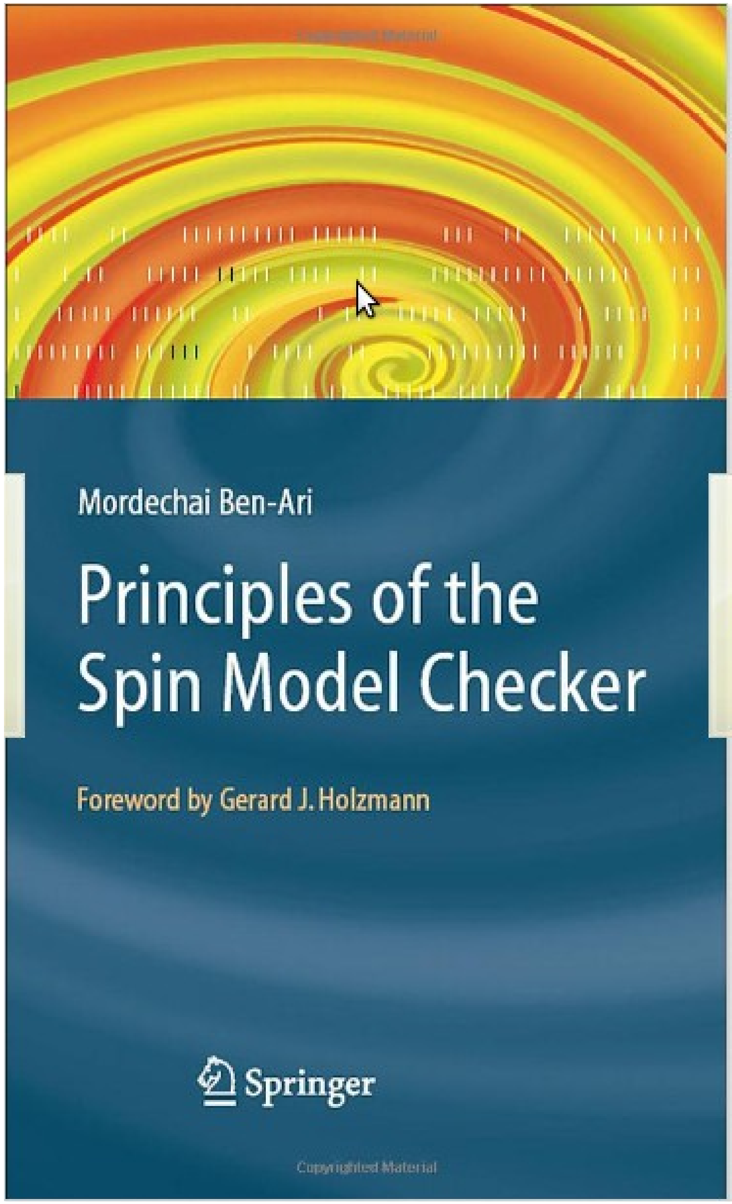


Mordechai Ben-Ari

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 1

Sequential Programing

in PROMELA

SPIN is a ***model checker*** – a software tool for verifying models of physical sistems, in particular, computerized systems.

First, a model is written that describes the behavior of the system;

then, correctness properties that express requirements on the system's behavior are specifies;

finally, the model checker is run to check if the correctness properties hold for the model, and if not, to provide a counterexample: a computation that does not satisfy a correctness property.

1.1 A first program in PROMELA

PROMELA (**Pro**cess or **Pro**tocol **Me**ta **La**nguage) is, in effect, **a simple programming language**.

Programs in PROMELA are composed of **a set of *processes***. Processes may have parameters.

The statements of the process are written between the braces **{** and **}**.

Comments are enclosed between **/*** and ***/**.

init

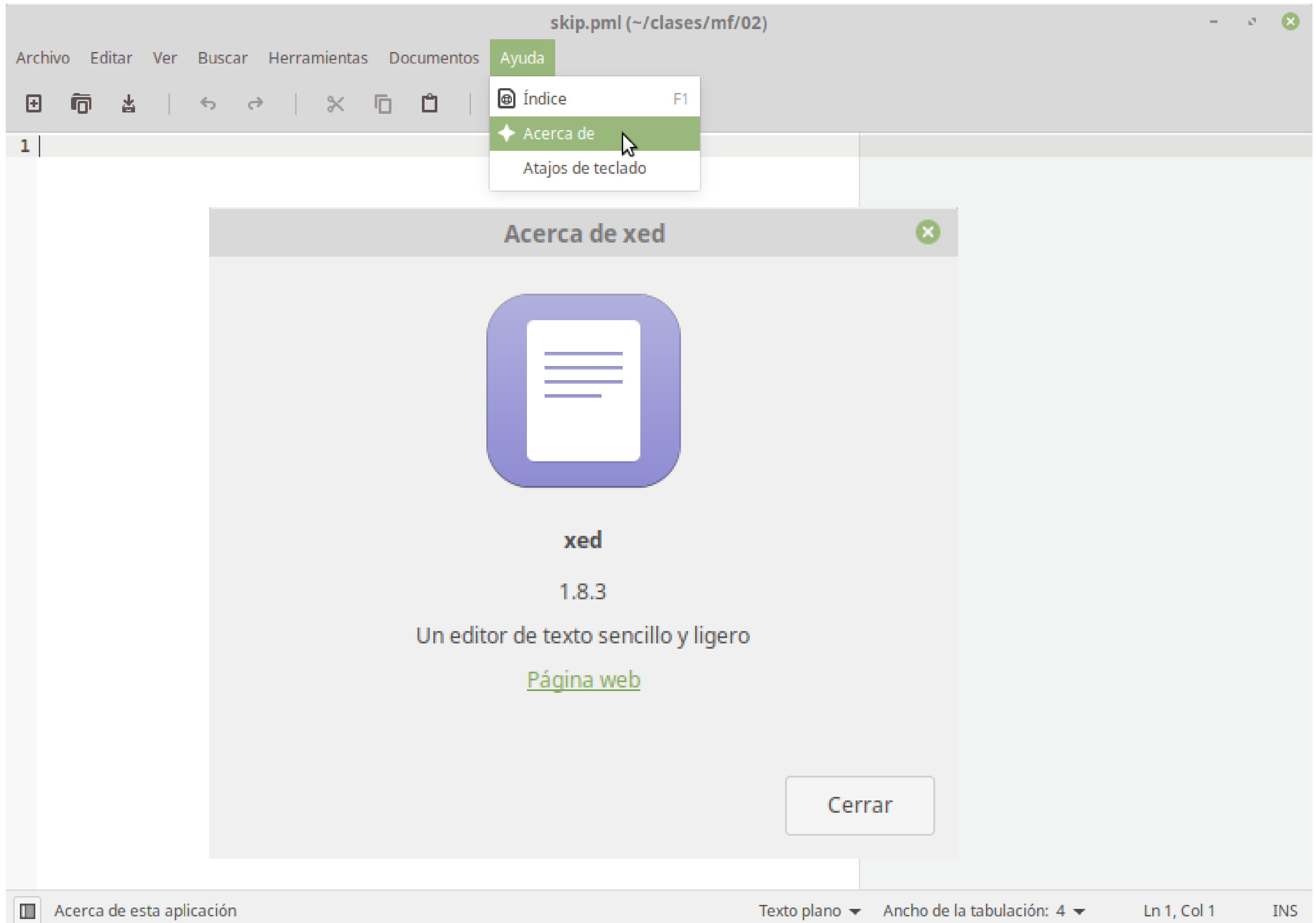
NAME
init - for declaring an initial process.

