# 経営情報システム学特論１
# １３－１５．まとめ・補足・議論

SS専攻 経営情報システム学講座 客員
石川 冬樹
f-ishikawa@nii.ac.jp

# 目次

- 講義で学んだこと
- 補足・発展事項
  - 今ならわかるオートマトンの基礎理論
  - 一般的なプロセス代数
  - 状態爆発への対処
  - 実際の活用プロセス
  - 様々なモデル検査ツール

# まとめ： 並行システムのモデル化

- LTSAツールにおけるFSP（Finite State Process）言語を用いた並行システムのモデル化を学んだ
  - 制御フローの表現： 順次実行，選択（非決定的でもよい），再帰的な定義
  - 並行性の表現： 全プロセスの遷移の「かけ合わせ」によるシステム全体における遷移の合成
  - 並行性に関する制御の表現： 共有ラベルによる同期，優先度の定義
  - 部品化の表現： 外部と内部のラベル区別による隠蔽，ラベルの置換による複数部品の識別や部品間の接続

# まとめ（2）：並行システムの検証

- LTSAツールにおける特定キーワードおよび時相論理式を用いた並行システムの検証を学んだ
  - 安全性の基本的な検証：デッドロックやエラー状態に到達しないことの検証，指定されたプロセスの遷移と「同一」であることの検証
  - 到達性の基本的な検証：最終的に陥る無限サイクルに特定の遷移が起きる可能性がある（確率が偏っていなければ何度でも起きる）ことの検証
  - 論理式に基づいた検証：状態に対する述語の紐付け，時相論理のうちLTL（Linear Temporal Logic）を用いた検証式を与えての検証

# まとめ（3）：並行システムの実装

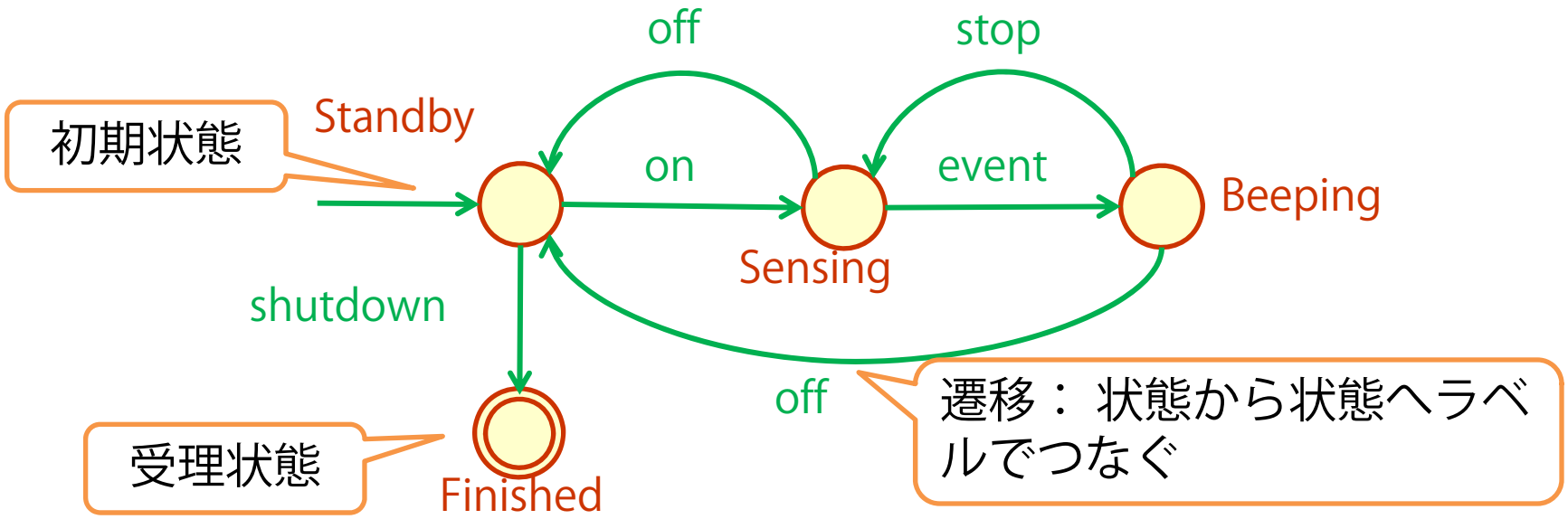
- Javaのマルチスレッドプログラミングにおける基本的な制御構造と典型的な記述パターンを学んだ
  - 単純な排他制御： synchronizedによる単純なロック確保・解放動作の実現
  - 同期制御： wait/notifyによる待ち合わせの実現と，待つ際の適切なロックの解放
  - 様々なライブラリと記述パターン： セマフォ，生産者と消費者におけるバッファ・キュー，読み書きロックの制御など

# 目次

- 講義で学んだこと
- 補足・発展事項
  - 今ならわかるオートマトンの基礎理論
  - 一般的なプロセス代数
  - 状態爆発への対処
  - 実際の活用プロセス
  - 様々なモデル検査ツール

# オートマトン： 基本定義

- オートマトン：
  - 全状態の集合（初期状態・受理状態の各集合）
  - 状態遷移のラベルの集合・状態間の有向遷移関係

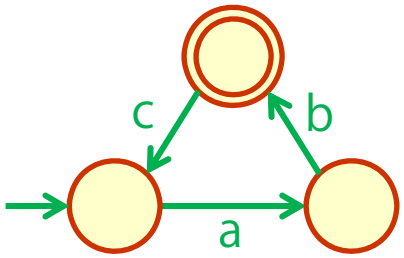



初期状態から受理状態に至るラベル列の例
on . event . stop . off . shutdown

# オートマトン： 基本定義

- オートマトンの言語（Language）
  - 言語（Language）： 受理状態に到達する語（Word, ラベル列）の集合
- 有限オートマトン： 状態が有限

  ※ 有限状態機械（Finite State Machine）とも
  - 有限オートマトンと正規表現は対応
    （任意の有限オートマトンの言語は，正規表現で表すことができ，また，任意の正規表現を受理言語とする有限オートマトンが存在する）

