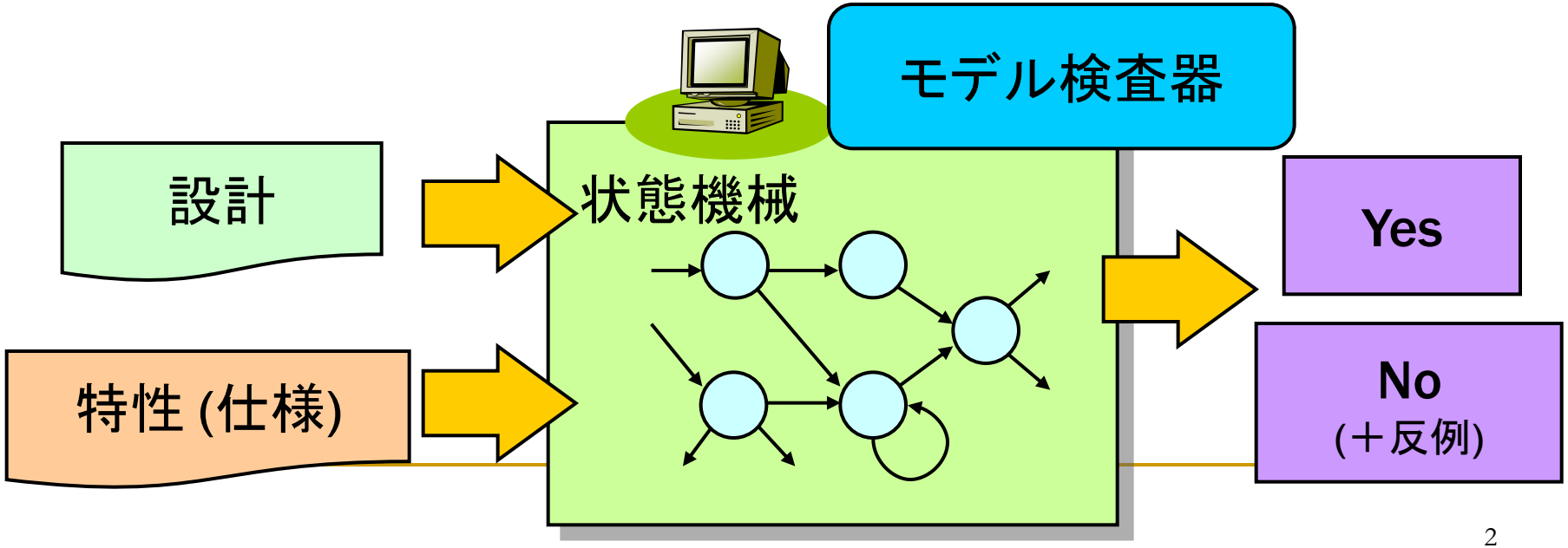
原理から学ぶモデル検査

土屋達弘（大阪大学)

モデル検査とは

- 形式的検証手法
  - 2007 Turing Award (Clarke, Emerson, Sifakis)
  - 入力: 設計 + 特性 (仕様)
  - 出力: Yes or No
  - 方法: 状態探索



モデル検査の実例：SPIN

- SPIN
  - 代表的なモデル検査器
  1. Promela言語による記述
  2. Cプログラムへ変換
     - spin -a -f "¥! []mutex" sample.pml
  3. Cプログラムをコンパイル，実行
     - cc pan.c -o pan
     - pan -a

```
#define mutex !(P0@cs && P1@cs)

byte turn = 0;
active proctype P0() {
  do
  :: skip;
     (turn == 0);
 cs: turn = 1
  od
}

active proctype P1() {
  do
  :: skip;
     (turn == 1)
 cs: turn = 0
  od
}
```

テーマ

1. 基本的な言葉に注意する
   - 状態，遷移，遷移関係，状態系列，．．．
2. 検証したい性質の書き方
   - 時相論理は難しい？

時相論理式

3. どのモデル検査ツールを使うか？

1. 基本的な言葉に注意する

- アルゴリズムの例

```
int x = 0;
1:  int tmp = x;
2:  x = tmp + 1;
(end)
```

- コントロールポイント
  - 現在の制御の位置
    - 次に実行される命令を指す

```
1:  int tmp = x;
2:  x = tmp + 1;
(end)
```

