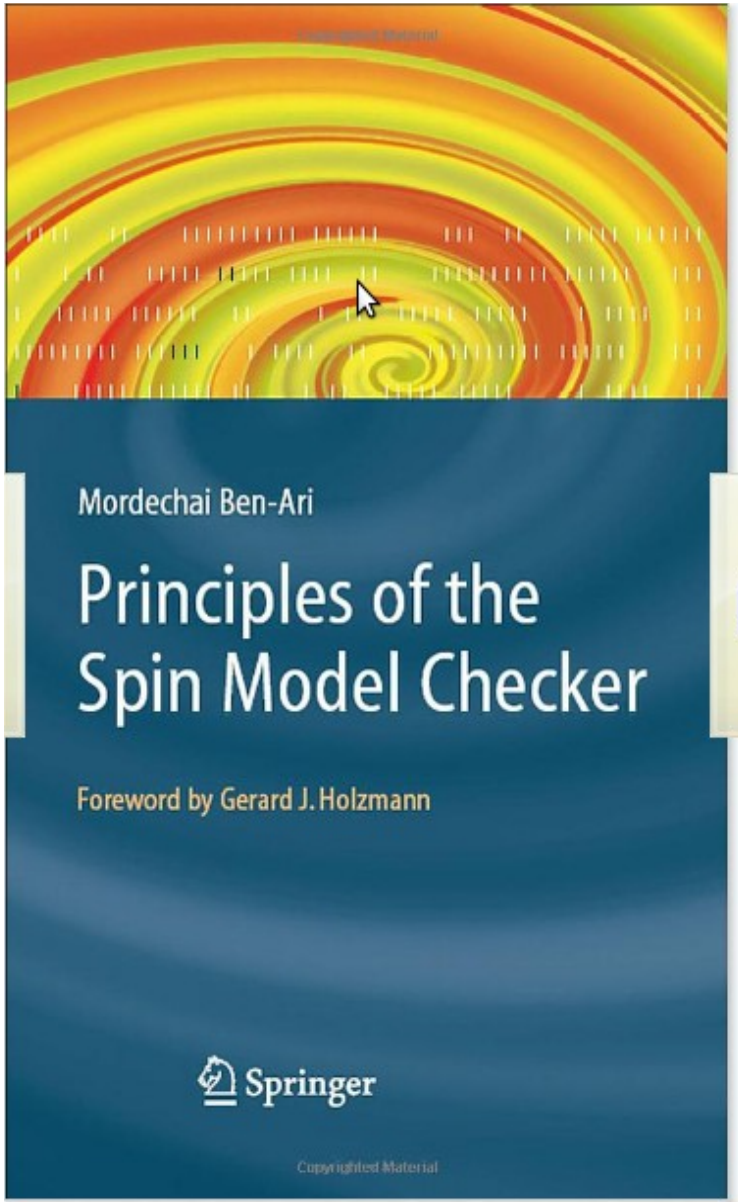


Mordechai Ben-Ari

Principles of the Spin Model Checker

Springer, 2008

ISBN: 978-1-84628-769-5

http://www.springer.com/computer/swe/book/978-1-84628-769-5

Supplementary material (zip, 38 kB)

978-1-84628-769-5-additional material.zip

Chapter 2

Verification of Sequential Programs

Section 2.1

Assertions

Listing 1.1. Reversing digits (rev.pml)

```
1  active proctype P() {
2    int value = 123; /* int or byte? */
3    int reversed;
4    reversed =
5      (value % 10) * 100 +
6      ((value / 10) % 10) * 10 +
7      (value / 100);
8    printf("value = %d, reversed = %d\n", value, reversed)
9  }
```

A ***state*** of a program is

a set of values of its variables and location counters.

For example, a state of the program in Listing 1.1 is a triple such as (**123**, **321**, **8**), where

the first element is the value of the variable **`value`**,

the second is the value of of the variable **`reversed`**, and

the third shows that the **location counter** is before the **`printf`** statement in line 8.

A ***computation*** of a program is

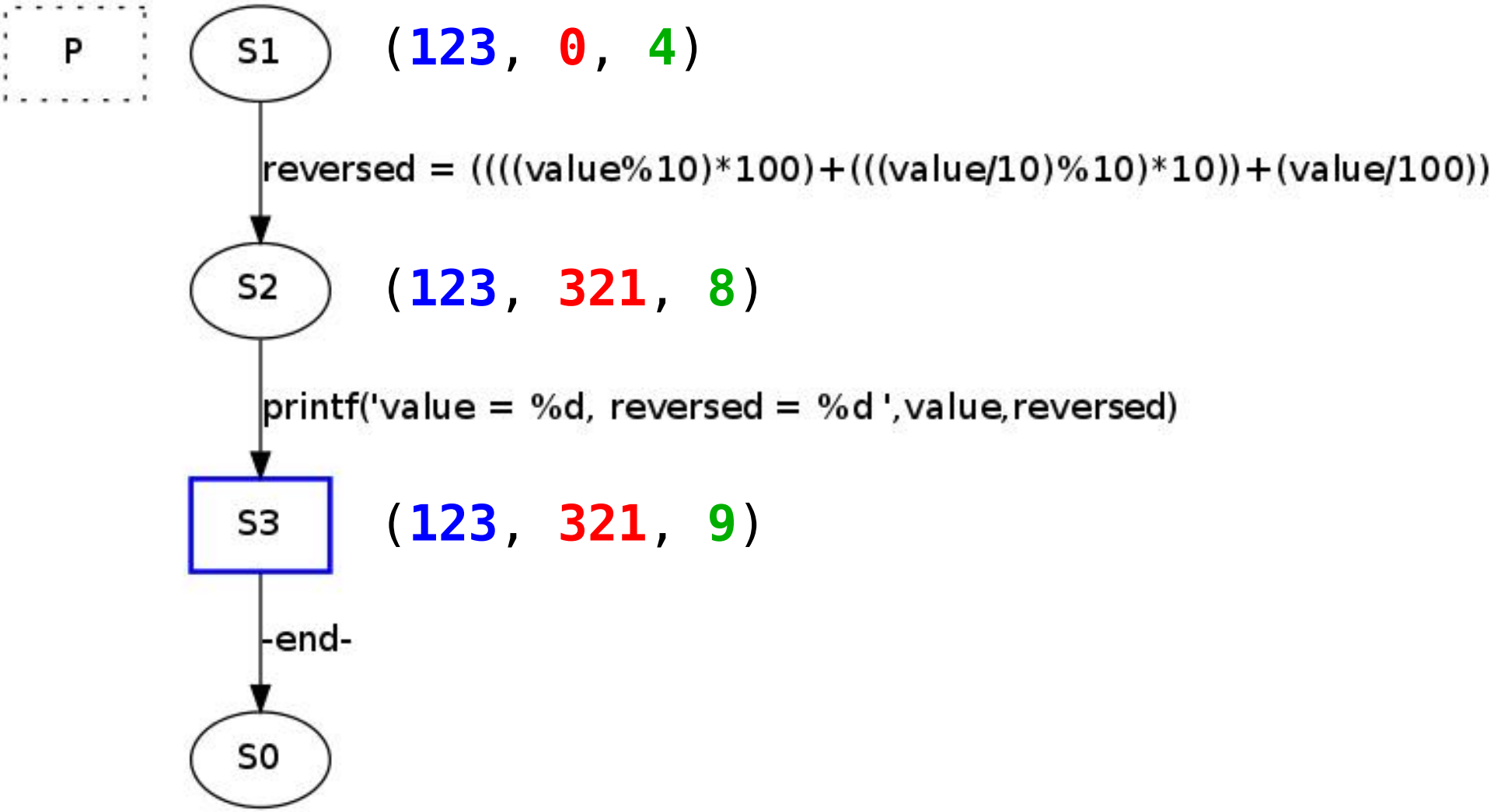
a sequence of states beginning with an initial state and continuing with the states that occur as each statement is executed.

The ***state space*** of a program is

the set of states that can *possibly* occur during a computation.

There is only one computation for the program in Listing 1.1:

```
(123, 0, 4) -> (123, 321, 8) -> (123, 321, 9)
```



```
$ ls -l rev.pml
-rw------- 1 vk vk 277 jul 23 02:22 rev.pml

$ spin -o3 -a rev.pml     # generate a verifier in pan.c


$ spin -H                                        # also spin --
use: spin [-option] ... [-option] file
Note: file must always be the last argument
...
-a generate a verifier in pan.c
...
-o3 turn off statement merging in verifier    # optimization
...
```

```
$ ls -l pan.* rev.pml
-rw-r--r-- 1 vk vk    276 ago 29 15:11 pan.b
-rw-r--r-- 1 vk vk 327860 ago 29 15:11 pan.c
-rw-r--r-- 1 vk vk  15907 ago 29 15:11 pan.h
-rw-r--r-- 1 vk vk   1538 ago 29 15:11 pan.m
-rw-r--r-- 1 vk vk  56161 ago 29 15:11 pan.p
-rw-r--r-- 1 vk vk  18165 ago 29 15:11 pan.t
-rw------- 1 vk vk    277 jun  3 02:22 rev.pml


$ gcc -o pan pan.c


$ ls -l pan
-rwxr-xr-x 1 vk vk  90944 ago 29 15:15 pan
```

```
$ ./pan -V
Generated by Spin Version 6.4.8 -- 2 March 2018
Compiled as: cc -o pan pan.c

pan: elapsed time 0 seconds


$ ./pan -H                                              # also ./pan --
saw option -H
Spin Version 6.4.8 -- 2 March 2018
Valid Options are:
        -a  find acceptance cycles
        -A  ignore assert() violations
        -b  consider it an error to exceed the depth-limit
        -cN stop at Nth error (defaults to -c1)
        -D  print state tables in dot-format and stop
        -d  print state tables and stop
        -e  create trails for all errors
        -E  ignore invalid end states
...
```

```
...
-f  add weak fairness (to -a or -l)
-hN use different hash-seed N:0..499 (defaults to -h0)
-hash generate a random hash-polynomial for -h0
      using a seed set with -Rsn (default 12345)
-i  search for shortest path to error
-I  like -i, but approximate and faster
-J  reverse eval order of nested unlesses
-l  find non-progress cycles -> disabled, requires compilation with -DNP
-mN max depth N steps (default=10k)
-n  no listing of unreached states
-QN set time-limit on execution of N minutes
-q  require empty chans in valid end states
-r  read and execute trail - can add -v,-n,-PN,-g,-C
-r trailfilename  read and execute trail in file
-rN read and execute N-th error trail
-C  read and execute trail - columnated output (can add -v,-n)
-r -PN read and execute trail - restrict trail output to proc N
-g  read and execute trail + msc gui support
-S  silent replay: only user defined printfs show
-RSn use randomization seed n
-rhash use random hash-polynomial and randomly choose -p_rotateN,
       -p_permute, or p_reverse
...
```

```
...
-T  create trail files in read-only mode
-t_reverse  reverse order in which transitions are explored
-tsuf replace .trail with .suf on trailfiles
-V  print SPIN version number
-v  verbose -- filenames in unreached state listing
-wN hashtable of 2^N entries (defaults to -w24)
-x  do not overwrite an existing trail file

options -r, -C, -PN, -g, and -S can optionally be followed by
a filename argument, as in '-r filename', naming the trailfile
```

print state tables in dot-format and stop

```
$ ./pan -D
digraph p_P {
size="8,10";
  GT [shape=box,style=dotted,label="P"];
  GT -> S1;
        S1 -> S2  [color=black,style=solid,label="reversed = ((((value
%10)*100)+(((value/10)%10)*10))+(value/100))"];
        S2 -> S3  [color=black,style=solid,label="printf('value = %d,
reversed = %d ',value,reversed)"];
        S3 -> S0  [color=black,style=solid,label="-end-"];
  S3 [color=blue,style=bold,shape=box];
}
```

pasar la tablas a la entrada de dot
(filter for drawing undirected graphs)

```
$ ./pan -D | dot
digraph p_P {
        graph [bb="0,0,405,370",
                size="8,10"
        ];
        node [label="\N"];
        GT       [height=0.5,
                label=P,
                pos="27,352",
                shape=box,
                style=dotted,
                width=0.75];
        S1       [height=0.5,
                pos="27,279",
                width=0.75];
        GT -> S1         [pos="e,27,297.03 27,333.81 27,325.79
27,316.05 27,307.07"];
        S2       [height=0.5,
                pos="27,192",
                width=0.75];
```