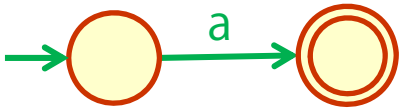



a b(cab)*

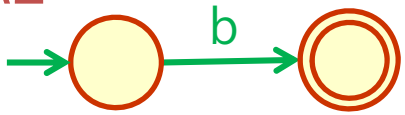
# オートマトン： 基本的な演算

- 有限オートマトンの和： 単に合わせる
- 受理言語の和集合： どちらかで受理

FA1



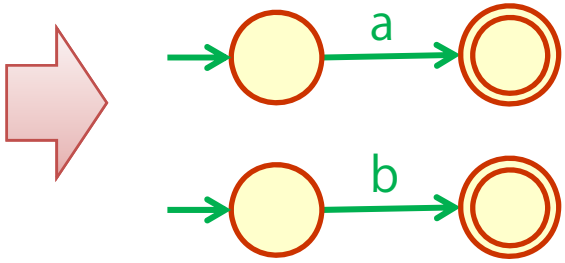

FA2




LANG(FA1) = {a}

LANG(FA2) = {b}

FA1+FA2

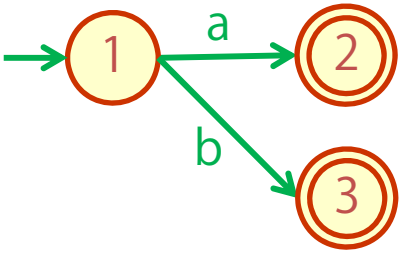



LANG(FA1+FA2) = LANG(FA1) ∪ LANG(FA2)
= {a, b}

# オートマトン： 基本的な演算

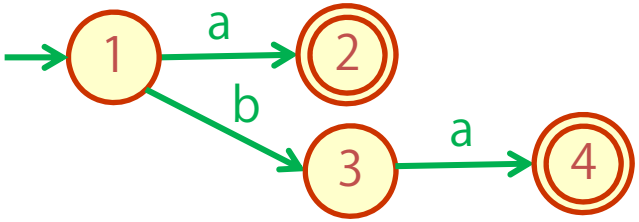
- 有限オートマトンの積： 両方で可能な遷移のみを抜き出す
- 受理言語の積集合： 両方に受理

FA1



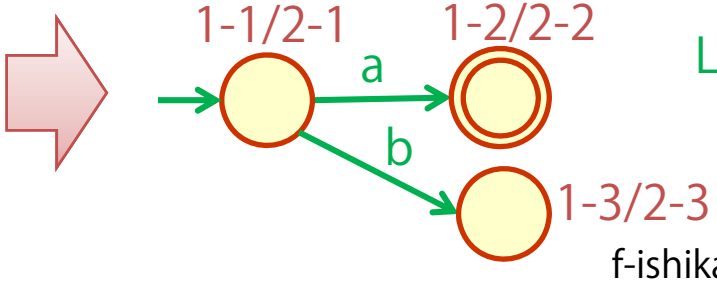

LANG(FA1) = {a, b}

FA2




LANG(FA2) = {a, ba}

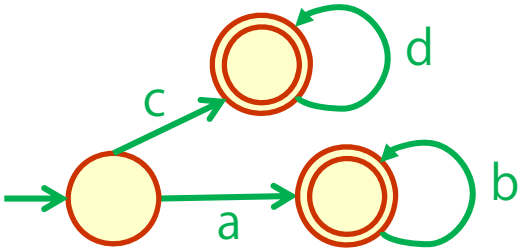
FA1×FA2




LANG(FA1×FA2) = LANG(FA1) ∩ LANG(FA2)
= {a}

# オートマトン：関連する拡張

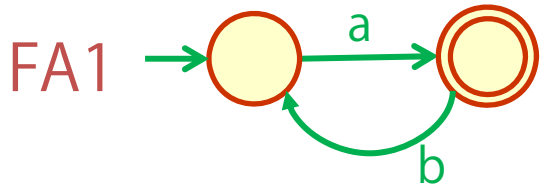
- Büchiオートマトン
  - 受理される語も無限長でよい：受理状態で「終わる」のではなく，受理状態を「無限回通る」語を，受理される語と定義する
  - 受理状態を通らない無限長の語に注目し，「いつも」「いつか」（起きる・起きない）という性質が議論できる
  - 正規表現に「無限回の繰り返しω（＊は有限回）」を導入したω正規表現と対応

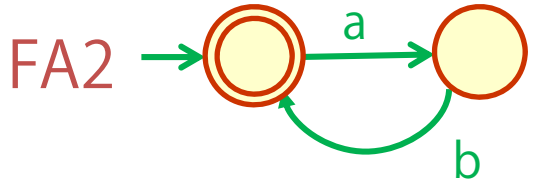



$ab^{\omega} + cd^{\omega}$

# オートマトン：関連する拡張

- Büchiオートマトンにおける積
  - 有限オートマトンのように定義しても意味がない
  - 例：下記は同じ言語を受理するが，FA1×FA2を作ると共通の受理状態がなく，受理言語は空集合



