Atomic Statement

```
atomic { stat1; stat2; ... statn }
```

- An `atomic{}` statement can be used to group statements into an atomic sequence;
- all statements are executed in a single sequence (no interleaving with statements of other processes), though each step is taken.
- The statement is executable if `stat1` is executable
- If a `stat i` (with i>1) is blocked, the "atomicity" is temporarily lost and other processes may do a step.

Sumofhellos made atomic

- We can use the **`atomic`** construct to easily produce a version of sumofhellos that works properly (**`atm_sumofhellos.pml`**).
  - This is the advantage of a modelling language over a program language: We can say "make it so!" (to some extent, at least).
- We can run simulations of these, but how do we know we have fixed things?
- The real strength of SPIN is it can build the state diagram and check every possible path through it, so determining the outcome of every possible interleaving
  - **`$> spin -run atm_sumofhellos.pml`**

`#define` and `inline`

- SPIN uses the C pre-processor (CPP) to process Promela files
  - So all the CPP facilities are available, such as **`#define`**, **`#if`**, **`#ifdef`**, etc.
- There is also something similar called **`inline`**.
  - It has to be defined at the top level

```
inline name(arg1, …, argN){
   stat1; stat2; ... statn
}
```

  - It can only be called where a statement can occur

```
name(val1, …, valN);
```

  - An **`inline`** can contain calls to ***other*** **`inline`**s (but cannot be recursive).

`inline behaviour`

```
inline name(arg1, …, argN){
   stat1; stat2; ... statn
}
```

- Important: **`inline`** describes ***textual substitution***.
  - Just like the way **`#define`** operates.
- The inline code does not represent a function/procedure that can be called.
  - A call **`name(val1, …, valN);`** results in the texts "**`val1`**", .. "**`valN`**" being substituted for the occurrences of "**`arg1`**", .. "**`argN`**", wherever they appear.
- In particular, if a statement like **`stat1`** (say) declares a variable, then that variable has ***global*** scope.

`#define` vs. `inline`

- What is the difference?
- If an error occurs in code produced by a macro defined using **`#define`**,
    - the error is reported at the point of use in the expanded macro text
- If an error occurs in code produced by a macro defined using **`inline`**,
    - the error is reported at the relevant line in the **`inline`** definition itself
        - Generally much more useful.

Modelling Mutexes in Promela

- We can model mutexes in Promela as a variable whose state is locked or unlocked with the check for being not locked done atomically with locking it. We also record the **`_pid`** of the process with the lock.

```
mtype = { unlocked, locked }
mtype mutex = unlocked ;
int mid = 0;
```

- We want to check a mutex is unlocked and then lock it atomically:

```
atomic{ mutex==unlocked -> mutex = locked; mid = _pid } ;
```

- To unlock, we should have the mutex lock, so we need to check our **`_pid`**

```
atomic {
  assert(mid==_pid);
  mutex = unlocked;
  mid = 0;
}
```

Sumofhellos using mutexes

- We can use the inline construct to easily produce a version of sumofhellos that uses mutexes. (**`mtx_sumofhellos.pml`**).
    - This is the advantage of a modelling language over a program language: We can say "make it so!" (to some extent, at least).
- As before, we can run simulations of this.
- Again, we can build the state diagram and check every possible path through it, so determining the outcome of every possible interleaving
    - **`$> spin -run mtx_sumofhellos.pml`**