状態と遷移

- 状態
  - コントロールポイントと変数値

```
int x = 0;
1:  int tmp = x;
2:  x = tmp + 1;
(end)
```

cp = 1, x = 0,
tmp = 0

- 遷移
  - 状態と次状態のペア
    - 正確には順序対（入れかえたら別のもの）

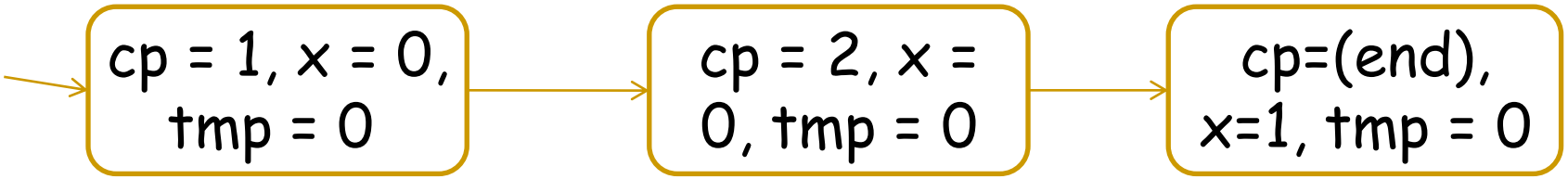
cp = 1, x = 0,
tmp = 0

cp = 2, x = 0,
tmp = 0

状態遷移系

- 状態遷移系
  - グラフで表現可能
    - 頂点: 状態
    - 辺 : 遷移

```
int x = 0;
1:  int tmp = x;
2:  x = tmp + 1;
(end)
```



- 遷移関係
  - 遷移の集合
    - 集合V上の関係＝Vの要素のペアの集合

並行動作

- 例

```
int x = 0;
```

スレッド0

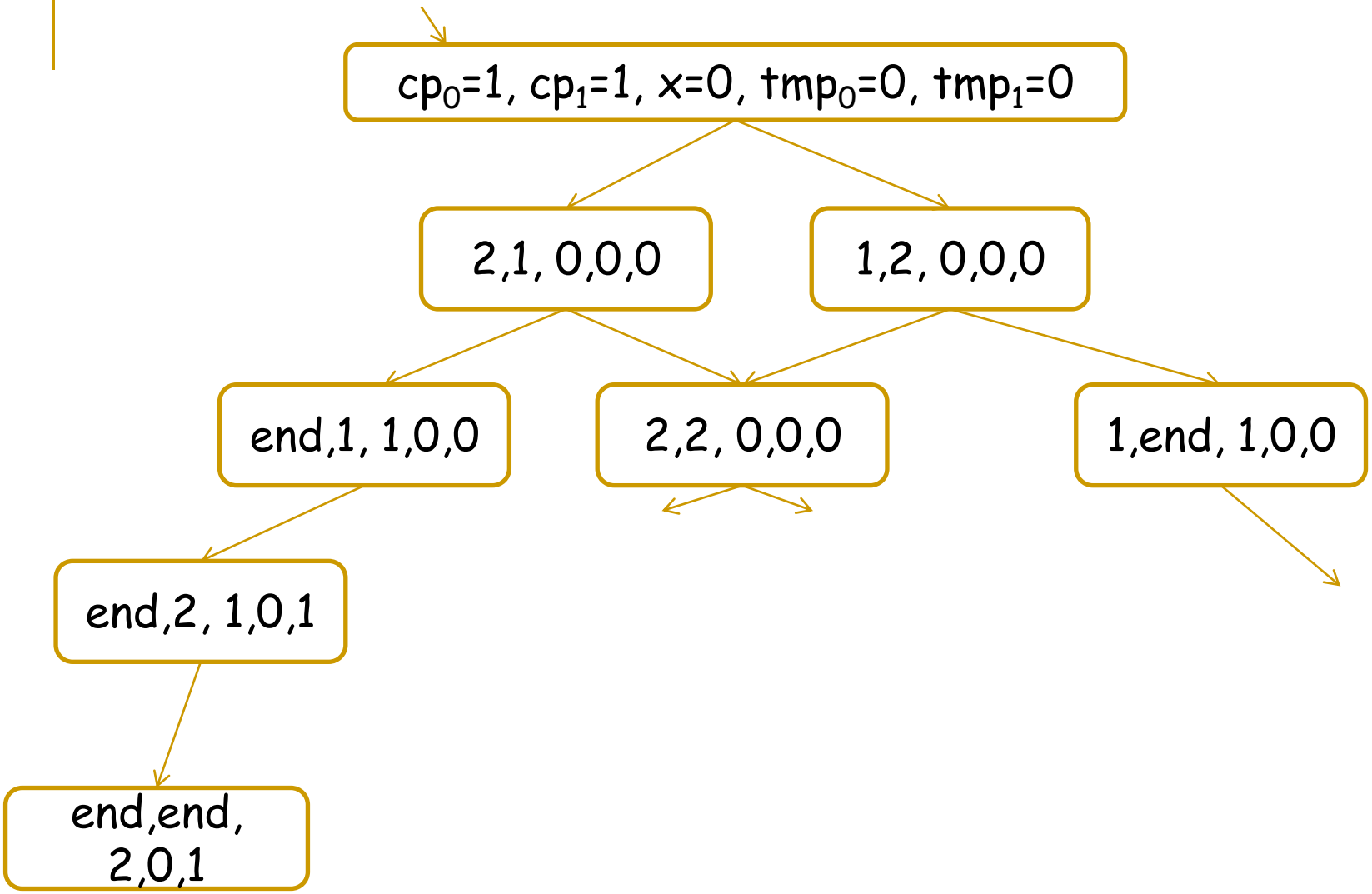
```
1:  int tmp = x;
2:  x = tmp + 1;
(end)
```