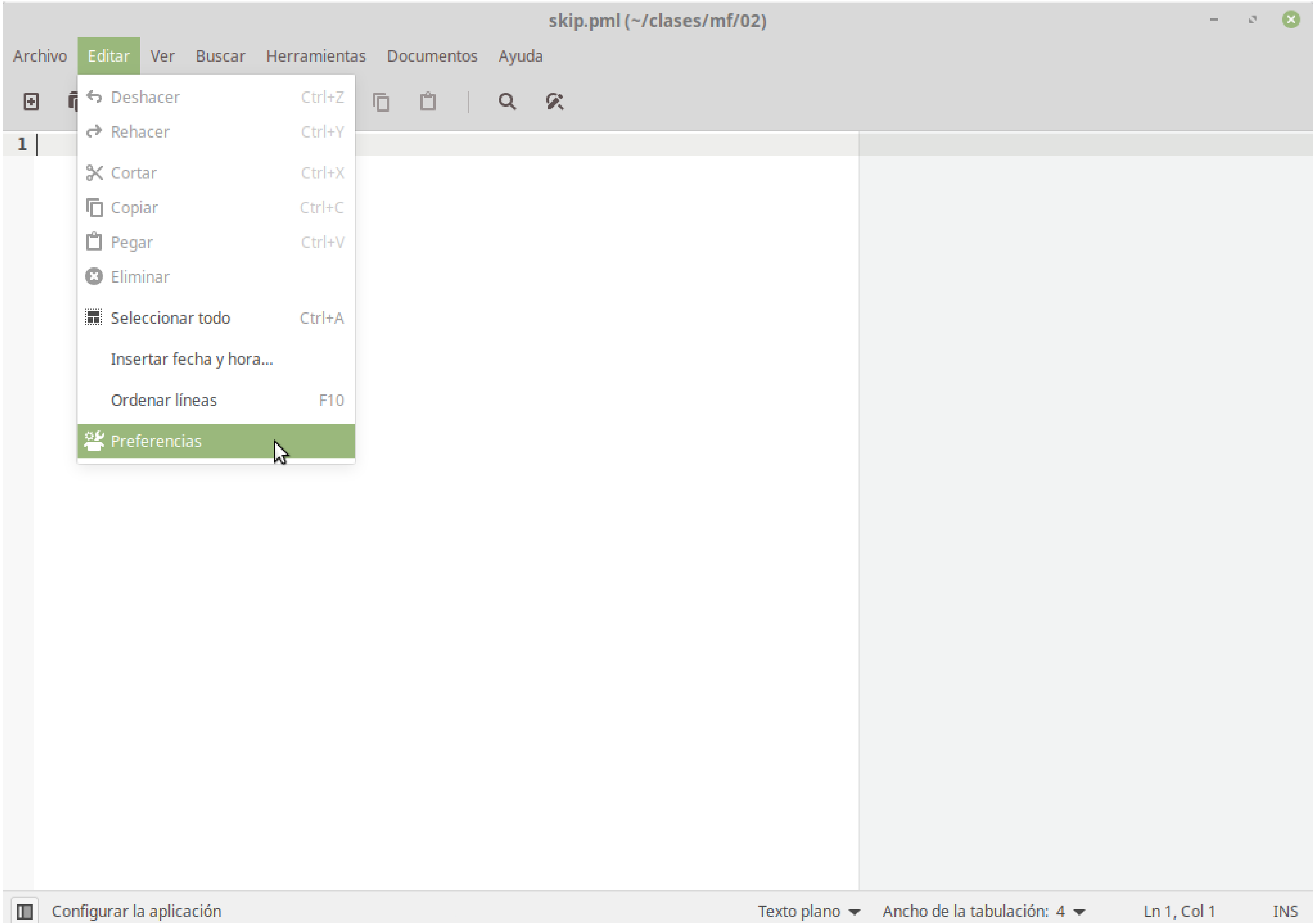
SYNTAX
init { *sequence* }

DESCRIPTION
The init keyword is used to declare the behavior of a process that is active in the initial system state.

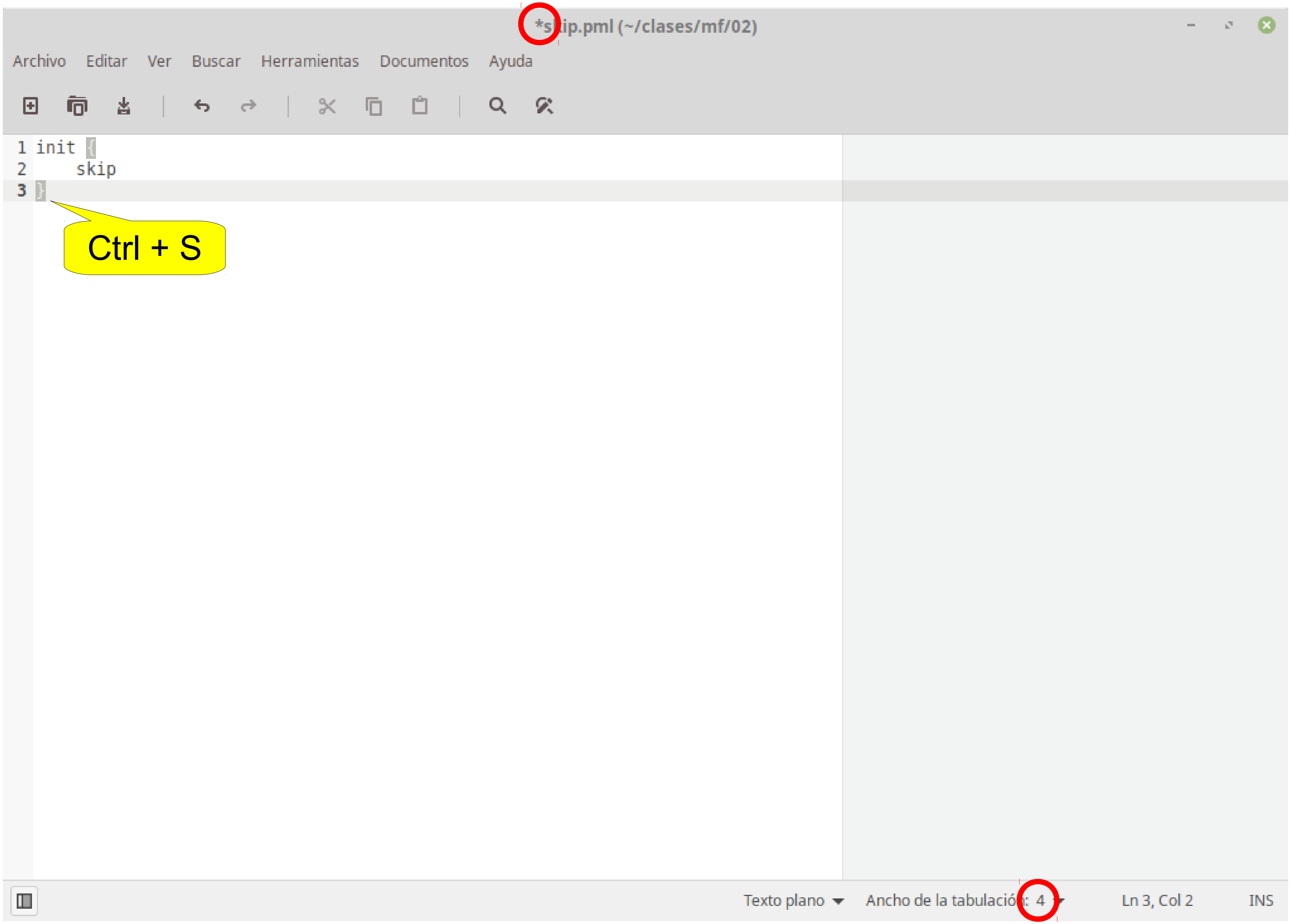
An init process has no parameters, and no additional copies of the process can be created (that is, the keyword cannot be used as an argument to the run operator).

```
$ pwd
/home/vk

$ mkdir -p clases/mf/02/ch01

$ cd clases/mf/02/

$ xed skip.pml &
```