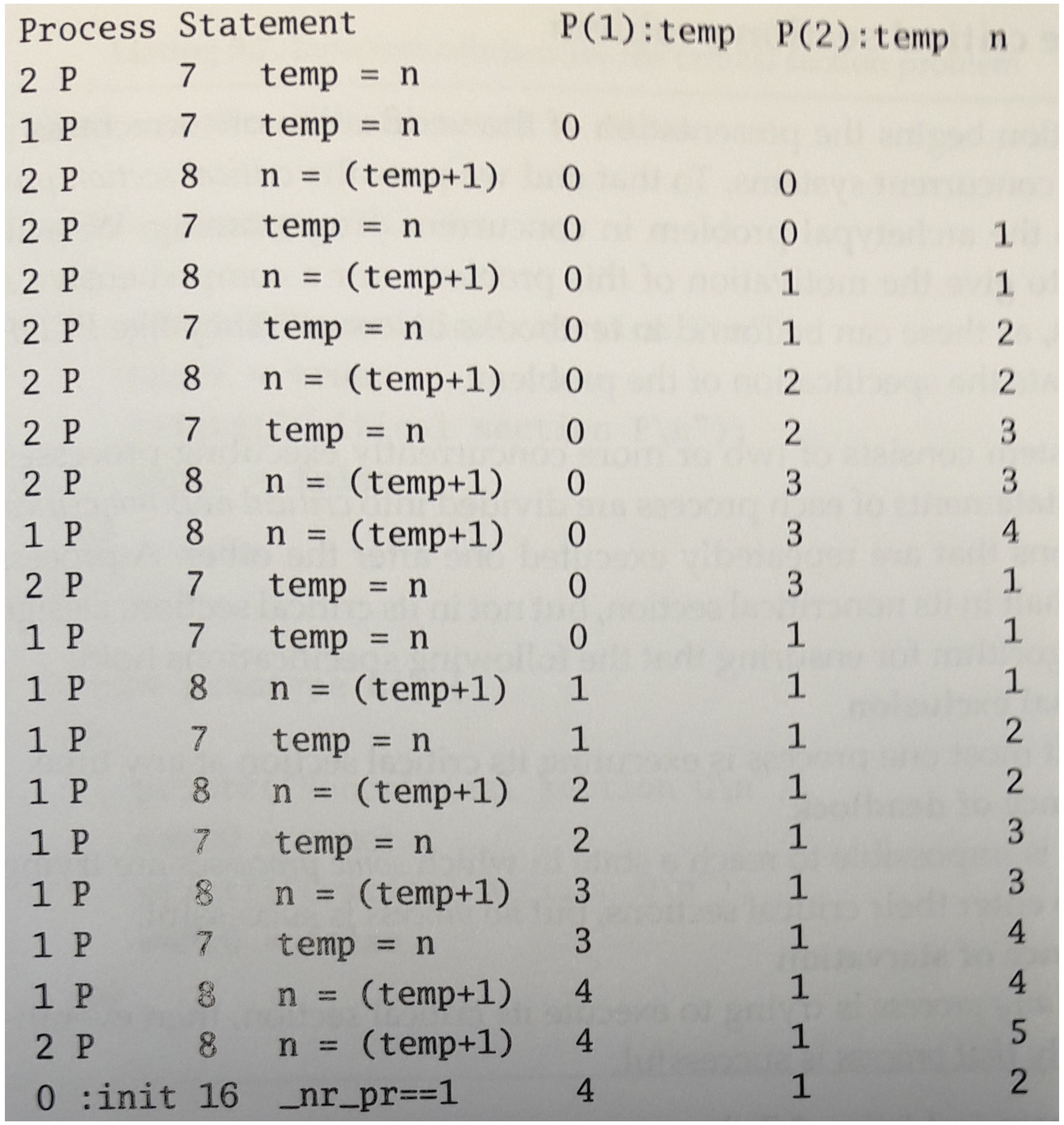
Concurrent Counting Algorithm (Revisited)

Example: Concurrent Counting Algorithm	
integer n ← 0;	
p	**q**
integer temp	integer temp
p1: do 10 times	q1: do 10 times
p2: temp ← n	q2: temp ← n
p3: n ← temp + 1	q3: n ← temp + 1

- Increments a global variable *n* 20 times, thus *n* should be 20 after execution.
- But, the program is faulty.
    - Proof: construct a scenario where *n is 2* afterwards.
- Wouldn't it be nice to get a program to do this analysis?

????

- Discovered by M. Ben-Ari during his concurrency course
  - Student puzzled him by observing a sum equal to 9
  - He modelled it and found it could be as low as 2, but no lower
  - On the right, running Promela
    - with a loop of length 5 rather than 10
    - a final assertion that n>2
    - This is the counterexample resulting in not(n>2), i.e., n=2.



```
Process Statement         P(1):temp  P(2):temp  n
2 P      7   temp = n
1 P      7   temp = n       0
2 P      8   n = (temp+1)   0          0
2 P      7   temp = n       0          0         1
2 P      8   n = (temp+1)   0          1         1
2 P      7   temp = n       0          1         2
2 P      8   n = (temp+1)   0          2         2
2 P      7   temp = n       0          2         3
2 P      8   n = (temp+1)   0          3         3
1 P      8   n = (temp+1)   0          3         4
2 P      7   temp = n       0          3         1
1 P      7   temp = n       0          1         1
1 P      8   n = (temp+1)   1          1         1
1 P      7   temp = n       1          1         2
1 P      8   n = (temp+1)   2          1         2
1 P      7   temp = n       2          1         3
1 P      8   n = (temp+1)   3          1         3
1 P      7   temp = n       3          1         4
1 P      8   n = (temp+1)   4          1         4
2 P      8   n = (temp+1)   4          1         5
0 :init 16   _nr_pr==1      4          1         2
```

Mordechai Ben-Ari, "Principles of the Spin Model-Checker", Springer, 2006

2 ways to run SPIN

- SPIN can be run in one of two modes: *Simulation* and *Verification*
- *Simulation*: SPIN performs ***one*** possible run of the system, making its own choices
  - such runs are often referred to as "*Scenarios*"
  - usually choices are random, and we can use command-line options to control the randomness
  - SPIN can also do a guided simulation, taking input from a so-called "trail" file (see below)
- *Verification*: SPIN systematically searches over ***all*** possible runs of the system
  - Checking for the truth of desirable properties
  - If a check fails, it outputs a *Counter-example*.
- *Counter-example:* a run of the system that leads to a property failure
  - output to a "trail" file

Sumofhellos mis-using mutexes

- Let's produce a version of sumofhellos that uses mutexes in an incorrect manner. (bad_`sumofhellos.pml`).
  - This is the advantage of a modelling language over a program language:
    We can say "make it so!" (to some extent, at least).
- As before, we can run simulations of this, and we observe failure.
- Again, we can build the state diagram and check every possible path through it, so determining the outcome of every possible interleaving
  - `$> spin -run bad_sumofhellos.pml`
- In this case we not only see an error indication, but a "trail" file has been created, which we can run with
  - `$> spin -p -k bad_sumofhellos.pml.trail bad_sumofhellos.pml`