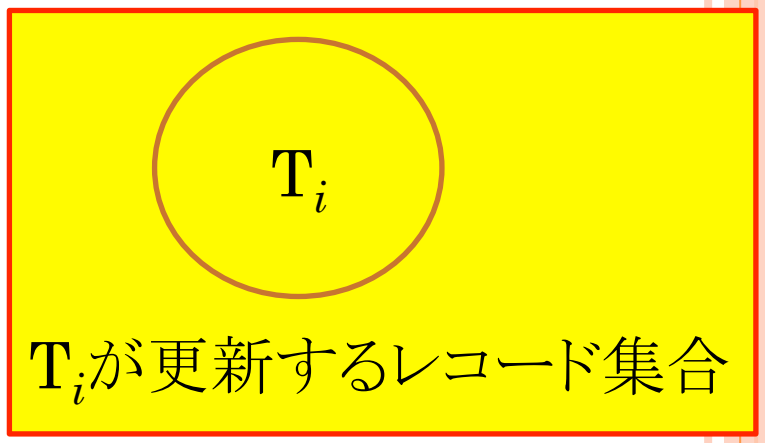
↓

トランザクション実行中の状態や整合性を定義し難い

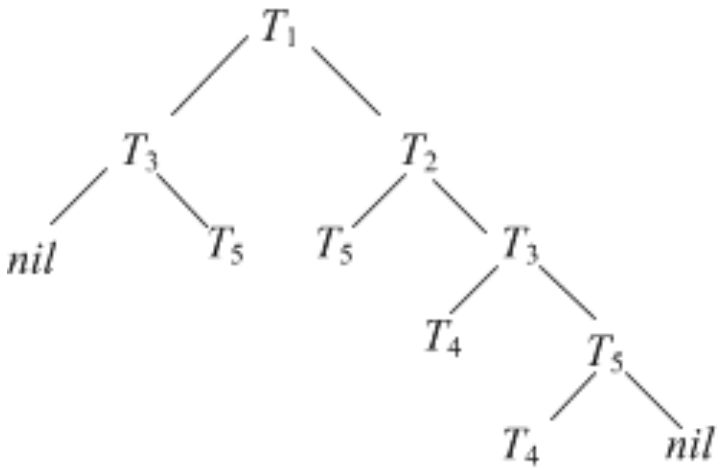
楽観的ロックにより将来の状態も非決定的

BASEにおける状態の非決定性



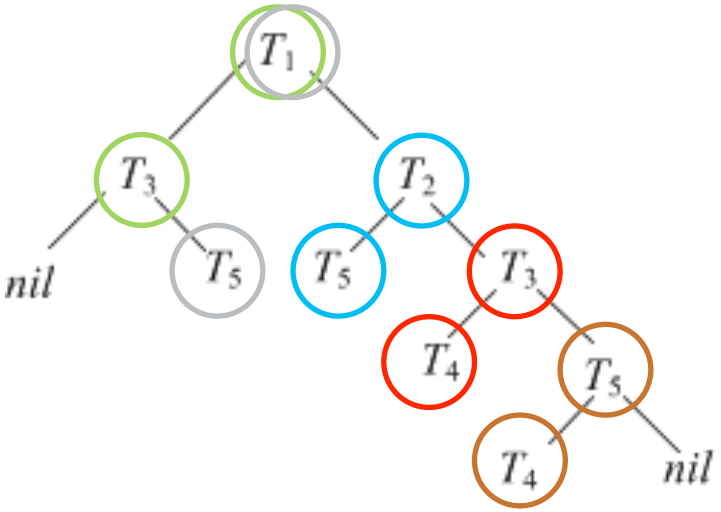


共通部分のあるトランザクションは
一方がcommit、一方がabort



競合のないトランザクション➔左
競合のあるトランザクション➔右
二分木から同時commit
可能なトランザクションセット

BASEにおける状態の非決定性



{T5, T1}, {T3, T1}, {T4, T5},
{T4, T3}, {T5, T2}
が同時にcommitする可能性のある
トランザクションセット
➔五つの可能な到達先

BASEでは、トランザクション稼働中の状態は定義できない
　➔従って整合性も定義できない
到達可能な遷移先の集合をその時点の状態と定義する
　➔状態の重ね合わせになる
　➔すべての可能な状態で整合性の条件が満たされる必要

VDM++による整合性定義

VDMとは…

システムの構成や設計を厳密かつ抽象的にモデル化、記述し、分析・検証する形式手法

記述言語

VDM－SL　　VDM＋＋　今回はこれを使用

- VDM＋＋

VDM－SLにオブジェクト指向を拡張した言語

幾つかの定義ブロックから構成

型定義ブロック、定数定義ブロック、インスタンス変数定義ブロック、
操作定義ブロック、関数定義ブロック

例

```
class Library
        types
                databaseLibrary = seq of Library;
        values
        instance variables
                static i : int := 0; -- databaseLibraryインデックス
                static dbLibrary : databaseLibrary := [];--データベース本体
                static bookSet : set of int := {}; -- bookId set
```