スレッド1

```
1:  int tmp = x;
2:  x = tmp + 1;
(end)
```

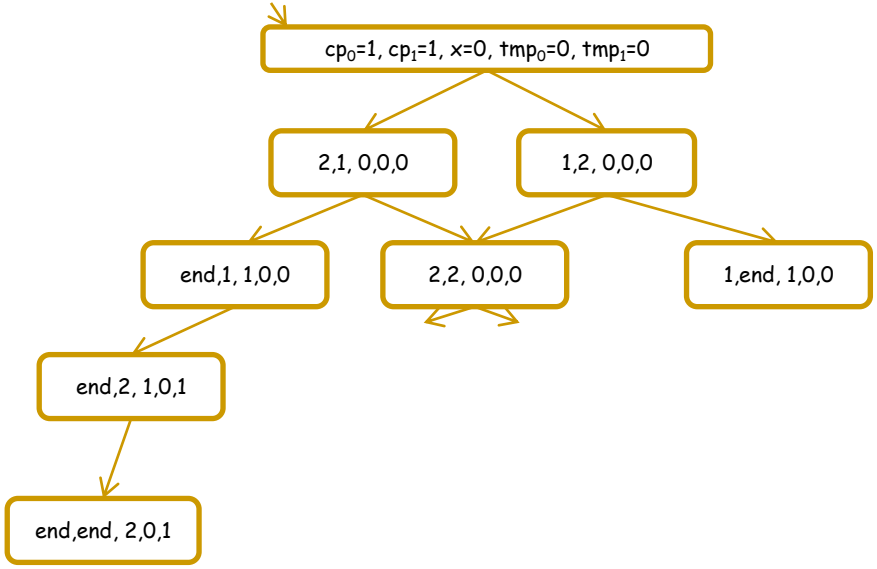
- 状態
  - すべての制御点とすべての変数値

cp0 = 1, cp1 = 1, x = 0,
tmp0 = 0, tmp1 = 0



ステップのインターリービング

計算



- 系列
  - 要素の並び a, b, c, ….
- 計算 (パス)
  - ある状態から起こりうる状態系列
    s0, s1, s2, ….
    - 本当は無限長の系列と定義
      した方が都合がよい　→後で説明
- 動作
  - 初期状態からはじまる計算