skip.pml (~/clases/mf/02)
Archivo
Editar
Ver
Buscar
Herramientas
Documentos
Ayuda
Deshacer Ctrl+Z
Rehacer Ctrl+Y
Cortar Ctrl+X
Copiar Ctrl+C
Pegar Ctrl+V
Eliminar
Seleccionar todo Ctrl+A
Insertar fecha y hora...
Ordenar líneas F10
Preferencias
Configurar la aplicación
Texto plano
Ancho de la tabulación: 4
Ln 1, Col 1
INS


Opciones de xed
Editor
Guardar
Tema
Complementos
Tipografía
Usar la fuente de ancho fijo del sistema (Monospace 10)
Visualizar
Mostrar números de línea
Mostrar mapa general
Mostrar el margen derecho
Margen derecho en la columna
80
Resaltar
Resaltar la línea actual
Resaltar corchetes correspondientes
Indentación
Ancho de tabulador
4
Usar espacions en lugar de tabuladores
Indentación automática
Ajuste de línea
Ajuste de línea
Dividir palabras en dos líneas
Desplazamiento de pestañas
Permitir cambiar de pestaña con la rueda del ratón
Ayuda
Cerrar


*skip.pml (~/clases/mf/02)
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
init {
skip
}
Ctrl + S
Texto plano
Ancho de la tabulación: 4
Ln 3, Col 2
INS

Listing 0.0 Smallest possible model

```
$ cat -n skip.pml
     1   init {
     2       skip
     3   }

$ spin skip.pml
1 process created
```

```
$ pwd
/home/vk/clases/mf/02/
```

Poblar la carpeta ch01 con los ejemplos de Ben-Ari

```
$ ls -l ch01
total 44
-rw------- 1 vk vk 270 jun  3  2007 counting.pml
-rw------- 1 vk vk 192 sep  6  2005 for.h
-rw------- 1 vk vk 244 jun  3  2007 for.pml
-rw------- 1 vk vk 304 jun  3  2007 gcd.pml
-rw------- 1 vk vk 359 jun  3  2007 if1.pml
-rw------- 1 vk vk 510 jun  3  2007 if2-conditional.pml
-rw------- 1 vk vk 535 jun  3  2007 if2.pml
-rw------- 1 vk vk 298 jun  3  2007 max.pml
-rw------- 1 vk vk 444 jun  3  2007 mtype1.pml
-rw------- 1 vk vk 321 jun  3  2007 mtype.pml
-rw------- 1 vk vk 277 jun  3  2007 rev.pml
```

Listing 0.1 Hello program

```
init {
  printf("hello world\n")
}
```

NAME
proctype - for declaring new process behavior.

SYNTAX
proctype _name_ ([_decl_list_]) { _sequence_ }

DESCRIPTION
All process behavior must be declared before it can be instantiated. The proctype construct is used for the ***declaration***.

Instantiation can be done either with the run operator, or with the prefix active that can be used at the time of declaration.

NAME
active - prefix for proctype declarations to instantiate an initial set of processes.

SYNTAX
active proctype *name* ([*decl_list*]) { *sequence* }
active '['*const*']' **proctype** *name* ([*decl_list*]) { *sequence* }

DESCRIPTION
The keyword active can be prefixed to any proctype declaration to define a set of processes that are required to be active (i.e., running) in the initial system state. At least one active process must always exist in the initial system state. Such a process can also be declared with the help of the keyword init .

Multiple instantiations of the same proctype can be specified with an optional array suffix of the active prefix. The instantiation of a proctype requires the allocation of a process state and the instantiation of all associated local variables. At the time of instantiation, a unique process instantiation number is assigned.

...

Processes that are instantiated through an active prefix cannot be passed arguments. It is, nonetheless, legal to declare a list of formal parameters for such processes to allow for argument passing in additional instantiations with a run operator. In this case, copies of the processes instantiated through the active prefix have all formal parameters initialized to zero. Each active process is guaranteed to have a unique _pid within the system.

...

In many Promela models, the init process is used exclusively to initialize other processes with the run operator. By using active prefixes instead, the init process becomes superfluous and can be omitted, which reduces the amount of memory needed to store global states.

```
$ cat -n active.pml
     1 active proctype A(int a) {
     2     printf("A: %d\n", a)
     3 }
     4
     5 active [4] proctype B() {
     6     printf("B: %d\n", _pid)
     7     run A(_pid)
     8 }
```

do not indent printf output

```
$ spin active.pml
         B: 3
   A: 0
       B: 2
           B: 4
     B: 1
                 A: 2
             A: 3
               A: 4
                   A: 1
9 processes created
```

```
$ spin -T active.pml
B: 2
B: 1
B: 3
A: 0
A: 2
A: 3
A: 1
B: 4
A: 4
9 processes created
```