VDM++による整合性定義

オブジェクトの状態との類似性よりVDM++による記述が可能

データベース➔class

DBスキーマ➔インスタンス変数定義

DB本体➔クラスを型とする静的な(static) 列

ロード・挿入➔コンストラクタでインスタンスを上記列に追加

検索・挿入・削除➔静的な操作定義

単一のデータベース上の制約➔静的なbool型操作

複数データベースに渡る制約➔他クラスの静的操作呼び出し

VDM++による整合性定義

```
class Library
        types
                databaseLibrary = seq of Library;
        values
        instance variables
                static i : int := 0; -- databaseLibraryインデックス
                static dbLibrary : databaseLibrary := [];--データベース本体
                static bookSet : set of int := {}; -- bookId set
                public bookId : int := 0; -- 書籍ID
                public lib : int := 1; -- 図書館コード
                public price : int := 0; -- 購入価格
                public lend : bool := false;
        operations
                public Library : int * int * int * bool  ==> Library
                Library (b, li, p, le ) ==
                        atomic (
                                bookId := b
                                lib := li
                                price := p
                                lend := le
                                dbLibrary(i) := self
                                bookSet := bookSet union {b};
                                i := i + 1
                        );
                public static constraint1 : Library ==> bool
                constraint1 (x) ==
                        if x.bookId > 0 and x.bookId < 10000 then return true
                        else return false;
```

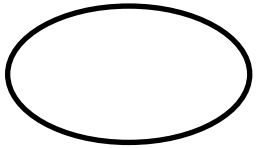
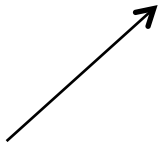
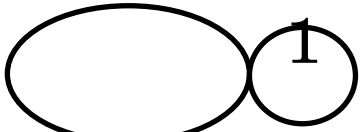
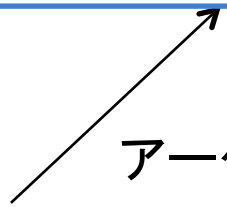
DBスキーマ

コンストラクタ

整合性制約

CPNによるBASEモデリング

ペトリネット

			
プレイス	トランジション	アーク	トークン
 [ガード関数]	 アーク関数		
ガード関数	アーク関数		

カラー定義

```
colset BookId = int;
colset Lib = int;
colset Price = int;
colset Lend = bool;
```