結局モデル検査とは？

- すべての動作（初期状態からのすべての計算（パス））を考慮して，
  与えられた性質の真偽を判定すること

- すべての動作**に対して**，とは言っていない点に注意
  - 性質は計算に対して真偽が決まるとは限らない（後でくわしく説明）

モデル検査の実例：SPIN

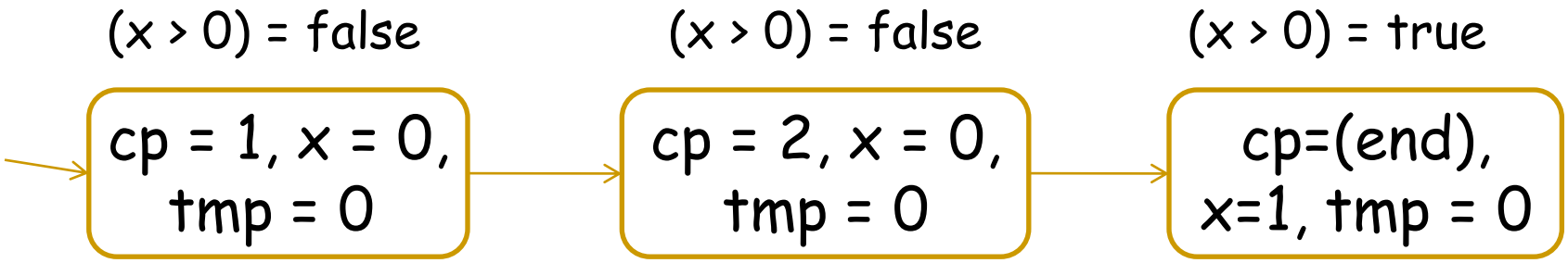
コントロールポイント
(SPINではロケーションカウンタとよぶ)

```
#define mutex !(P0@cs && P1@cs)

byte turn = 0;
active proctype P0() {
  do
  :: skip;
     (turn == 0);
cs: turn = 1
  od
}

active proctype P1() {
  do
  :: skip;
     (turn == 1);
cs: turn = 0
  od
}
```