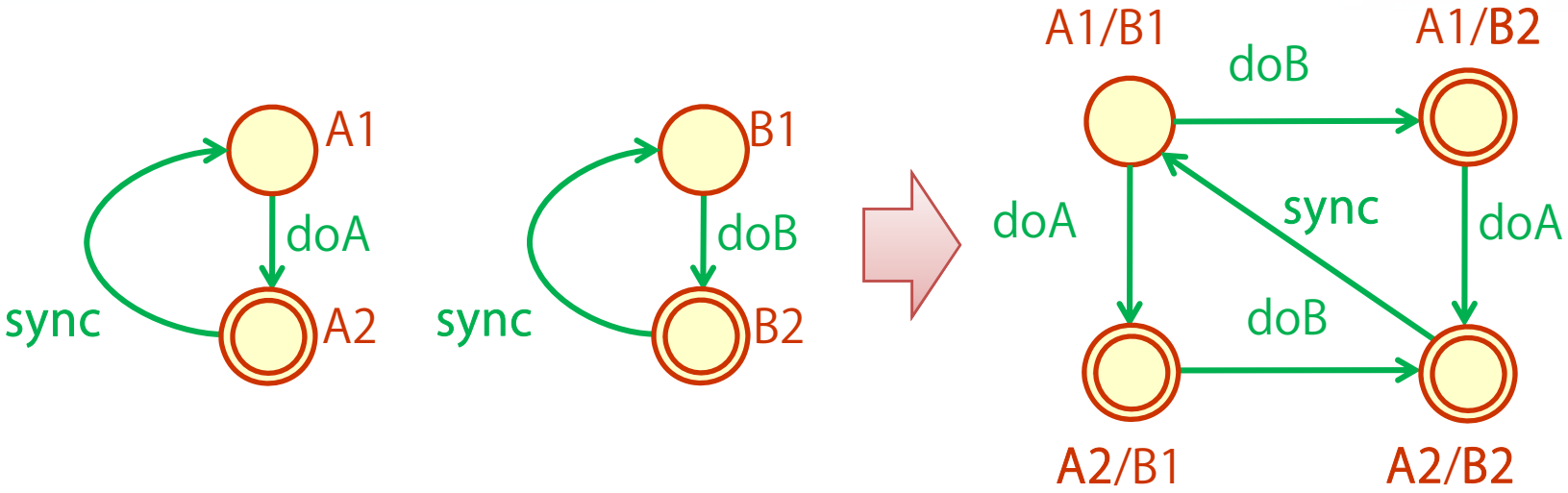




LANG(FA1) = {abababab…}　　LANG(FA2) = {abababab…}

  - 「受理状態にぴったり止まる」必要はなくなったので，受理状態を「揃える」意味もなくなっている
  - 「FA1由来の受理状態も無限回通るし，FA2由来の受理状態も無限回通る」というように受理の意味を定義すれば済む（拡張Büchiオートマトン・詳細略）

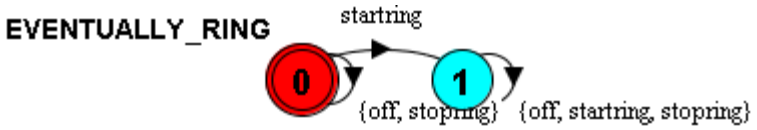
# オートマトン： 関連する拡張

- オートマトンにおける並行プロセスの表現
  - 共通のラベルを同期と見なす



# LTSAを振り返る

- 遷移がラベルで区別される状態遷移系（Labeled Transition System）
  - プロセス記述においては，受理状態は気にしない
  - 検証においては内部的に受理状態を定義し，それに基づき検証を行っていることも多く，前述の理論が活用されていることが垣間見える
  - 例： 第6回「ラベルstartingがいつか起きる」という論理式を検証
  - 下記オートマトンが作成され，元のオートマトンと合成した結果，受理状態を無限回通るサイクルがあるとすると，startringが起きない可能性があると判断

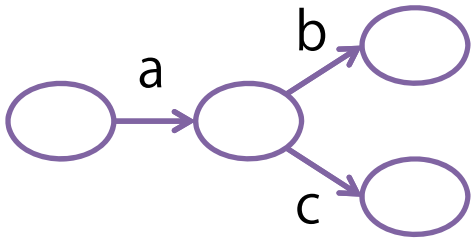


# 目次

- 講義で学んだこと
- **補足・発展事項**
  - 今ならわかるオートマトンの基礎理論
  - **一般的なプロセス代数**
  - 状態爆発への対処
  - 実際の活用プロセス
  - 様々なモデル検査ツール

# プロセス代数

- プロセス代数（プロセス計算・Process Algebra）
  - （相互作用する）並行プロセスの記述や分析・検証のための代数的な（記号による表現と記号に関する変形などを行う）記述
  - CSP（Communicating Sequential Process）が代表的で，π-calculusなどその発展も有名
  - Labelled Transition Systemが表現される




a . (b + c)

LTSAにおける
FSP言語だと
a -> (b | c)

# プロセス代数： 一般的な語彙

- 記法の違いはあるものの，下記のような語彙は共通的に用いられる

| | |
|---|---|
| *P* . *Q* | Sequential composition of *P* and *Q* |
| *P* \| *Q* | Parallel composition of *P* and *Q* |
| *P* + *Q* | Choice of P or Q |
| *c*<*x*> | Send *x* via *c* |
| *c*(*x*) | Receive into *x* via *c* |
| !*P* | Replication of *P* |
| 0 | Stop |

LTSAにおける
FSP言語では
直接扱わない
が表現可能

LTSAにおける
FSP言語では
扱っていない

# プロセス代数： Reduction

- Reductionと呼ばれる遷移の定義によって，記号上での遷移の表現が与えられる
  - 以下は可能なreductionの例

$$a.b.0 \xrightarrow{a} b.0 \xrightarrow{b} 0$$

$$!P \rightarrow P \mid !P \rightarrow P \mid P \mid !P$$

$$a.k(x).P \mid (b+c).\underline{k}\langle d\rangle.Q \xrightarrow[a]{} k(x).P \mid (b+c).\underline{k}\langle d\rangle.Q \xrightarrow[c]{} k(x).P \mid \underline{k}\langle d\rangle.Q \xrightarrow[k]{} P\{d/x\} \mid Q$$

# プロセス代数： 様々な表現力

- $\pi$-calculusの場合
  - 送受信に用いるチャンネル自身の送受信
  - ラベル・チャンネルが共有される範囲の明示

*カッコ内でのみ有効であり,*
*他のプロセスに知られない新しい秘密の名前*

new k ( $\overline{a}$<k> . k(x) . P ) | a(c) . $\overline{c}$<d>

→ new k ( $\overline{k}$(x) . P | $\overline{k}$<d> )

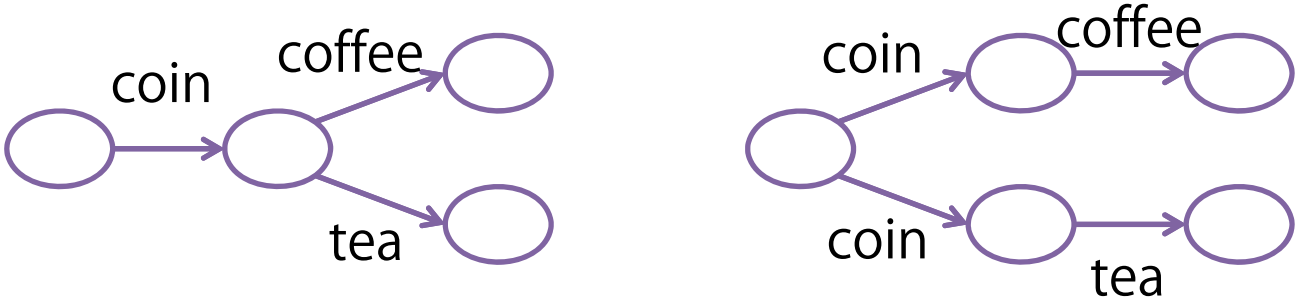
チャンネルaを通した
通信により秘密kが共有される

→ new k ( P{d/x} | 0 )