カラーペトリネット

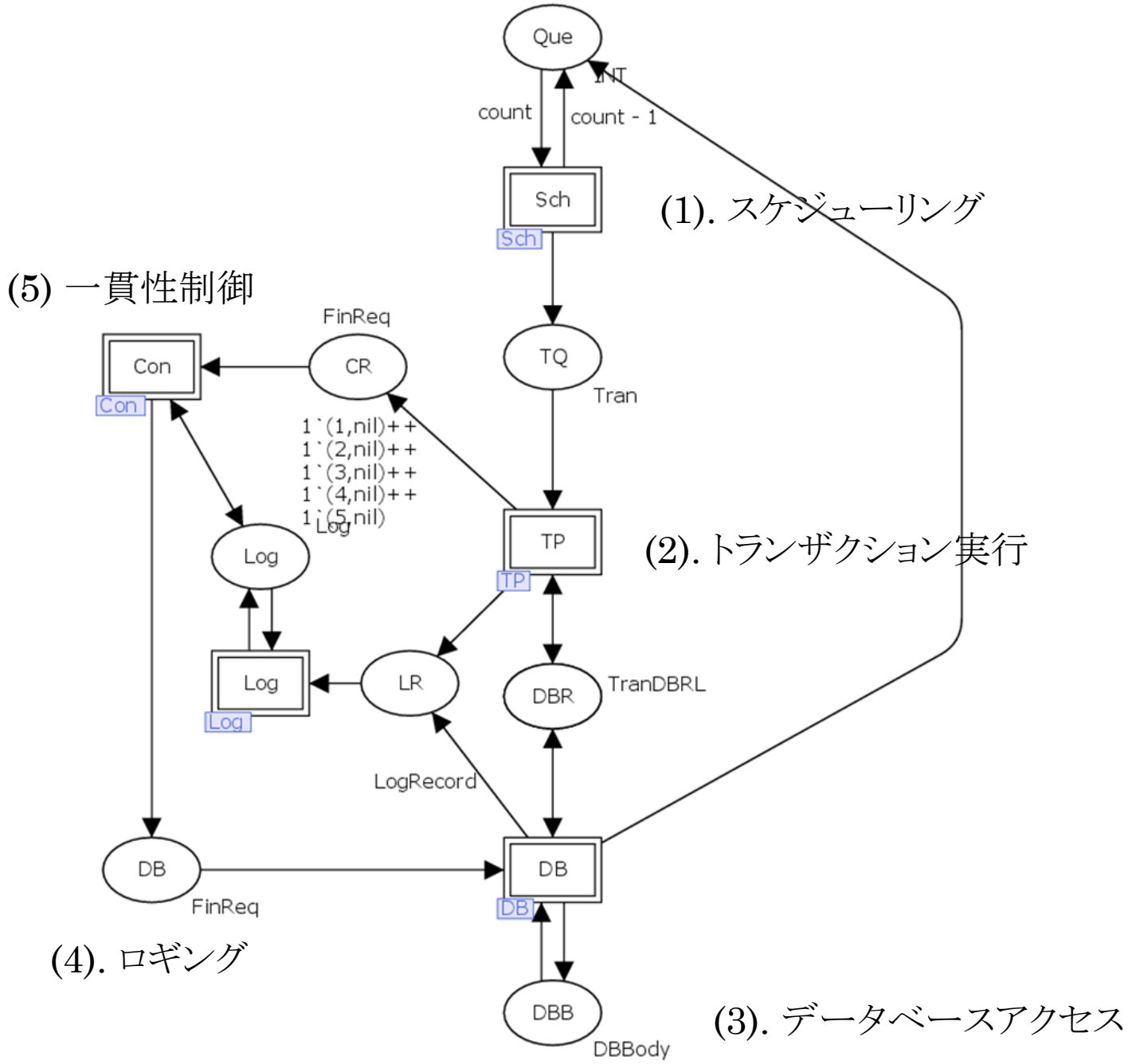
CPNによるBASEモデリング

BASEトランザクション処理の三つの視点を単一モデル表現

- トランザクション管理システム
  (1). スケジューリング
  (2). トランザクション実行
  (3). データベースアクセス
  (4). ロギング
  (5) 一貫性制御

  } CPN構造

- データベース
  VDM++インスタンス変数➔CPN MLカラー
  静的列➔CPN MLリスト
- トランザクション内部ロジック
  CPNトークン＋CPN ML関数


Que
INT
count
count - 1
Sch
Sch
(1). スケジューリング
(5) 一貫性制御
FinReq
Con
CR
Con
TQ
Tran
1`(1,nil)++
1`(2,nil)++
1`(3,nil)++
1`(4,nil)++
1`(5,nil)
Log
Log
TP
TP
(2). トランザクション実行
Log
LR
DBR
TranDBRL
Log
LogRecord
DB
DB
FinReq
DB
(4). ロギング
DBB
DBBody
(3). データベースアクセス

CPNでのデータベース表現

データベースごとにカラー(colset)定義
DBレコードを組型とし、このリストをカラーとする

VDM++

```
class Library

public bookId : int := 0; -- 書籍ID
public lib : int := 1; -- 図書館コード
public price : int := 0; -- 購入価格
public lend : bool := false;
```

CPN ML

```
colset BookId = int;
colset Lib = int;
colset Price = int;
colset Lend = bool;
colset Version = int; (* 楽観的ロック用 *)
colset  DBID = int; (* CPM ML アクセス用 *)
colset Library = product Version * BookId
                         *Lib * Price * Lend;
colset DbLibrary = product DBID * list Library
```