P0@cs:
プロセスP0のコントロールポイントがcsなら真

小難しい言葉

- クリプキ構造
  - ≒状態遷移系
- 原子命題
  - ≒状態毎に，状態だけを見て真か偽か決まるもの（状態述語）
    - 例．x > 0

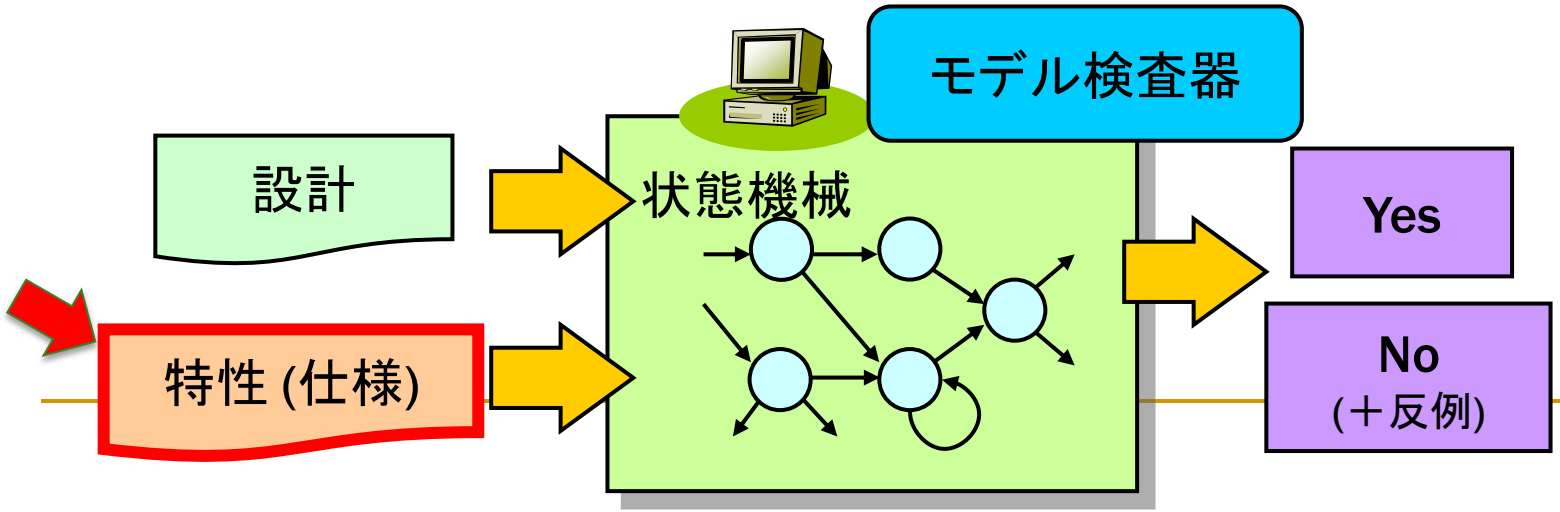


2. 検証したい性質の書き方

- 時相論理
  - 遷移の実行を考慮して性質を表す論理体系
    - 「いずれ」,「つねに」などを表現
    - 例. []mutex: 常にmutexが真
  - 正しく使われてない例が散見

時相論理式

spin -a -f "¥! []mutex" sample.pml



クイズ（経験者むけ）

- 演算子
  - G (□) つねに
  - F (◇) いずれ
- 性質の例：プロセスP0はスターブしない
  - クリティカルセクション(CS)をいずれ実行できる
- LTL論理式は？
  - ただし，P0@CSで，CS実行中であることを表現

問題と解決策

- 問題
  - ただしく性質が表現できなければ，検証は無意味

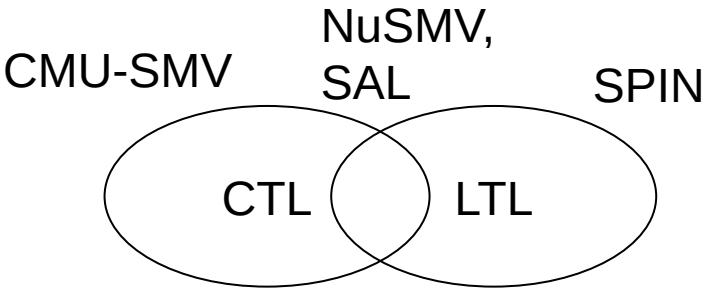
- 解決策

A) 時相論理をただしく理解する

B) 時相論理をつかわない

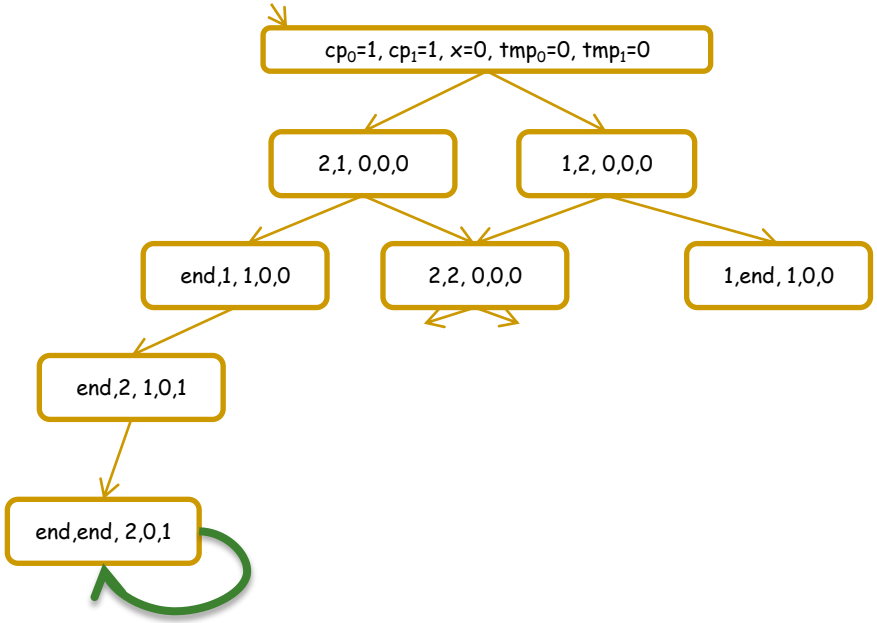
A: 時相論理をただしく理解する

- 代表的な時相論理
  - LTL
  - CTL
- 主要な演算子 (+ ブール演算子)
  - G (□) : つねに
  - F (◇) : いずれ
  - A: すべての計算(パス)で
  - E: ある計算(パス)で
    - CTL式の例: AF (EG (p = 1) ∨ AF (x = 0))