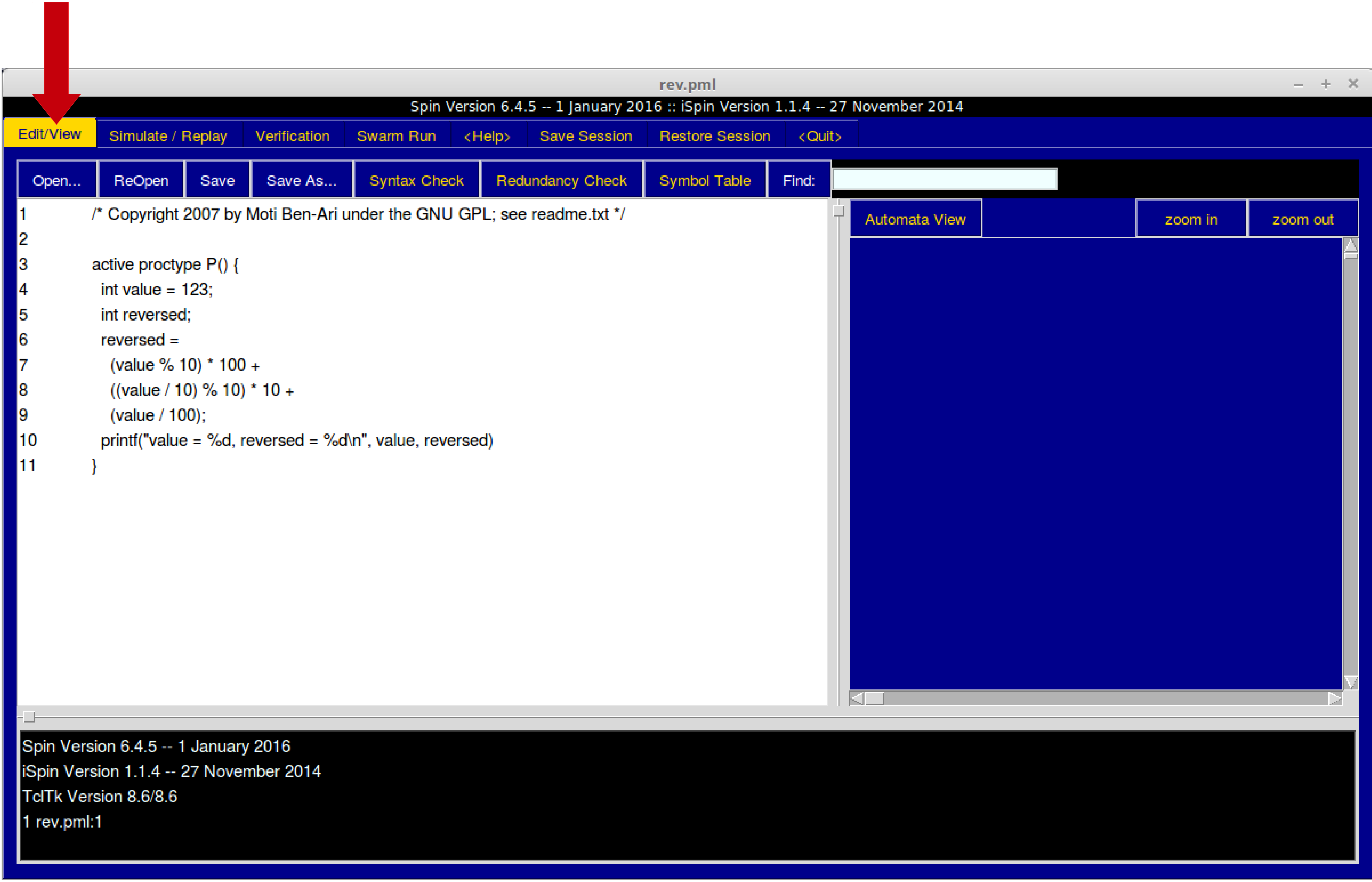
```
$ cat -n active_error.pml
     1	active proctype A(int a=7) {
     2		printf("A: %d\n", a)
     3	}
     4	
     5	active [4] proctype B() {
     6		printf("B: %d\n", _pid)
     7		run A(_pid)
     8	}

$ spin active_error.pml
spin: active_error.pml:1, Error: initializer in parameter list
```