CPNでのトランザクション表現

トランザクション　　idと引数リストの組で表現

Id　　各トランザクションを識別する整数値

引数　　整数値で表現

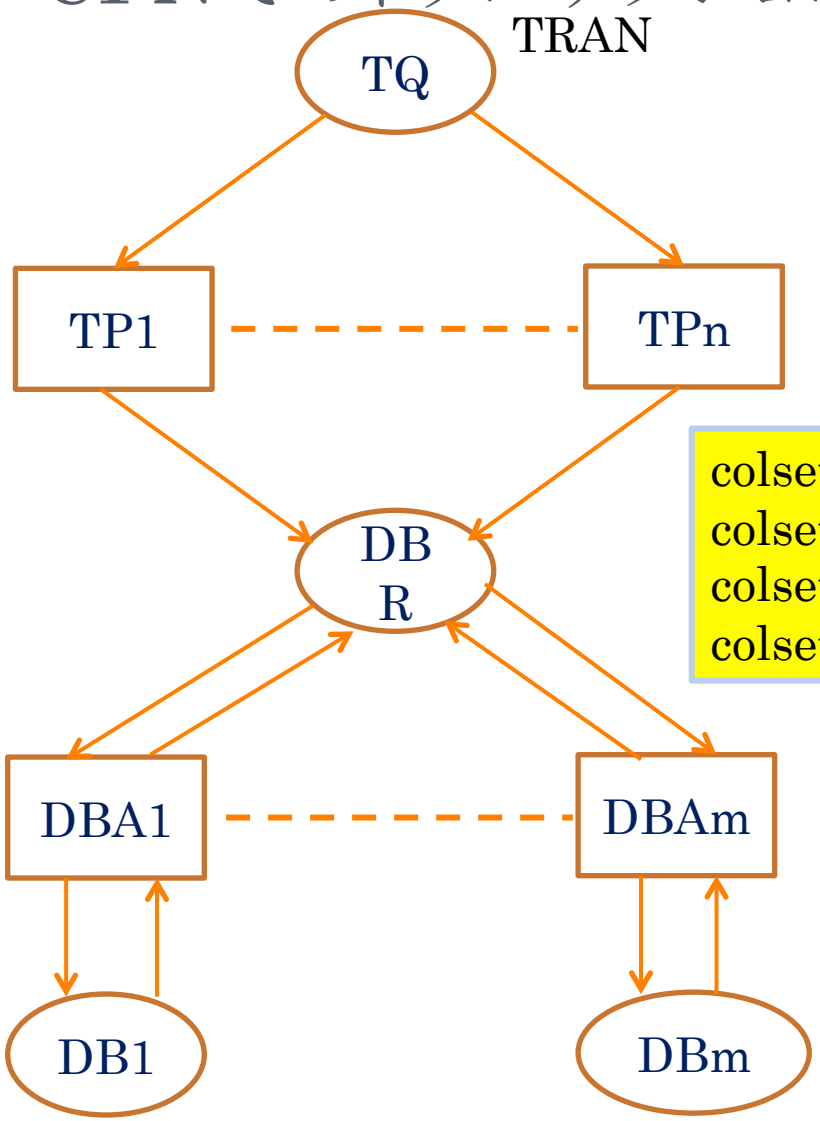
idに対応するCPN MLの関数を用意

- アーク関数より呼び出す
- この関数の戻り値をDBアクセス要求のリストとする
- DBトランジションが各アクセス要求を処理

トランザクションの内部ロジックはVDM++の仕様を基に作成

```
colset TranId = int;
colset Arg = int;
colset TRAN = product TranId * list Arg;
```

CPNでのトランザクション処理



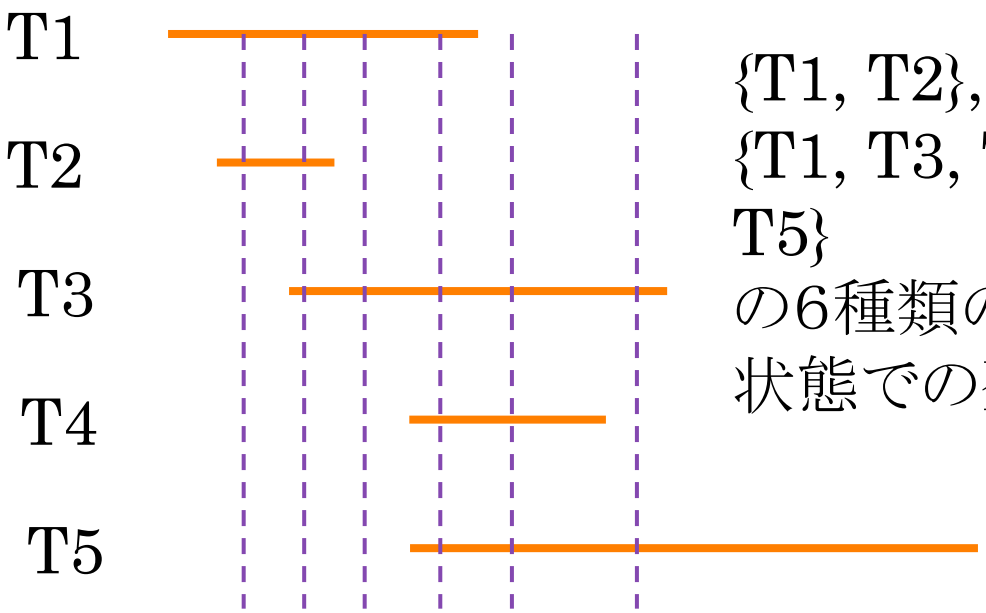
並行処理

```
colset  Key = int;
colset Attribute = int;
colset  DBA = product DBID * Key * list Attribute
colset DBR = product TranId * list DBA
```

並行処理

整合性の評価

BASEは各時刻の整合性ではなく、各時刻でそこから到達可能なすべての状態でデータ整合性が満たされることを検証する必要がある



{T1, T2}, {T1, T2, T3}, {T1, T3} {T1, T3, T4, T5},{T3, T4, T5}, {T3, T5}
の6種類の並行処理からの到達可能状態での整合性検証