ポイント

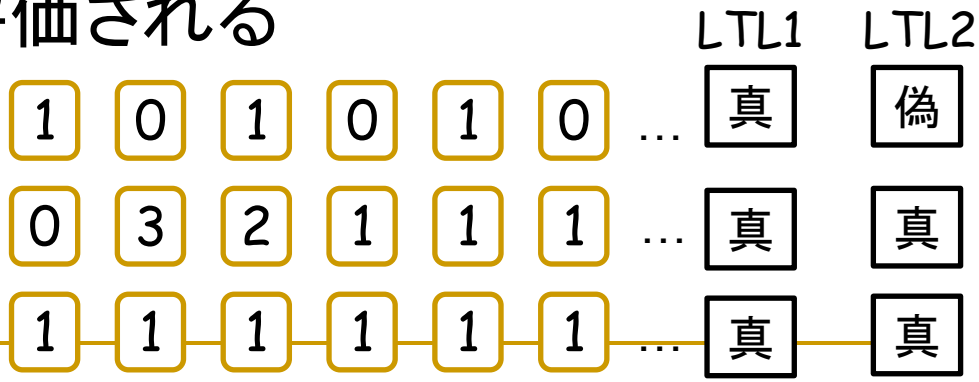
- 時相論理式が何に対して評価されるのかに注意！
  - 計算（パス），または，
  - 状態



- 計算は無限長と考える
  - 変化しない状態は，自己ループがあるものと考える

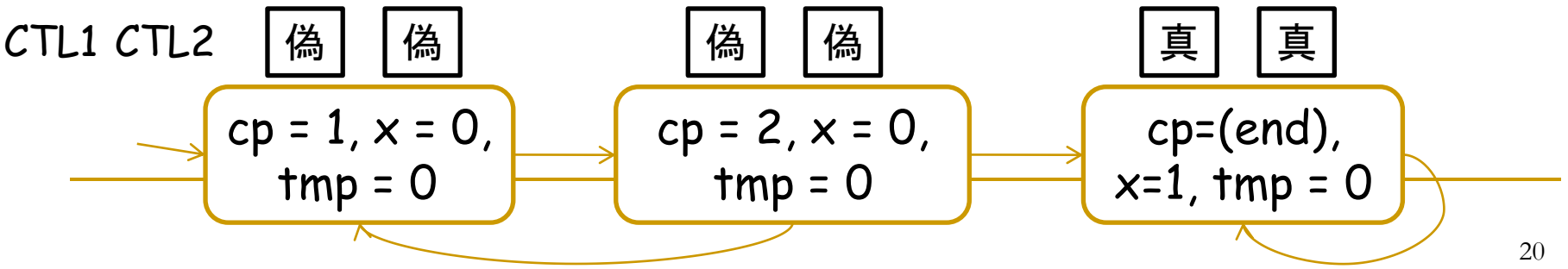
LTL

- A: すべての計算で，E: ある計算で　はつかわない
- G:つねに
  - 計算上のすべての状態で，または，
    計算上のすべての状態についてそこから始まる計算で
- F:いずれ
  - 計算上のどこかの状態で，または，
    計算上のどこかの状態についてそこから始まる計算で
- 計算（パス）に対して評価される
  - LTL1 := GF (x = 1)
  - LTL2 := FG (x = 1)



CTL

- 必ずAG, AF, EG, EFのようにAかEとペアにする
  - A: その状態から始まるすべての計算で, E: ある計算で
- G:つねに
  - 計算上のすべての状態で
- F:いずれ
  - 計算上のどこかの状態で
- 状態に対して評価される
  - CTL1 := AF (cp = (end)), CTL2 := AGEF (x = 1)