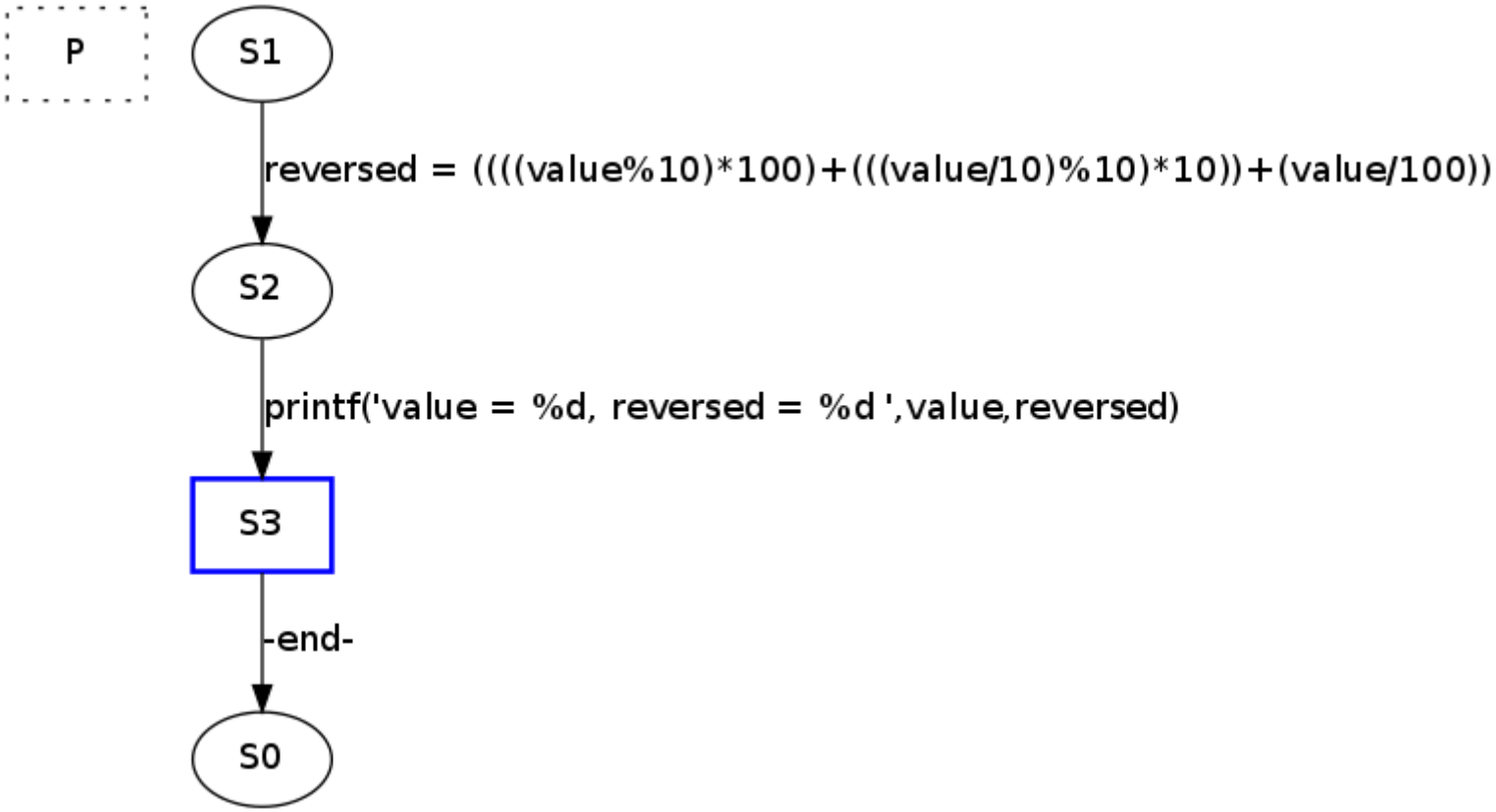
```
S1 -> S2          [color=black,
        label="reversed = ((((value%10)*100)+(((value/10)%10)*10))+
(value/100))",
        lp="216,235.5",
        pos="e,27,210.18 27,260.8 27,249.16 27,233.55 27,220.24",
        style=solid];
S3                [color=blue,
        height=0.5,
        pos="27,105",
        shape=box,
        style=bold,
        width=0.75];
S2 -> S3          [color=black,
        label="printf('value = %d, reversed = %d ',value,reversed)",
        lp="168,148.5",
        pos="e,27,123.18 27,173.8 27,162.16 27,146.55 27,133.24",
        style=solid];
S0       [height=0.5,
        pos="27,18",
        width=0.75];
S3 -> S0          [color=black,
        label="-end-",
        lp="41.5,61.5",
        pos="e,27,36.175 27,86.799 27,75.163 27,59.548 27,46.237",
        style=solid];
}
```

```
$ dot -?
Usage: dot [-Vv?] [-(GNE)name=val] [-(KTlso)<val>] <dot files>
(additional options for neato)    [-x] [-n<v>]
(additional options for fdp)      [-L(gO)] [-L(nUCT)<val>]
(additional options for memtest)  [-m]
(additional options for config)  [-cv]

 -V          - Print version and exit
 -v          - Enable verbose mode
 -Gname=val  - Set graph attribute 'name' to 'val'
 -Nname=val  - Set node attribute 'name' to 'val'
 -Ename=val  - Set edge attribute 'name' to 'val'
 -Tv         - Set output format to 'v'
 -Kv         - Set layout engine to 'v' (overrides default based on command name)
 -lv         - Use external library 'v'
 -ofile      - Write output to 'file'
 -O          - Automatically generate an output filename based on the input filename
with a .'format' appended. (Causes all -ofile options to be ignored.)
 -P          - Internally generate a graph of the current plugins.
 -q[l]       - Set level of message suppression (=1)
 -s[v]       - Scale input by 'v' (=72)
 -y          - Invert y coordinate in output

 -n[v]       - No layout mode 'v' (=1)
 -x          - Reduce graph
...
```

```
...

 -Lg         - Don't use grid
 -LO         - Use old attractive force
 -Ln<i>      - Set number of iterations to i
 -LU<i>      - Set unscaled factor to i
 -LC<v>      - Set overlap expansion factor to v
 -LT[*]<v>   - Set temperature (temperature factor) to v

 -m          - Memory test (Observe no growth with top. Kill when done.)
 -m[v]       - Memory test - v iterations.

 -c          - Configure plugins (Writes $prefix/lib/graphviz/config
               with available plugin information.  Needs write privilege.)
 -?          - Print usage and exit
```

```
$ ./pan -D | dot -Tpng -o rev.png
```

```
$ ls -l rev.png
-rw-r--r-- 1 vk vk 21698 ago 29 15:30 rev.png
```




```
$ ./pan -D | dot -Tjpg -o rev.jpg
$ ls -l rev.jpg
-rw-r--r-- 1 vk vk 19235 ago 29 15:34 rev.jpg
```

print state tables and stop

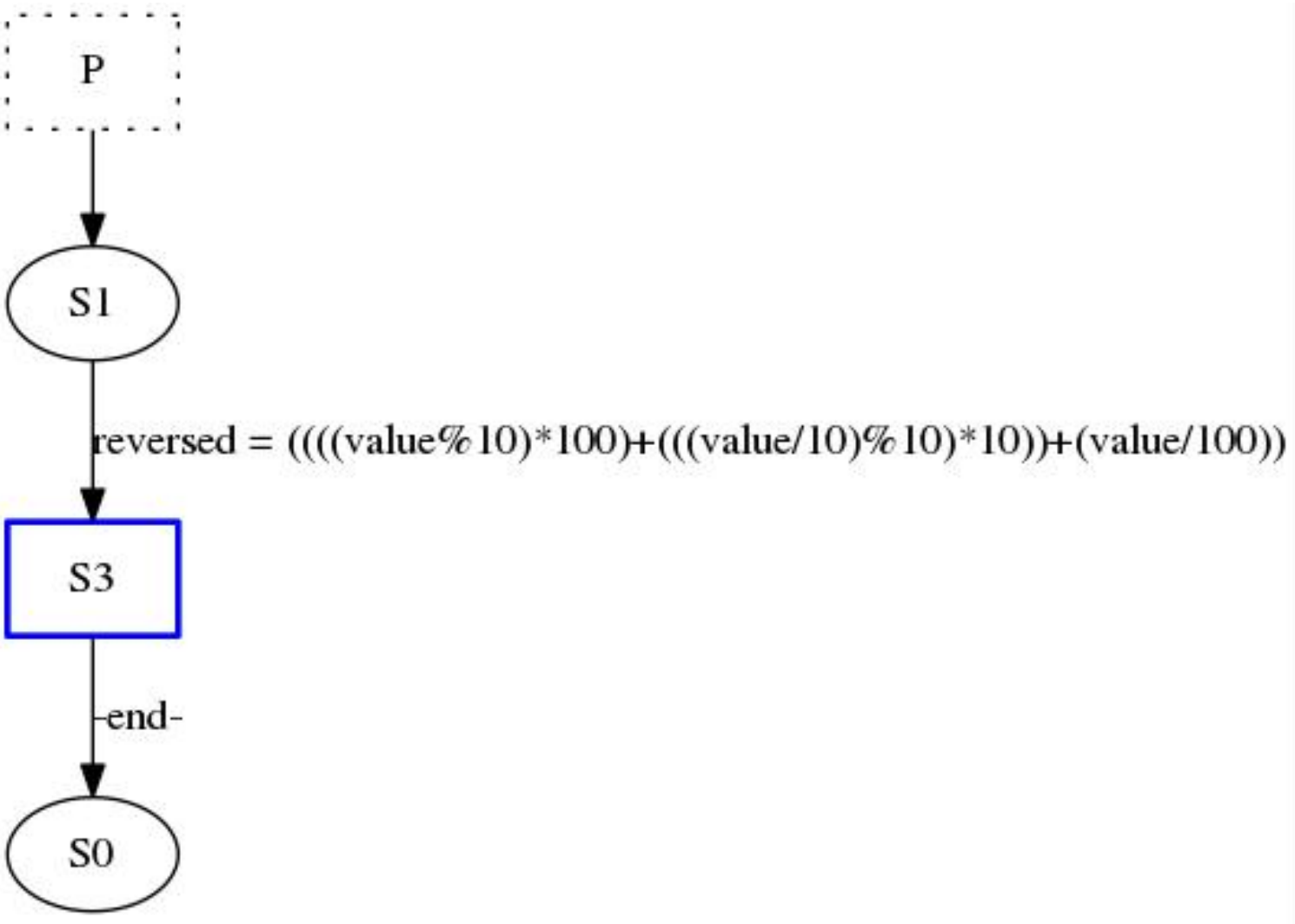
```
$ ./pan -d
proctype P
        state   1 -(tr   3)-> state   2  [id   0 tp   2] [----L]
rev.pml:6 => reversed = ((((value%10)*100)+(((value/10)%10)*10))+
(value/100))
        state   2 -(tr   4)-> state   3  [id   1 tp   2] [----L]
rev.pml:10 => printf('value = %d, reversed = %d\n',value,reversed)
        state   3 -(tr   5)-> state   0  [id   2 tp 3500] [--e-L]
rev.pml:11 => -end-

Transition Type: A=atomic; D=d_step; L=local; G=global
Source-State Labels: p=progress; e=end; a=accept;

pan: elapsed time 1.76e+07 seconds
pan: rate         0 states/second
```

turn ON statement merging in verifier

```
$ spin -a rev.pml      # generate a verifier in pan.c

$ gcc -o pan pan.c

$ ls -l pan
-rwxr-xr-x 1 vk vk 90944 ago 29 15:42 pan

$ ./pan -d
proctype P
        state   1 -(tr   3)-> state   3  [id   0 tp   2] [----L] rev.pml:6 =>
reversed = ((((value%10)*100)+(((value/10)%10)*10))+(value/100))
        state   3 -(tr   4)-> state   0  [id   2 tp 3500] [--e-L] rev.pml:11
=> -end-

Transition Type: A=atomic; D=d_step; L=local; G=global
Source-State Labels: p=progress; e=end; a=accept;
Note: statement merging was used. Only the first
      stmnt executed in each merge sequence is shown
      (use spin -a -o3 to disable statement merging)

pan: elapsed time 1.76e+07 seconds
pan: rate         0 states/second
```

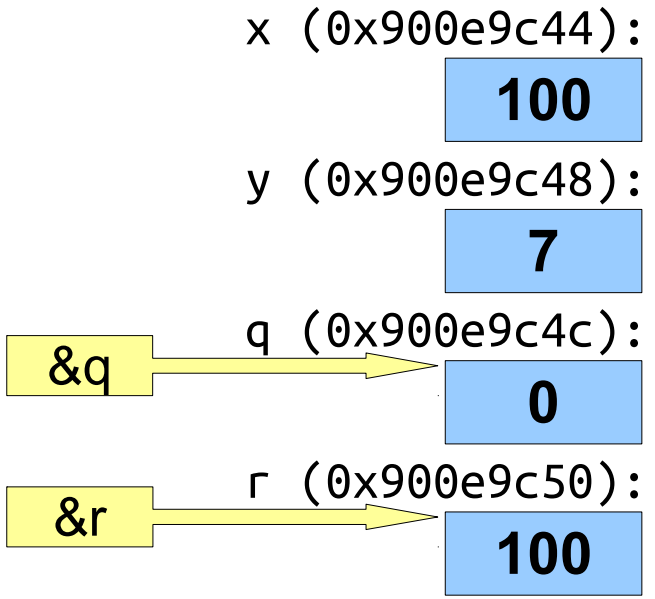
```
$ ./pan -D | dot -Tjpg -o rev.jpg
```

```
$ ls -l rev.jpg
-rw-r--r-- 1 vk vk 13795 ago 29 15:45 rev.jpg
```



Integer division (divmod_v0.c) (1/18)

```
$ cat -n divmod_v0.c
     1 #include <stdio.h>
     2
     3 void divmod(int x, int y, int* q, int* r)
     4 {
     5     *r=x; *q=0;
     6     while (*r>y) {
     7         *r=*r-y; (*q)++;
     8     }
     9 }
    10
    11 void main(void)
    12 {
    13     int x, y=7, q, r, i;
    14
    15     for (i=0; i<5; i++) {
    16         x=100+i;
    17         divmod(x,y,&q,&r);
    18         printf("%d = %d*%d + %d\n",x,y,q,r);
    19     }
    20 }
```



&r = 0x900e9c50
*(&r) = 100

Integer division (divmod_v0.c) (2/18)

```
$ gcc divmod_v0.c -o divmod_v0

$ ./divmod_v0
100 = 7*14 + 2
101 = 7*14 + 3
102 = 7*14 + 4
103 = 7*14 + 5
104 = 7*14 + 6
```

Realmente, 7*14 = 98
98 + 2 = 100

Integer division (divmod_v1.c) (3/18)

```
$ cat -n divmod_v1.c
     1 #include <stdio.h>
     2
     3 void divmod(int x, int y, int* q, int* r)
     4 {
     5     printf("dividend x=%d, divisor y=%d\n",x,y);
     6     *r=x; *q=0;
     7     while (*r>y) {
     8         *r=*r-y; (*q)++;
     9     }
    10     printf("y*q + r = %d\n\n",y*(*q)+(*r));
    11 }
    12
    13 void main(void)
    14 {
    15     int x, y=7, q, r, i;
    16
    17     for (i=0; i<5; i++) {
    18         x=100+i;
    19         divmod(x,y,&q,&r);
    20     }
    21 }
```

Para asegurarnos, quitamos printf de main() y los añadimos a la función

Integer division (divmod_v1.c) (4/18)

```
$ gcc divmod_v1.c -o divmod_v1

$ ./divmod_v1
dividend x=100, divisor y=7
y*q + r = 100

dividend x=101, divisor y=7
y*q + r = 101

dividend x=102, divisor y=7
y*q + r = 102

dividend x=103, divisor y=7
y*q + r = 103

dividend x=104, divisor y=7
y*q + r = 104

$
```

Una mala idea.
La función se usa muchas veces en bucles,
y hay demasiada impresión.