整合性の評価

CPNモデルでの検証

モデル実行中の

トランザクション並行度のモニタと記録

トランザクション間の競合状況の把握

が必要となる

並行度が変化するたびに、競合状況に基づき到達可能状態の集合を求め、各状態での整合性を検証

並行処理されているトランザクション

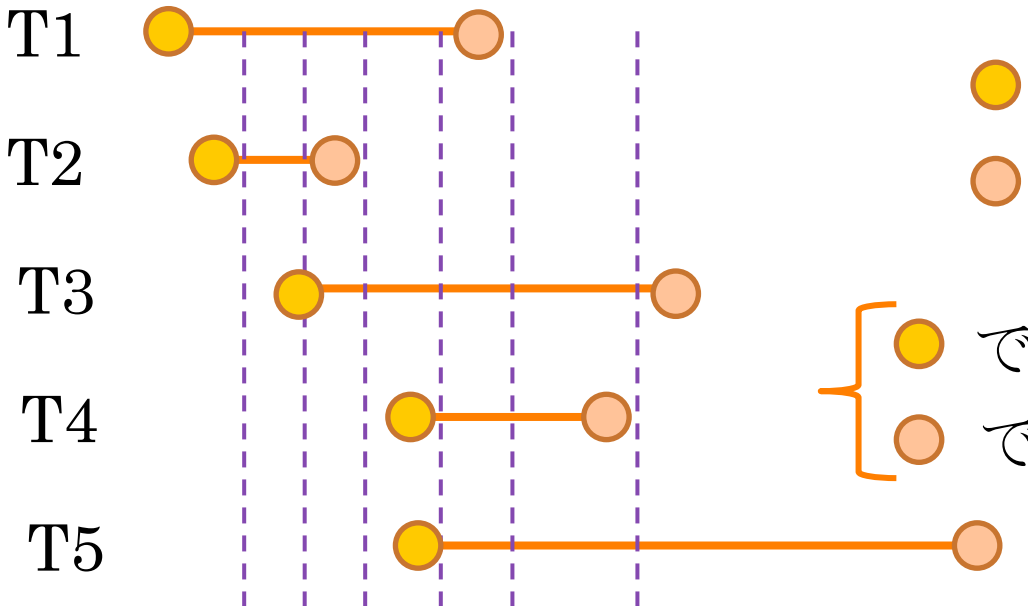
$T = \{T1, T2, \cdots, Tn\}$に対し$Tc \subseteq T$が同時コミット可能な集合とすると、$Ti \in Tc$をコミット、$Ti \notin Tc$をアボートして整合性を検証

整合性の評価

並行度の変化の検出

並行度変化の記録

```
colset  TranSeq = int;  (*シミュレーションでunique*)
colset  Hist = list TranId;
```



- Transaction Start
- Transaction End

- で現在のHistにTranId追加
- で現在のHistからTranId削除

時間経過

現行Histの追加・削除毎に以前のリストと比較
これまでにない組み合わせなら整合性検証プロセス開始

整合性検証プロセス(CPN)