秘密のkをチャンネルとして用いた通信

# プロセス代数： 同一性判断

- 以下の自動販売機は「同じ」と考えるべきだろうか？（実用上，同じと見なしたいだろうか，異なると見なしたいだろうか？）
  - すべての状態が受理状態だと考え（まぁ最後だけでもよいが）オートマトンと見なすと，受理言語が同じなので同じと見なす

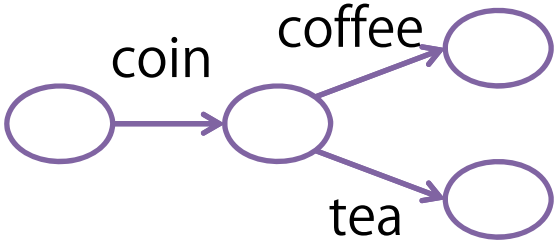


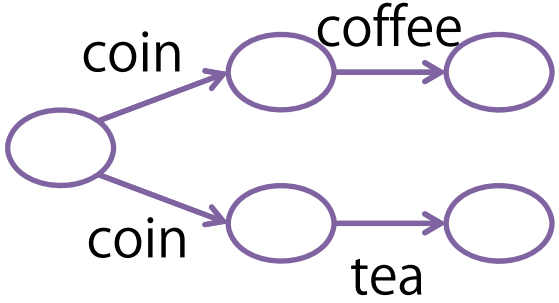
# プロセス代数： 同一性判断

- 初期状態からのラベル列だけ見ると，途中過程で発生する「選択肢」を識別できない

  コインを入れた後,

  - 左はコーヒーか紅茶か選べる
  - 右はコーヒーか紅茶かどちらかしか選べないように決まってしまっている



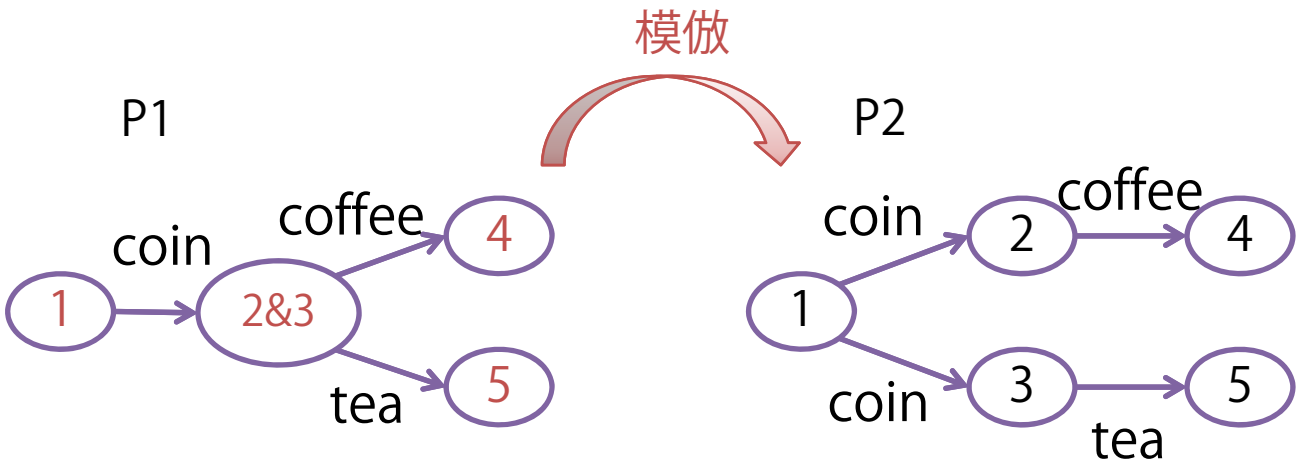



# プロセス代数： 同一性判断

- 模倣性（Simulation）
  - 定義： q simulates p if there exists S such that pSq, and:
    if $p \xrightarrow{a} p'$ then there exists $q' \in Q$
    such that $q \xrightarrow{a} q'$ and $p'Sq'$
  - 元のプロセスの各状態に対し，それに対応する状態が模倣するプロセスにあると見なすことができて
  - 元のプロセスにおいてある状態から可能な遷移に対し，模倣プロセスも対応する状態から遷移を行うことができ，遷移後も対応関係が維持される
- 双模倣性（Bisimulation）
  - これが双方向に成り立つ

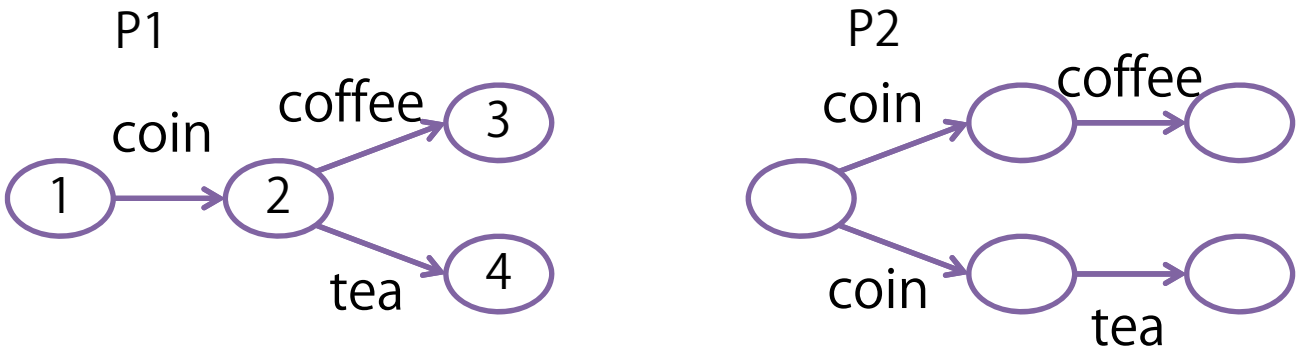
# プロセス代数： 同一性判断

- P1はP2を模倣しているか？
  - P1に付けた数字のように対応関係をとればよい
  - 例： P2における遷移「状態1からcoinで状態2へ」に対応するような遷移がP1にある：
    「状態1（P2の状態1に対応）からcoinで,
    状態2&3（P2の状態2に対応）へ」



