# システムソフトウェア

2020年度

第8回 (11/2)

月曜7-8限・木曜7-8限(Zoom)

講義担当：渡部卓雄 (Takuo Watanabe)

http://titech-os.github.io

e-mail: takuo∅c.titech.ac.jp

# 本日のメニュー

- 相互排除プログラムの検証

# 相互排除アルゴリズムの検証

- 相互排除アルゴリズムの正しさ
  - 安全性(safety)
  - 活性(liveness)
- 検証の方針
  - 複数のスレッドからなるシステムをオートマトンとしてモデル化する.
    - 各スレッドもオートマトンとしてモデル化
    - システム全体はそれらの合成
  - そのオートマトンで受理可能な全ての動作が所定の性質をみたすことを網羅的検査によって示す.

# 例：アルゴリズム0の場合(1)

```
while (true) {
  p0: NC
  p1: while (in_use);
  p2: in_use = true;
  p3: CS
  p4: in_use = false;
}
```

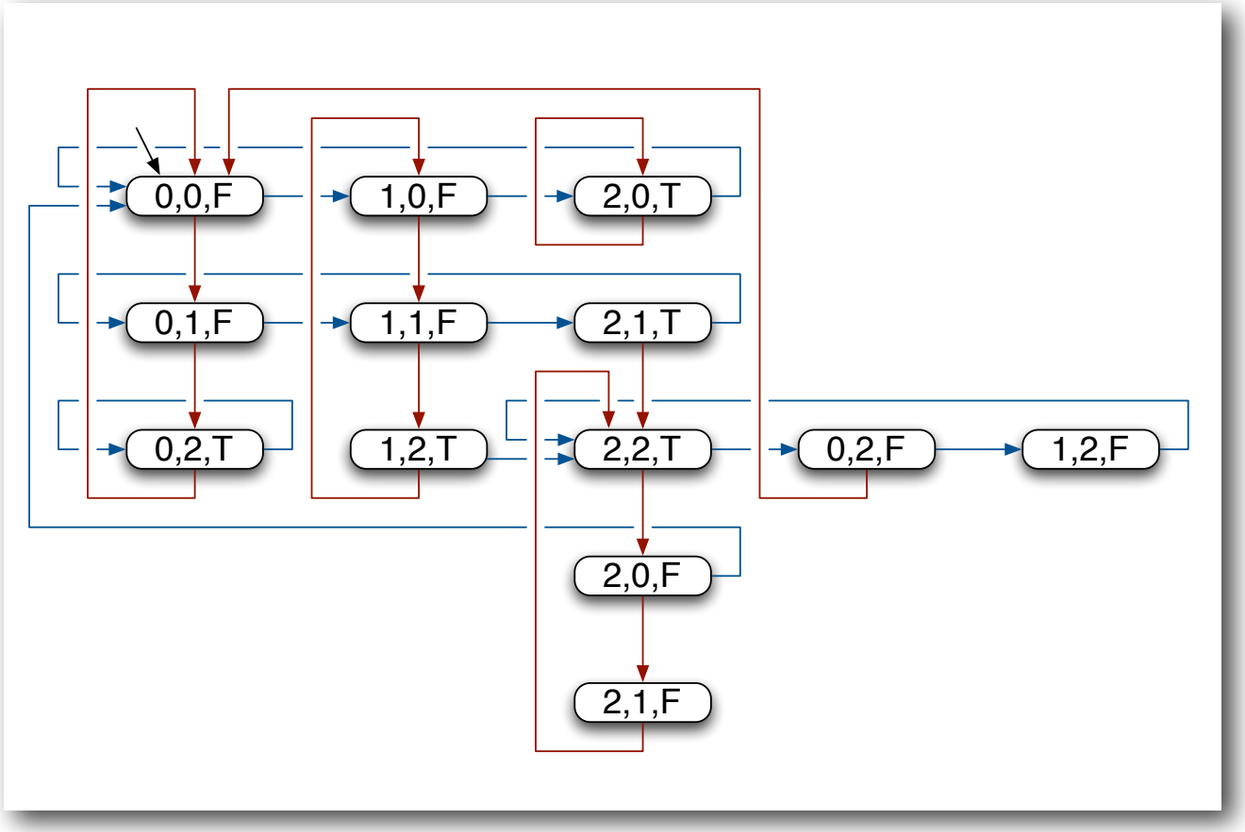
```
while (true) {
  q0: while (in_use);
  q1: in_use = true;
  q2: in_use = false;
}
```

簡単化

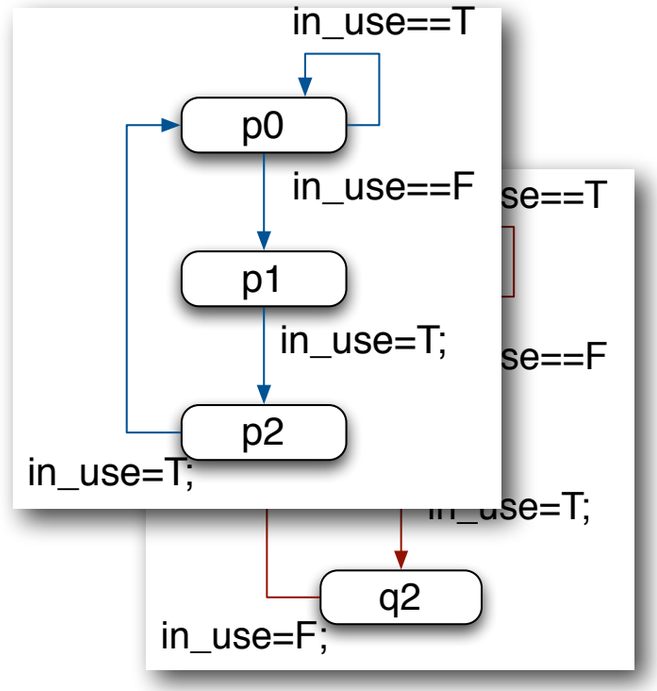
```
while (true) {
  p0: while (in_use);
  p1: in_use = true;
  p2: in_use = false;
}
```

```
while (true) {
  q0: while (in_use);
  q1: in_use = true;
  q2: in_use = false;
}
```