Integer division (divmod_v2.c) (5/18)

```
$ cat -n divmod_v2.c
     1 #include <assert.h>
     2 #include <stdio.h>
     3
     4 void divmod(int x, int y, int* q, int* r)
     5 {
     6     assert(y>0);
     7     *r=x; *q=0;
     8     while (*r>y) {
     9         *r=*r-y; (*q)++;
    10     }
    11     assert(x==y*(*q)+(*r));
    12 }
    13
    14 void main(void)
    15 {
    16     int x, y=7, q, r, i;
    17
    18     for (i=0; i<5; i++) {
    19         x=100+i;
    20         divmod(x,y,&q,&r);
    21     }
    22 }
```

Una mejor idea:
la macro assert

Integer division (divmod_v2_error.c) (6/18)

```
$ cat -n divmod_v2.c
     1 #include <assert.h>
     2 #include <stdio.h>
     3
     4 void divmod(int x, int y, int* q, int* r)
     5 {
     6     assert(y>0);
     7     *r=x; *q=0;
     8     while (*r>y) {
     9         *r=*r-y; (*q)++;
    10     }
    11     assert(x==y*(*q)+(*r));
    12 }
    13
    14 void main(void)
    15 {
    16     int x, y=0, q, r, i;
    17
    18     for (i=0; i<5; i++) {
    19         x=100+i;
    20         divmod(x,y,&q,&r);
    21     }
    22 }
```

Y, si hay un error, …

Integer division (divmod_v2_error.c) (7/18)

```
$ gcc divmod_v2_error.c -o divmod_v2_error

$ ./divmod_v2_error
divmod_v2_error: divmod_v2_error.c:6: divmod: Assertion 'y>0' failed.
Abortado ('core' generado)
```

… el error se captura.

Integer division (divmod_v3.c) (8/18)

```
$ cat -n divmod_v3.c
     1 #include <assert.h>
     2 #include <stdio.h>
     3
     4 void divmod(int x, int y, int* q, int* r)
     5 {
     6     assert(y>0);
     7     *r=x; *q=0;
     8     while (*r>y) {
     9         *r=*r-y; (*q)++;
    10     }
    11     assert(x==y*(*q)+(*r));
    12 }
    13
    14 void main(void)
    15 {
    16     int x, y=7, q, r, i;
    17
    18     for (i=0; i<7; i++) {
    19         x=100+i;
    20         divmod(x,y,&q,&r);
    21         printf("%d = %d*%d + %d\n",x,y,q,r);
    22     }
    23 }
```

Reestablecemos el divisor y aumentemos la cantidad de iteraciones …

Integer division (divmod_v3.c) (9/18)

```
$ gcc divmod_v3.c -o divmod_v3

$ ./divmod_v3
100 = 7*14 + 2
101 = 7*14 + 3
102 = 7*14 + 4
103 = 7*14 + 5
104 = 7*14 + 6
105 = 7*14 + 7
106 = 7*15 + 1
```

… obteniendo un resto imposible

Integer division (divmod_v4.c) (10/18)

```
$ cat -n divmod_v4.c
     1 #include <assert.h>
     2 #include <stdio.h>
     3
     4 void divmod(int x, int y, int* q, int* r)
     5 {
     6     assert(y>0);
     7     *r=x; *q=0;
     8     while (*r>y) {
     9         *r=*r-y; (*q)++;
    10     }
    11     assert(x==y*(*q)+(*r) && *r<y);
    12 }
    13
    14 void main(void)
    15 {
    16     int x, y=7, q, r, i;
    17
    18     for (i=0; i<7; i++) {
    19         x=100+i;
    20         divmod(x,y,&q,&r);
    21         printf("%d = %d*%d + %d\n",x,y,q,r);
    22     }
    23 }
```

Ajustamos la postcondición, …

Integer division (divmod_v4.c) (11/18)

```
$ gcc divmod_v4.c -o divmod_v4

$ ./divmod_v4
100 = 7*14 + 2
101 = 7*14 + 3
102 = 7*14 + 4
103 = 7*14 + 5
104 = 7*14 + 6
divmod_v4: divmod_v4.c:11: divmod: Assertion 'x==y*(*q)+(*r) && *r<y'
failed.
Abortado ('core' generado)
```

… y capturamos el error

Integer division (divmod_v5.c) (12/18)

```
$ cat -n divmod_v5.c
     1 #include <assert.h>
     2 #include <stdio.h>
     3
     4 void divmod(int x, int y, int* q, int* r)
     5 {
     6     assert(y>0);
     7     *r=x; *q=0;
     8     while (*r>=y) {
     9         *r=*r-y; (*q)++;
    10     }
    11     assert(x==y*(*q)+(*r) && *r<y);
    12 }
    13
    14 void main(void)
    15 {
    16     int x, y=7, q, r, i;
    17
    18     for (i=0; i<7; i++) {
    19         x=100+i;
    20         divmod(x,y,&q,&r);
    21         printf("%d = %d*%d + %d\n",x,y,q,r);
    22     }
    23 }
```

Este error nos acompañaba mucho tiempo

Integer division (divmod_v5_error.c) (13/18)

```
$ cat -n divmod_v5_error.c
     1 #include <assert.h>
     2 #include <stdio.h>
     3
     4 void divmod(int x, int y, int* q, int* r)
     5 {
     6     assert(y>0);
     7     *r=x; *q=0;
     8     while (*r>=y) {
     9         *r=*r-y; (*q)++;
    10     }
    11     assert(x==y*(*q)+(*r) && *r<y);
    12 }
    13
    14 void main(void)
    15 {
    16     int x, y=7, q, r, i;
    17
    18     for (i=0; i<7; i++) {
    19         x=-100+i;
    20         divmod(x,y,&q,&r);
    21         printf("%d = %d*%d + %d\n",x,y,q,r);
    22     }
    23 }
```

Todavía tenemos errores …

Integer division (divmod_v5_error.c) (14/18)

```
$ gcc divmod_v5_error.c -o divmod_v5_error

$ ./divmod_v5_error
-100 = 7*0 + -100
-99 = 7*0 + -99
-98 = 7*0 + -98
-97 = 7*0 + -97
-96 = 7*0 + -96
-95 = 7*0 + -95
-94 = 7*0 + -94
```

… que son muy graves

Integer division (divmod_v6.c) (15/18)

```
$ cat -n divmod_v6.c
     1 #include <assert.h>
     2 #include <stdio.h>
     3
     4 void divmod(int x, int y, int* q, int* r)
     5 {
     6     assert(0<=x && 0<y);
     7     *r=x; *q=0;
     8     while (*r>=y) {
     9         *r=*r-y; (*q)++;
    10     }
    11     assert(x==y*(*q)+(*r) && *r<y);
    12 }
    13
    14 void main(void)
    15 {
    16     int x, y=7, q, r, i;
    17
    18     for (i=0; i<7; i++) {
    19         x=-100+i;
    20         divmod(x,y,&q,&r);
    21         printf("%d = %d*%d + %d\n",x,y,q,r);
    22     }
    23 }
```

… pero se puede capturarlos

Integer division (divmod_v6.c) (16/18)

```
$ gcc divmod_v6.c -o divmod_v6

$ ./divmod_v6
divmod_v6: divmod_v6.c:6: divmod: Assertion '0<=x && 0<y' failed.
Abortado ('core' generado)
```

Integer division (divmod_v7.c) (17/18)

```
$ cat -n divmod_v7.c
     1 #include <assert.h>
     2 #include <stdio.h>
     3
     4 void divmod(int x, int y, int* q, int* r)
     5 {
     6     assert(0<=x && 0<y);
     7     *r=x; *q=0;
     8     while (*r>=y) {
     9         *r=*r-y; (*q)++;
    10     }
    11     assert(x==y*(*q)+(*r) && *r<y);
    12 }
    13
    14 void main(void)
    15 {
    16     int x, y=7, q, r, i;
    17
    18     for (i=0; i<7; i++) {
    19         x=100+i;
    20         divmod(x,y,&q,&r);
    21         printf("%d = %d*%d + %d\n",x,y,q,r);
    22     }
    23 }
```

Parece que ya no hay errores …
¿Pero el bucle estará correcto?
Ya hemos hecho un error …

Integer division (divmod_v8.c) (18/18)

```
$ cat -n divmod_v8.c
     1  #include <assert.h>
     2  #include <stdio.h>
     3
     4  void divmod(int x, int y, int* q, int* r)
     5  {
     6      assert(0<=x && 0<y);
     7      *r=x; *q=0;
     8      assert(0<=*r && 0<y && x==y*(*q)+(*r));
     9      while (*r>=y) {
    10          assert(0<=*r && 0<y && y<=*r && x==y*(*q)+(*r));
    11          *r=*r-y; (*q)++;
    12          assert(0<=*r && 0<y && x==y*(*q)+(*r));
    13      }
    14      assert(0<=*r && *r<y && x==y*(*q)+(*r));
    15  }
    16
    17  void main(void)
    18  {
    19      int x, y=7, q, r, i;
    20
    21      for (i=0; i<7; i++) {
    22          x=100+i;
    23          divmod(x,y,&q,&r);
    24          printf("%d = %d*%d + %d\n",x,y,q,r);
    25      }
    26  }
```

El invariante del bucle es cierto a su inicio, se mantiene en cada iteración y, al salir del bucle, se convierte en la post-condición.

Integer division (divmod_v0.py) (1/7)

```
$ cat -n divmod_v0.py | expand
     1  def divMod(x,y):
     2      r,q= x,0
     3      while r>y:
     4          r,q= r-y,q+1
     5      return q,r
     6
     7  q,r= divmod(22,7)
     8  print()
     9  print("divmod (builtin function): 22 = 7 *",q,'+',r)
    10  print()
    11
    12  for i in range(5):
    13      q,r= divMod(100+i,7)
    14      print("%d = %d*%d + %d" % (100+i,7,q,r))

$ python3 divmod_v0.py

divmod (builtin function): 22 = 7 * 3 + 1

100 = 7*14 + 2
101 = 7*14 + 3
102 = 7*14 + 4
103 = 7*14 + 5
104 = 7*14 + 6
```

Integer division (divmod_v2.py) (2/7)

```
$ cat -n divmod_v2.py | expand
     1  def divMod(x,y):
     2      assert y>0, "The divisor is 0"
     3      r,q= x,0
     4      while r>y:
     5          r,q= r-y,q+1
     6      assert x==y*q+r, "Function error"
     7      return q,r
     8
     9  q,r= divmod(22,7)
    10  print()
    11  print("divmod (builtin function): 22 = 7 *",q,'+',r)
    12  print()
    13
    14  for i in range(5):
    15      q,r= divMod(100+i,7)
    16      print("%d = %d*%d + %d" % (100+i,7,q,r))
```

Integer division (divmod_v2_error.py) (3/7)

```
$ cat -n divmod_v2_error.py | expand
     1  def divMod(x,y):
     2      assert y>0, "The divisor is 0"
...
    15      q,r= divMod(100+i,0)
...


$ python3 divmod_v2_error.py

divmod (builtin function): 22 = 7 * 3 + 1

Traceback (most recent call last):
  File "divmod_v2_error.py", line 15, in <module>
    q,r= divMod(100+i,0)
  File "divmod_v2_error.py", line 2, in divMod
    assert y>0, "The divisor is 0"
AssertionError: The divisor is 0
```

Integer division (divmod_v3.py) (4/7)

```
$ cat -n divmod_v3.py | expand
     1  def divMod(x,y):
     2      assert y>0, "The divisor is 0"
     3      r,q= x,0
     4      while r>y:
     5          r,q= r-y,q+1
     6      assert x==y*q+r, "Function error"
     7      return q,r
     8
     9  for i in range(7):
    10      q,r= divMod(100+i,7)
    11      print("%d = %d*%d + %d" % (100+i,7,q,r))

$ python3 divmod_v3.py
100 = 7*14 + 2
101 = 7*14 + 3
102 = 7*14 + 4
103 = 7*14 + 5
104 = 7*14 + 6
105 = 7*14 + 7
106 = 7*15 + 1
```

Integer division (divmod_v4.py) (5/7)

```
$ cat -n divmod_v4.py | expand
     1  def divMod(x,y):
     2      assert y>0, "The divisor is 0"
     3      r,q= x,0
     4      while r>y:
     5          r,q= r-y,q+1
     6      assert x==y*q+r and r<y, "Function error"
     7      return q,r
     8
...

$ python3 divmod_v4.py
100 = 7*14 + 2
101 = 7*14 + 3
102 = 7*14 + 4
103 = 7*14 + 5
104 = 7*14 + 6
Traceback (most recent call last):
  File "divmod_v4.py", line 10, in <module>
    q,r= divMod(100+i,7)
  File "divmod_v4.py", line 6, in divMod
    assert x==y*q+r and r<y, "Function error"
AssertionError: Function error
```

Integer division (divmod_v6.py) (6/7)

```
$ cat -n divmod_v6.py | expand
     1  def divMod(x,y):
     2      assert x>=0 and y>0, "Wrong input"
     3      r,q= x,0
     4      while r>=y:
     5          r,q= r-y,q+1
     6      assert x==y*q+r and r<y, "Function error"
     7      return q,r
     8
     9  for i in range(7):
    10      q,r= divMod(100+i,7)
    11      print("%d = %d*%d + %d" % (100+i,7,q,r))
```