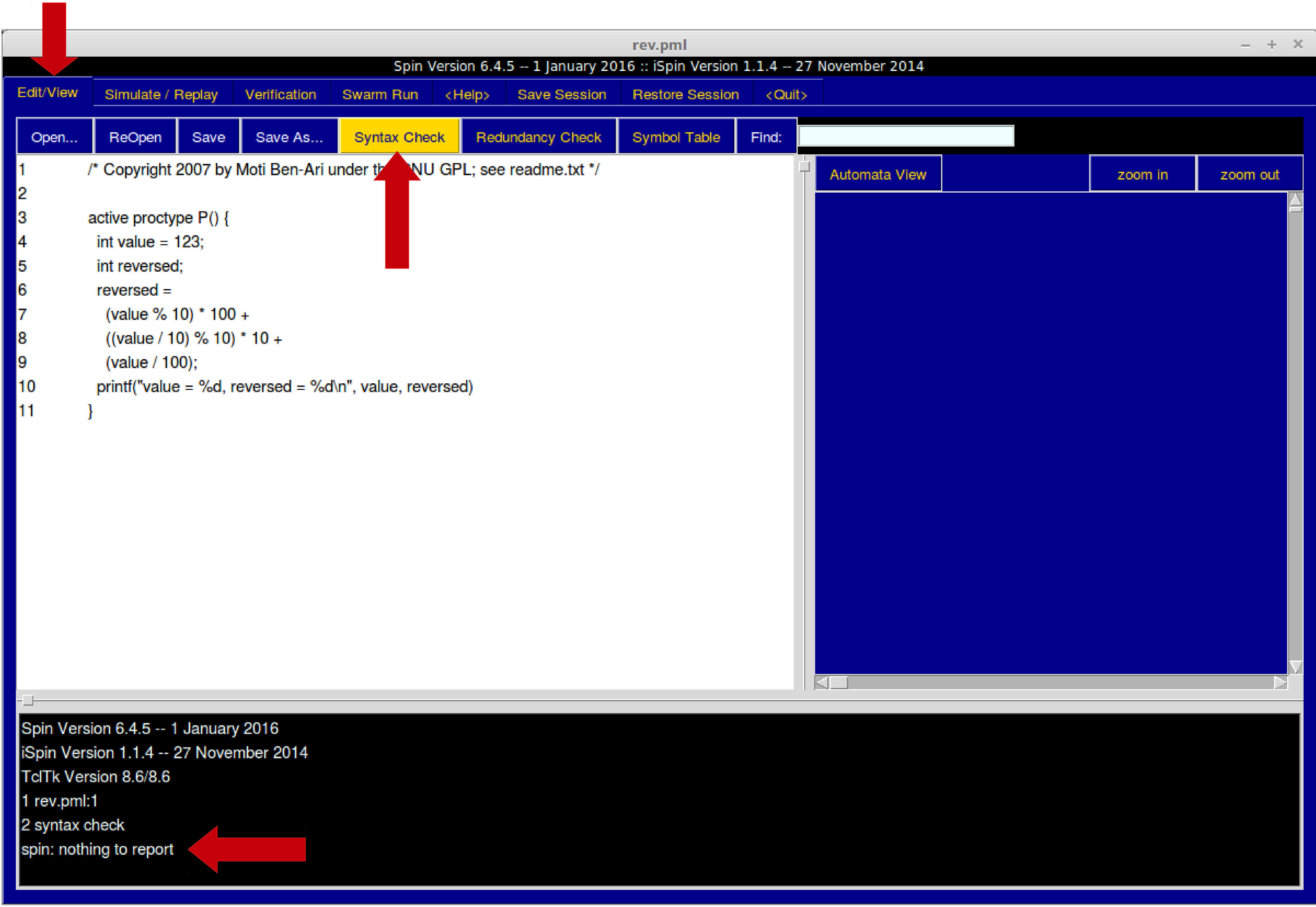
Listing 1.1 Reversing digits

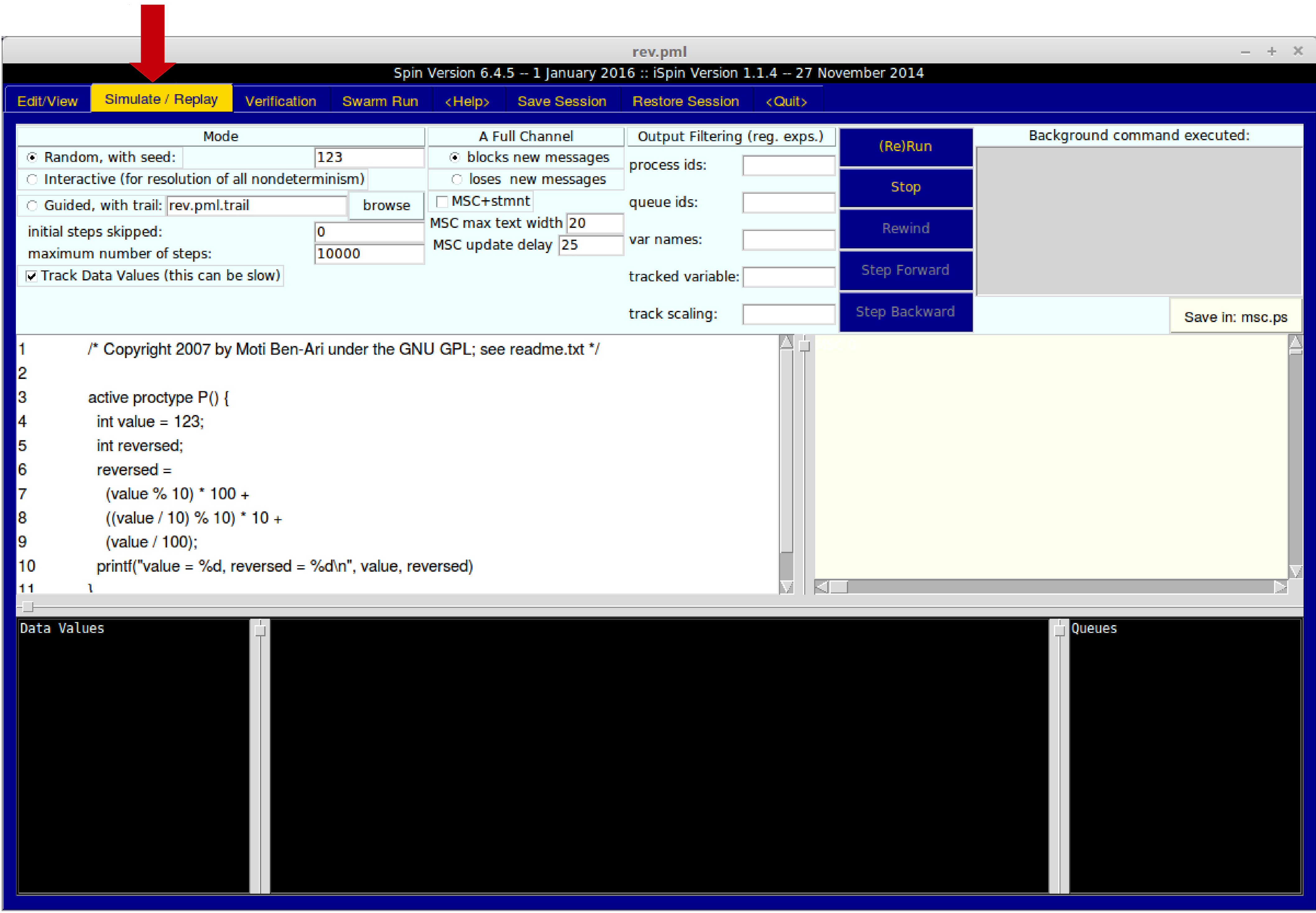
```
active proctype P() {
  int value = 123; /* int or byte? */
  int reversed;
  reversed =
    (value % 10) * 100 +
    ((value / 10) % 10) * 10 +
    (value / 100);
  printf("value = %d, reversed = %d\n", value, reversed)
}
```