オートマトンで表現

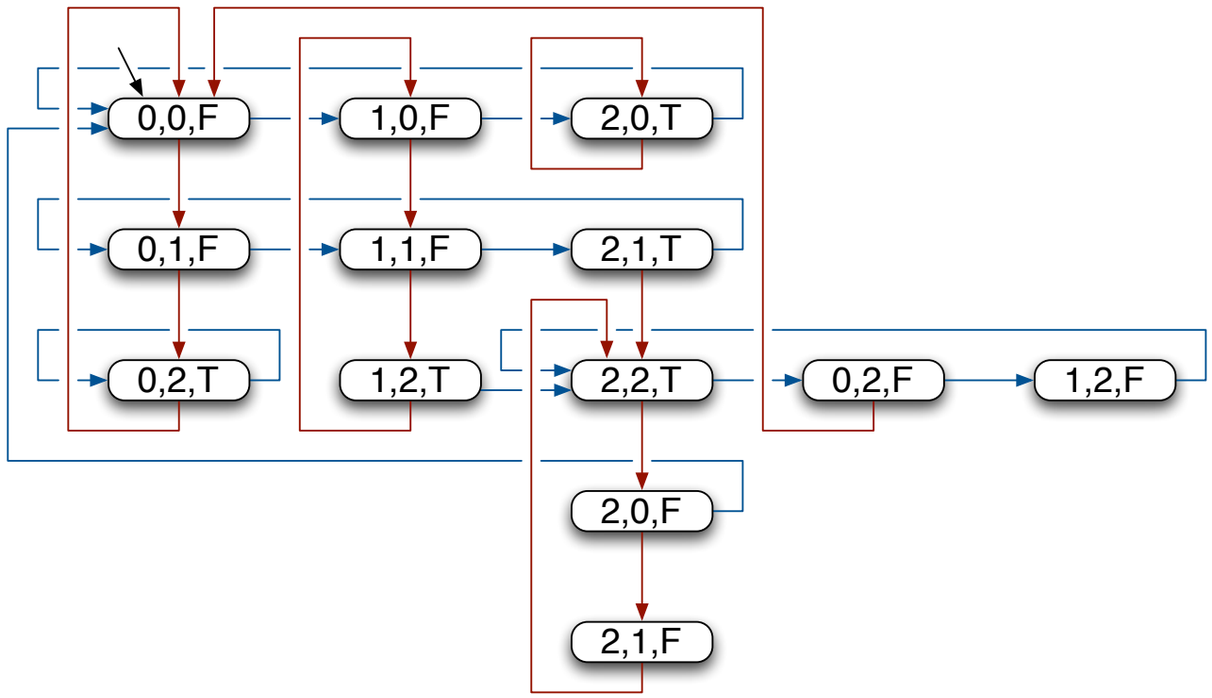



合成



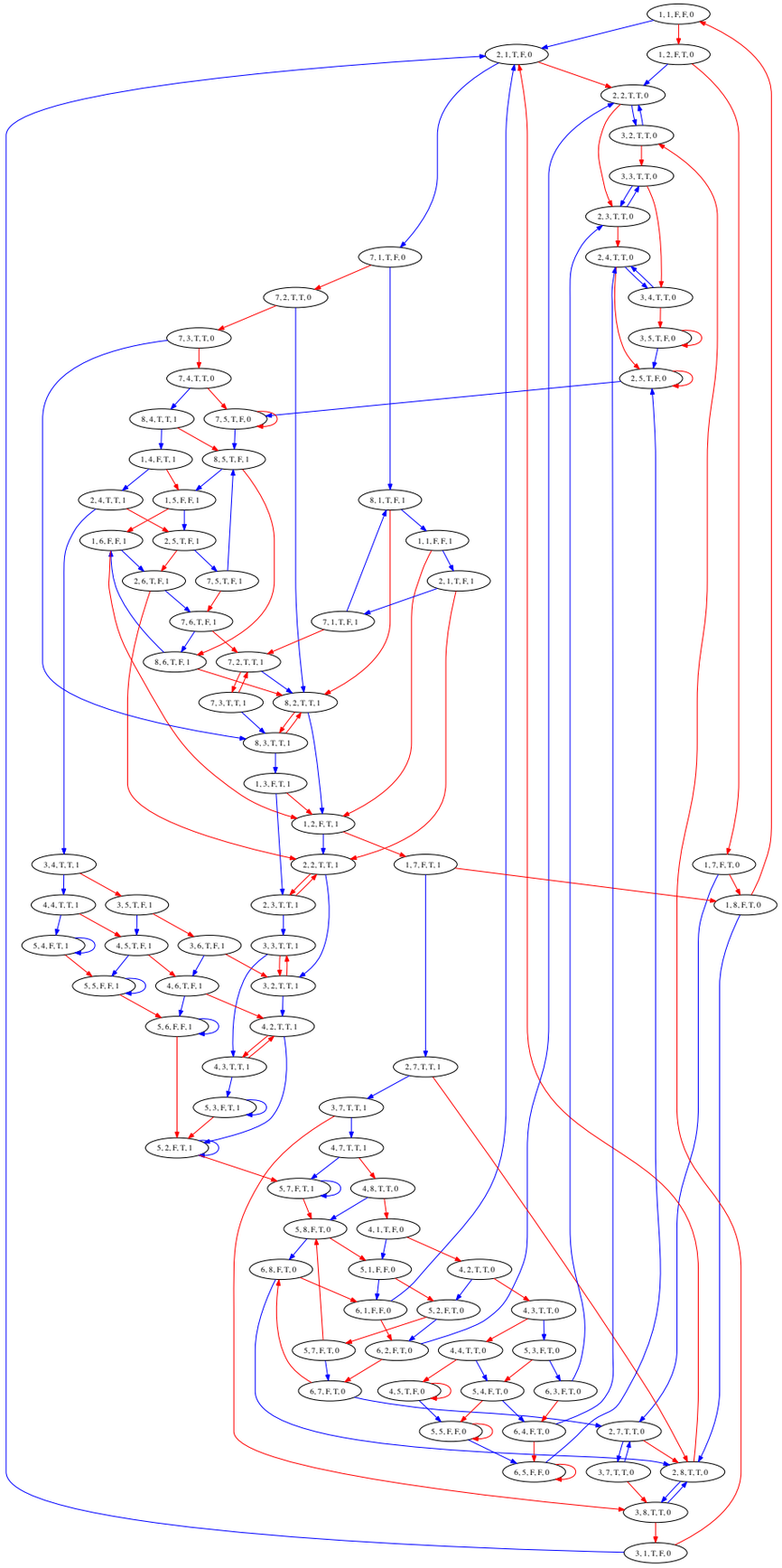
# 例：アルゴリズム0の場合(2)

安全性の検証：2つのスレッドが同時にCSに入れないことを網羅的探索によって示す．具体的には，両スレッドがCSに入っている状態(反例)への経路を探索し，みつからなければよい．



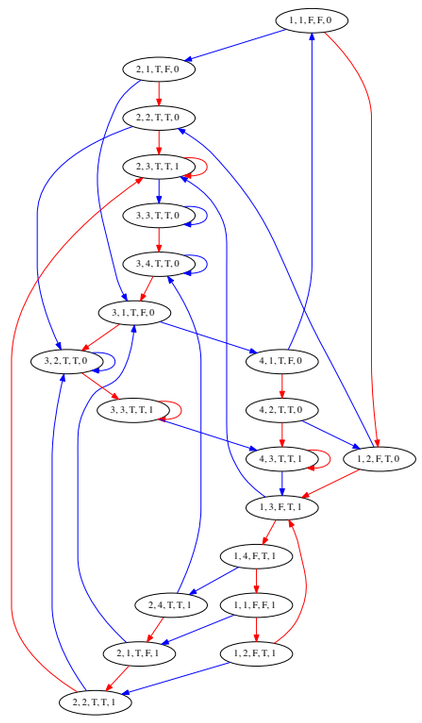

この例では，
(0,0,F)→(1,0,F)→(1,1,F)→(2,1,T)→(2,2,T) という反例が見つかる．つまり安全性は成り立たない．

# Dekker/Petersonのアルゴリズムの場合




←Dekkerのアルゴリズム

↓Pertersonのアルゴリズム



# 検証の自動化

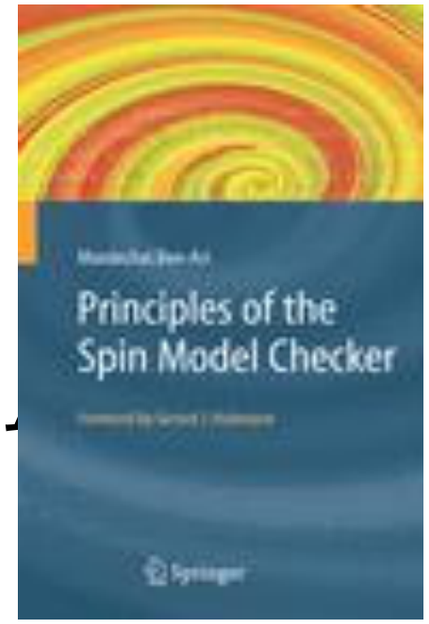
- プロセスをモデル化したオートマトンの作成・合成や網羅的探索を人力でやるのは大変
  - 以前レポートでDekkerのアルゴリズムのオートマトンを手書きしてきた人がいた
  - 採点は大変だったが，ちゃんと合っていた
- こういった作業は自動化できるはず.
- 実際にそのためのツールが存在する.
  - モデル検査(model checking)ツール
- 実際に，モデル検査ツールはハードウェアやソフトウェアの検証に用いられている.

# モデル検査ツール Spin

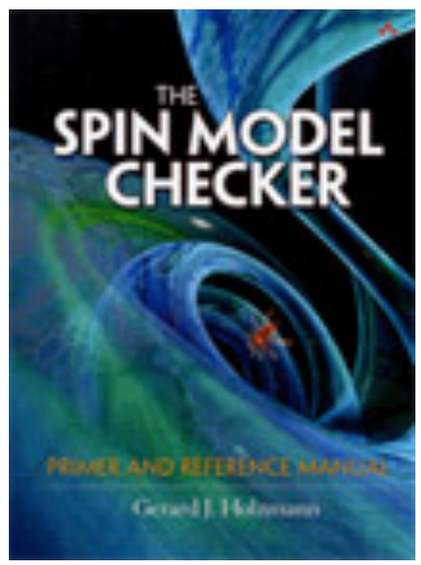
- 現在最も広く用いられているモデル検査ツールで，G. J. Holzmann 博士らによって1980年代より開発されている.
  - Spinオフィシャルサイト
    - http://spinroot.com
  - マニュアルの日本語訳
    - http://www.asahi-net.or.jp/~hs7m-kwgc/spin/Man/Manual_japanese.html

# マニュアル・参考書


- SPIN モデル検査: 検証モデリング技法
  - 中島震, 近代科学社, 2008.
- Principles of the Spin Model Cheker
  - M. Ben-Ari, Springer-Verlag, 2008.
  - 邦訳: SPINモデル検査入門, 中島震他訳, オー
    2010.




- The Spin Model Checker: Primer and Refernce Manual
  - G. J. Hozlmann, Addison-Wesley, 2003.



# インストール

- バイナリ (Linux, Mac, Windows)
  - http://spinroot.com/spin/Bin
    - 実行形式を適当な場所(/usr/local/bin等)に置くだけ
- Homebrew
  - `brew install spin`

- 本スライドで用いる例題
  - https://github.com/titech-os/spin_tutorial

```
git clone https://github.com/titech-os/spin_tutorial.git
```

# Dockerでの実行

- Spinでの検証に必要なものを集めたDockerイメージを用意してある

```
$ docker pull wtakuo/spin-env
$ cd spin_tutorial
$ docker run -it --rm -v $(pwd):/home/spin/tutorial wtakuo/spin-env
```

- 起動後の作業例

```
spin@cb117e5bb2af:~$ cd tutorial
spin@cb117e5bb2af:~$ spin hello.pml
...
spin@cb117e5bb2af:~$ spin -a alg0_2.pml
spin@cb117e5bb2af:~$ gcc -o pan pan.c
spin@cb117e5bb2af:~$ ./pan
```

# モデリング言語 Promela

- Spin でシステムの検証を行う際に，システムのモデルを記述するための言語
  - (少しだけ)Cに似た構文

hello.pml

```
active proctype Foo () {
  printf("Hello, World.\n");
}
```

# Promelaプログラムの実行

## (シミュレーション実行)

```
$ spin hello.pml
      Hello, World.
1 process created
$ spin -p hello.pml
  0:    proc  - (:root:) creates proc  0 (Foo)
      Hello, World.
  1:    proc  0 (Foo:1) hello.pml:2 (state 1) [printf('Hello, World.\\n')]
  1:    proc  0 (Foo:1)       terminates
1 process created
```

# 2つのプロセスの例

pqgreet.pml

```
active proctype P () {
  printf("P: Hello!\n");
  printf("P: This is P.\n");
}

active proctype Q () {
  printf("Q: Hello!\n");
  printf("Q: This is Q.\n");
}
```