時相論理式が検証対象に対して評価されるとき

- （本来の）モデル検査
  - 時相論理式を検証対象に対して評価したときの真偽を判定
- LTL式
  - 初期状態からはじまるすべての計算について，LTL式が真か，そうでないか
- CTL式
  - すべての初期状態において，CTL式が真か，そうでないか

B: 時相論理を使わない

- アサーション
  - 評価されるときには，常に真であるべき条件

```
byte turn = 0;
byte critical = 0;
active proctype P0() {
  do
  :: skip;
     (turn == 0);
     critical++;
     assert(critical <= 1);
     critical--;
     turn = 1
  od
}
active proctype P1() {
  do
  :: skip;
     (turn == 1);
     critical++;
     assert(critical <= 1);
     critical--;
     turn = 0
  od
}
```

時相論理を使わない

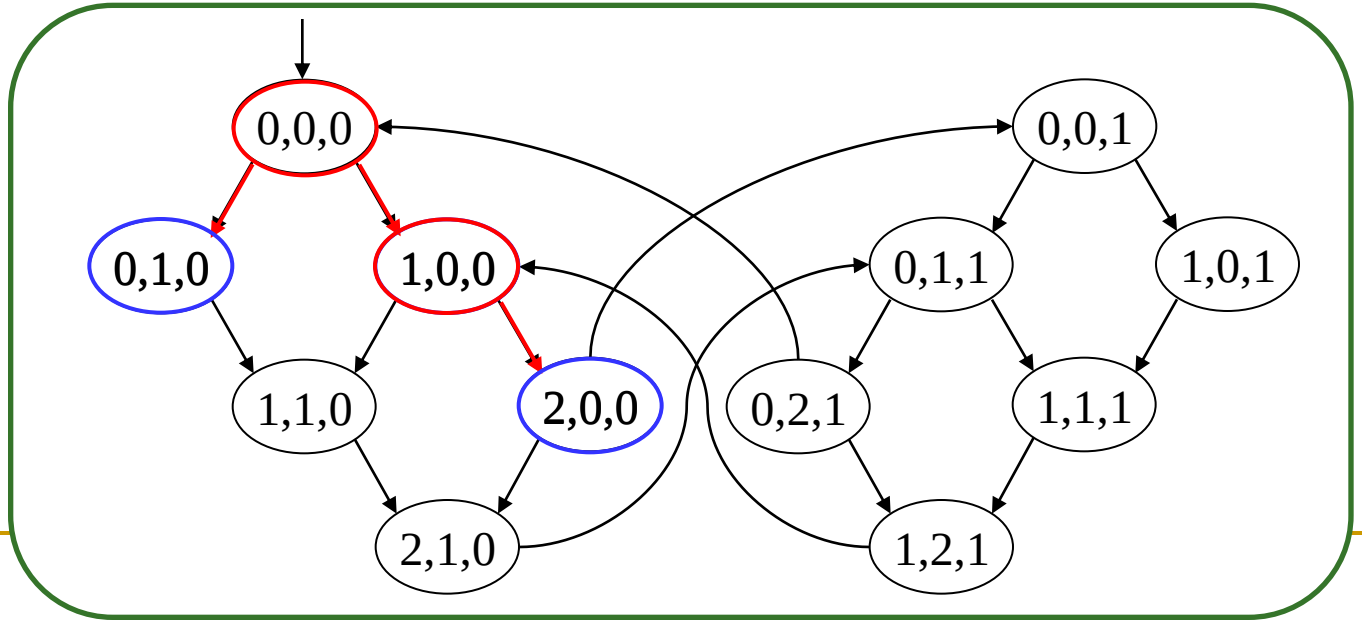
- インバリアント
  - 起こりうる状態で常に成り立つべき条件
    - 例. critical <= 2
  - 個々の状態を見るだけ真偽が決定できる
    - 到達性だけをみればよい
      →状態は1度見るだけでよいので検証が高速
      - NuSMV: INVARSPEC critical <= 2
      - SPIN: [](critical <= 2)
  - assertionはspecial case
    - コントロールポインタ + 式　というinvariant