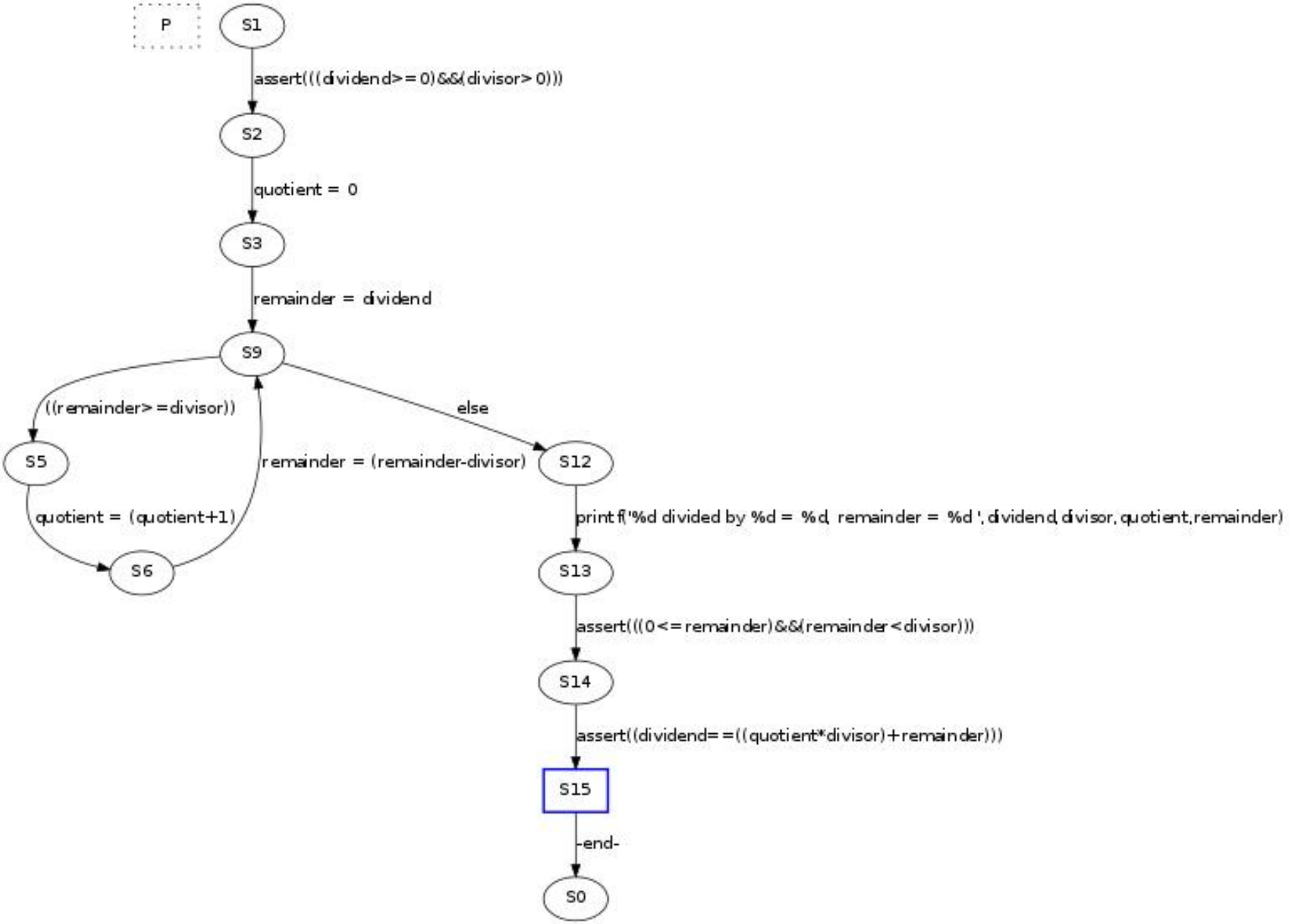
Integer division (divmod_v8.py) (7/7)

```
$ cat -n divmod_v8.py | expand
     1  def divMod(x,y):
     2      assert x>=0 and y>0, "Wrong input"  # precondition
     3      r,q= x,0
     4      while r>=y:
     5          assert r>=0 and y>0 and y<=r and x==y*q+r, "Invariant
error"
     6          r,q= r-y,q+1
     7          assert r>=0 and y>0 and x==y*q+r, "Invariant error"
     8      assert x==y*q+r and 0<=r and r<y, "Function error"   #
postcondition
     9      return q,r
    10
    11  for i in range(7):
    12      q,r= divMod(100+i,7)
    13      print("%d = %d*%d + %d" % (100+i,7,q,r))
```

Listing 2.1. Integer division (`divide1.pml`)

```
 1  active proctype P() {
 2    int dividend = 15;
 3    int divisor  = 4;
 4    int quotient, remainder;
 5
 6    assert (dividend >= 0 && divisor > 0); /* precondition */
 7
 8    quotient = 0;
 9    remainder = dividend;
10    do
11    :: remainder >= divisor ->
12         quotient++;
13         remainder = remainder - divisor
14    :: else ->
15         break
16    od;
17    printf("%d divided by %d = %d, remainder = %d\n",
18           dividend, divisor, quotient, remainder);
19    /* postcondition */
20    assert (0 <= remainder && remainder < divisor);
21    assert (dividend == quotient * divisor + remainder);
22  }
```

Listing 2.1. Integer division (divide1.pml)



```
$ spin divide1.pml
      15 divided by 4 = 3, remainder = 3
1 process created
```

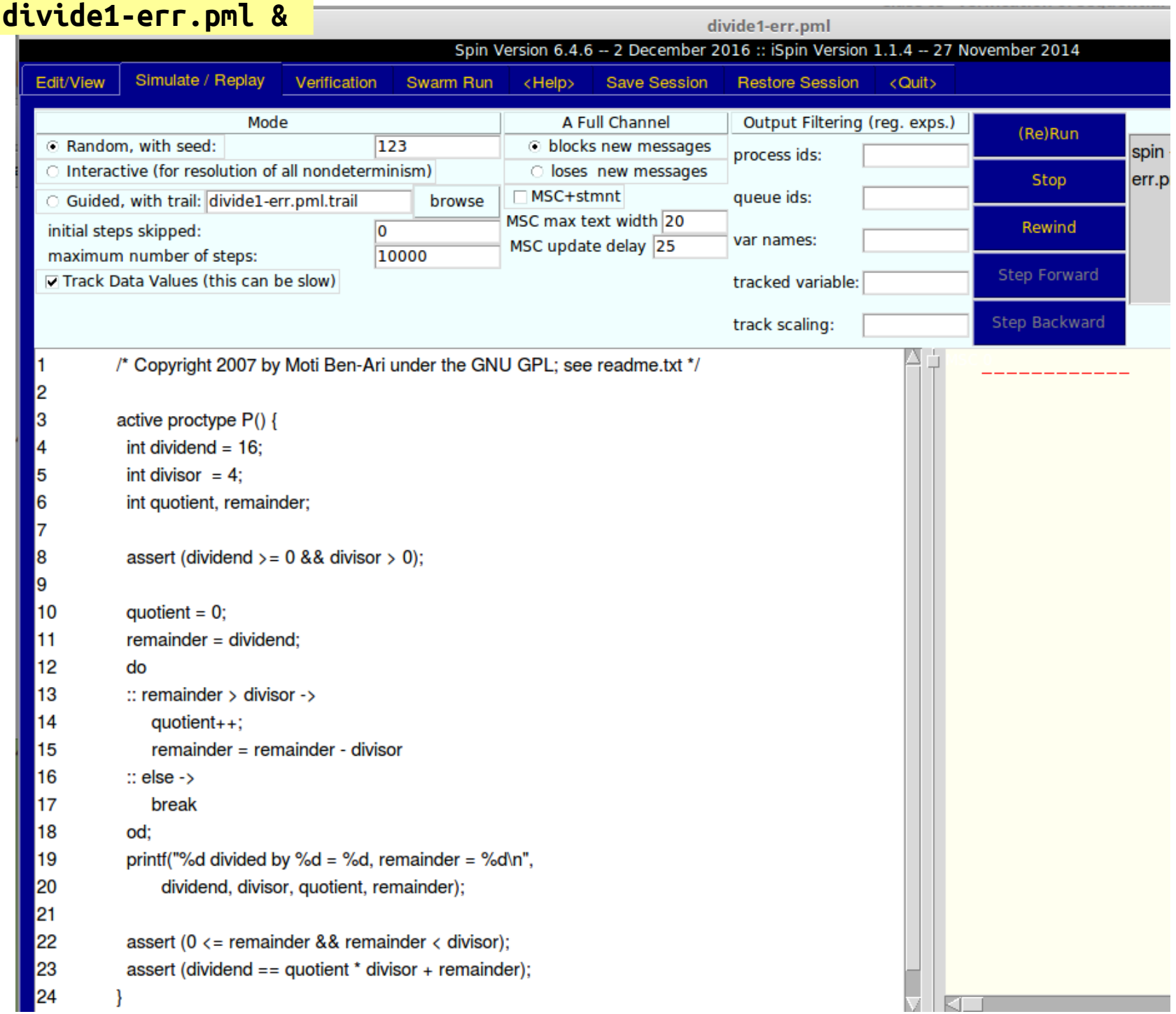
Change:

```
    2  int dividend = 15;                2  int dividend = 16;
   11 :: remainder >= divisor ->        11 :: remainder > divisor ->
```

```
$ spin divide1-err.pml
      16 divided by 4 = 3, remainder = 4
spin: divide1-err.pml:22, Error: assertion violated
spin: text of failed assertion: assert(((0<=remainder)&&(remainder<divisor)))
#processes: 1
 20:    proc  0 (P:1) divide1-err.pml:22 (state 13)
1 process created
```

```
$ spin -l divide1-err.pml                    # print all local variables
      16 divided by 4 = 3, remainder = 4
spin: divide1-err.pml:22, Error: assertion violated
spin: text of failed assertion: assert(((0<=remainder)&&(remainder<divisor)))
#processes: 1
 20:    proc  0 (P:1) divide1-err.pml:22 (state 13)
               P(0):remainder = 4
               P(0):quotient = 3
               P(0):divisor = 4
               P(0):dividend = 16
1 process created
```

```
$ ispin divide1-err.pml &
```



Data Values

```
[variable values, step  15]

 P(0):quotient  =  3
 P(0):remainder  =  4
```

Simulation Output

```
 0:      proc  - (:root:) creates proc  0 (P)
 1:      proc  0 (P:1) divide1-err.pml:8 (state 1) [assert(((dividend>=0)&&(divisor>0)))]
 2:      proc  0 (P:1) divide1-err.pml:10 (state 2) [quotient = 0]
 3:      proc  0 (P:1) divide1-err.pml:11 (state 3) [remainder = dividend]
 5:      proc  0 (P:1) divide1-err.pml:13 (state 4) [((remainder>divisor))]
 6:      proc  0 (P:1) divide1-err.pml:14 (state 5) [quotient = (quotient+1)]
 7:      proc  0 (P:1) divide1-err.pml:15 (state 6) [remainder = (remainder-divisor)]
 9:      proc  0 (P:1) divide1-err.pml:13 (state 4) [((remainder>divisor))]
10:      proc  0 (P:1) divide1-err.pml:14 (state 5) [quotient = (quotient+1)]
11:      proc  0 (P:1) divide1-err.pml:15 (state 6) [remainder = (remainder-divisor)]
13:      proc  0 (P:1) divide1-err.pml:13 (state 4) [((remainder>divisor))]
14:      proc  0 (P:1) divide1-err.pml:14 (state 5) [quotient = (quotient+1)]
15:      proc  0 (P:1) divide1-err.pml:15 (state 6) [remainder = (remainder-divisor)]
17:      proc  0 (P:1) divide1-err.pml:16 (state 7) [else]
19:      proc  0 (P:1) divide1-err.pml:19 (state 12) [printf('%d divided by %d = %d, remainder = %d\\n',dividen
d,divisor,quotient,remainder)]
spin: divide1-err.pml:22, Error: assertion violated
spin: text of failed assertion: assert(((0<=remainder)&&(remainder<divisor)))
#processes: 1
20:      proc  0 (P:1) divide1-err.pml:22 (state 13)
1 processes created
```

Listing 2.2. Another program for integer division (`divide2.pml`)

```
 3  active proctype P() {
 4  int dividend = 15, divisor  = 4;
 5  int quotient = 0, remainder = 0;
 6  int n = dividend;
 7
 8  assert (dividend >= 0 && divisor > 0);
 9
10  do
11  :: n != 0 ->
12   assert (dividend == quotient * divisor + remainder + n);
13   assert (0 <= remainder && remainder < divisor);
14    if
15    :: remainder + 1 == divisor ->
16        quotient++;
17        remainder = 0
18    :: else ->
19        remainder++
20    fi;
21    n--
22  :: else ->
23      break
24  od;
```

Listing 2.2. Another program for integer division (`divide2.pml`)

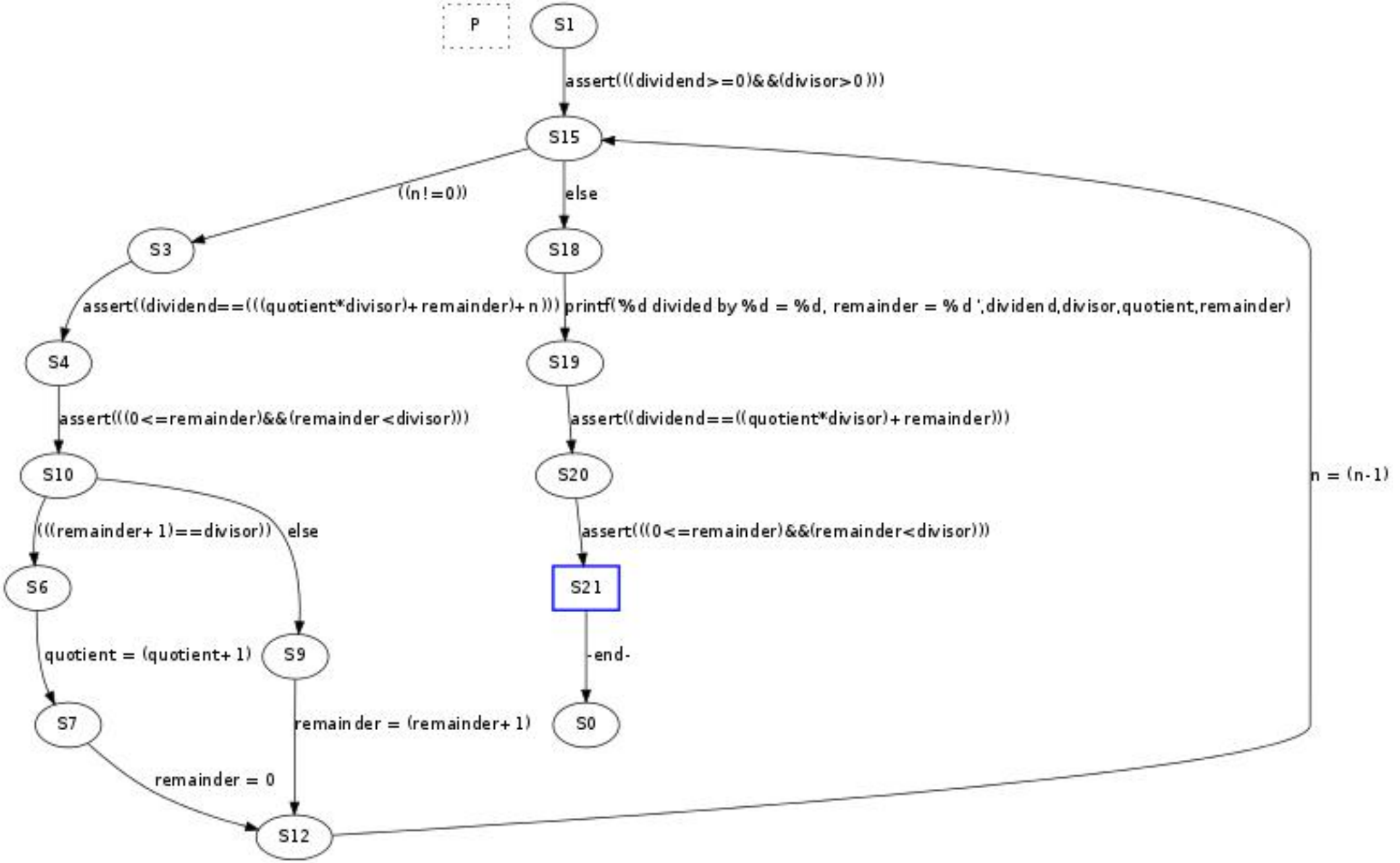
```
25  printf("%d divided by %d = %d, remainder = %d\n",
26         dividend, divisor, quotient, remainder);
27  assert (dividend == quotient * divisor + remainder);
28  assert (0 <= remainder && remainder < divisor);
29 }
```

n: 15, mientras es > 0, decrece en 1. Serán 15 iteraciones.

remainder: crece de 0 a 3, se resetea a 0 y otra vez crece.

quotient: crece en 1 cada vez cuando remainder llega a 3.