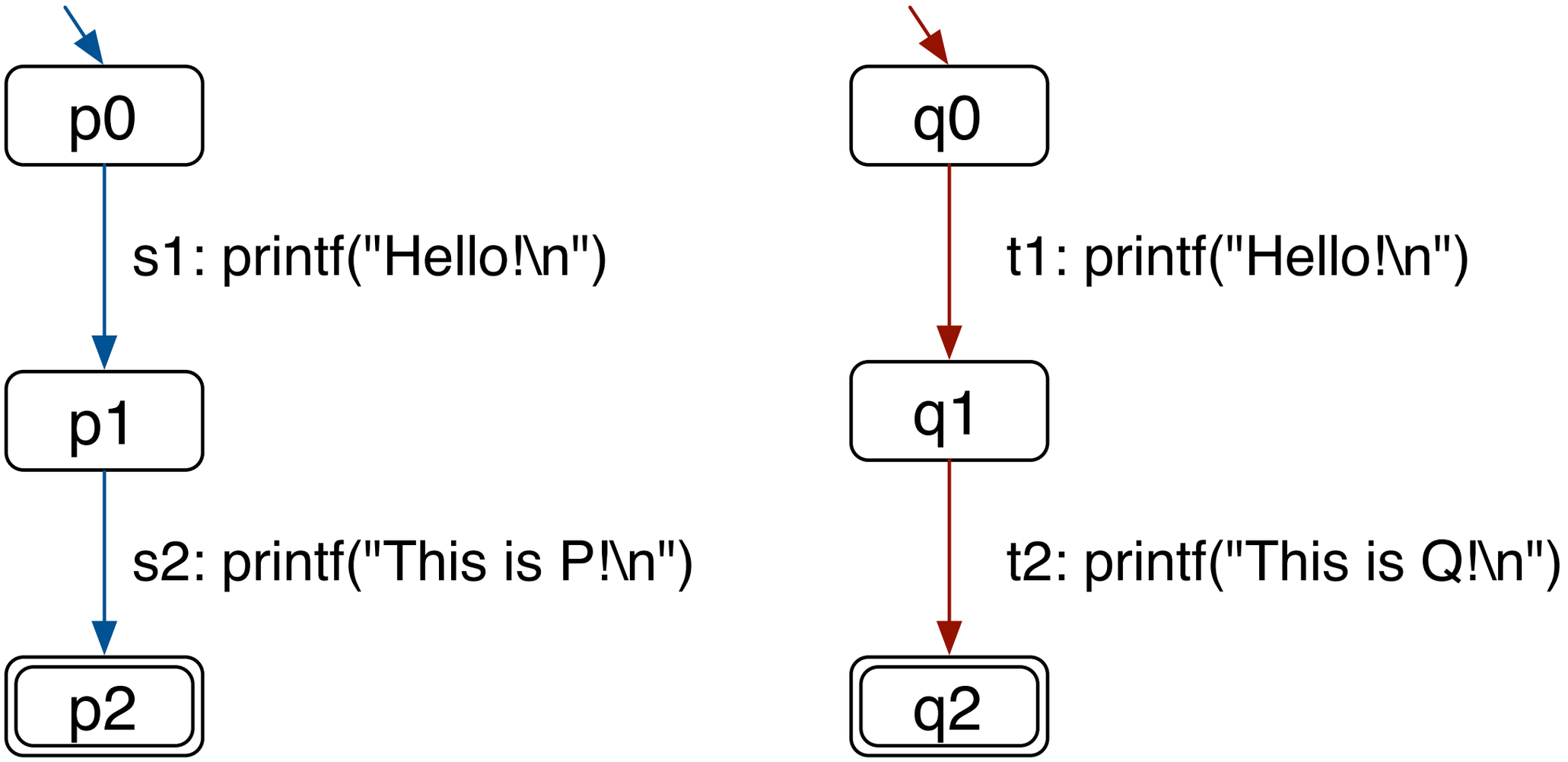
# pqgreet.pml の実行例

```
$ spin pqgreet.pml
        P: Hello!
            Q: Hello!
        P: This is P.
            Q: This is Q.
2 processes created
$ spin pqgreet.pml
        P: Hello!
            Q: Hello!
            Q: This is Q.
        P: This is P.
2 processes created
$ spin pqgreet.pml
            Q: Hello!
        P: Hello!
        P: This is P.
            Q: This is Q.
2 processes created
```

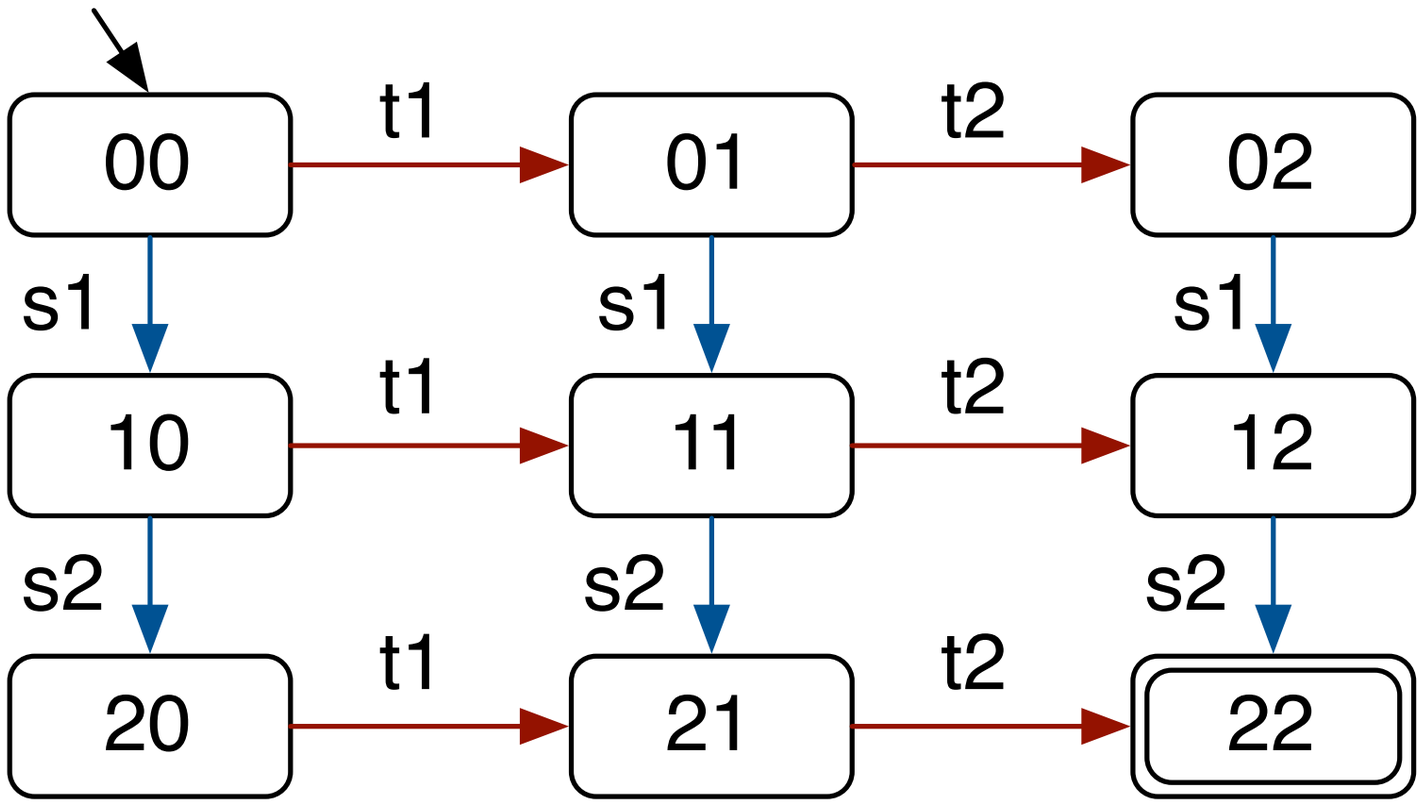
起動する毎に異なる出力が得られている．これは実行可能な文をランダムに選んでいるため．このような実行をシミュレーション実行という．

-p オプションを付けると実行した文を表示する．

# プロセス P, Q


p0
s1: printf("Hello!\n")
p1
s2: printf("This is P!\n")
p2
q0
t1: printf("Hello!\n")
q1
t2: printf("This is Q!\n")
q2

# P, Q の合成




可能な実行の列は s1s2t1t2, s1t1s2t2, s1t1t2s2, t1s1s2t2, t1s1t2s2, t1t2s1s2 の6通り．Spinのシミュレーション実行ではこれらをランダムに選んでいる．

# アルゴリズム0のPromera記述

alg0_1.pml

```
/* shared variable */
bool in_use = false;

active proctype P () {
  do
    :: printf("P: NC\n");
       in_use == false;
       in_use = true;
       printf("P: CS\n");
       in_use = false;
  od
}

...
```

# alg0_1.pmlの実行例

```
$ spin alg0_1.pml
        P: NC
             Q: NC
             Q: CS
        P: CS
             Q: NC
        P: NC
             Q: CS
             Q: NC
        P: CS
        P: NC
        P: CS
             Q: CS
             Q: NC
        P: NC
             Q: CS
        P: CS
^C
```

シミュレーション実行を行うと，2つのプロセスが同時にCSに入っている状態があることがわかる.

-p -g オプションを付けると，実行した文と共有変数の値を表示する.

# Spinの基本データ型

| | | |
|---|---|---|
| bit / bool | ビット(1bit) | 0～1 (false/true) |
| byte | 符号なし8bit整数 | 0～$2^{8}$-1 |
| short | 符号付き16bit整数 | -$2^{15}$ .. $2^{15}$-1 |
| int | 符号付き32bit整数 | -$2^{31}$ .. $2^{31}$-1 |

# 変数宣言

```
/* shared variables */
int count1 = 0, count2 = 0;

active proctype P () {
  /* local variables */
  int x, y;
  x = count1;
  y = count2;
  count1 = x + 1;
  count2 = y + 1;
  printf("count1=%d\n", count1);
}
```

共有変数と局所変数の宣言

# 代入文

- Cと似た構文：変数 = 式;
- 代入文はアトミックに実行される．したがって，cが共有変数の場合，c = c + 1; という文を実行することでcの値をアトミックに1増やすことになる．
  - c++; という文も同様.
- 共有変数の read-modify-write をアトミックに実行したくない場合は，一旦局所変数に代入する.
  - 例：c = c + 1; における共有変数cの読み出しと書き込みを非アトミックに行う場合は，局所変数 x を用いて x = c; c = x + 1; のようにする.

# 実行可能性(1)

- Promelaでは，条件式は文として扱われる.
  - 条件が成り立っているとき，その文は実行可能(executable)であるという.
  - 条件が成り立っていない場合，その文を実行しようとしたプロセスは条件が満たされるまでブロックされる.
- 例えば in_use == true という文は，変数 in_use の値が true (=1) でない限り実行可能ではない.
- 代入文，printf文はいつでも実行可能.