# プロセス代数： 同一性判断

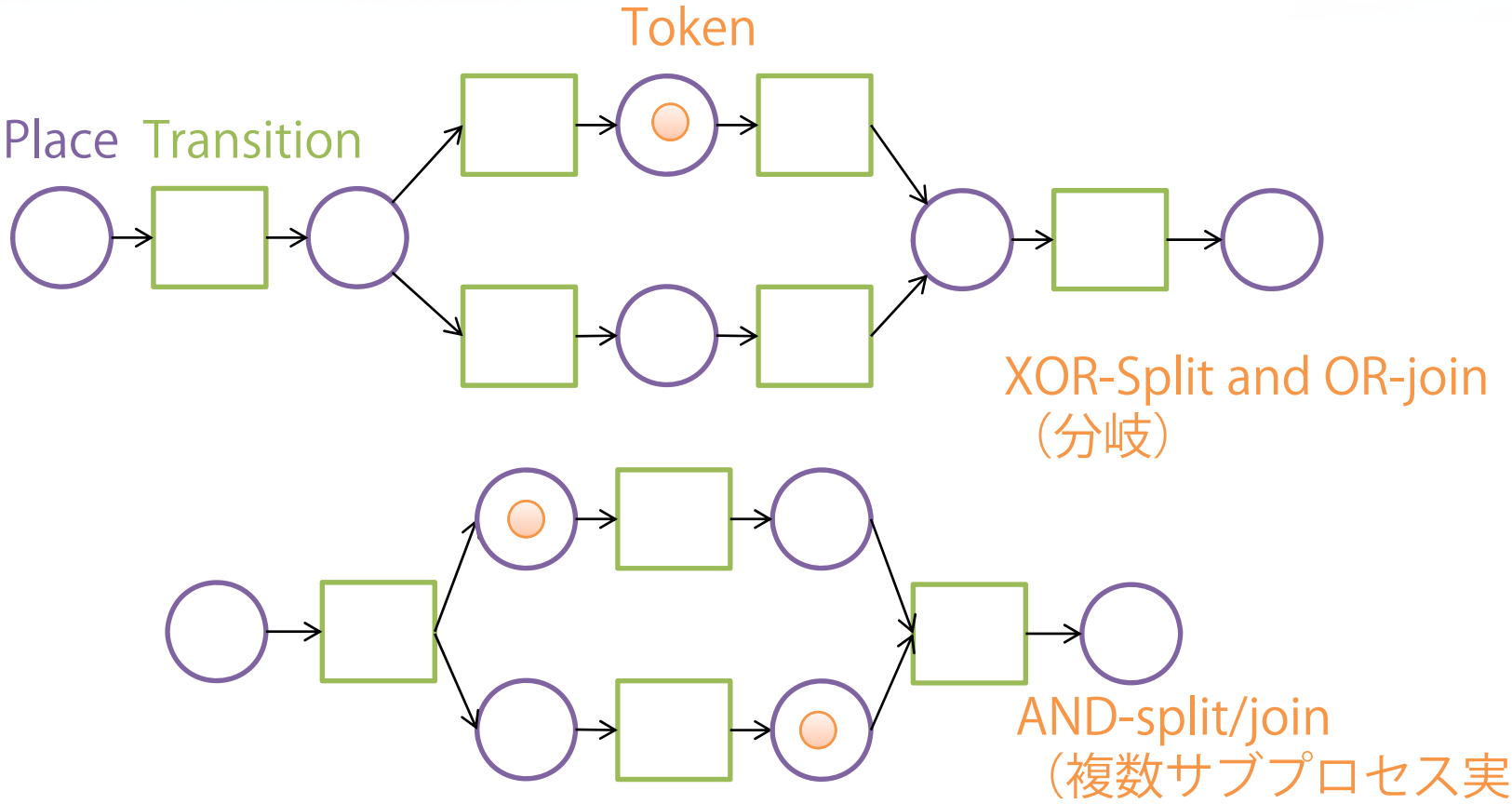
- 逆にP2はP1を模倣しているか？
  - coffeeもteaもできるP1の状態2に対し，それに対応する状態を選ぶことができないので模倣していない




  - これらは，双模倣ではなく，振る舞いが同一とは見なすべきでないと考えることができる
  - P1はP2を模倣するので，P1はP2の「代わりに使える」が，その逆はできないということ

# おまけ：ペトリネット

- ペトリネット：Tokenが流れる場として表現
  - Split/Joinが明示的に表現される



# 目次

- 講義で学んだこと
- **補足・発展事項**
  - 今ならわかるオートマトンの基礎理論
  - 一般的なプロセス代数
  - **状態爆発への対処**
  - 実際の活用プロセス
  - 様々なモデル検査ツール

# モデル検査の出力

- 改めて，モデル検査の出力は
  - エラーなし： 与えられたモデルがとりうる状態において，検証となる性質が成り立たない状況はない（ことが断言できた）
  - エラー発見： 与えられたモデルがとりうる状態において，検証となる性質が成り立たない状況（反例）が見つかった
  - 状態爆発： 実行時間あるいは使用メモリが，定められた限界を超えたが，すべての状態を探索することはできておらず，今のところ反例は見つかっていない

# 状態爆発への対処

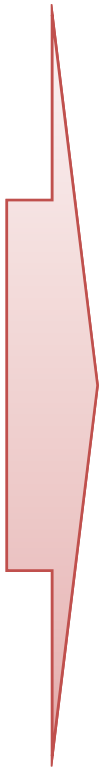
- 様々な手法やツール設定により状態削減
  - 検証シナリオの設定
    - 例： ユーザは２名に限る（実際は10,000人でも）
  - 様々な抽象化手法
    - 例： int型変数 x があり，条件分岐においては
      x>0，x=0，x<0 の3つの場合を考えている
      ⇨ x = {neg, zero, pos} と列挙型に変更（次頁）
  - 他にツールの様々な機能も
    - bit単位でうまくメモリに詰め込む工夫を使う
    - 対称動作や内部動作等は並び替えを網羅しない
      （「・・・→Aの内部動作3→Bの内部動作1→・・・」）