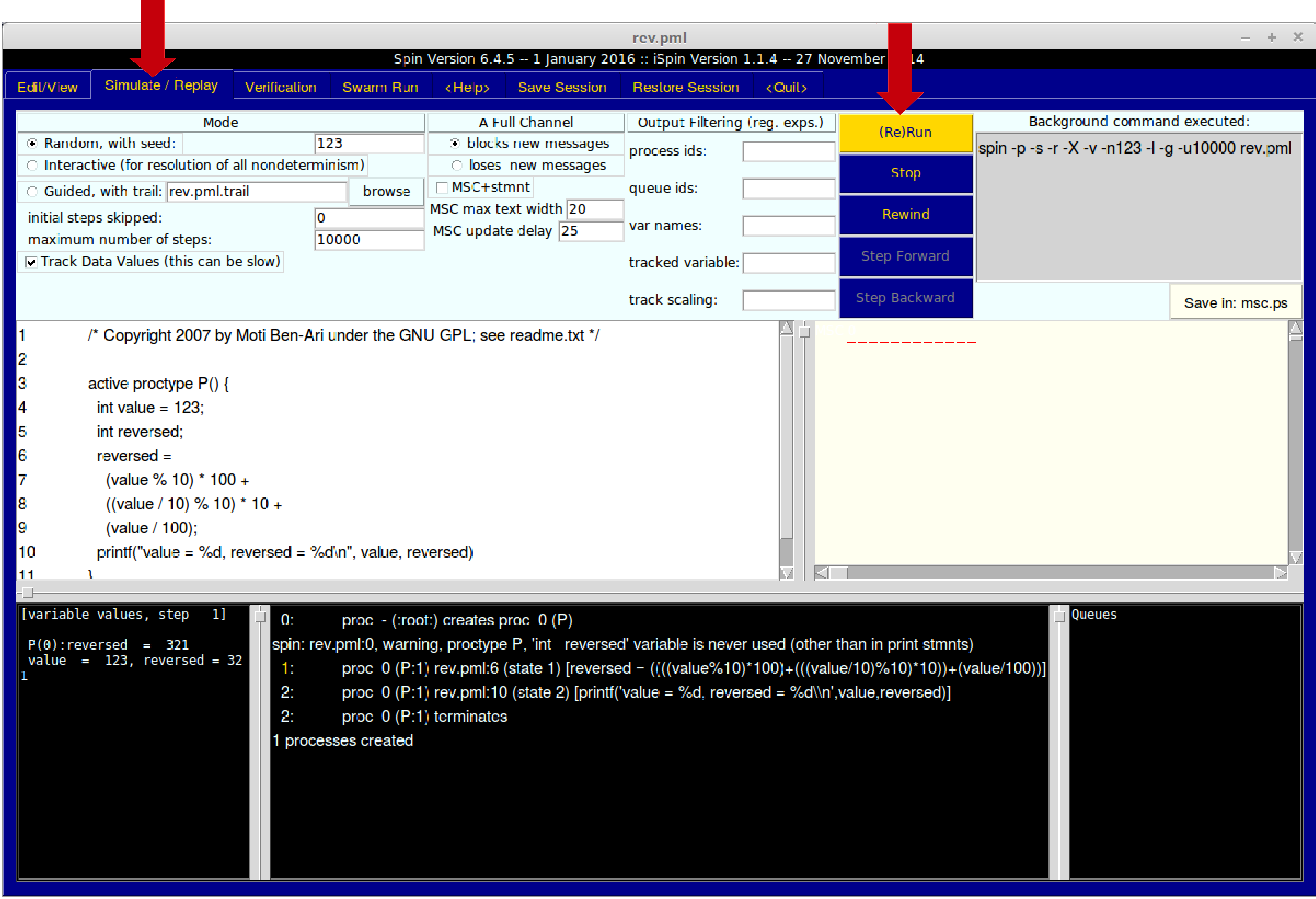
```
$ cat -n rev.pml
     1   /* Copyright 2007 by Moti Ben-Ari under the GNU GPL;
see readme.txt */
     2
     3   active proctype P() {
     4     int value = 123;
     5     int reversed;
     6     reversed =
     7       (value % 10) * 100 +
     8       ((value / 10) % 10) * 10 +
     9       (value / 100);
    10     printf("value = %d, reversed = %d\n", value,
reversed)
    11   }
```

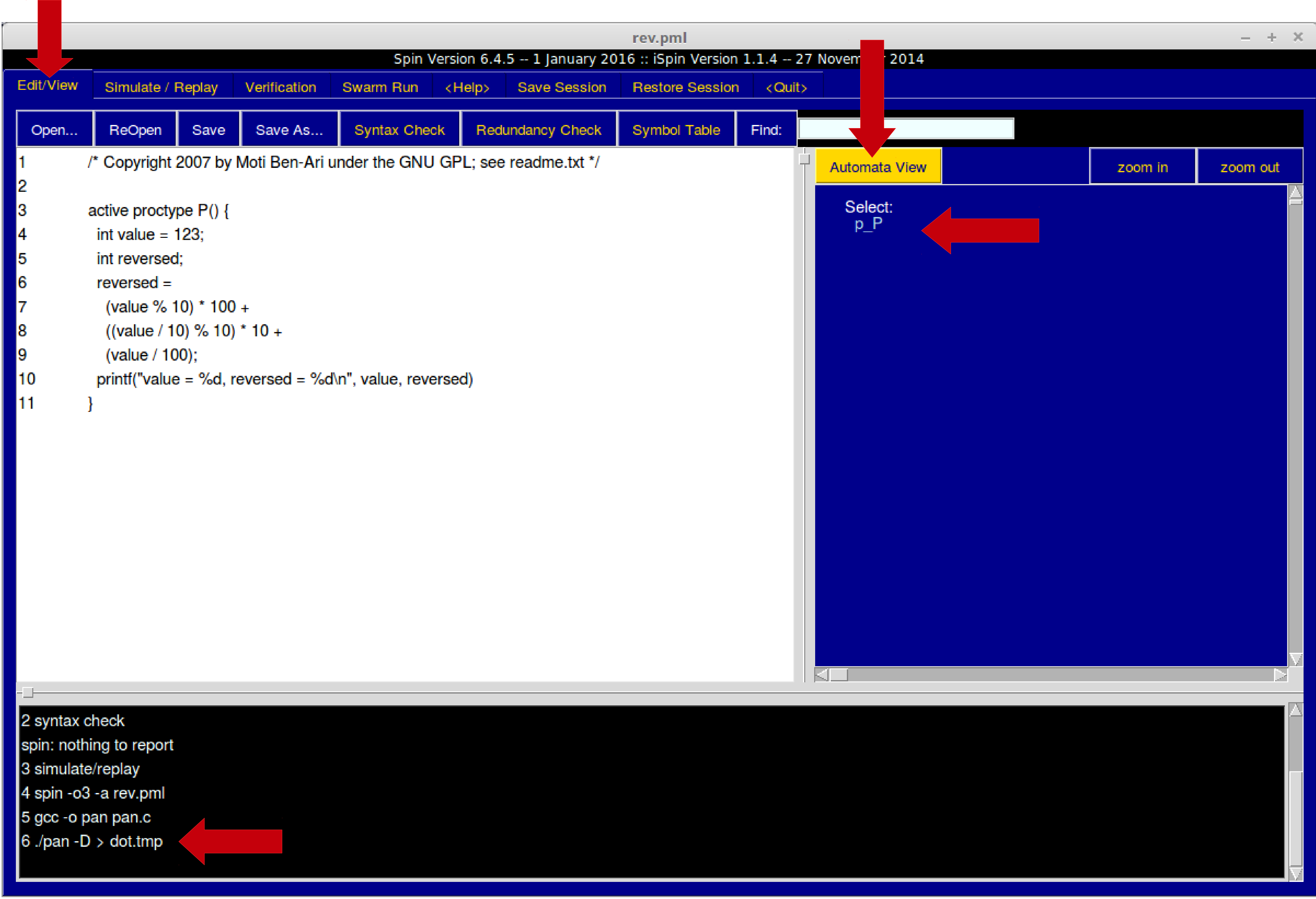
```
$ spin rev.pml
      value = 123, reversed = 321
1 process created

$ spin rev.pml > rev.out

$ cat rev.out
      value = 123, reversed = 321
1 process created

$ ls -l rev.{pml,out}
-rw-rw-r-- 1 vk vk  52 ago 25 01:16 rev.out
-rw------- 1 vk vk 277 jun  3  2007 rev.pml

$ ispin rev.pml
```