# Cとの比較

C

```
void P (void) {
  while (true) {
    printf("NC\n");
    while (in_use == true);
    in_use = true;
    printf("CS\n");
    in_use = false;
  }
}
```

Promela

```
active proctype P () {
  do
    :: printf("NC\n");
       in_use == false;
       in_use = true;
       printf("CS\n");
       in_use = false;
  od
}
```

# 条件文

```
if
  :: S00; S01; ... ; S0k1
  :: S10; S11; ... ; S1k2
        ...
  :: Sn0; Sn1; ... ; Snkn
fi
```

各節の先頭の文(ガードと呼ぶ) $s_{i0}$ (i=0...n)の中に実行可能なものがあれば，そのうちのひとつを選んで $s_{i0}$, $s_{i1}$, ..., $s_{iki}$ を実行する.

```
if
  :: n >= 0; s1; s2; ...
  :: n <= 0; t1; t2; ...
fi
```

複数のガードが実行可能なとき(この例ではn==0の場合)は，そのうちどれかが非決定的に選ばれる.

```
if
  :: n > 0; s1; s2; ...
  :: n < 0; t1; t2; ...
fi
```

どのガードも実行可能でないとき(この例ではn==0の場合)は，if文全体の実行がどれかのガードが成り立つまでブロックされる.

```
if
  :: n > 0; s1; s2; ...
  :: n < 0; t1; t2; ...
  :: else; u1; u2; ...
fi
```

ガードが else である節は，他のどの節のガードも成り立たないときに選択される.

```
if
  :: n > 0 -> s1; s2; ...
  :: n < 0 -> t1; t2; ...
  :: else  -> u1; u2; ...
fi
```

セミコロンの代わりに -> と書いてもよい.

# 繰り返し

**do**

:: $S_{00}$; $S_{01}$; ... ; $S_{0k1}$
:: $S_{10}$; $S_{11}$; ... ; $S_{1k2}$
...
:: $S_{n0}$; $S_{n1}$; ... ; $S_{nkn}$

**od**

各節のガード $S_{i0}$ の中に実行可能なものがあればそのうちのひとつを選んで $S_{i0}$, $S_{i1}$, ..., $S_{iki}$ を実行する．以上を繰り返す.

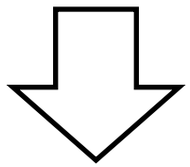
ifと同様に，複数のガードが実行可能なときは，どれかが非決定的に選ばれる．またどのガードも実行可能でないときは，いずれかのガードが成り立つまでブロックされる．

```
do
  :: n > 0 -> s1; s2; ...
  :: n < 0 -> t1; t2; ...
  :: else -> break
od
```

break を実行すると do をぬけることができる．

whileによる記述(C)

```
int a = 1;
while (n > 0) {
  a = a * n;
  n = n - 1;
}
```



doによる記述(Promela)

```
int a = 1;
do
  :: n > 0 ->
     a = a * n;
     n = n - 1;
  :: else -> break;
od
```

# シミュレーション実行による アルゴリズム0の安全性検査

alg0_2.pml

```
bool in_use = false;
int ncs = 0;

active proctype P () {
  do
    :: /* printf("P: NC\n"); */
       in_use == false;
       in_use = true;
       ncs++;
       /* printf("P: CS\n"); */
       assert(ncs == 1);
       ncs--;
       in_use = false;
  od
}
```

alg0_3.pml

```
bool in_use = false;
int ncs = 0;

active[2] proctype Proc () {
  do
    :: in_use == false;
       in_use = true;
       ncs++;
       assert(ncs == 1);
       ncs--;
       in_use = false;
  od
}
```

プロセス P, Q の定義は同じなので，上のように定義をまとめることもできる.

# alg0_2.pmlの実行

```
$ spin alg0_2.pml
spin: alg0_2.pml:24, Error: assertion violated
spin: text of failed assertion: assert((ncs==1))
#processes: 2
        in_use = 1
        ncs = 2
  7:   proc  1 (Q:1) alg0_2.pml:24 (state 4)
  7:   proc  0 (P:1) alg0_2.pml:11 (state 4)
2 processes created
```

# 表明(アサーション)

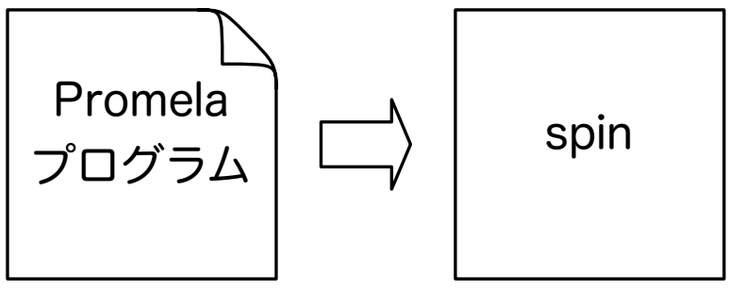
- 表明文：assert(条件式)
  - 実行したときに条件が成り立っていなければエラー(表明違反)となる.
- この例では，変数 ncs はCSに入ったプロセスの数を表している．したがって相互排除のためには ncs == 1 という表明が成り立たなければならない.

# シミュレーション実行の限界

- シミュレーション実行では，実行可能な文をランダムに選んで実行している.
- したがって，表明違反に至る実行経路がある場合でも，その経路を構成する文がいつまでたっても選ばれない可能性がある.
  - 何回も実行すればよいというものではない.
- そこで，可能な実行経路をシステマティックに選択して全て検査する必要がある．Spinはそのための機能を備えている.

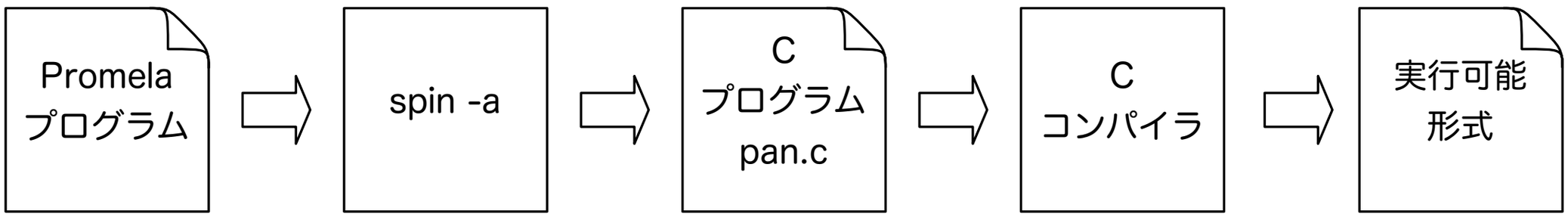
# シミュレーション実行と網羅的探索

シミュレーション実行




網羅的探索をするには -a オプションを使う．これにより，Promelaプログラムから網羅的探索のためのCプログラムが生成される．

網羅的探索



# 網羅的探索の方法

```
$ spin -a alg0_2.pml
$ ls pan.c
pan.c
$ gcc -o pan pan.c
$ ./pan
pan:1: assertion violated (ncs==1) (at depth 18)
pan: wrote alg0_2.pml.trail

(Spin Version 6.5.2 -- 6 December 2019)
Warning: Search not completed
    + Partial Order Reduction

...

Stats on memory usage (in Megabytes):
    0.001   equivalent memory usage for states (stored*(State-vector
+ overhead))
    0.291   actual memory usage for states
  128.000   memory used for hash table (-w24)
    0.534   memory used for DFS stack (-m10000)
  128.730   total actual memory usage

pan: elapsed time 0 seconds
```

# 網羅的探索の結果を用いたシミュレーション実行(ガイド付き実行)

```
$ ls alg0_2.pml.trail
alg0_2.pml.trail
$ spin -p -g -t alg0_2.pml
Starting P with pid 0
Starting Q with pid 1
  1:   proc  1 (Q) line  20 "alg0_2.pml" (state 1)
[((in_use==0))]
  2:   proc  0 (P) line   7 "alg0_2.pml" (state 1)
[((in_use==0))]
...
```

網羅的探索で表明違反が見つかった場合，トレイルファイルと呼ばれるファイルが生成される．シミュレーション実行時に -t オプションをつけることで，表明違反に至実行経路(探索結果)を表示することができる.

# atomic文

alg0_a.pml

```
bool in_use = false;
int ncs = 0;

active[2] proctype Proc () {
  do
    :: atomic {
         in_use == false;
         in_use = true;
       };
       ncs++;
       assert(ncs == 1);
       ncs--;
       in_use = false;
  od
}
```

atomic { ... } で指定された部分はアトミックに実行される.

```
$ spin alg0_a.pml
^C
```

シミュレーション実行は停止しない.

```
$ spin -a alg0_a.pml
$ gcc -o pan pan.c
$ ./pan
  ....
Full statespace search for:
   never claim             - (none specified)
   assertion violations  +
   acceptance   cycles     - (not selected)
   invalid end states +
  ...
pan: elapsed time 0 seconds
```

