Producers/Consumers Extended

Back to the flawed Producers/Consumers

```
mtype = { P, C };

mtype turn = P;

int msgs;

active [2] proctype producer()
{
  do
    :: (turn == P) ->
         printf("Produce\n");
         msgs++;
         turn = C
  od
}
```

```
active [2] proctype consumer()
{
  do
    :: (turn == C) ->
         printf("Consume\n");
         msgs--;
         turn = P
  od
}


active proctype monitor() {
  assert(msgs >= 0 && msgs <= 1)
}
```

```
> spin -a prodcons2_flaw.pml && gcc -o pan pan.c && ./pan
```

Producers/Consumers Extended (Trail File)

Trail File

prodcons2_flaw.pml.trail contains SPIN's transition markers corresponding to the contents of the stack of transitions leading to error states

Meaning:

- Step number in execution trace
- Id of the process moved in the current step
- Id of the transition taken in the current step

```
-4:-4:-4
1:1:0
2:1:1
3:1:2
4:1:3
5:3:8
6:3:9
7:3:10
8:2:8
9:2:9
10:3:11
11:2:10
12:4:16
```

```
> spin -t -p prodcons2_flaw.pml
```

The Mutual Exclusion problem

General algorithm

```
active [2] proctype mutex()
{
again:
        /* trying section */

        cnt++;
        assert(cnt == 1);           /* critical section */
        cnt--;

        /* exit section */
        goto again
}
```

The Mutual Exclusion problem (First tentative)

```
bit flag;  /* signal entering/leaving the section */
byte cnt;  /* # procs in the critical section */

active [2] proctype mutex() {
again:
  flag != 1;  /* It models "while (flag == 1) wait!" */
  flag = 1;
  cnt++;
  assert(cnt == 1);
  cnt--;
  flag = 0;
  goto again
}
```

The Mutual Exclusion problem (First tentative)

```
bit flag;  /* signal entering/leaving the section */
byte cnt;  /* # procs in the critical section */

active [2] proctype mutex() {
again:
  flag != 1;  /* It models "while (flag == 1) wait!" */
  flag = 1;
  cnt++;
  assert(cnt == 1);
  cnt--;
  flag = 0;
  goto again
}
```

Assertion violation: Both processes can pass the `flag != 1` before flag is set to 1.

The Mutual Exclusion problem (Second tentative)

```
bit x, y;   /* signal entering/leaving the section */
byte cnt;

active proctype A() {
again:
  /* A waits for B to end */
  x = 1;
  y == 0;
  cnt++;
  /* critical section */
  assert(cnt == 1);
  cnt--;
  x = 0;
  goto again
}
```

```
active proctype B() {
again:
  y = 1;
  x == 0;
  cnt++;
  /* critical section */
  assert(cnt == 1);
  cnt--;
  y = 0;
  goto again
}
```

The Mutual Exclusion problem (Second tentative)

```
bit x, y;   /* signal entering/leaving the section */
byte cnt;

active proctype A() {
again:
  /* A waits for B to end */
  x = 1;
  y == 0;
  cnt++;
  /* critical section */
  assert(cnt == 1);
  cnt--;
  x = 0;
  goto again
}
```

```
active proctype B() {
again:
  y = 1;
  x == 0;
  cnt++;
  /* critical section */
  assert(cnt == 1);
  cnt--;
  y = 0;
  goto again
}
```

Invalid-end-state: Both processes can execute `x = 1` and `y = 1` at the same time and will then be waiting for each other.

Dekker/Dijkstra algorithm

```
/* trying section */
flag[i] = true;
do
   :: flag[j] ->
      if
         :: turn == j ->
            flag[i] = false;
            !(turn == j);
            flag[i] = true
         :: else -> skip
      fi
   :: else ->
      break
od;
```

Dekker/Dijkstra algorithm

```
/* trying section */
flag[i] = true;
do
    :: flag[j] ->
        if
            :: turn == j ->
                flag[i] = false;
                !(turn == j);
                flag[i] = true
            :: else -> skip
        fi
    :: else ->
        break
od;
```

```
/* initialization */
pid i = _pid;
pid j = 1 - _pid;
```

```
/* exit session */
turn = j;
flag[i] = false;
```

Dekker/Dijkstra algorithm

Verification:

```
> spin -a dekker.pml
> cc -o pan pan.c
> ./pan
...
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 20 byte, depth reached 67, errors: 0
...
```

Doran&Thomas change

Is the outer loop really necessary?

```
flag[i] = true;
if
:: flag[j] ->
        if
        :: turn == j ->
                flag[i] = false;
                !(turn == j);
                flag[i] = true
        :: else -> skip
        fi
:: else
fi;
```

Doran&Thomas change

Verification:

```
> spin -a doran.pml
> cc -o pan pan.c
> ./pan
...
pan: assertion violated (cnt==1) (at depth 117)
pan: wrote doran.pml.trail
...
```

`doran.pml.trail` contains a counterexample with length 117.

Doran&Thomas change

We can use a breadth-first search to find the shortest counterexample:

```
> cc -DBFS -o pan pan.c
> ./pan
...
pan: assertion violated (cnt==1) (at depth 12)
pan: wrote doran.pml.trail
...
```

Doran&Thomas change

Now, we can perform a guided simulation:

```
> spin -p -t doran.pml
  1:     proc  1 (mutex) line   8 ... [i = _pid]
  2:     proc  1 (mutex) line   9 ... [j = (1-_pid)]
  3:     proc  1 (mutex) line  11 ... [flag[i] = 1]
  4:     proc  1 (mutex) line  21 ... [else]
  5:     proc  1 (mutex) line  24 ... [cnt = (cnt+1)]
  6:     proc  0 (mutex) line   8 ... [i = _pid]
  7:     proc  0 (mutex) line   9 ... [j = (1-_pid)]
  8:     proc  0 (mutex) line  11 ... [flag[i] = 1]
  9:     proc  0 (mutex) line  13 ... [(flag[j])]
 10:     proc  0 (mutex) line  19 ... [else]
 11:     proc  0 (mutex) line  19 ... [(1)]
 12:     proc  0 (mutex) line  24 ... [cnt = (cnt+1)]
```

Peterson algorithm

A correct improvement:
trying session

```
        flag[i] = true;
        turn = i;
        !(flag[j] && turn == i) ->
```

exit session

```
        flag[i] = false;
```

Verification:

```
> spin -a peterson.pml
> cc -o pan pan.c
> ./pan
...
State-vector 20 byte, depth reached 41, errors: 0
...
```