# モデル検査： 状態爆発への対処

x := 0

x > 0 ->

x++



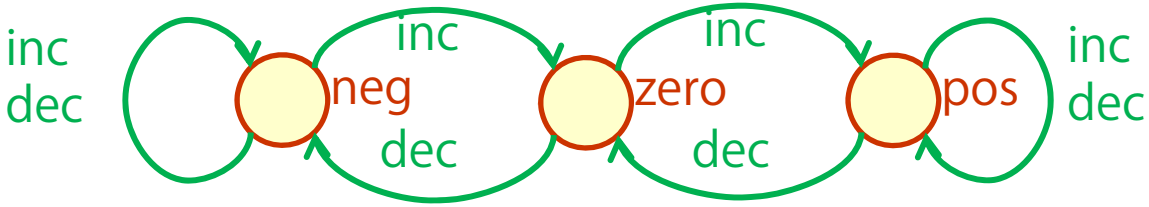
x := zero

x == positive –>

:: x == neg ->
  if
    :: x := neg
    :: x := zero
  fi
:: x == zero -> x = pos
:: else skip

# モデル検査： 状態爆発への対処

- 検証の意味が変わるのが難しい！
  - 例： ユーザは２名に限る
  - ➡ ３名が関わって初めて起きる不整合はない？
  - 例： int x を x = {neg, zero, pos} に
  - ➡ 起きうること・起きえないことが変わらない？




-1でも-100でも同じnegであり，1回のincでzeroになれる

*この例では抽象版のモデルでしか起きないことがあり，抽象版で起きないことは元のモデルでも起きない*

# 目次

- 講義で学んだこと
- **補足・発展事項**
  - 今ならわかるオートマトンの基礎理論
  - 一般的なプロセス代数
  - 状態爆発への対処
  - **実際の活用プロセス**
  - 様々なモデル検査ツール

# 実際の活用

- 講義では，LTSA版とJava版双方を扱う場合でも，双方は頭の中で対応づけ，「同様に」書いた
  - 概念的な説明のため
  - LTSA版では適宜モデル化（省略や，LTSAのために有限化など）している部分も
- もちろん「設計の考え方に対する確信度を高める」ためにこの進め方をとることもできる
  - が，対応関係が保証されているわけではなく，モデルを検証してもコードがそれとずれているかも
  - 同じロジックを二種類書くことを，「時間をかけるべき整理」と感じるか，「二度手間」と感じるか

# 実際の活用

- 典型的なツールの方向性（１）：
  形式モデルからコードを生成
  - 「条件が成り立つまでブロッキング」といった処理などは，自動で生成してしまえばよい
  - JCSPと呼ばれるツールでは，CSP記述からJavaコードを生成可能
  - 生成するのは，排他制御や同期制御などのみに関連するコードのスケルトンと見なし，並行制御に関係ない部分を埋めていくというのも現実的

# 実際の活用

- 典型的なツールの方向性（２）：
  コードから形式モデルを抽出
  - 現在一般的なプロセスにおいて使いやすい（「コードを書く前に設計モデル検証の時間を追加しよう」よりも）
  - 検証の目的に応じて状態爆発を防ぐため，本質のみを抜き出すことが必要
    - 例： 特定変数に関係する行だけを抜き出す
    - 例： 特定APIの呼び出し順序（例えば，ファイルのopen/write/close）だけ見る
    - 例： ライブラリやOS部分などをモックにする

# 実際の活用

- 典型的なツールの方向性（２‘）：
  コードを網羅的に検証
  - 特に抽出などせずコードの実行を網羅することも考えられる
  - より状態爆発が起こりやすくなり，探索するステップ数に上限を設けるなどする（有界モデル検査）
  - コードを1ステップずつ実行，バックトラックなどを行っていくような実装をしたツールもある

# 実際の活用

- 典型的なツールの方向性（３）：
  実用的な（準形式）モデルから形式モデルを抽出
  - UMLステートチャート図，ビジネスプロセス記述言語（BPMNなど）から，モデル検査ツールの入力に変換する
  - 皆が知っているモデルで書いた設計モデルを検証できる
  - ただし，厳密化・検証補助のために文法を拡張することもある
  - 言語によっては，意味論が厳密に定義されていないので，ツールを作る人の解釈になる

# 目次

- 講義で学んだこと
- 補足・発展事項
  - 今ならわかるオートマトンの基礎理論
  - 一般的なプロセス代数
  - 状態爆発への対処
  - 実際の活用プロセス
  - 様々なモデル検査ツール

# 他のモデル検査ツール

- 「モデル検査」と呼んでいる場合，先のオートマトンに基づくもの，時相論理に基づくものが多い
  - 基本的には同様
  - 送受信を言語として扱っていることも多い
    （LTSAでは同期ラベルだけがあり，「送受信」として読み手の人間が受け取ると考えた）

# 他のツール： SPIN

- SPIN（ツール名）
  - 記述言語はPROMELA（Process Meta Language）と呼ばれる
    - 変数，#defineなど，C言語に近い記述
    - 送受信チャンネル，送受信動作などの語彙が用意されており，通信プロセスの記述が行いやすい
  - SPINは，検査を行う際にその都度，その検査を効率的に行うためのCプログラムを出力する
    - それをコンパイル，実行する

http://spinroot.com/spin/whatispin.html

# 他のツール： SPIN

```
#define SIZE 4

byte msg[SIZE];

chan s2r = [2] of {byte};

proctype Sender() {
  byte i;
  do
    :: i == SIZE -> break;
    :: else -> s2r ! msg[i];
                i++;
  od
}
```

4個のデータを送信してみる

通信チャンネル（有限バッファ長）や送受信に関する語彙を用意

送信

この例は決定的な分岐だが，非決定的な振る舞いを書く

```
proctype Receiver() {
  byte j;
  byte rmsg;
  do
    :: j == SIZE -> break;
    :: else -> s2r ? rmsg;
       assert (rmsg == msg[j]);
       j++;
  od
}

proctype Lost() {
  byte drop;
  do
    :: s2r ? drop;
  od
}
```