表明違反は起こらず，安全性をみたすことがわかる.

```
mtype = { TurnP, TurnQ };
mtype turn = TurnP;
bool wantp = false, wantq = false;
int ncs = 0;

active proctype P () {
  do :: wantp = true;
        do :: wantq ->
              if :: (turn == TurnQ) ->
                    wantp = false;
                    turn == TurnP;
                    wantp = true;
                 :: else -> skip;
              fi
           :: else -> break;
        od;
progress:
        ncs++;
        assert(ncs == 1);
        ncs--;
        turn = TurnQ;
        wantp = false;
  od
}
```

dekker.pml

# アルゴリズム3(再)

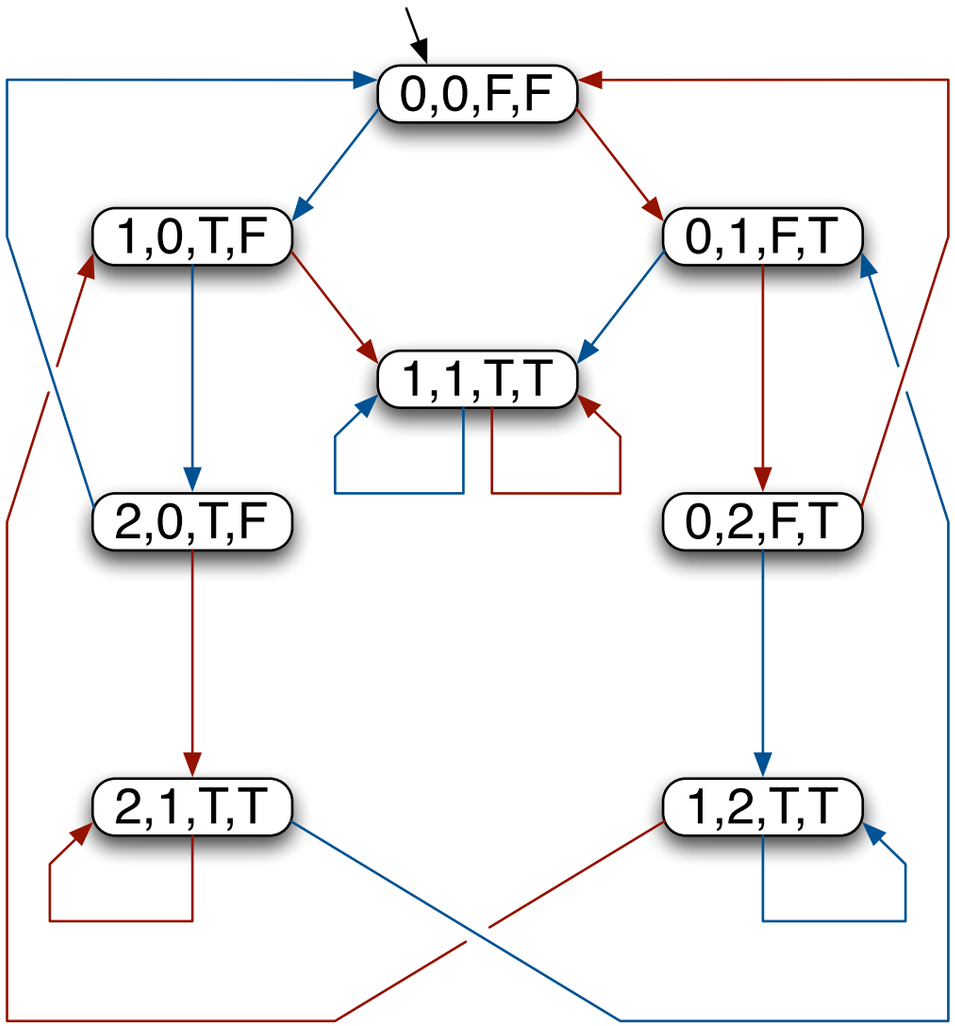
```
// shared variables
bool wantp = false, wantq = false;
```

```
// thread p
while (true) {
  p0: NC
  p1: wantp = true;
  p2: while (wantq);
  p3: CS
  p4: wantp = false;
}
```

```
// thread q
while (true) {
  q0: NC
  q1: wantq = true;
  q2: while (wantp);
  q3: CS
  q4: wantq = false;
}
```

アルゴリズム２のp1(q1)とp2(q2)を交換したもの.

# アルゴリズム3の活性




合成した系の(1,1,T,T)という状態からはどの状態にも遷移できない．よってデッドロック(ライブロック)が生じる．

# alg3.pml

```
bool wantp = false, wantq = false;
int ncs = 0;

active proctype P () {
  do
    :: wantp = true;
       wantq == false;
       ncs++;
       assert(ncs == 1);
       ncs--;
       wantp = false;
  od
}
```

# alg3.pmlの検証結果

```
$ spin -a alg3.pml
$ gcc -o pan pan.c
$ ./pan
...
pan: invalid end state (at depth 7)
pan: wrote alg3.pml.trail

(Spin Version 6.2.5 -- 3 May 2013)
Warning: Search not completed
   + Partial Order Reduction
...
```

# 終了状態

- 各プロセスの正しい終了状態とは，end で始まるラベルで停止している状態か，プロセスの最後の状態(停止する場合)をいう.
- 正しい終了状態以外で停止する場合，invalid end state というエラーメッセージが表示され，その状態に至るトレイルファイルが生成される.

```
$ spin -p -g -t alg3.pml
using statement merging
  1:   proc  1 (Q) alg3.pml:17 (state 1)  [wantq = 1]
       wantq = 1
  2:   proc  1 (Q) alg3.pml:18 (state 2)  [((wantp==0))]
  3:   proc  1 (Q) alg3.pml:19 (state 3)  [ncs = (ncs+1)]
       ncs = 1
  4:   proc  1 (Q) alg3.pml:20 (state 4)  [assert((ncs==1))]
  5:   proc  1 (Q) alg3.pml:21 (state 5)  [ncs = (ncs-1)]
       ncs = 0
  6:   proc  0 (P) alg3.pml:6 (state 1)   [wantp = 1]
       wantp = 1
  7:   proc  1 (Q) alg3.pml:22 (state 6)  [wantq = 0]
       wantq = 0
  8:   proc  1 (Q) alg3.pml:17 (state 1)  [wantq = 1]
       wantq = 1
spin: trail ends after 8 steps
#processes: 2
       wantp = 1
       wantq = 1
       ncs = 0
  8:   proc  1 (Q) alg3.pml:18 (state 2)
  8:   proc  0 (P) alg3.pml:7 (state 2)
2 processes created
```

デッドロックに至る実行系列

# alg3t.pml

```
bool wantp = false, wantq = false;

active proctype P () {
  wantp = true;
  wantq == false;
  wantp = false;
}
```

無限ループの中を一回だけ実行するようモデル化してみる．この場合も，各プロセスが最後まで実行されれば正常な終了状態である．

# endラベルのある例

```
byte x = 1, y = 1;

active proctype GCD() {
  do
    :: x < 100 -> x++
    :: y < 100 -> y++
    :: break
  od;
loop:
  assert(0 < x && 0 < y);
  printf("x=%d, y=%d\n", x, y);
end:
  if
    :: (x > y) -> x = x - y
    :: (x < y) -> y = y - x
  fi;
  goto loop;
}
```

1～100までの2つの数のGCDがこの方法で正しくもとめられることを検証する.

if 文のどちらの条件にもあてはまらないときに停止する(実行不可能になる)が，その状態にendラベルを付けている.

# timeout

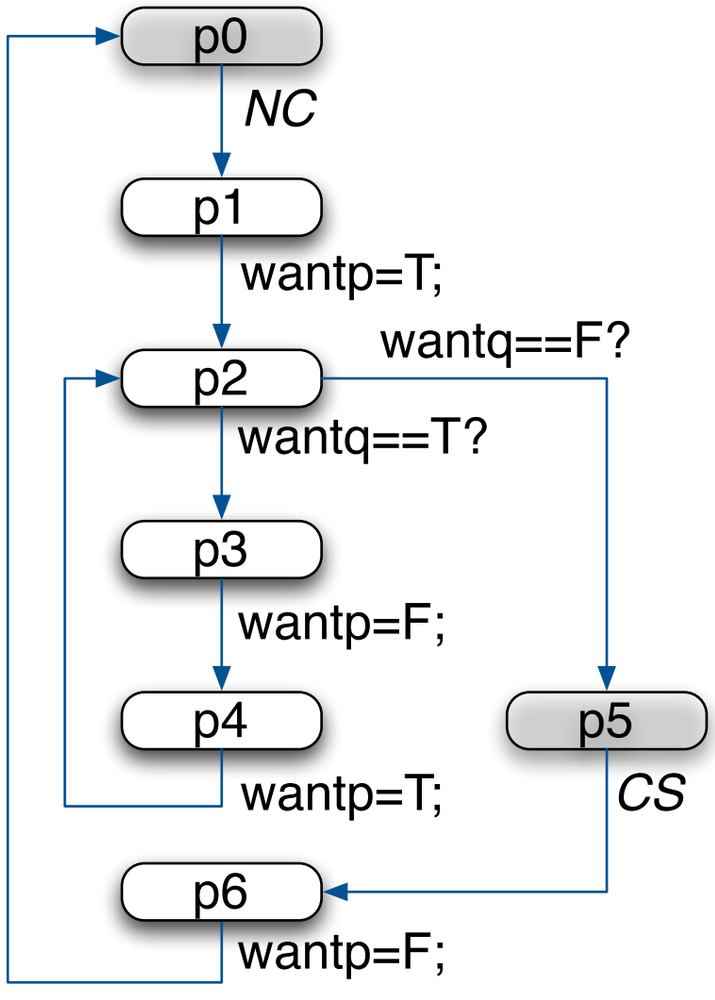
```
active proctype Watcher() {
  do
    :: timeout ->
       assert(x == y);
       printf("x=%d, y=%d\n", x, y);
       break;
  od
}
```

上のようなプロセスを付加することで，他のどのプロセスも実行可能でなくなった状態(timeout)の様子を観測することができる.

# アルゴリズム４

```
// shared variables
bool wantp = false, wantq = false;
```

```
// thread p
while (true) {
   p0: NC
   p1: wantp = true;
   p2: while (wantq) {
   p3:       wantp = false;
   p4:       wantp = true;
       }
   p5: CS
   p6: wantp = false;
}
```



# alg4.pml

```
bool wantp = false, wantq = false;
int ncs = 0;

active proctype P () {
  do
    :: wantp = true;
       do :: wantq == true ->
             wantp = false;
             wantp = true;
          :: else -> break;
       od;
progress:
       ncs++;
       assert(ncs == 1);
       ncs--;
       wantp = false;
  od
}
```

# 活性検査

```
$ spin -a alg4.pml
$ gcc -DNP -o pan pan.c
$ ./pan -l
pan: non-progress cycle (at depth 4)
pan: wrote alg4.pml.trail
...
```

pan.c のコンパイル時に -DNP, 実行時に -l オプションを付ける．これによって，ラベル progress を付けた文が無限にしばしば(infinitely often)実行されないことが検査できる.

```
$ ./pan -l
pan:1: non-progress cycle (at depth 4)
pan: wrote alg4.pml.trail

(Spin Version 6.2.5 -- 3 May 2013)
Warning: Search not completed
    + Partial Order Reduction

Full statespace search for:
    never claim          + (:np_:)
    assertion violations + (if within scope of claim)
    non-progress cycles  + (fairness disabled)
    invalid end states   - (disabled by never claim)

State-vector 36 byte, depth reached 9, errors: 1
        5 states, stored
        0 states, matched
        5 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.000    equivalent memory usage for states (stored*(State-vector + overhead))
    0.290    actual memory usage for states
  128.000    memory used for hash table (-w24)
    0.534    memory used for DFS stack (-m10000)
  128.730    total actual memory usage

pan: elapsed time 0 seconds
```

# 公平性(fairness)(1)

fairtest.pml

```
active proctype P () {
  do
    :: printf("RED\n");
progress:
       printf("BLUE\n");
  od
}

active proctype Q () {
  do
    :: printf("RED\n");
       printf("BLUE\n");
  od
}
```

# 公平性(fairness)(2)

- fairtest.pml を検査すると，公平でない実行(プロセスQのみの実行)も検査されるため，以下のようなエラーとなる.

```
$ spin -a fairtest.pml
$ gcc -DNP -o pan pan.c
$ ./pan -l
pan: non-progress cycle (at depth 2)
pan: wrote fairtest.pml.trail
```

- pan の実行時に -f を付加することで，公平な実行のみを検査できる.