Listing 2.2. Another program for integer division (divide2.pml)



Listing 2.2. Another program for integer division (`divide2.pml`)

De n hay que extraer el cociente y el resto.
Al terminar de extraer la variable n se queda vacía (0).

Pero la suma de lo extraído y lo que se queda en la variable n **no se cambia**. (¿La ley de conservación de la materia?)

Postcondición es equivalente al ***invariante*** del bucle con n = 0:

```
    (dividend == quotient * divisor + remainder) &&
    (0 <= remainder && remainder < divisor)
≡
    (dividend == quotient * divisor + remainder + n) &&
    (0 <= remainder && remainder < divisor)
```

El invariante **se cumple** al inicio con

```
    (dividend == quotient * divisor + remainder + n) &&
    (0 <= remainder && remainder < divisor)
≡
    (15 == 0 * 4 + 0 + 15) &&
    (0 <= 0 && 0 < 4)
```

Al inicio del bucle `n == dividend`.

Entonces, según la precondición para `dividend`, `n >= 0`.

Nuestro objetivo es construir un bucle para llevar n a 0 convertiendo así el invariante a la postcondición.

Entonces, mientras n no es 0, se necesita ejecutar el cuerpo del bucle.
Por eso, la protección del bucle será `n != 0`.

Dentro del bucle n es mayor que 0, y le podemos restar 1 (`n--`).

Pero el cualquier cambio de n "desbalanceará" el invariante.
Por eso nuestro objetivo en el bucle es mantener el invariante para que él sea el invariante como tal.

Lo haremos incrementando `remainder` pero manteniéndolo siempre menor que `divisor`.

Ya tenemos el código del programa.

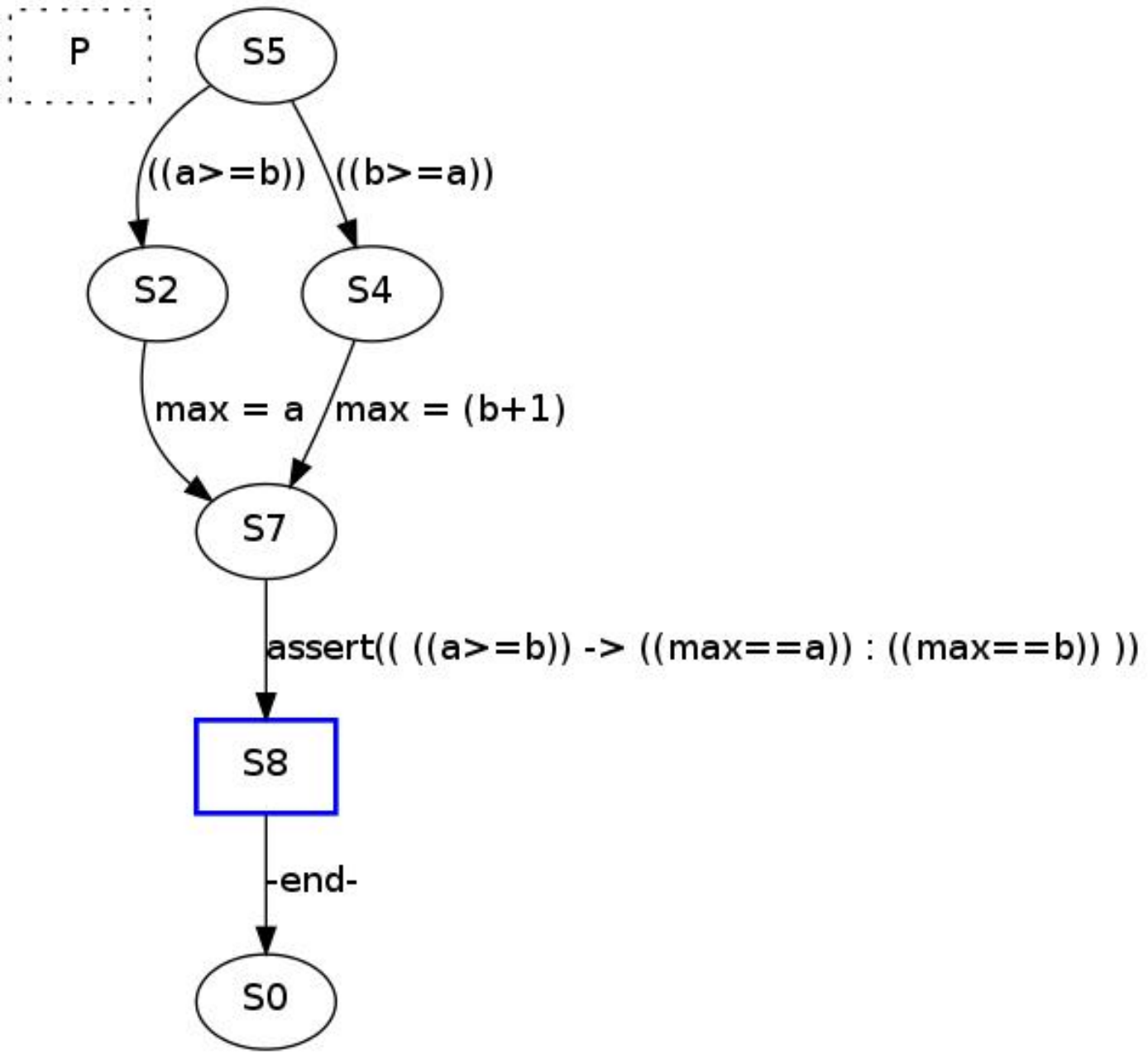
Section 2.2

Verifying a program in Spin

Listing 2.3. Maximum with an error (`max1.pml`)

```
 3  active proctype P() {
 4    int a = 5, b = 5, max;
 5    if
 6    :: a >= b -> max = a;
 7    :: b >= a -> max = b+1;
 8    fi;
 9    assert (a >= b -> max == a : max == b);
10  }
```

Listing 2.3. Maximum with an error (max1.pml)



```
$ # Random simulation

$ spin max1.pml
1 process created

$ spin max1.pml
1 process created

$ spin max1.pml
spin: max1.pml:9, Error: assertion violated
spin: text of failed assertion: assert(( ((a>=b)) -> ((max==a)) :
((max==b)) ))
#processes: 1
  4:   proc  0 (P:1) max1.pml:9 (state 7)
1 process created
```

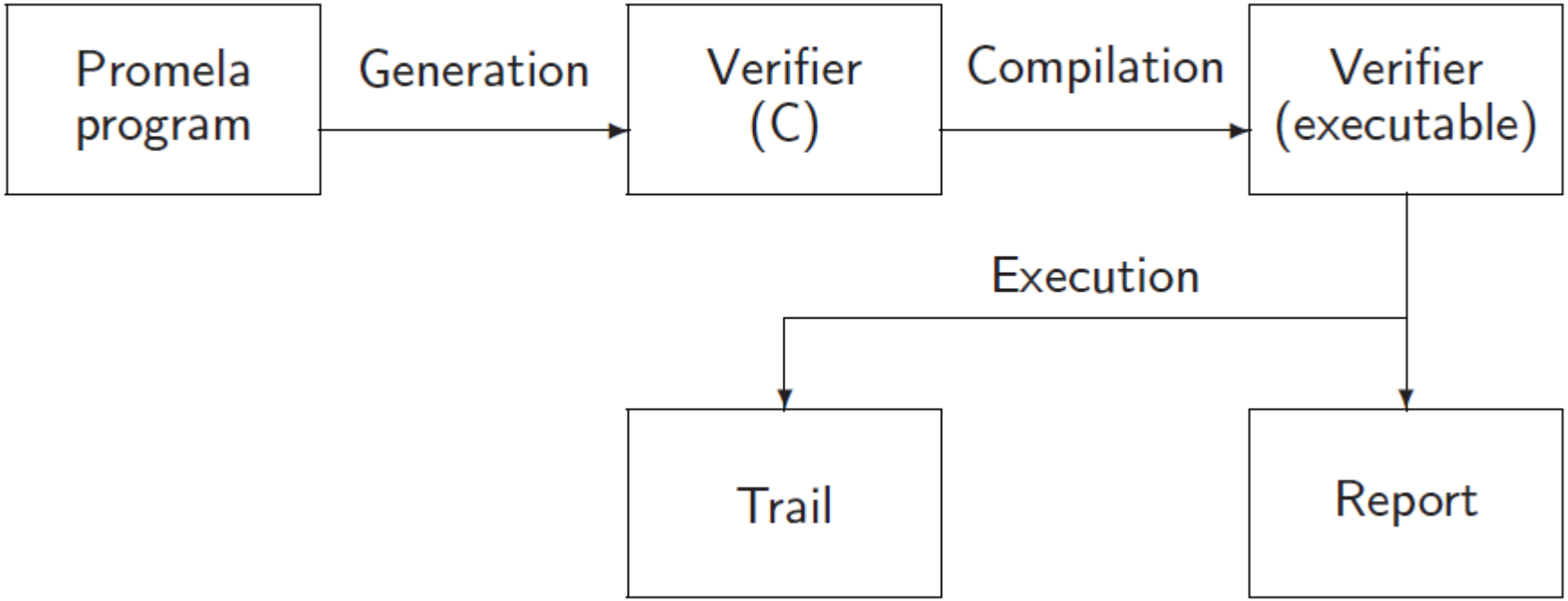
Verification in Spin is a three-step process:

- Generate the ***verifier*** (the optimized program) from the PROMELA source code.
  The verifier is a program written in C.

- Compile the verifier using a C compiler.

- Execute the verifier. The result of the execution of the verifier is a report that *all* computations are correct or else that *some* computation contains an error.

Fig. 2.1. The architecture of SPIN



```
$ cat max.pml
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

active proctype P() {
        int a = 5, b = 5;
        int max;
        int branch;
        if
        :: a >= b -> max = a; branch = 1;
        :: b >= a -> max = b; branch = 2;
        fi;
        printf("The maximum of %d and %d = %d by branch %d\n",
                        a, b, max, branch);
}
$ spin -a max.pml
$ ls -l pan.*
-rw-rw-r-- 1 vk vk    550 ago 31 22:25 pan.b
-rw-rw-r-- 1 vk vk 328142 ago 31 22:25 pan.c
-rw-rw-r-- 1 vk vk  15936 ago 31 22:25 pan.h
-rw-rw-r-- 1 vk vk   3067 ago 31 22:25 pan.m
-rw-rw-r-- 1 vk vk  56161 ago 31 22:25 pan.p
-rw-rw-r-- 1 vk vk  18573 ago 31 22:25 pan.t
$ gcc -o pan pan.c
$ ls -l pan
-rwxr-xr-x 1 vk vk 95072 ago 31 22:27 pan
```

```
$ ./pan

(Spin Version 6.4.8 -- 2 March 2018)
        + Partial Order Reduction

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

State-vector 28 byte, depth reached 2, errors: 0
        4 states, stored
        1 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))
    0.292   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

unreached in proctype P
        (0 of 10 states)

pan: elapsed time 0 seconds
```

Pan's Output Format (http://spinroot.com/spin/Man/Pan.html#C)

This is what each line in this listing means:

```
(Spin Version 6.4.8 -- 2 March 2018)
```

Identifies the version of Spin that generated the pan.c source from which this verifier was compiled.

```
        + Partial Order Reduction
```

The plus sign means that the default partial order reduction algorithm was used. A minus sign would indicate compilation for exhaustive, non-reduced, verification with option `-DNOREDUCE` .

```
Full statespace search for:
```

Indicates the type of search. The default is a full statespace search. Large models can also be verified with a Bitstate search, which is approximate.

```
        never claim             - (none specified)
```

The minus sign indicates that no never claim, or LTL formula was used for this run. If a never claim was part of the model, it could have been suppressed with the compiler directive `-DNOCLAIM` .

```
        assertion violations    +
```

The plus indicates that the search checked for violations of user specified assertions, which is the default.

Pan's Output Format (http://spinroot.com/spin/Man/Pan.html#C)

```
acceptance   cycles     - (none specified)
```

The minus indicates that the search did not check for the presence of acceptance or non-progress cycles. To do so would require a run-time option `-a` or compilation with `-DNP` combined with the run-time option `-l`.

```
invalid end states      +
```

The plus indicates that a check for invalid endstates was done (i.e., for absence of deadlocks).

```
State-vector 28 byte, depth reached 2, errors: 0
```

The complete description of a global system state required 28 bytes of memory (per state). The longest depth-first search path contained 2 transitions from the root of the tree (i.e., from the initial system state). No errors were found in this search.

```
4 states, stored
```

A total of 4 unique global system states were stored in the statespace (each represented effectively by a vector of 28 bytes).

```
1 states, matched
```

In 1 case did the search return to a previously visited state in the search tree.

Pan's Output Format (http://spinroot.com/spin/Man/Pan.html#C)

```
        5 transitions (= stored+matched)
```

A total of 5 transitions were explored in the search, which can serve as a statistic for the amount of work that has been performed to complete the verification.

```
        0 atomic steps
```

No one of the transitions was part of an atomic sequence, all were outside atomic sequences.

```
hash conflicts: 0 (resolved)
```

In 0 cases the default hashing scheme (a weaker version than what is used in bitstate hashing) encountered a collision, and had to place the states into a linked list in the hash-table.