3.どのモデル検査ツールを使うか？

- 動作原理から考える
  - A) 明示的な状態探索 (explicit model checking)
    - SPIN, TLC
  - B) BDDによる状態探索 (symbolic model checking)
    - NuSMV, SAL
  - C) SAT/SMT
    - FSM
      - NUSMV, SAL
    - プログラムコード
      - CBMC

A. 明示的な状態探索

1. 未展開の状態sを選択
2. sの各遷移について
   1. 次状態s'を計算
   2. s'がはじめて現れたなら記録

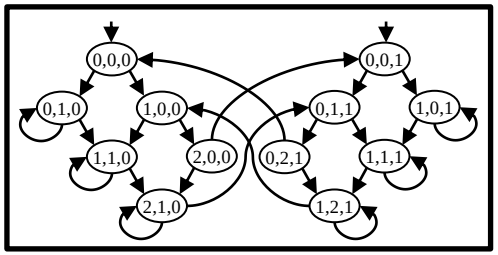


特徴

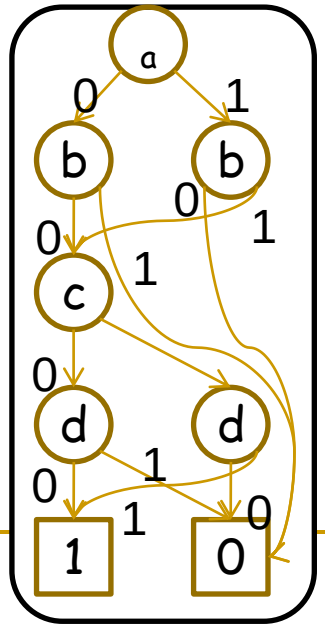
- 柔軟
  - 複雑な処理も記述可能
  - 変数のレンジをあまり気にしなくてよい
    - キューなども扱える
    - BDDの場合と対照的
  - 逐次プログラムでも，並行システムでも検証可能
- BDDよりは扱える状態は少ない
  - ただし，lossyだが効果的な圧縮手法はある
    - 例．Bitstate hashing

B. BDDによる状態探索

- ブール関数で状態集合や遷移関係を表現
- BDDでブール関数を表現
- BDDの操作によって検証を実現



$(lc_0=0)\wedge(lc_0'=1)\wedge(lc_1'=lc_1)\wedge$
$(turn'=turn)\vee\ldots$



NuSMVの実行例

```
MODULE main
VAR
  x: 0..15;
INIT
  x = 1 | x = 3
TRANS
     (x <= 3 & next(x) = x + 1)
  | (x > 3 & next(x) = x)

SPEC  -- xが4より大きくなることがある
  EF (x > 4)
SPEC -- いつかxが4未満になる
  AF (x < 4)
```

```
-- specification EF x > 4  is false
-- as demonstrated by the following execution
sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  x = 1
-- specification AF x < 4  is true
```

特徴

- 変数のレンジを最初に定める必要がある
  - 使われるブール変数が多くなればBDDも大きくなる
- キューなどのデータ構造は扱いが困難
  - 予めブール変数でエンコードする必要があるため
- 状態数の少ない多数のFSMからなる場合，非常に大きな状態空間を検証可能
  - デジタル回路
  - ステートチャート

C. SAT/SMT

- 充足可能性判定を用いたモデル検査
  - ブール値を持つ式を真にすることができるかどうかを判定
- FSM
  - BDDの方が速い
- プログラムコード
  - 小規模な逐次プログラムの全動作を検証可能
  - 並行プログラムは発展途上
  - モデル検査を超えた応用
    - 例. Concolic testing (Pexなど)

ソフトウェアモデル検査

- プログラム自体に適用

1. 充足可能性判定に基づく記号実行
   - CBMC
2. 抽象化
   - BLAST, SLAM
3. 仮想マシン上での実行 (in-situモデル検査)
   - Java Pathfinder, eXplode

まとめ

1. 基本的な言葉に注意する
   - 状態，遷移，遷移関係，状態系列
2. 検証したい性質の書き方
   - 時相論理を使う，使わない
     - 時相論理式が何に対して評価されるのか
3. どのモデル検査ツールを使うか？
   A) 明示的な状態探索　　SPIN, TLC
   B) BDDによる状態探索　NuSMV, SAL
   C) SAT/SMT　　CBMC