# 公平性の種類

- 無条件公平性(unconditional fairness)
  - 無限実行列上で各プロセスが無限回実行される.
- 弱公平性(weak fairness)
  - プロセスがずっと（＝常に）実行可能であれば，いつかは必ず実行される.
  - -f オプションで仮定される公平性はこれ.
- 強公平性(strong fairness)
  - プロセスが無限回実行可能(ずっと実行可能とは限らない)であれば，いつかは必ず実行される.

# アルゴリズム0

alg0_0.pml

```
bool in_use = false;

active[2] proctype P() {
  do
    :: /* NC */
       in_use == false;
       in_use = true;
       /* CS */
       in_use = false;
  od
}
```

# 表明による検証

alg0_3.pml

```
bool in_use = false;
int ncs == 0;

active[2] proctype P() {
  do
    :: in_use == false;
       in_use = true;
       ncs++;
       assert(ncs == 1);
       ncs--;
       in_use = false;
  od
}
```

# Spinによるテストと検証

シミュレーション実行によるテスト

```
$ spin alg0_3.pml
spin: alg0_3.pml:9, Error: assertion violated
spin: text of failed assertion: assert((ncs==1))
#processes: 2
```

網羅的探索による検証

```
$ spin -a alg0_3.pml
$ gcc -o pan pan.c
$ ./pan
...
pan:1: assertion violated (ncs==1) (at depth 18)
pan: wrote alg0_3.pml.trail
...
```

# 安全性と不変式

- このシステムでの安全性は，2つのプロセスが同時にCSに入らないことである.
- 安全性が成り立つのなら，ncs <= 1 は不変式(invariant)である.
- 「（不変式である）ncs <= 1 が常に成立する」という事実は他のプロセスからみても変らないはず.

# 観測プロセス

alg0_4.pml

```
bool in_use = false;
int ncs == 0;

active[2] proctype P() {
  do
    :: in_use == false;
       in_use = true;
       ncs++;
       ncs--;
       in_use = false;
  od
}

active proctype Observer() {
  !(ncs <= 1) -> assert(false)
}
```

ここにあった表明を削除し，代りに観測プロセスに表明の検査をさせる.

観測プロセス：
不変式(ncs <= 1)を観測し，成り立たなくなったときに表明違反とする.

# シミュレーション実行

```
$ spin alg0_4.pml
spin: alg0_4.pml:15, Error: assertion violated
spin: text of failed assertion: assert(0)
#processes: 3
        in_use = 1
        ncs = 1
267:   proc  2 (Observer) alg0_4.pml:15 (state 2)
267:   proc  1 (Proc) alg0_4.pml:9 (state 4)
267:   proc  0 (Proc) alg0_4.pml:10 (state 5)
3 processes created
```

観測プロセスによって表明違反を検出できた. ここで ncs = 1 になっているが, これはObserverが表明違反を検出してから assert(false) を実行するまでの間に他のプロセスが ncs-- を実行したため.

# 観測プロセスの改良

alg0_5.pml (一部)

```
active proctype Observer() {
  atomic { !(ncs <= 1) -> assert(false) }
}
```

こうすることで，表明違反を報告した時点で ncs > 1 であることが保証できる.

```
$ spin alg0_5.pml
spin: alg0_5.pml:15, Error: assertion violated
spin: text of failed assertion: assert(0)
#processes: 3
        in_use = 1
        ncs = 2
176:   proc  2 (Observer) alg0_5.pml:15 (state 2)
176:   proc  1 (Proc) alg0_5.pml:9 (state 4)
176:   proc  0 (Proc) alg0_5.pml:9 (state 4)
3 processes created
```

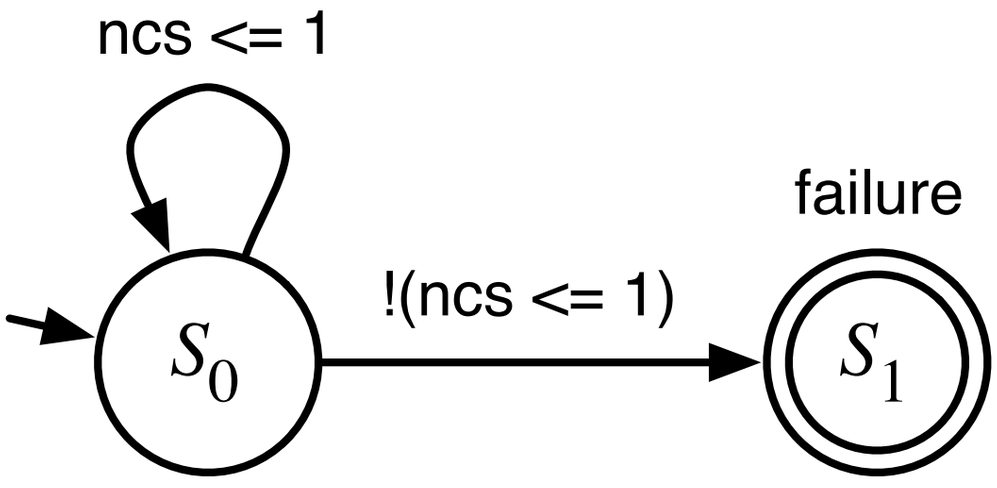
# 観測プロセスの他の定義方法

```
active proctype Observer() {
  do
    :: ncs <= 1 -> skip
    :: else -> break
  od;
  assert(false)
}
```

```
active proctype Observer() {
  do
    :: assert(ncs <= 1)
  od
}
```

# 観測プロセスの性質

```
active proctype Observer() {
  do
    :: ncs <= 1 -> skip
    :: else -> break
  od;
  assert(false)
}
```




この形の観測プロセスは，最後まで実行すると表明違反となる．オートマトンとして表すと左のようになる．

# 観測プロセスを使った検証

- あるシステムについて，検証したい性質（例えば，ncs <=1 が不変式であること）の否定が成立するときに，エラー（表明違反）で実行を終了するような観測プロセスを導入しておく．
- システムと観測プロセスを合成した系を実行，あるいは網羅的探索を行い，観測プロセスが表明違反に至れば検証したい性質は成立しないことがわかる．

# ゴースト変数(ghost variables)

- ncs のように，検証対象の動作とは関係なく，検証目的で導入される変数のこと.
- ゴースト変数の読み出しや代入もシステムの状態遷移動作に含まれるため，状態数が多くなってしまう.

# ゴースト変数を使わない検証

alg0_6.pml

```
bool in_use = false;

active[2] proctype P() {
  do
    :: in_use == false;
       in_use = true;
cs:    in_use = false;
  od
}

active proctype Observer() {
  atomic { P[0]@cs && P[1]@cs -> assert(false) }
}
```

# リモート参照

- プロセス名@ラベル
  - プロセス名で表されるプロセスの実行位置がラベルで表される箇所にあることを表す命題
- P[0]@cs && P[1]@cs
  - P[0], P[1]は，active[2] proctype P ... によって生成された２つのプロセスを表す.
  - P[0]@cs によって，プロセスP[0]がラベルcsの位置，つまりクリティカルセクションにあることを表している.
  - よって，この式全体で２つのプロセスが同時にクリティカルセクションにあることを表している.

alg0_7.pml

```
bool in_use = false;

active proctype P() {
  do
    :: in_use == false;
       in_use = true;
cs:    in_use = false;
  od
}

active proctype Q() {
 ... /* same as P */
}

active proctype Observer() {
  atomic { P@cs && Q@cs -> assert(false) }
}
```

このように書く事もできる.

# Never Claim

- 観測プロセスは，検証したい性質の否定が成り立つかどうかを常にチェックしている.
- 言い換えると，動作の系列が観測プロセスのオートマトンによって受理されると，システムは検証したい性質を満たさないことになる.
- このような，観測プロセスのオートマトンによって表されるような性質（=検証したい性質の否定）を never claimと呼ぶ.
- Spinにはnever claimのための構文がある.

# Never Claimを使った検証(1)

alg0_8.pml

```
bool in_use = false;

active[2] proctype P () {
  do
    :: in_use == false;
       in_use = true;
cs:    in_use = false;
  od
}

never {
  do
    :: P[0]@cs && P[1]@cs -> break
    :: else -> skip
  od
}
```

# Never Claimを使った検証(2)

```
$ spin -a alg0_8.pml
$ gcc -o pan pan.c
$ ./pan
warning: for p.o. reduction to be valid the never claim
must be stutter-invariant
(never claims generated from LTL formulae are stutter-
invariant)
pan:1: end state in claim reached (at depth 21)
pan: wrote alg0_8.pml.trail
```

シミュレーション実行ではnever claimによる検査はできないので，網羅的探索を行う必要がある.