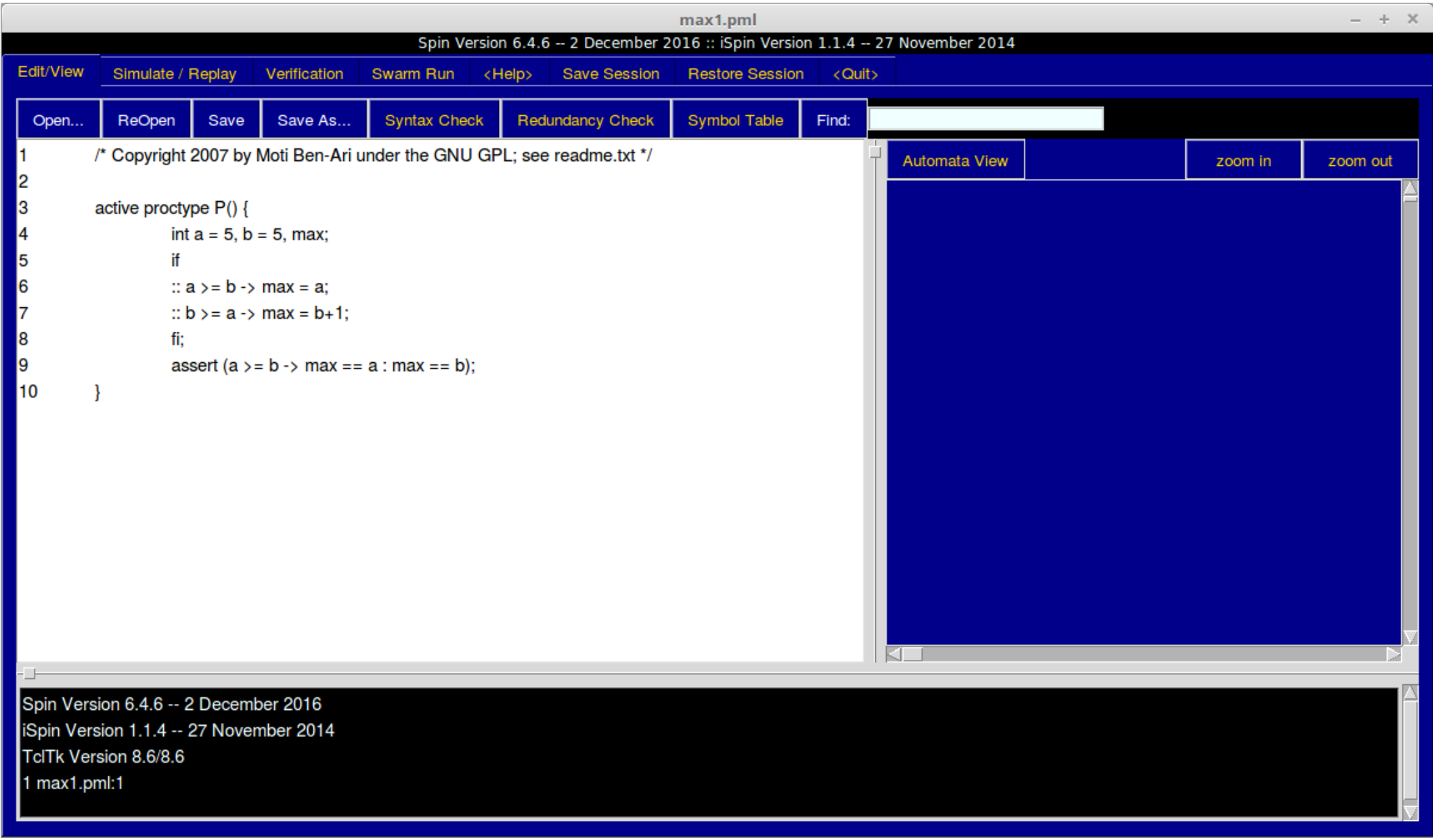
```
Stats on memory usage (in Megabytes):
    ...

unreached in proctype P
        (0 of 10 states)
```

A listing of the state numbers and approximate line numbers for the basic statements in the specification that were not reached. Since this is a full statespace search that ran to completion this means that these transitions are effectively unreachable (dead code).

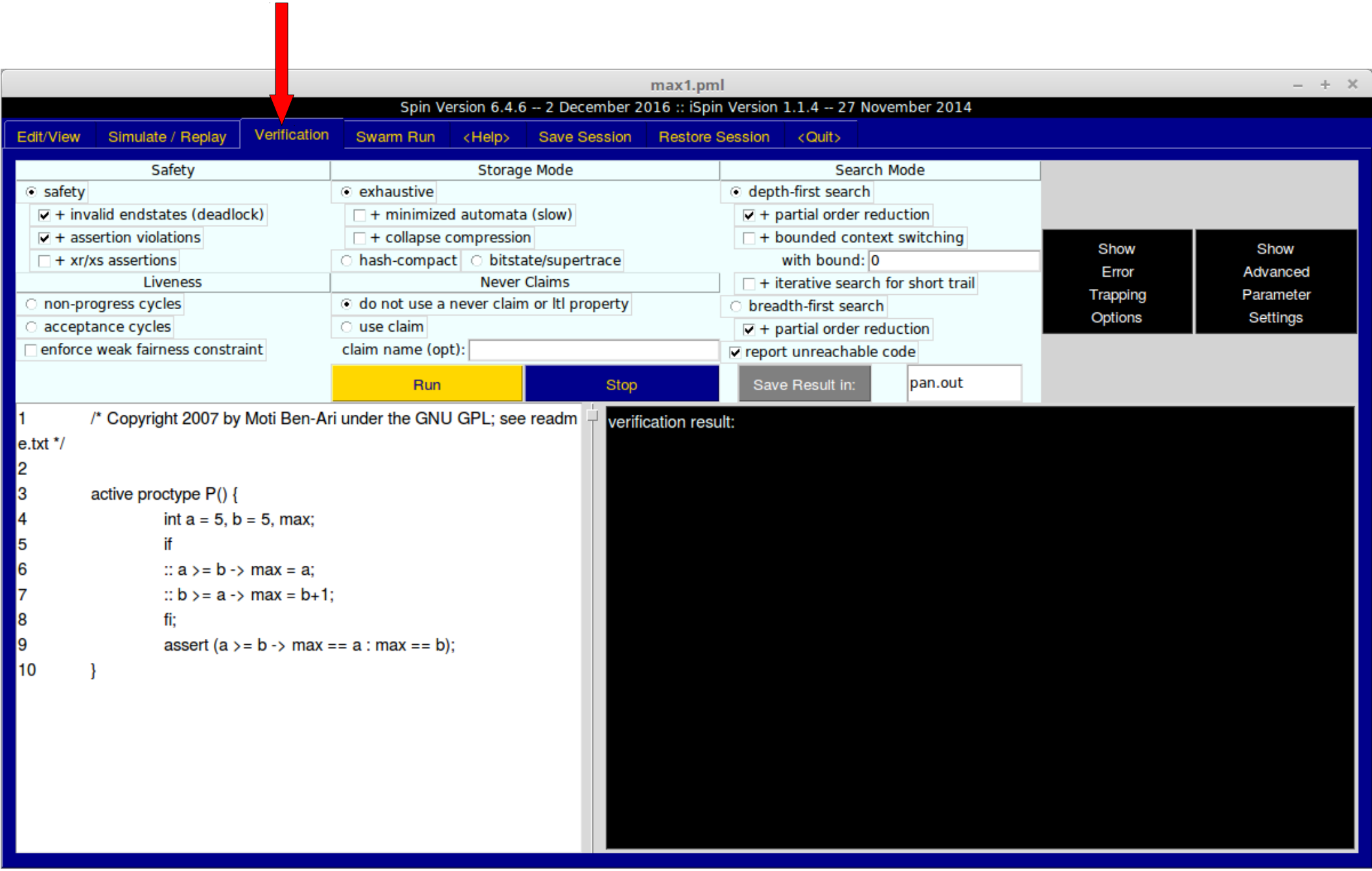
```
$ spin -a max1.pml
$ gcc -o pan pan.c
$ ./pan
pan:1: assertion violated ( ((a>=b)) ? ((max==a)) : ((max==b)) ) (at depth 0)
pan: wrote max1.pml.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
        + Partial Order Reduction
...
State-vector 24 byte, depth reached 2, errors: 1
        3 states, stored
        0 states, matched
        3 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))
    0.292   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 0.02 seconds
Pan: rate       150 states/second
```

```
$ ls -l max1.pml.trail
-rw-r--r-- 1 vk vk 15 ago 31 22:34 max1.pml.trail

$ spin -t max1.pml
spin: max1.pml:9, Error: assertion violated
spin: text of failed assertion: assert(( ((a>=b)) -> ((max==a)) : ((max==b)) ))
spin: trail ends after 1 steps
#processes: 1
  1:    proc  0 (P:1) max1.pml:10 (state 8) <valid end state>
1 process created

$ spin -t -l max1.pml
using statement merging
spin: max1.pml:9, Error: assertion violated
spin: text of failed assertion: assert(( ((a>=b)) -> ((max==a)) : ((max==b)) ))
spin: trail ends after 1 steps
#processes: 1
  1:    proc  0 (P) max1.pml:10 (state 8) <valid end state>
                P(0):max = 6
                P(0):b = 5
                P(0):a = 5
1 process created
```

```
$ ispin max1.pml &
```