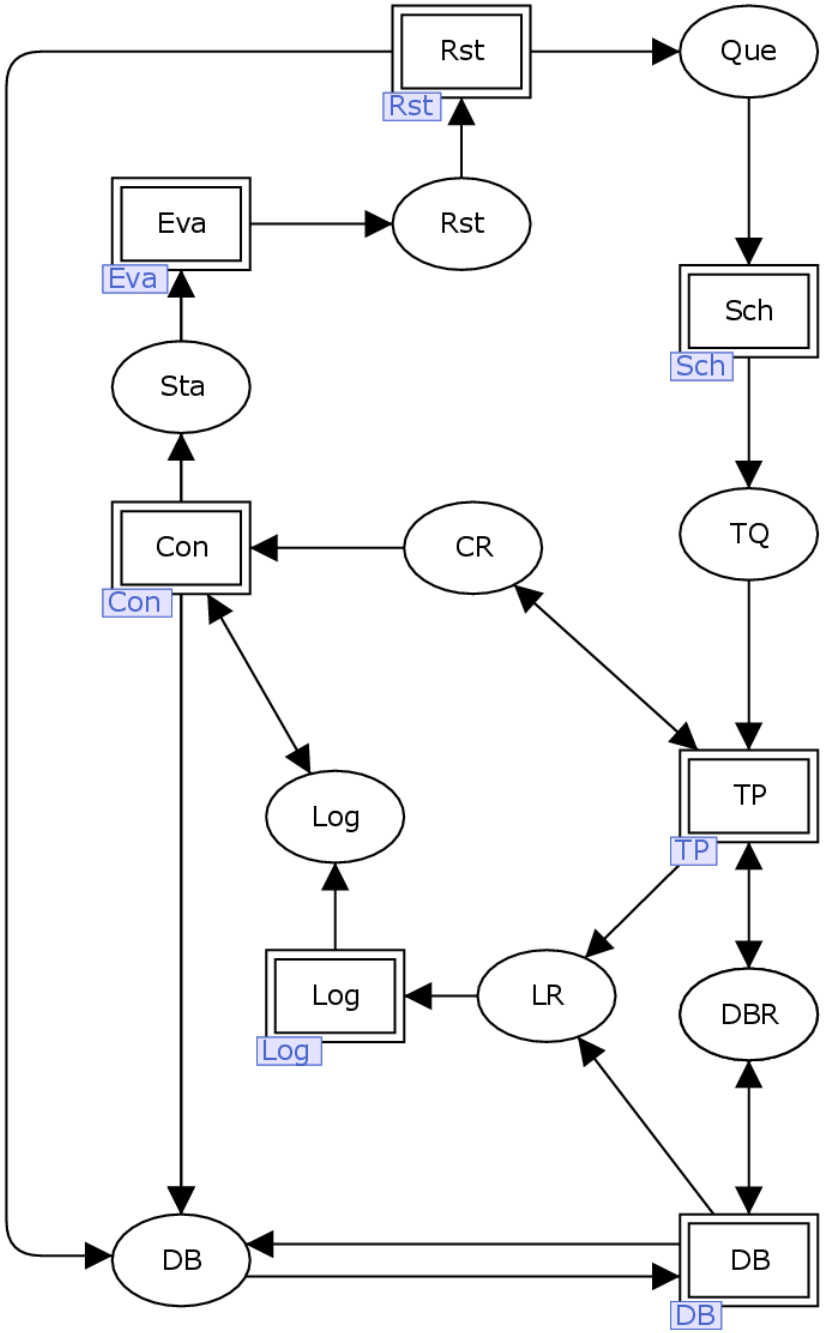
1. 新規のトランザクションスケジューリングの停止

2. 実行中の各トランザクションからのDBアクセス情報を取得
   - DBRプレースのトークンより各トランザクションがアクセスするデータベースおよびそのキーの一覧を得る
   - 同じキーを(Writeモードで)アクセスするトランザクションは競合状態となり前出の二分木が(CPN MLで)作成可能

3. これより同時commit可能なトランザクションのセットを得る
   各セットにつき、適切なcommit/abortを行いDBを更新
   すべての結果が整合性制約を満たせばOK

4. 次の可能なトランザクションの並行処理を見つけるため
   マーキングを初期状態に戻す

検証プロセス概要


Rst
Que
Rst
Eva
Rst
Eva
Sch
Sch
Sta
Con
CR
TQ
Con
Log
TP
TP
Log
LR
DBR
Log
DB
DB
DB

結論

- BASEトランザクション状態
  - 可能性に基づく多重化された状態
- BASEトランザクションでのデータ整合性
  - 将来のすべての可能な状態で成立すべき
- 述語論理➔VDM++による整合性定義
  - 可読性・理解性の向上とCPN MLの仕様
- CPNによる三つの視点からのモデリング
  - プラットフォーム・データベース・トランザクション内部ロジック
- 上記CPNモデルに検証用モデルを付加
  - すべての可能な状態での検証を行った

今後の課題

- トランザクション整合性も考慮
- シミュレーションの効率化
- 個々のクラウド環境（GAE、Amazon EC2、Microsoft Azure等）のモデリング

データベース整合性の定義（一般化）

$$\mathcal{Q}_1 \cdots \mathcal{Q}_n \left(\bigvee_j \bigwedge_i P_{ij} \left(t_1^{(ij)} \cdots t_{m_{(ij)}}^{(ij)} \right) \right)$$

$\mathcal{Q}_1 \cdots \mathcal{Q}_n$ ：限量子

⇒“∀”または“∃”

P_{ij} ：述語

⇒比較（≦，＝，＞）など

$t_k^{(ij)}$ ：関数の値を引数とする項

⇒SUM，AVGなど

すべての時点においてデータベースがこの制約式を充足することが整合性の条件

- データベースレコードの値に対する制約

$$((\forall x P(8,x) \wedge \forall y P(4,y)) \vee (\forall y Q(0,y) \wedge \forall x P(10,x))) \wedge \forall x \forall y P(x,y)$$

$$(\forall x \forall y (P(8,x) \wedge P(4,y)) \vee \forall y \forall x (Q(0,y) \wedge P(10,x))) \wedge \forall x \forall y P(x,y)$$

$$\forall x \forall y \forall a \forall b \forall c \forall d (((P(8,x) \wedge P(4,y)) \vee (Q(0,b) \wedge P(10,a))) \wedge P(c,d))$$

- データベースレコードの集合に対する制約

$$\forall x P(10000000, f(x)) \vee \forall x P(300000, g(x))$$

$$\forall x \forall y (P(10000000, f(x)) \vee P(300000, g(y)))$$

- データベースレコードの存在に対する制約

$$\forall r DB_i(r) \rightarrow \exists s DB_j(s) \wedge (u_1^{(m)}(s) = u_1^{(m)}(r))$$

$$\neg(\forall r DB_i(r)) \vee \exists s DB_j(s) \wedge (u_1^{(m)}(s) = u_1^{(m)}(r))$$

$$\exists r(\neg DB_i(r)) \vee \exists s DB_j(s) \wedge (u_1^{(m)}(s) = u_1^{(m)}(r))$$

$$\exists r \exists s ((\neg DB_i(r)) \vee DB_j(s) \wedge (u_1^{(m)}(s) = u_1^{(m)}(r)))$$