# アルゴリズム4

alg4_1.pml

```
bool wantp = false, wantq = false;

active proctype P () {
  do
     :: skip;
try:    wantp = true;
        do :: wantq == true ->
              wantp = false;
              wantp = true;
           :: else -> break;
        od;
cs:     wantp = false;
  od
}

active proctype Q () { ... }
```

構文上，ガード文にはラベルをつけられないのでskip(何もしない)を入れておく．

# Never Claimによる活性の検証

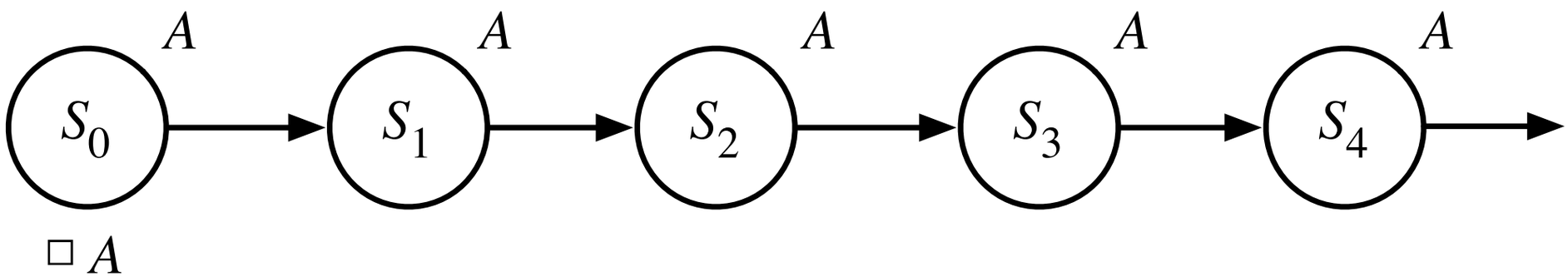
- 検証したい性質(活性)：
  - プロセスPの実行がtryに到達したら，いつかは必ずPはcsに到達する.
- この性質の否定を表すnever claim（のオートマトン）を書けばよいのだが，実は結構面倒.
- ここでは，線形時相論理(LTL)の式からnever claimを生成する方法を示す.

# 時相論理(Temporal Logic)

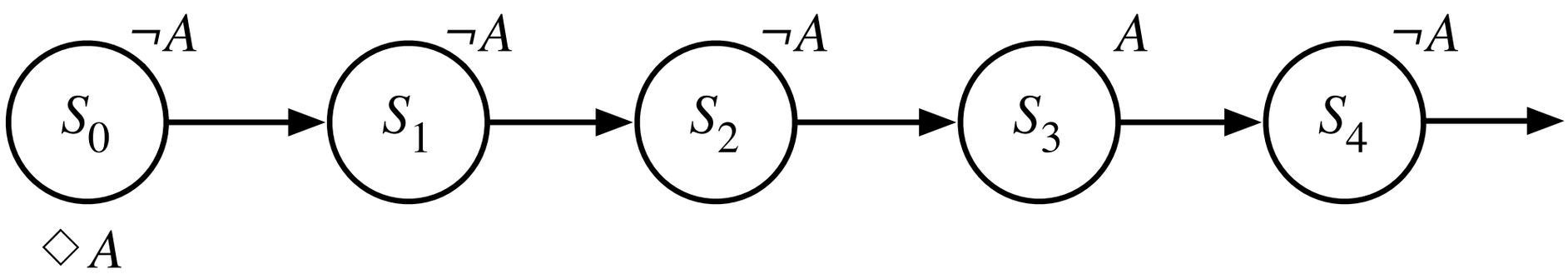
- 時間（実行状態）を考慮した論理
- 時相論理(LTL)における論理記号の例
  - □A
    - この先Aは常に成り立つ(=現在の状態から到達可能な状態すべてにおいてAが成り立つ).
    - “box A” と読む.
  - ◇A
    - この先いつか必ずAが成り立つ(=現在の状態から，Aが成り立つ状態に有限ステップで必ず到達できる).
    - “diamond A” と読む.
- 他にも○A, A U B, A W B 等の論理記号がある.

# 線形時相論理(LTL)

- □A　この先ずっとAが成り立つ.




- ◇A　この先いつか必ずAが成り立つ.



# □と◇の性質

- □□A ⇔ □A
- ◇◇A ⇔ ◇A
- □◇□A ⇔ ◇□A
- ◇□◇A ⇔ □◇A
- □(A ∧ B) ⇔ □A ∧ □B
- ◇(A ∨ B) ⇔ ◇A ∨ ◇B
- □A ⇔ ¬◇¬A
- ◇A ⇔ ¬□¬A

# よく用いられる例

- □A
  - 常にAが成り立つこと．不変条件.
  - 例: □(¬(P@cs ∧ Q@cs))
- □◇A
  - Aが無限にしばしば(infinitely often)成り立つ.
  - 例：□◇(P@cs)
- □(A → ◇B)
  - Aが成り立てばいつか必ずBになる.
  - 例：□(P@try → ◇P@cs)

# SpinにおけるLTL式の構文

- 構文
  - []A,
  - <>A,
  - A -> B,
  - !A, A && B, A || B
  - Promelaの条件式
- 例
  - [](!(P@cs && Q@cs))
  - [](ncs <= 1)
  - []<>P@cs
  - [](P@try -> <>P@cs)

# LTL式からnever claimを生成する

spin -a -f LTL式

検証したい性質の否定

```
$ spin -a -f '![]!(P[0]@cs && P[1]@cs)'
never  {    /* ![]!(P[0]@cs && P[1]@cs) */
T0_init:
    do
    :: atomic { ((P[0]@cs && P[1]@cs)) ->
                        assert(!((P[0]@cs && P[1]@cs))) }
    :: (1) -> goto T0_init
    od;
accept_all:
    skip
}
```

標準出力にnever claimの記述が出力される.

alg0_9.pml

```
bool in_use = false;

active[2] proctype P () {
  do
    :: in_use == false;
       in_use = true;
cs:    in_use = false;
  od
}

never  {    /* ![]!(P[0]@cs && P[1]@cs) */
T0_init:
   do
   :: atomic { ((P[0]@cs && P[1]@cs)) ->
                      assert(!((P[0]@cs && P[1]@cs))) }
   :: (1) -> goto T0_init
   od;
accept_all:
   skip
}
```

出力されたnever claimの記述をコピー

# 生成したNever Claimを使う検証

```
$ spin -a alg0_9.pml
$ gcc -o pan pan.c
$ ./pan -a
...
pan:1: assertion violated  !
(((P[0]._p==cs)&&(P[1]._p==cs))) (at depth 8)
pan: wrote alg0_7.pml.trail
...
```

-a (acceptance)オプションによって, never claimによって受理されるか否かを検証する.

# 名前付きのNever Claim (1)

```
never safety {
    . . .
}

never liveness {
    . . .
}
```

neverと{の間に識別子（名前）を書いて，名前をつけたnever claimを複数個書くことができる.

# 名前付きのNever Claim (2)

```
$ spin -a alg0_9.pml
  the model contains 2 never claims: liveness, safety
  only one claim is used in a verification run
  choose which one with ./pan -a -N name (defaults to -N
safety)
$ gcc -o pan pan.c
$ ./pan -a -N safety
...
pan:1: assertion violated  !
(((P[0]._p==cs)&&(P[1]._p==cs))) (at depth 8)
pan: wrote alg0_9.pml.trail
...
```

-N 名前 で使用するnever claimを指定する.

# LTL式の指定 (1)

- Spinの第6版以降では，Promelaのソースファイルにltlの式を直接書けるようになった.
  - 従って，前ページまでに説明した，検証したいLTL式の否定からnever claimを生成してソースファイルにコピーする作業は必要ない.
    - 自動でやってくれる.
  - Never claimでは検証したい性質の否定を書くが，LTL式の指定では検証したい性質そのものを書く.
  - ただし，LTL式で表現できないnever claimを書きたいときはneverを用いる.

# LTL式の指定 (2)

alg0_10.pml

```
bool in_use = false;

active[2] proctype P() {
  do
    :: in_use == false;
       in_use = true;
cs:    in_use = false;
  od
}

ltl safety { []!(P[0]@cs && P[1]@cs) }
```

検証したい性質をLTL式で書く.

# 活性の検証

alg4_1.pml

```
bool wantp = false, wantq = false;

active proctype P() { . . . }
active proctype Q() { . . . }

ltl safety { []!(P@cs && Q@cs) }

ltl liveness1 { []<>P@cs }

ltl liveness2 { [](P@try -> <>P@cs) }
```

安全性(safety)と2種類の活性(liveness1, liveness2)を書いておく.

# 検証したい性質

- safety : []!(P@cs && Q@cs)
  - 常に（=初期状態から到達可能な全ての状態において），PとQの実行が同時にcsに至ることはない.
- liveness1 : []<>P@cs
  - Pの実行は無限にしばしばcsに至る.
    - 初期状態から到達可能な全ての状態において，その状態から先いつか必ずPの実行がcsに至る.
- liveness2 : [](P@try -> <>P@cs)
  - Pの実行がtryに至れば，その先いつか必ずPはcsに至る.

# 安全性(safety)

```
$ spin -a alg4_1.pml
ltl safety: [] (! (((P@cs)) && ((Q@cs))))
ltl liveness1: [] (<> ((P@cs)))
ltl liveness2: [] ((! ((P@try))) || (<> ((P@cs))))
  the model contains 3 never claims: liveness2,
liveness1, safety
  only one claim is used in a verification run
  choose which one with ./pan -a -N name (defaults to -N
safety)
$ gcc -o pan pan.c
$ ./pan -a -N safety
```

安全性は成立するので，エラーは出力されない．

# 活性(liveness1)の検査

```
$ ./pan -a -N liveness1
pan:1: acceptance cycle (at depth 8)
pan: wrote alg4_1.pml.trail
...
```

- acceptance cycleというメッセージが出る.
- これはLTL式の否定であるnever claimによって受理される実行があること，つまりliveness1がなりたたないことを表している.

```
$ spin -p -g -t alg4_1.pml
…
starting claim 3
using statement merging
Never claim moves to line 12   [(!((P._p==cs)))]
  2:    proc  1 (Q) alg4_1.pml:18 (state 1)    [(1)]
Never claim moves to line 17   [(!((P._p==cs)))]
  4:    proc  1 (Q) alg4_1.pml:19 (state 2)    [wantq = 1]
        wantq = 1
  6:    proc  0 (P) alg4_1.pml:5 (state 1)[(1)]
  8:    proc  0 (P) alg4_1.pml:6 (state 2)[wantp = 1]
        wantp = 1
  <<<<<START OF CYCLE>>>>>
 10:    proc  1 (Q) alg4_1.pml:20 (state 3)    [((wantp==1))]
 12:    proc  1 (Q) alg4_1.pml:21 (state 4)    [wantq = 0]
        wantq = 0
 14:    proc  1 (Q) alg4_1.pml:22 (state 5)    [wantq = 1]
        wantq = 1
spin: trail ends after 14 steps
#processes: 2
        wantp = 1
        wantq = 1
        cs = 0
        try = 0
 14:    proc  1 (Q) alg4_1.pml:20 (state 6)
 14:    proc  0 (P) alg4_1.pml:7 (state 8)
 14:    proc  - (liveness1) _spin_nvr.tmp:16 (state 10)
2 processes created
```

プロセスQのみでループしていることがわかる.