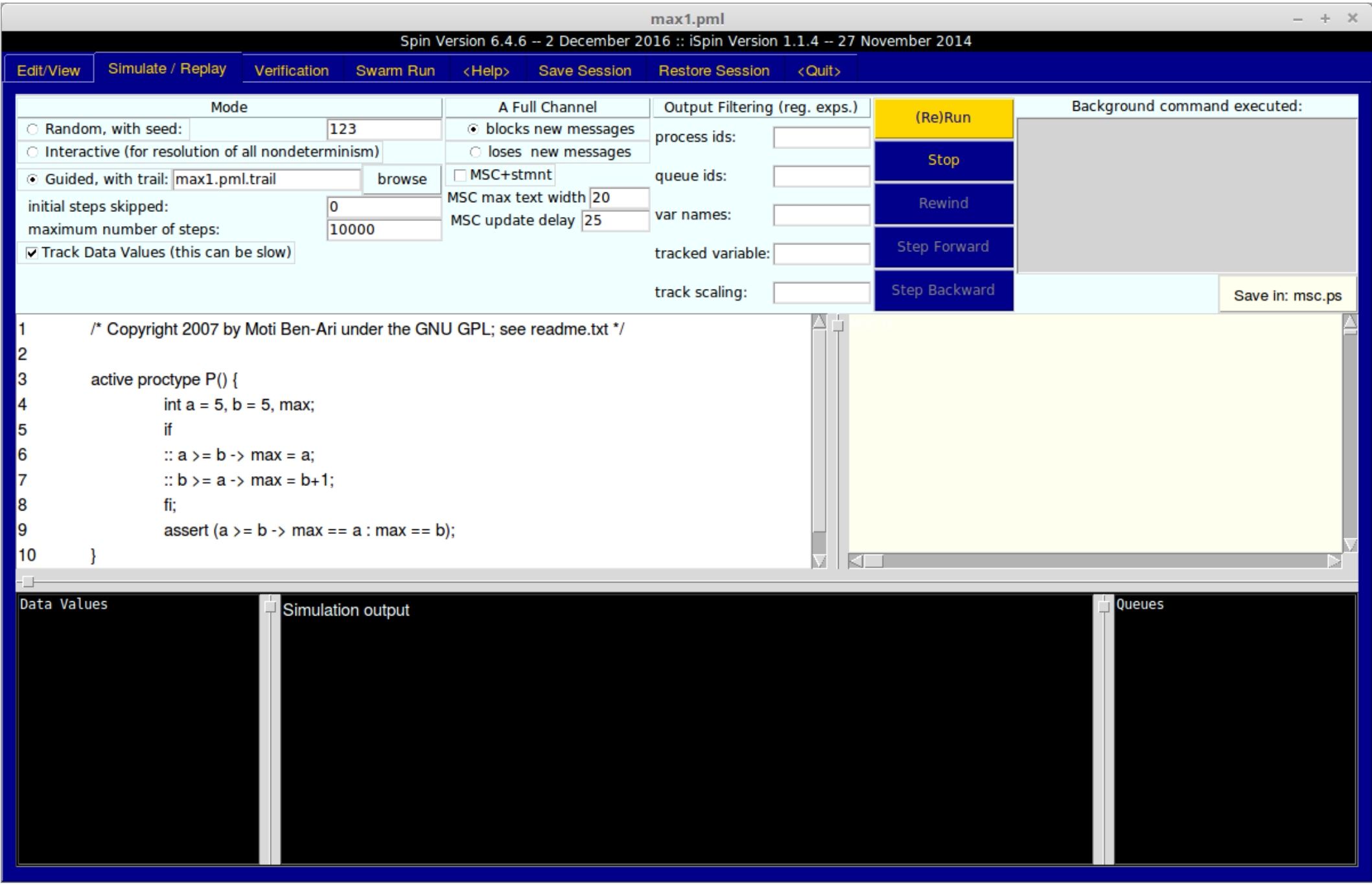
Verification tab



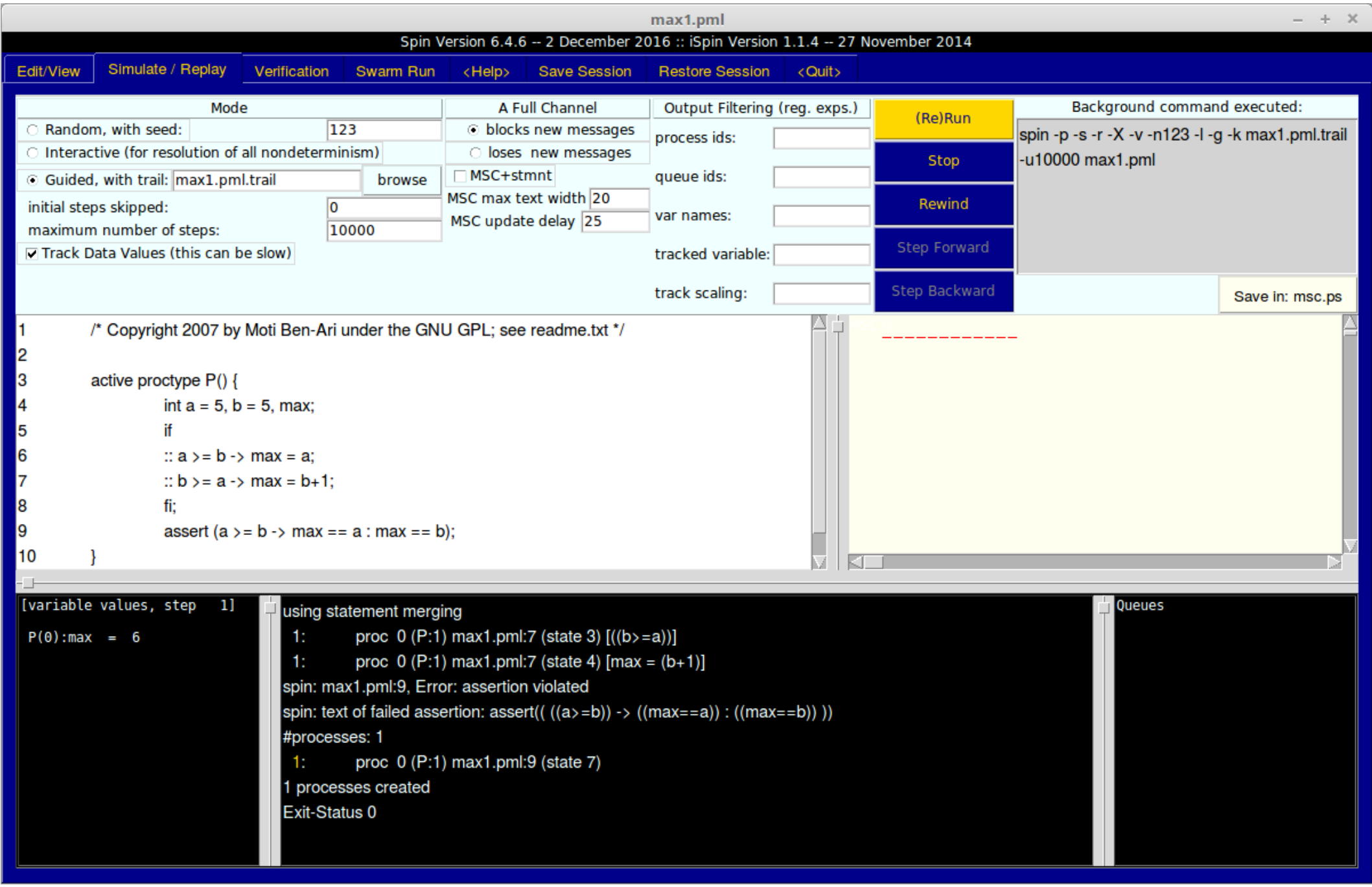
Verification, Run click

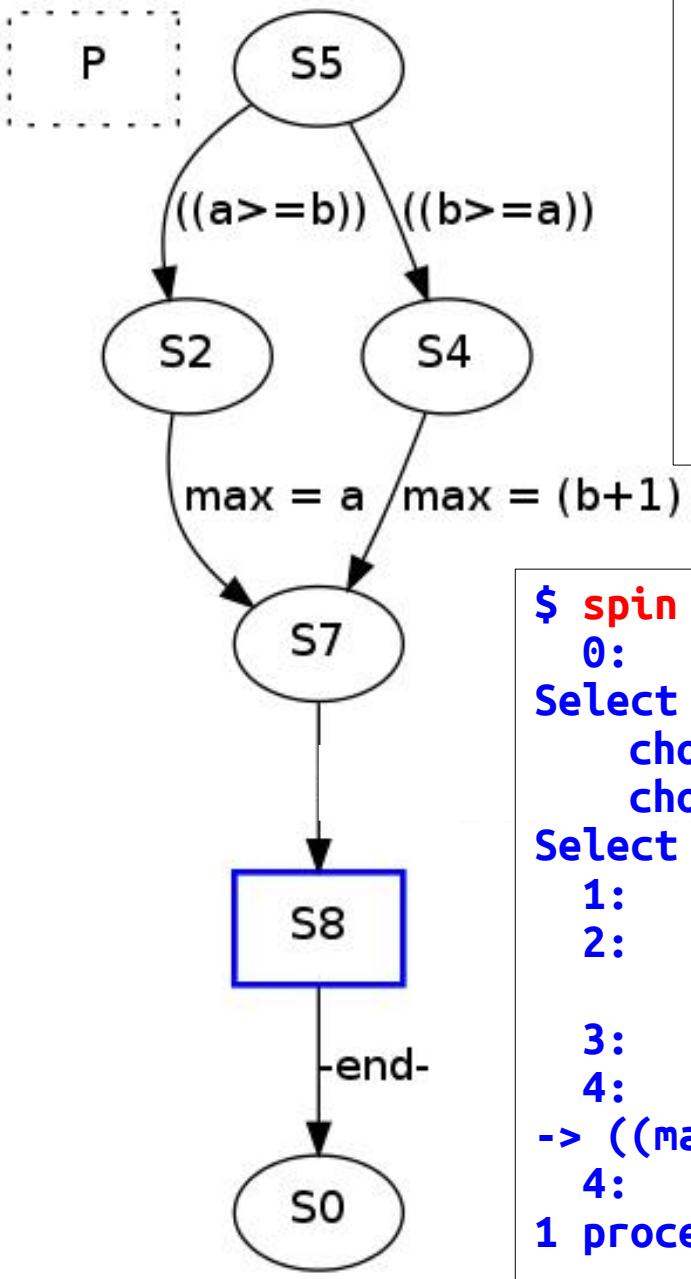
```
verification result:
spin -a  max1.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 1420
pan:1: assertion violated ( ((a>=b)) ? ((max==a)) : ((max==b)) ) (at depth 0)
pan: wrote max1.pml.trail

(Spin Version 6.4.6 -- 2 December 2016)
Warning: Search not completed
            + Partial Order Reduction

Full statespace search for:
            never claim         - (not selected)
            assertion violations +
            cycle checks        - (disabled by -DSAFETY)
            invalid end states +

State-vector 24 byte, depth reached 2, errors: 1
       3 states, stored
       0 states, matched
       3 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))
    0.292   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
To replay the error-trail, goto Simulate/Replay and select "Run"
```

Simulate / Replay tab, Mode: Guided



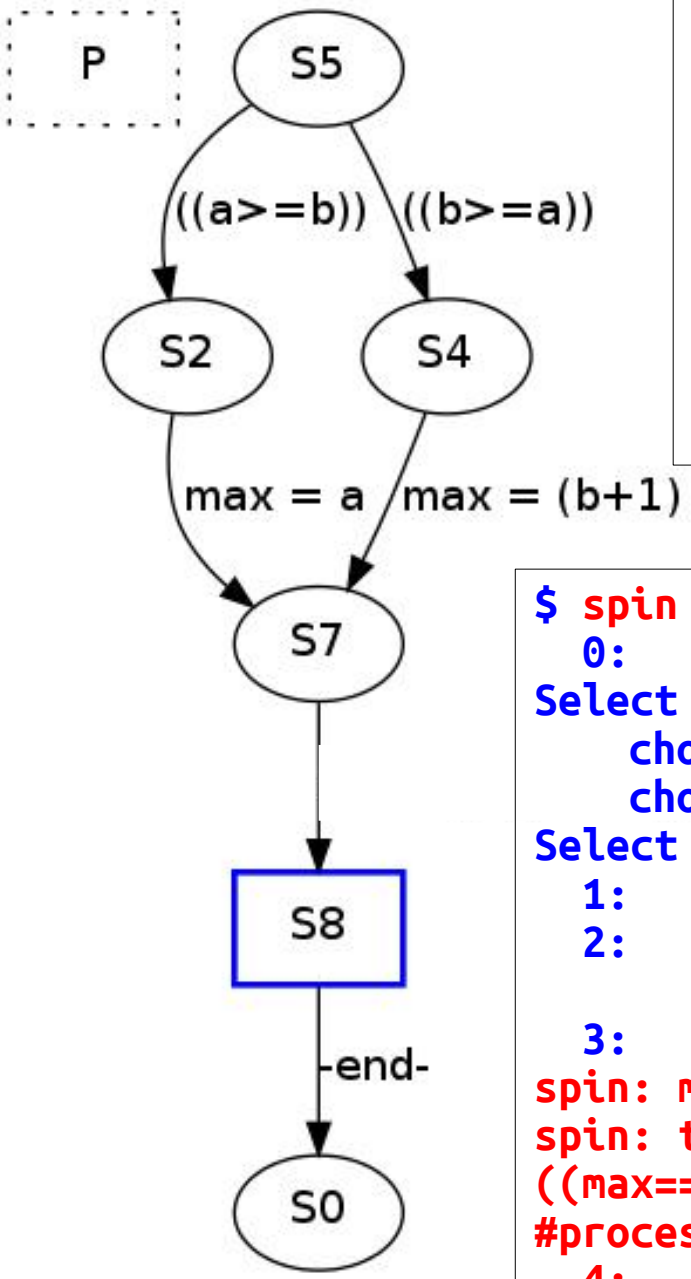
Simulate / Replay tab, (Re)Run click






```
 1  /* Copyright 2007 by Moti Ben-Ari under the ... */
 2
 3  active proctype P() {
 4      int a = 5, b = 5, max;
 5      if
 6      :: a >= b -> max = a;
 7      :: b >= a -> max = b+1;
 8      fi;
 9      assert (a >= b -> max == a : max == b);
10  }
```

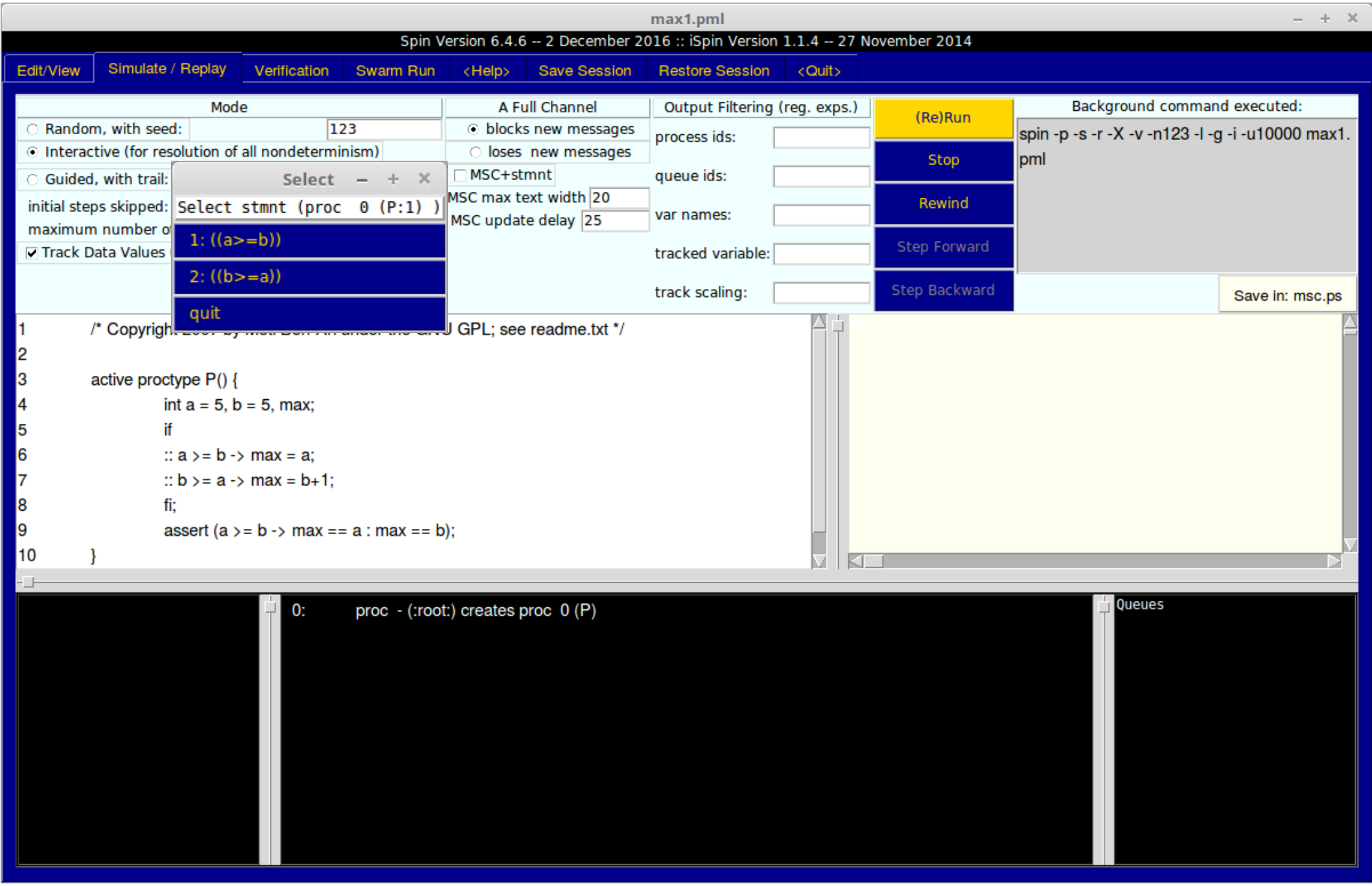
```
$ spin -p -l -g -i max1.pml
  0:    proc  - (:root:) creates proc  0 (P)
Select stmnt (proc  0 (P:1) )
    choice 1: ((a>=b))
    choice 2: ((b>=a))
Select [0-2]: 1
  1:    proc  0 (P:1) max1.pml:6 (state 1) [((a>=b))]
  2:    proc  0 (P:1) max1.pml:6 (state 2) [max = a]
                P(0):max = 5
  3:    proc  0 (P:1) max1.pml:9 (state 6) [.(goto)]
  4:    proc  0 (P:1) max1.pml:9 (state 7) [assert(( ((a>=b))
-> ((max==a)) : ((max==b)) ))]
  4:    proc  0 (P:1)       terminates
1 process created
```