・データベースレコードの値に対する制約

⇒レコード内の属性値に対する制約

例)【条件】

扶養者数は8人以下　かつ　同居扶養者数が4人以下

または

同居扶養者数が0人のときは扶養者数は10人以下

かつ

扶養者数≧同居扶養者

社員データベース

社員名	扶養者数	同居扶養者数	…
A	9	0	…
B	2	2	…
C	5	4	…

x: 扶養者数

y: 同居扶養者数

述語P(a,b): a≧b

述語Q(a,b): a=b

$$((\forall x P(8,x) \wedge \forall y P(4,y)) \vee (\forall y Q(0,y) \wedge \forall x P(10,x))) \wedge \forall x \forall y P(x,y)$$

・データベースレコードの集合に対する制約

⇒レコードの集合から得られる値に関する制約

例)【条件】

社員全員の給与額の合計値は1千万円以下

かつ

社員全員の給与額の平均値は30万円以下

給与データベース

社員名	給与	…
A	200,000	…
B	350,000	…
C	250,000	…

x：給与額

述語P(a,b)：a≧b

f(x)：合計値を返す関数

g(x)：平均値を返す関数

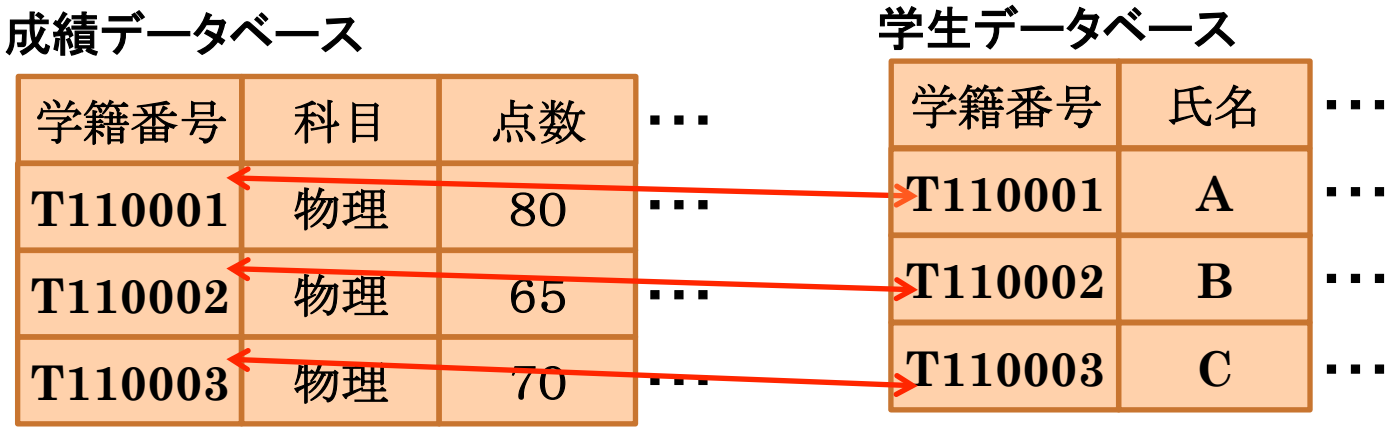
$$\forall x P(10000000, f(x)) \wedge \forall x P(300000, g(x))$$

- データベースレコードの存在に関する制約

⇒関連レコードに関する制約

例)【条件】

成績データベースの学籍番号は
学生データベースの学籍番号に存在しなければならない



$DB_i(r)$: 成績DBにrが属する

$DB_j(s)$: 学生DBにsが属する

$u_1^{(m)}(a)$: m個ある属性の中から1番目の要素aを取り出す

$$\forall r DB_i(r) \rightarrow \exists s DB_j(s) \wedge \left(u_1^{(m)}(s) = u_1^{(m)}(r) \right)$$

図書館ソース(1/4)

```
class Library ---------●Libraryクラス●----------------------------------------------------------------

----------------------------型定義ブロック-----------------------------------------------------------

                types
-- TODO Define types here
                                databaseLibrary = seq of Library;  --databaseLibrary型、Libraryのリスト

        --*seq of  リスト

----------------------------定数定義ブロック-----------------------------------------------------------
                values
-- TODO Define values here

        --*ここでは値を入れるとき、「:=」ではなく「=」を使用することを注意！

-------------------------インスタンス変数定義ブロック-----------------------------------------------

                instance variables
-- TODO Define instance variables here
                                static i : int := 0; -- databaseLibraryインデックス、制御変数
                                static dbLibrary : database Library := [];--データベース本体
                                static bookSet : set of int := {}; -- bookId set
                                public bookId : int := 0; -- 書籍ID
--                              public title : seq of char := []; -- タイトル => Bookへ移動
                                public lib : int := 1; -- 図書館コード
                                public price : int := 0; -- 購入価格
                                public lend : bool := false;  --貸出の有無
```