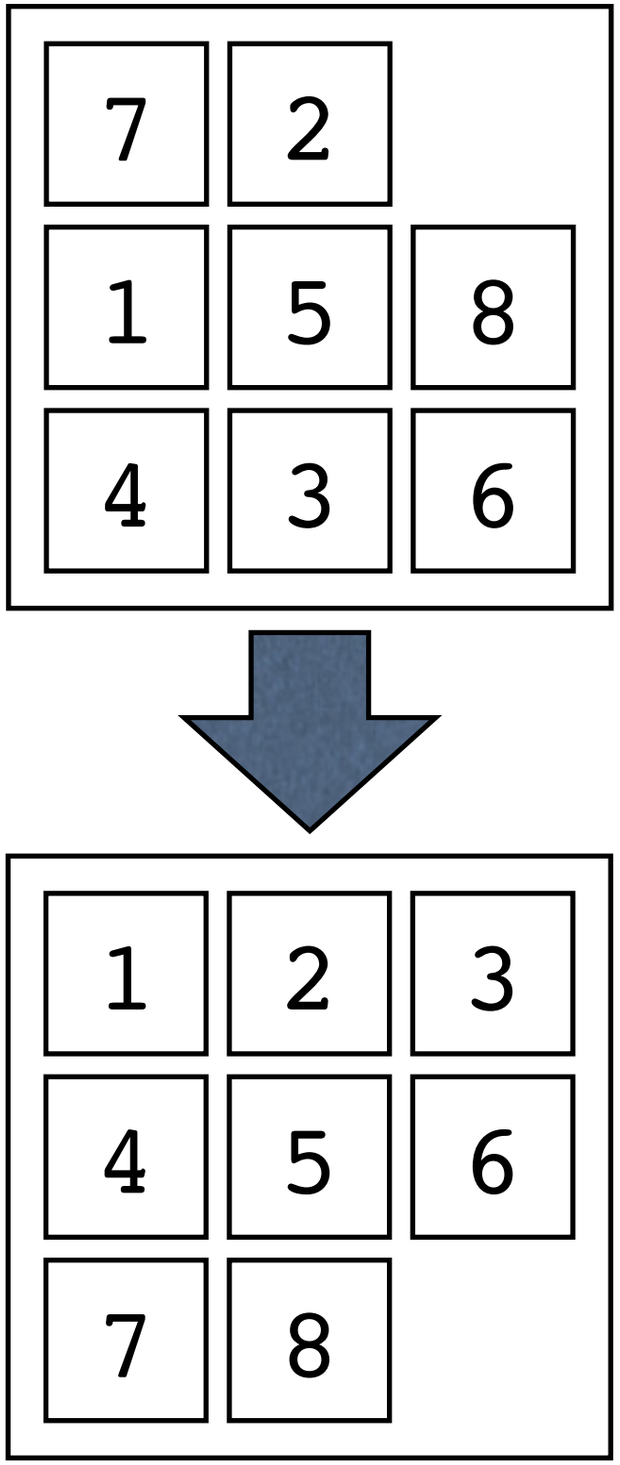
# 公平性を仮定して検査する

```
$ ./pan -a -f -N liveness1
pan:1: acceptance cycle (at depth 22)
pan: wrote alg4_1.pml.trail
...
```

- -fオプションをつけて，弱公平性を仮定して実行した場合を検査する.
  - 弱公平性
    - 常に実行可能なプロセスは必ず実行される.
- この場合もacceptance cycleというメッセージが出力される．つまりアルゴリズム4はliveness1が成り立たない.

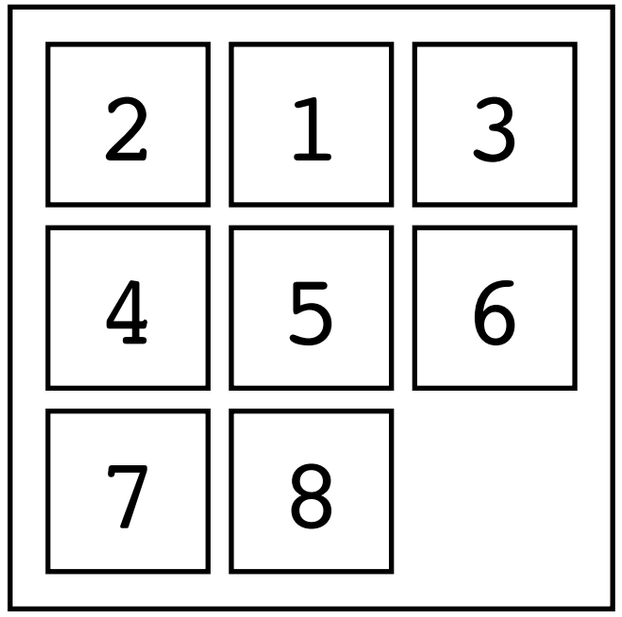
```
spin -p -g -t alg4_1.pml
…
 <<<<<START OF CYCLE>>>>>
 24:    proc  1 (Q) alg4_1.pml:22 (state 5)    [wantq = 1]
        wantq = 1
 26:    proc  0 (P) alg4_1.pml:9 (state 5) [wantp = 1]
        wantp = 1
 28:    proc  1 (Q) alg4_1.pml:20 (state 3)    [((wantp==1))]
 30:    proc  1 (Q) alg4_1.pml:21 (state 4)    [wantq = 0]
        wantq = 0
 32:    proc  1 (Q) alg4_1.pml:22 (state 5)    [wantq = 1]
        wantq = 1
 34:    proc  0 (P) alg4_1.pml:7 (state 3) [((wantq==1))]
 36:    proc  1 (Q) alg4_1.pml:20 (state 3)    [((wantp==1))]
 38:    proc  1 (Q) alg4_1.pml:21 (state 4)    [wantq = 0]
        wantq = 0
 40:    proc  0 (P) alg4_1.pml:8 (state 4) [wantp = 0]
        wantp = 0
spin: trail ends after 40 steps
#processes: 2
        wantp = 0
        wantq = 0
        cs = 0
        try = 0
 40:    proc  1 (Q) alg4_1.pml:22 (state 5)
 40:    proc  0 (P) alg4_1.pml:9 (state 5)
 40:    proc  - (liveness1) _spin_nvr.tmp:16 (state 10)
2 processes created
```

# 例：8パズル(1)




- 1～8の数字が書かれた8個の正方形タイルを3×3のマスに適当に置く
- 空いている場所に，隣のタイルをスライドして動かすことができる
- 左下のようになったら完成

# 例：8パズル(2)




- 問題：与えられた初期配置から，ゴールに至ることができるか否かを判定せよ
  - 例えば左の配置からゴールに至ることは不可能
- 方針：初期値からゴールに至る経路がないことを網羅的探索で示す

```
#define COLS 3
#define ROWS 3
#define POS(x, y) ((x) + (y) * COLS)
#define GOAL ( b[0] == 1 && b[1] == 2 && ... && b[7] == 8 && b[8] == 0 )

active proctype EightPuzzle() {
  byte b[COLS * ROWS] = {  /* problem (space : 0) */
    2, 1, 3,
    4, 5, 6,
    7, 8, 0
  };
  byte x = 2, y = 2; /* initial position of the space */

  do
    :: GOAL ->  assert(false)
    :: else -> if
                  :: x > 0 ->
                     atomic { /* move right */
                       b[POS(x, y)] = b[POS(x - 1, y)];
                       b[POS(x - 1, y)] = 0;
                       x--
                     }
                  :: x < COLS - 1 ->
                     atomic { /* move left */
                       b[POS(x, y)] = b[POS(x + 1, y)];
                       b[POS(x + 1, y)] = 0;
                       x++
                     }
                  :: y > 0 -> ...
                  :: y < ROWS - 1 -> ...
               fi
  od
}
```

8p.pml

```
$ spin -a 8p.pml
$ gcc -o pan pan.c
$ ./pan -m1000000

(Spin Version 6.5.2 -- 6 December 2019)
     + Partial Order Reduction

Full statespace search for:
     never claim             - (none specified)
     assertion violations    +
     acceptance   cycles     - (not selected)
     invalid end states +

State-vector 24 byte, depth reached 344792, errors: 0
   846720 states, stored
   302401 states, matched
  1149121 transitions (= stored+matched)
        0 atomic steps
hash conflicts:      3823 (resolved)

Stats on memory usage (in Megabytes):
   41.990 equivalent memory usage for states (stored*(State-vector + overhead))
   36.035 actual memory usage for states (compression: 85.82%)
          state-vector as stored = 17 byte + 28 byte overhead
  128.000 memory used for hash table (-w24)
   53.406 memory used for DFS stack (-m1000000)
  217.343 total actual memory usage


unreached in proctype EightPuzzle
     8p.pml:45, state 29, "-end-"
     (1 of 29 states)

pan: elapsed time 0.59 seconds
pan: rate 1435118.6 states/second
```

# 今回説明していない重要な概念

- Spinの構文
  - インライン
  - atomic, d_step
  - チャネルと通信
- 探索アルゴリズム
  - Partial Order Reduction (POR)

# まとめ

- 相互排除アルゴリズムの検証
- モデル検査ツールSpinによる検証
  - モデリング言語Promela
  - 安全性の検証
    - 線形時相論理式(LTL)
    - Never Claim
    - Never Claim/LTLを使った検証
  - 活性の検証