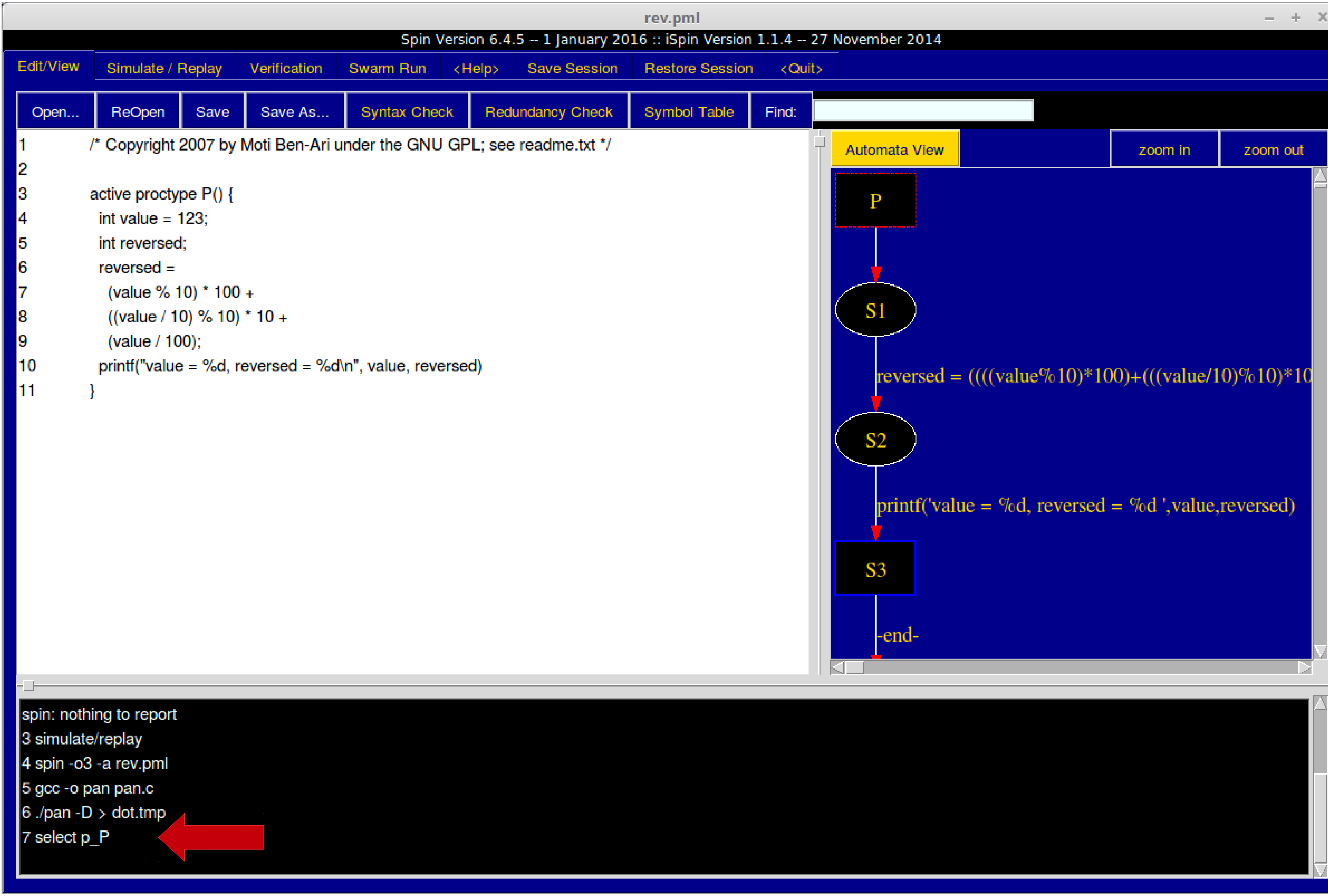

rev.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Open...
ReOpen
Save
Save As...
Syntax Check
Redundancy Check
Symbol Table
Find:
Automata View
zoom in
zoom out
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
active proctype P() {
int value = 123;
int reversed;
reversed =
(value % 10) * 100 +
((value / 10) % 10) * 10 +
(value / 100);
printf("value = %d, reversed = %d\n", value, reversed)
}
Spin Version 6.4.5 -- 1 January 2016
iSpin Version 1.1.4 -- 27 November 2014
TclTk Version 8.6/8.6
1 rev.pml:1


rev.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Open...
ReOpen
Save
Save As...
Syntax Check
Redundancy Check
Symbol Table
Find:
Automata View
zoom in
zoom out
1 /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
2
3 active proctype P() {
4 int value = 123;
5 int reversed;
6 reversed =
7 (value % 10) * 100 +
8 ((value / 10) % 10) * 10 +
9 (value / 100);
10 printf("value = %d, reversed = %d\n", value, reversed)
11 }
Spin Version 6.4.5 -- 1 January 2016
iSpin Version 1.1.4 -- 27 November 2014
TclTk Version 8.6/8.6
1 rev.pml:1
2 syntax check
spin: nothing to report


rev.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Mode
Random, with seed: 123
Interactive (for resolution of all nondeterminism)
Guided, with trail: rev.pml.trail
browse
initial steps skipped: 0
maximum number of steps: 10000
Track Data Values (this can be slow)
A Full Channel
blocks new messages
loses new messages
MSC+stmnt
MSC max text width 20
MSC update delay 25
Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:
(Re)Run
Stop
Rewind
Step Forward
Step Backward
Background command executed:
Save in: msc.ps
1 /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
2
3 active proctype P() {
4 int value = 123;
5 int reversed;
6 reversed =
7 (value % 10) * 100 +
8 ((value / 10) % 10) * 10 +
9 (value / 100);
10 printf("value = %d, reversed = %d\n", value, reversed)
Data Values
Queues


rev.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Mode
Random, with seed: 123
Interactive (for resolution of all nondeterminism)
Guided, with trail: rev.pml.trail
browse
initial steps skipped: 0
maximum number of steps: 10000
Track Data Values (this can be slow)
A Full Channel
blocks new messages
loses new messages
MSC+stmnt
MSC max text width 20
MSC update delay 25
Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:
(Re)Run
Stop
Rewind
Step Forward
Step Backward
Background command executed:
spin -p -s -r -X -v -n123 -l -g -u10000 rev.pml
Save in: msc.ps
1 /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
2
3 active proctype P() {
4 int value = 123;
5 int reversed;
6 reversed =
7 (value % 10) * 100 +
8 ((value / 10) % 10) * 10 +
9 (value / 100);
10 printf("value = %d, reversed = %d\n", value, reversed)
[variable values, step 1]
P(0):reversed = 321
value = 123, reversed = 321
0: proc - (:root:) creates proc 0 (P)
spin: rev.pml:0, warning, proctype P, 'int reversed' variable is never used (other than in print stmnts)
1: proc 0 (P:1) rev.pml:6 (state 1) [reversed = ((((value%10)*100)+(((value/10)%10)*10))+(value/100))]
2: proc 0 (P:1) rev.pml:10 (state 2) [printf('value = %d, reversed = %d\\n',value,reversed)]
2: proc 0 (P:1) terminates
1 processes created
Queues