図書館ソース(2/4)

```
---------------------  操作定義ブロック…システムの状態を読み書きできる  ---------------------------------------

            operations
-- TODO Define operations here

   --*********コンストラクタ**********
public Library : int * int * int * bool  ==> Library
            Library (b, li, p, le ) ==  --   b=書籍ID、li=図書コード、p=購入価格、le=貸出の有無
                        atomic (  --atomic 多重代入文...この間、不当条件のチェックを挟まない。代入は :=を使う。
                                    bookId := b
--                                  title := t
                                    lib := li
                                    price := p
                                    lend := le
                                    db Library(i) := self  --データベースに自分を入れる
                                    bookSet := bookSet union {b};  --union:結合、書籍IDリストに書籍IDリストを加える
                                    i := i + 1  --制御変数に1を足す。
                                    );


            --***************制約条件***************

                        --制約条件1、参照整合性制約、ブックIDの制約
                        public static constraint1 : Library ==> bool
                        constraint1 (x) ==
                                    if x.bookId > 0 and x.bookId < 10000 then return true
                                    else return false;
                                                :::
```

図書館ソース(3/4)

```
--**************検索、追加、更新、消去など*************

 --検索          IDがなければ、-1を、あれば、IDを返す。
public static search : int * int ==> int
search (key, n) ==
                if len dbLibrary <= n then return -1  --lenとはlength
                else
                                if dbLibrary (n).bookId = key then return n
                                else search(key, n + 1);  --次をサーチ

 --追加
public static insert : Library ==> int
insert (x) ==
                if search (x.bookId, 0) >= 0 then return -1  --所蔵書があるなら、-1を返す。
                else
                (
                atomic (dbLibrary (i) := x;  --atomic 多重代入文...この間、不当条件のチェックを挟まない。代入は :=を使う
                bookSet := bookSet union {x.bookId};  --ブックID追加
                i := i + 1);
                return i - 1
                );
```

図書館ソース(4/4)

```
---------------------------- 関数定義ブロック…同じ引数を渡すと常に同じ戻り値を返す ----------------------------------

        functions
-- TODO Define functiones here

    --リストbがリストaを含んでいるか調べる関数。
        public static refInt : set of int * set of int -> bool
        refInt (a, b) == forall x in set a & exists1 y in set b & y = x;
--xにaのすべての値を当てはめる。かつ、bとyがxの値と等しくなるような要素をちょうど1つだけもつyがある。

-- | forall X in set P     XにPのすべての値を当てはめる。
-- | &はかつ
-- | exist1 Y in set Q     集合Y内にある条件Qをみたす要素がちょうど1つある。＊existだと少なくとも1つ以上

-- TODO Define Combinatorial Test Traces here

end Library--------------------------------クラス終了宣言----------------------------------------------------
```

ACID特性

- Atomicity(原子性)

  個々のデータに対する操作が『すべて実行される』か『一つも実行されない』かのどちらかの状態になるという性質

- Consistency(整合性)

  データはある有効な状態から別の有効な状態に変換されなければならないという性質

- Isolation(独立性)

  トランザクションの実行中には他の操作からは何らかの影響も受けず，隠蔽される性質

- Durability(永続性)

  トランザクションの完了通知をユーザが受け取った時点で，どんな障害が発生してもそのトランザクションの結果が永久に保障されていなければならないという性質

BASE特性

- BASE特性とは、クラウド環境でトランザクションを実装するにあたり、次の3つの特性の頭文字を取ったもの
  - Basically Available（可用性が基本）
  - Soft-state（厳密でない状態遷移）
  - Eventual consistency(結果として整合性が保たれる)
- BASE特性はシステムの可用性を最優先する一方、システムの連携をなるべく緩やかにし、即時でなく最終的にシステム全体が整合性を保った状態になるような処理特性

CPNの特徴

- 離散分散環境を数学的に表現する手法
- CPN選択理由
  - CPNは並行動作を含む動的システムを構造面，機能面，動作面の3つから厳密にモデリング可能である
  - 支援ツールのCPN Toolsによりシミュレーションによる評価が可能である
- 有限オートマトンとの違い
  - モデルを合成した際にサイズが大きくなりすぎない
  - 並行性の表現が可能

モデル化の利点

- 実際のデータベースやアプリケーションを作ること無く、整合性が成り立つか確かめられる

整合性の成り立たない例

- バッファ処理がうまくモデリングできていないので異なるサイトのバッファ処理間の問題などで不整合がでるかと予測している

クラウドトランザクション

- GAEは中小企業をターゲットとしていて、例が公開されていないが、普及しつつある
- 地銀とかでNECのクラウドサービスを使っている

クラウドの利用形態


アプリケーション
アプリケーション
プラットフォーム
OS
仮想計算機
ハードウェア
IaaS
（Amazon EC2）
アプリケーション
アプリケーション
プラットフォーム
OS
仮想計算機
ハードウェア
PaaS
（GAE，Azure）
アプリケーション
アプリケーション
プラットフォーム
OS
仮想計算機
ハードウェア
SaaS
（Google：Gmail）
：ユーザー
：サービス提供者