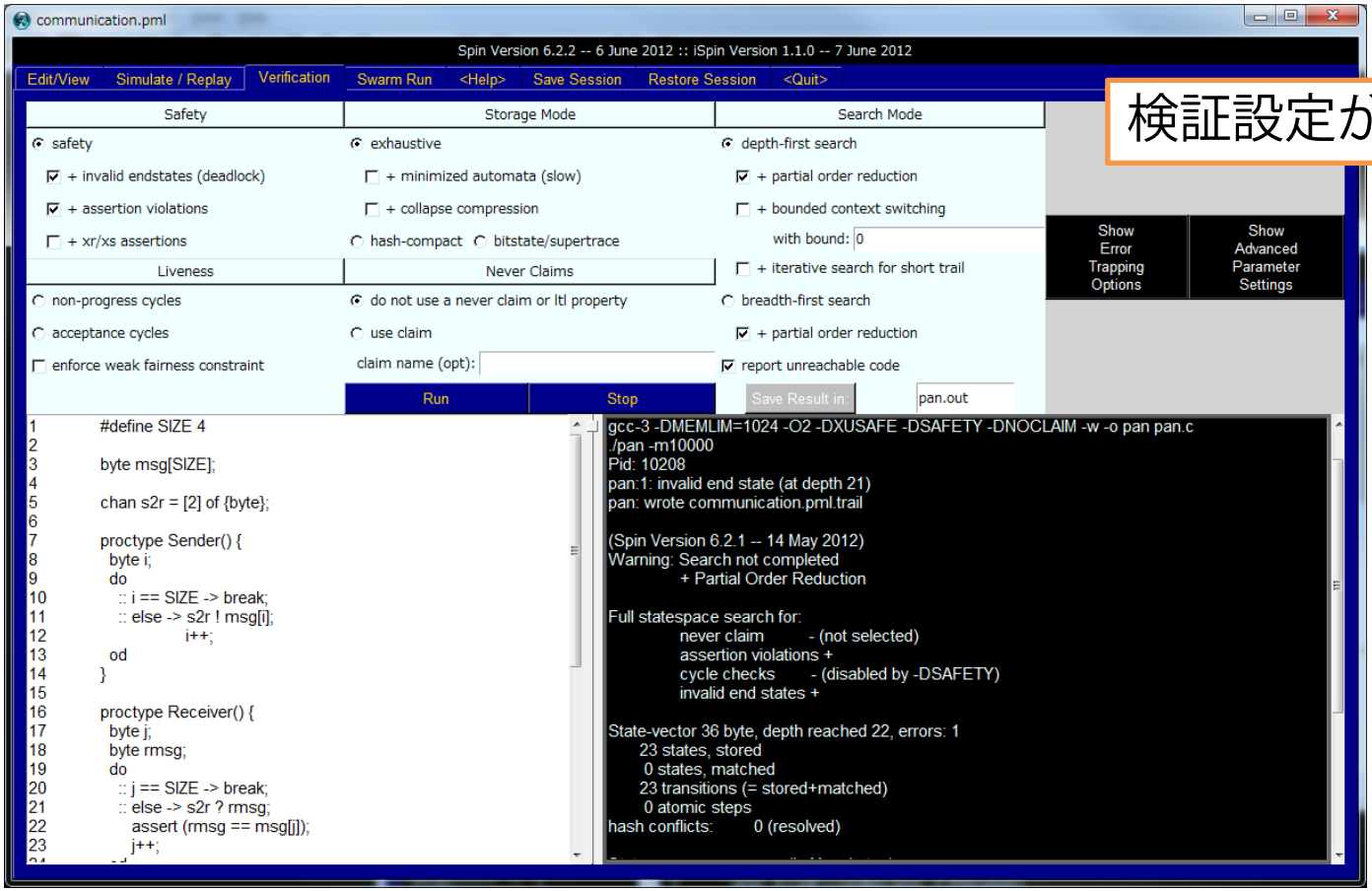
受信

順番通りに来ているかをアサーションで検証（成り立たない）

メッセージを別のプロセスも受信しうるとして通信失敗の可能性を表現

# 他のツール： SPIN

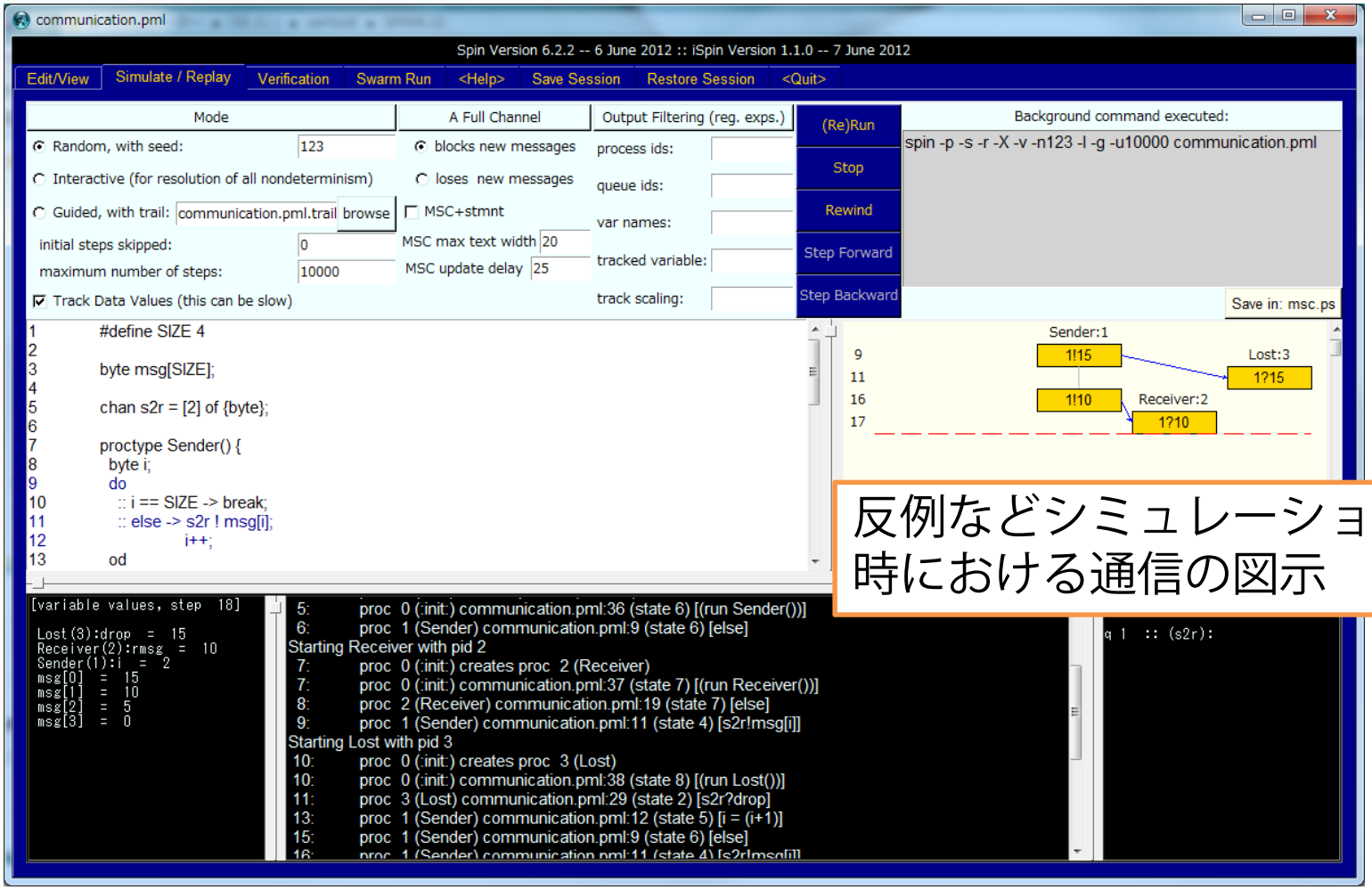
- SPIN自体はC言語を出力するツールでしかなく，いくつかのGUI実装がある（以下はiSPIN）