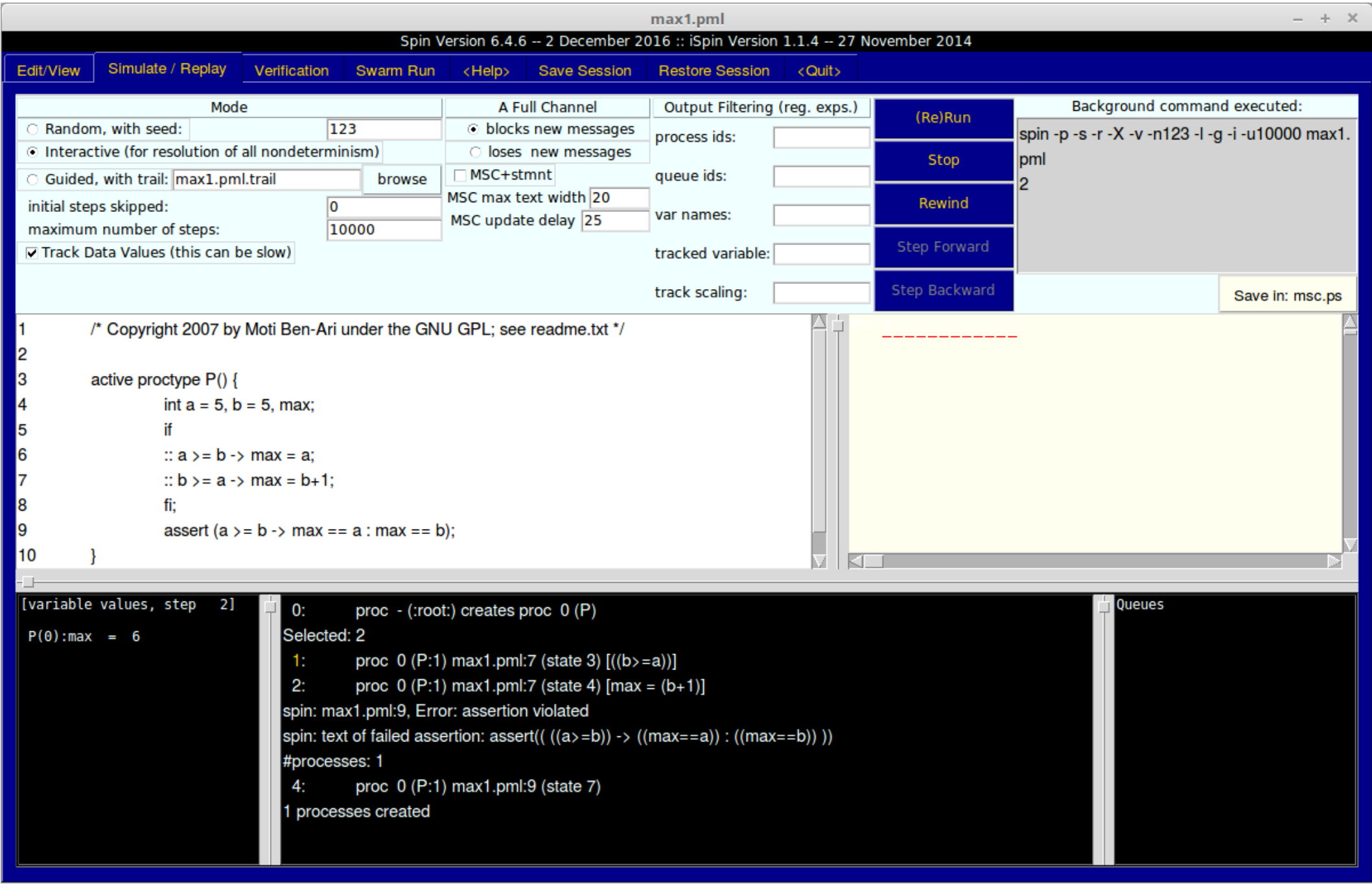

```
 1  /* Copyright 2007 by Moti Ben-Ari under the ... */
 2
 3  active proctype P() {
 4      int a = 5, b = 5, max;
 5      if
 6      :: a >= b -> max = a;
 7      :: b >= a -> max = b+1;
 8      fi;
 9      assert (a >= b -> max == a : max == b);
10  }
```

```
$ spin -p -l -g -i max1.pml
  0:    proc  - (:root:) creates proc  0 (P)
Select stmnt (proc  0 (P) )
    choice 1: ((a>=b))
    choice 2: ((b>=a))
Select [0-2]: 2
  1:    proc  0 (P:1) max1.pml:7 (state 3) [((b>=a))]
  2:    proc  0 (P:1) max1.pml:7 (state 4) [max = (b+1)]
                P(0):max = 6
  3:    proc  0 (P:1) max1.pml:9 (state 6) [.(goto)]
spin: max1.pml:9, Error: assertion violated
spin: text of failed assertion: assert(( ((a>=b)) ->
((max==a)) : ((max==b)) ))
#processes: 1
  4:    proc  0 (P:1) max1.pml:9 (state 7)
1 process created
```

Simulate / Replay tab, Interactive, (Re)Run click



Simulate / Replay tab, Interactive, (Re)Run click, Selected: 2



Maximum with two errors (max2.pml)

```
 3  active proctype P() {
 4    int a = 5, b = 5, max
 5    if
 6    :: a >= b -> max = a+2
 7    :: b >= a -> max = b+1
 8    fi
 9    assert (a >= b -> max == a : max == b)
10  }
```

```
$ spin -a max2.pml

$ gcc -o pan pan.c

$ ./pan -c3 -e
pan:1: assertion violated ( ((a>=b)) ? ((max==a)) : ((max==b)) ) (at depth 0)
pan: wrote max2.pml1.trail
pan: wrote max2.pml2.trail

(Spin Version 6.4.8 -- 2 March 2018)
        + Partial Order Reduction
...
State-vector 24 byte, depth reached 2, errors: 2
        4 states, stored
        1 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))
    0.292   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
```

-cN causes the verifier to stop at the *N*th error rather than the first, while the argument **-c0** requests the verifier to ignore all errors and not to generate a trail file. The argument **-e** to pan causes trails for all errors to be created.

-t[N] follow [Nth] simulation trail

```
$ spin -p -l -t1 max2.pml
using statement merging
  1:    proc  0 (P:1) max2.pml:6 (state 1)      [((a>=b))]
  1:    proc  0 (P:1) max2.pml:6 (state 2)      [max = (a+2)]
                P(0):max = 7
spin: max2.pml:9, Error: assertion violated
spin: text of failed assertion: assert(( ((a>=b)) -> ((max==a)) : ((max==b)) ))
  1:    proc  0 (P) max2.pml:9 (state 7)    [assert(( ((a>=b)) -> ((max==a)) : ((max==b)) ))]
                P(0):max = 7
spin: trail ends after 1 steps
#processes: 1
  1:    proc  0 (P:1) max2.pml:10 (state 8) <valid end state>
                P(0):max = 7
1 process created
```

```
$ spin -p -l -t2 max2.pml
using statement merging
  1:    proc  0 (P:1) max2.pml:7 (state 3)      [((b>=a))]
  1:    proc  0 (P:1) max2.pml:7 (state 4)      [max = (b+1)]
                P(0):max = 6
spin: max2.pml:9, Error: assertion violated
spin: text of failed assertion: assert(( ((a>=b)) -> ((max==a)) : ((max==b)) ))
  1:    proc  0 (P) max2.pml:9 (state 7)    [assert(( ((a>=b)) -> ((max==a)) : ((max==b)) ))]
                P(0):max = 6
spin: trail ends after 1 steps
#processes: 1
  1:    proc  0 (P:1) max2.pml:10 (state 8) <valid end state>
                P(0):max = 6
1 process created
```

```
$ spin -H      # or spin --
use: spin [-option] ... [-option] file
        Note: file must always be the last argument
        -A apply slicing algorithm
        -a generate a verifier in pan.c
        -B no final state details in simulations
        -b don't execute printfs in simulation
        -C print channel access info (combine with -g etc.)
        -c columnated -s -r simulation output
        -d produce symbol-table information
        -Dyyy pass -Dyyy to the preprocessor
        -Eyyy pass yyy to the preprocessor
        -e compute synchronous product of multiple never claims (modified by -L)
        -f "..formula.."  translate LTL into never claim
        -F file  like -f, but with the LTL formula stored in a 1-line file
        -g print all global variables
        -h at end of run, print value of seed for random nr generator used
        -i interactive (random simulation)
        -I show result of inlining and preprocessing
        -J reverse eval order of nested unlesses
        -jN skip the first N steps in simulation trail
        -k fname use the trailfile stored in file fname, see also -t
        -L when using -e, use strict language intersection
        -l print all local variables
        -M print msc-flow in tcl/tk format
        -m lose msgs sent to full queues
        -N fname use never claim stored in file fname
...
```

```
...
        -nN seed for random nr generator
        -O use old scope rules (pre 5.3.0)
        -o1 turn off dataflow-optimizations in verifier
        -o2 don't hide write-only variables in verifier
        -o3 turn off statement merging in verifier
        -o4 turn on rendezvous optiomizations in verifier
        -o5 turn on case caching (reduces size of pan.m, but affects reachability
reports)
        -o6 revert to the old rules for interpreting priority tags (pre version 6.2)
        -o7 revert to the old rules for semi-colon usage (pre version 6.3)
        -Pxxx use xxx for preprocessing
        -p print all statements
        -pp pretty-print (reformat) stdin, write stdout
        -qN suppress io for queue N in printouts
        -r print receive events
        -replay  replay an error trail-file found earlier
                if the model contains embedded c-code, the ./pan executable is used
                otherwise spin itself is used to replay the trailfile
                note that pan recognizes different runtime options than spin itself
        -run  (or -search) generate a verifier, and compile and run it
              options before -search are interpreted by spin to parse the input
              options following a -search are used to compile and run the verifier pan
                    valid options that can follow a -search argument include:
                    -bfs        perform a breadth-first search
                    -bfspar     perform a parallel breadth-first search
                    -dfspar     perform a parallel depth-first search, same as -
DNCORE=4
                    -bcs        use the bounded-context-switching algorithm
...
```

```
...
                -bitstate   or -bit, use bitstate storage
                -biterateN,M use bitstate with iterative search refinement (-
w18..-w35)
                             perform N randomized runs and increment -w every M
runs
                             default value for N is 10, default for M is 1
                             (use N,N to keep -w fixed for all runs)
                             (add -w to see which command will be executed)
                             (add -w if ./pan exists and need not be recompiled)
                -swarmN,M like -biterate, but running all iterations in parallel
                -link file.c  link executable pan to file.c
                -collapse   use collapse state compression
                -noreduce   do not use partial order reduction
                -hc         use hash-compact storage
                -noclaim    ignore all ltl and never claims
                -p_permute  use process scheduling order permutation
                -p_rotateN  use process scheduling order rotation by N
                -p_reverse  use process scheduling order reversal
                -rhash      randomly pick one of the -p_... options
                -ltl p      verify the ltl property named p
                -safety     compile for safety properties only
                -i          use the dfs iterative shortening algorithm
                -a          search for acceptance cycles
                -l          search for non-progress cycles
            similarly, a -D... parameter can be specified to modify the
compilation
            and any valid runtime pan argument can be specified for the
verification
...
```

```
...
        -S1 and -S2 separate pan source for claim and model
        -s print send events
        -T do not indent printf output
        -t[N] follow [Nth] simulation trail, see also -k
        -Uyyy pass -Uyyy to the preprocessor
        -uN stop a simulation run after N steps
        -v verbose, more warnings
        -w very verbose (when combined with -l or -g)
        -[XYZ] reserved for use by xspin interface
        -V print version number and exit
```

Programming Pearls by Jon Louis Bentley

Column 4: Writing Correct Programs, page 42

6. [C. Scholten] David Gries calls this the "Coffee Can Problem" in his *Science of Programming*. You are initially given a coffee can that contains some black beans and some white beans and a large pile of "extra" black beans. You then repeat the following process until there is a single bean left in the can.

> Randomly select two beans from the can. If they are the same color, throw them both out and insert an extra black bean. If they are different colors, return the white bean to the can and throw out the black.

Prove that the process terminates. What can you say about the color of the final remaining bean as a function of the numbers of black and white beans originally in the can?

The Scholten/Dijkstra Pebble Game ... by Wolfgang Reisig

... Edsger W. Dijkstra talked about *Reasoning about programs*. As an example, Dijkstra presents a "Pebble Game" as an example of a nondeterministic algorithm. Gries refers the problem to Carl Scholten, due to a letter from Dijkstra in fall 1979. Scholten plays the game with black and white beans in a coffee can. Dijkstra models this algorithm as a guarded command program and proves its decisive properties.

Figure 1 represents the algorithm as a nondeterministic guarded command program. B and W are the number of white and black pebbles in the initial state.

$$
\begin{array}{l}
b := B;\ w := W;\\
\\
\underline{\text{do}}\ w \geq 1 \wedge b \geq 1 \rightarrow b := b - 1\\
\Box\ \ b \geq 2 \rightarrow\\
\qquad\qquad b := b - 1\\
\Box\ \ w \geq 2 \rightarrow\\
\qquad\qquad w := w - 2;\ b := b + 1\\
\underline{\text{od}}
\end{array}
$$

Fig. 1. Dijkstra's solution to the pebble game

The Scholten/Dijkstra Pebble Game ... by Wolfgang Reisig

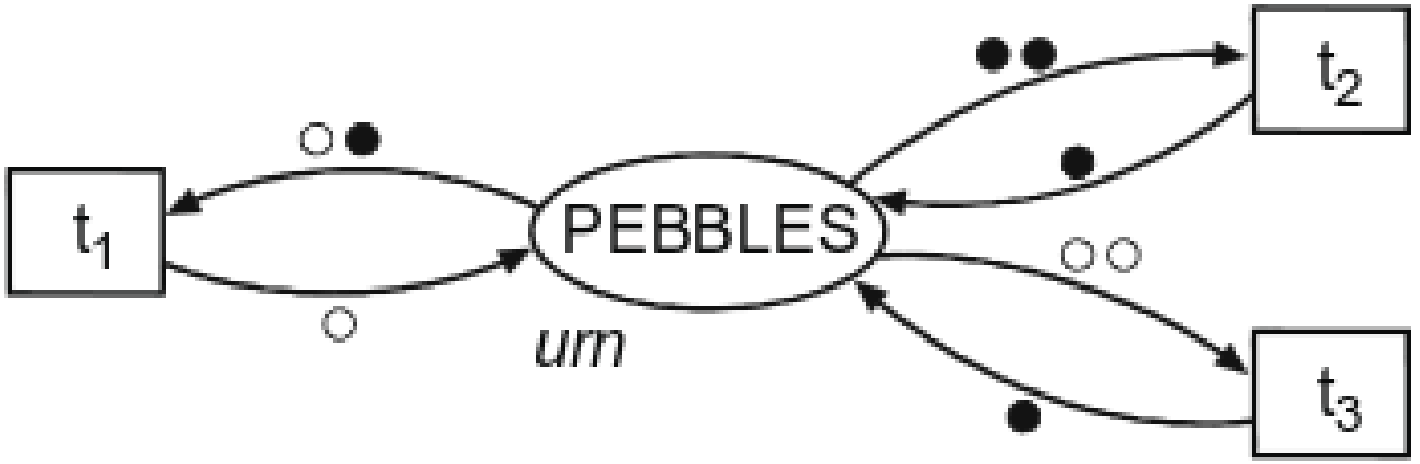


Fig. 2. The basic version of the algorithm

Construct this model in Promela.

Add the asserts (pre- and postconditions), verify the model.

Add the loop invariant, verify the model.

Bridge Crossing Problem #1

Three people begin on the same side of a bridge. You must help them across to the other side. It is night. There is one flashlight. A maximum of two people can cross at a time. Any party who crosses, either one or two people, must have the flashlight to see. The flashlight must be walked back and forth, it cannot be thrown, etc. Each person walks at a different speed. A pair must walk together at the rate of the slower person's pace, based on this information: Person 1 takes t1 = 1 minute to cross, and the other persons take t2 = 2 minutes, and t3 = 5 minutes to cross, respectively.

Construct this model in Promela.

Add the asserts (pre- and postconditions), verify the model.

How many walks are necessary?

What is the maximum (minimum) time of the bridge crossing?

Bridge Crossing Problem #2

There are 4 persons with t1 = 1 minute, t2 = 2 minutes, t3 = 5 minutes, and t4 = 10 minutes to cross, respectively.

Construct this model in Promela.

Add the asserts (pre- and postconditions), verify the model.

How many walks are necessary?

What is the maximum (minimum) time of the bridge crossing?