検証設定が多彩

# 他のツール： SPIN

- 続（以下はiSPIN）




反例などシミュレーション時における通信の図示

# 他のツール： SMV

- SMV（ツール名）
  - 最近の実装はNuSMVと呼ばれる
  - 複数の変数値が1ステップごとに値を変えていく様を記述
  - LTLではなくCTLの検証（第6回参照）

http://nusmv.fbk.eu/

# 他のツール： SMV

```
MODULE main
VAR
  cabin : 0 .. 3 ;
  dir : { up, down }


ASSIGN
  next(cabin) := case
                    dir = up & cabin < 3        : cabin + 1 ;
                    dir = down & cabin > 0      : cabin – 1 ;
                    1                           : cabin ;
                 esac
  next(dir)     := case
                    dir = up & cabin = 2        : down
                    dir = down & cabin = 1      : up ;
                    1                           : dir ;
                 esac
```

単純に上下を行き来する
エレベータの動きを表現
（実際はともかく3Fまで）

次の状態における各変数の
値をどう決めるか，という
観点で状態遷移を記述

# 他のツール： UPPAAL

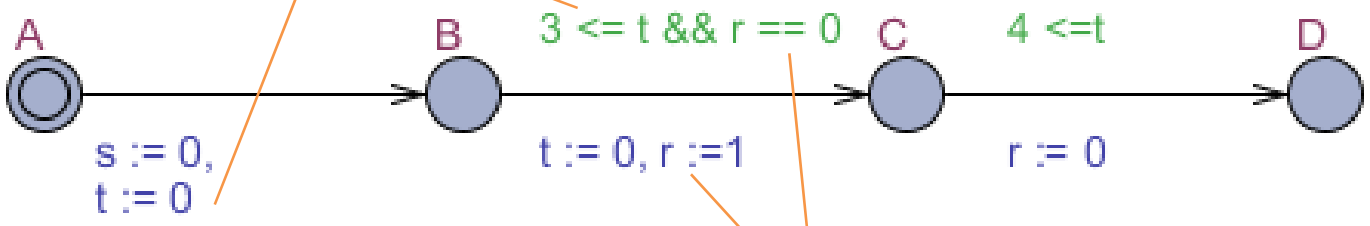
- UPPAAL（ツール名）
  - 時間オートマトンを扱うことができる（実時間モデル検査・timed model checking）
    - 所要時間（状態遷移が発生する時間の下限や上限）をオートマトンに付加
  - 時間CTLの検証を行う（第6回で述べたCTLに，時間に関する制約を加えたもの）
  - 実時間を扱うため，状態爆発がより起きやすく，許される検証式の形式も限られている（AGやEFの任意の入れ子などはできない）

http://www.uppaal.org/

# 他のツール： UPPAAL

状態B到達時に時間変数
（時間カウンター）tをリセット
→ Bを出てCに進むときには
時間が3以上経過している

所要時間（のぶれの可能性）
も含めて状態遷移を記述




共有資源が確保できる（r == 0）ときのみ状態Cに進め,
その際には確保（ r := 1 ）

# 他のツール： PRISM

- PRISM（ツール名）
  - 確率オートマトンを扱うことができる（確率モデル検査・probabilistic model checking）
    - 状態遷移が発生する確率をオートマトンに付加
  - 確率付きCTLなどの検証を行う（発生確率に関する性質）
  - 基礎理論的にはマルコフ過程などの領域になる

http://www.uppaal.org/

# 他のツール： PRISM

```
module client
  // 状態 (1:リクエスト送信中，2:結果通知受信中，3:成功通知受信・終了，
  //       4：失敗通知受信・終了，5:タイムアウト，6:結果不明・終了 )
  t : [1..6] init 1;
  retried : [0..1] init 0;

  [comm_request] t=1 -> (t'=2);
  [comm_success] t=2 -> 0.9:(t'=3)+0.1:(t'=5);
  [comm_failure] t=2 -> 0.9:(t'=4)+0.1:(t'=5);
  [exit_success] t=3 -> (t'=3);
  [exit_failure] t=4 -> (t'=4);
  [retry] t=5&retried=0 -> (t'=1)&(retried'=1);
  [giveup] t=5&retried=1 -> (t'=6);
  [exit_unkown] t=6 -> (t'=6);

endmodule

// サーバー側が成功するがクライアント側は不明となる確率
// P=? [ ( F s=3 ) & ( F t=6 ) ]
```

# ソフトウェアモデル検査

- プログラムコードに対して直接モデル検査を行う場合，ソフトウェアモデル検査と呼ばれる
  - SPINなどに変換するか，そのまま1ステップずつ実行し探索する
- 例： CBMC
  - Cコードに対してモデル検査を行うツール

  http://www.cprover.org/cbmc/
- 例： Java PathFinder
  - Javaコードに対してモデル検査を行うツール

  http://javapathfinder.sourceforge.net/