rev.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Open...
ReOpen
Save
Save As...
Syntax Check
Redundancy Check
Symbol Table
Find:
Automata View
zoom in
zoom out
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
active proctype P() {
int value = 123;
int reversed;
reversed =
(value % 10) * 100 +
((value / 10) % 10) * 10 +
(value / 100);
printf("value = %d, reversed = %d\n", value, reversed)
}
Select:
p_P
2 syntax check
spin: nothing to report
3 simulate/replay
4 spin -o3 -a rev.pml
5 gcc -o pan pan.c
6 ./pan -D > dot.tmp


rev.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Open...
ReOpen
Save
Save As...
Syntax Check
Redundancy Check
Symbol Table
Find:
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
active proctype P() {
int value = 123;
int reversed;
reversed =
(value % 10) * 100 +
((value / 10) % 10) * 10 +
(value / 100);
printf("value = %d, reversed = %d\n", value, reversed)
}
Automata View
zoom in
zoom out
P
S1
reversed = ((((value%10)*100)+(((value/10)%10)*10
S2
printf('value = %d, reversed = %d ',value,reversed)
S3
-end-
spin: nothing to report
3 simulate/replay
4 spin -o3 -a rev.pml
5 gcc -o pan pan.c
6 ./pan -D > dot.tmp
7 select p_P

iSpin, Automata View:

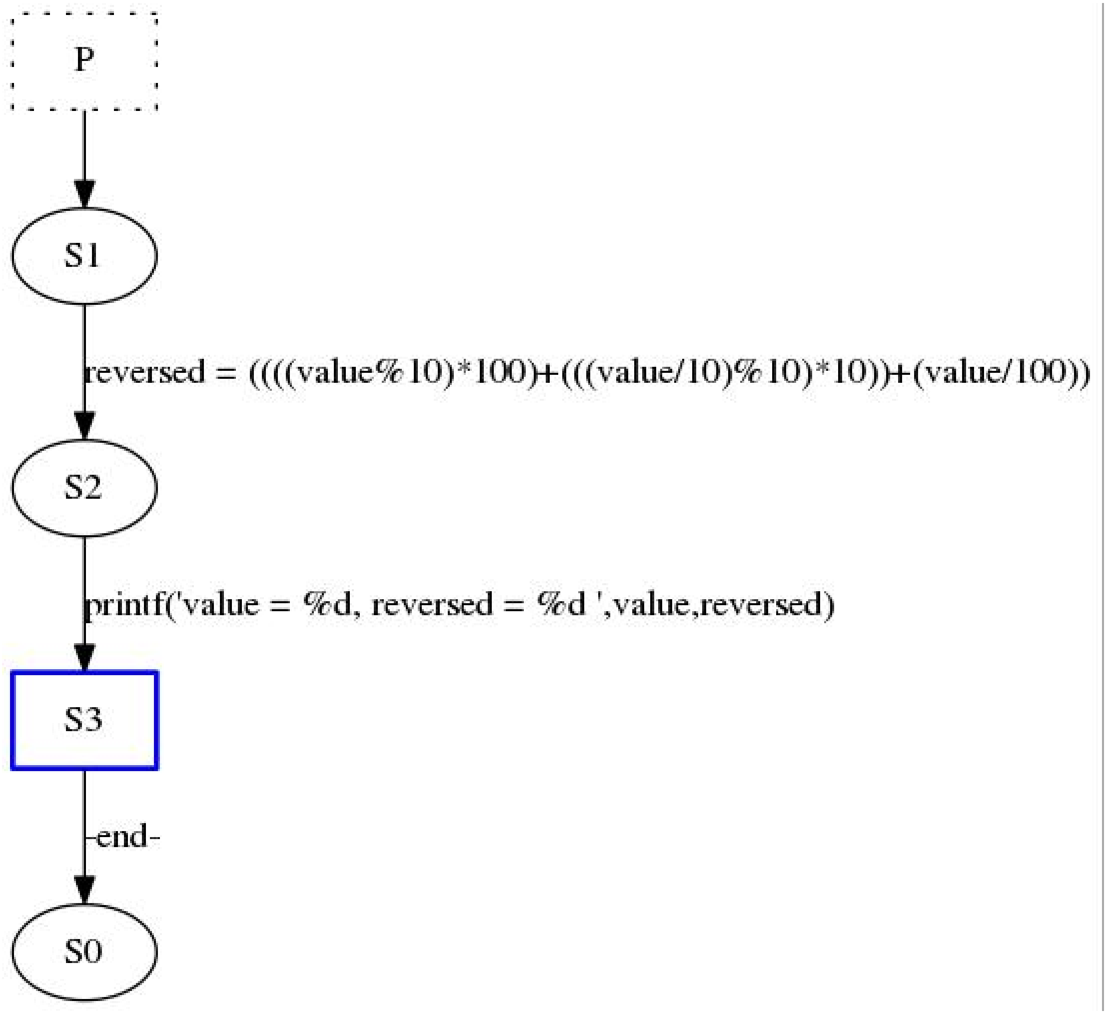
```
$ spin -o3 -a rev.pml        # generate a verifier in pan.c
$ gcc -o pan pan.c           # compile the verifier
$ ./pan -D > dot.tmp         # print state tables in dot-format and stop
```

Salida de la opción **D** de pan:

```
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];
}
```

```
$ dot < dot.tmp > dot.out

$ cat dot.out
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,
...
```

```
$ dot -Tps dot.tmp -o rev.ps

$ dot -Tjpg dot.tmp -o rev.jpg
```



Numeric data types

Type	Values	Size (bits)
bit, bool	0, 1, false, true	1
byte	0..255	8
pid	0..255	8
short	-32768..32767	16
int	$-2^{31}..2^{31}-1$	32
unsigned	$0..2^{n}-1$	≤ 32

typename name [= *anyexpr*]

unsigned *name* : constant [= *anyexpr*]

```
unsigned x : 5 = 15;   /* stored in 5 bits */
```

Warning

All variables are initialized by default to zero, but it is recommended that explicit initial values always be given in variable declarations.

- There is no separate character type in PROMELA. Literal character values can be assigned to variables of type **byte** and printed using the **%c** format specifier.
- There are no string variables in PROMELA. Messages are best modeled using just a few numeric codes and the full text is not needed. In any case, **printf** statements are only used as a convenience during simulation and are ignored when SPIN performs a verification.
- There are no floating-point data types in PROMELA. Floating-point numbers are generally not needed in models because the exact values are not important; it is better to model a numeric variable by a handful of discrete values such as minimum, low, high, maximum.

Initial values of variables

The recommendation to give explicit initial values is driven not only by good programming practice; it can also affect the size of models in SPIN.

For example, if you need to model positive integer values and write

```
byte n;
n = 1;
```

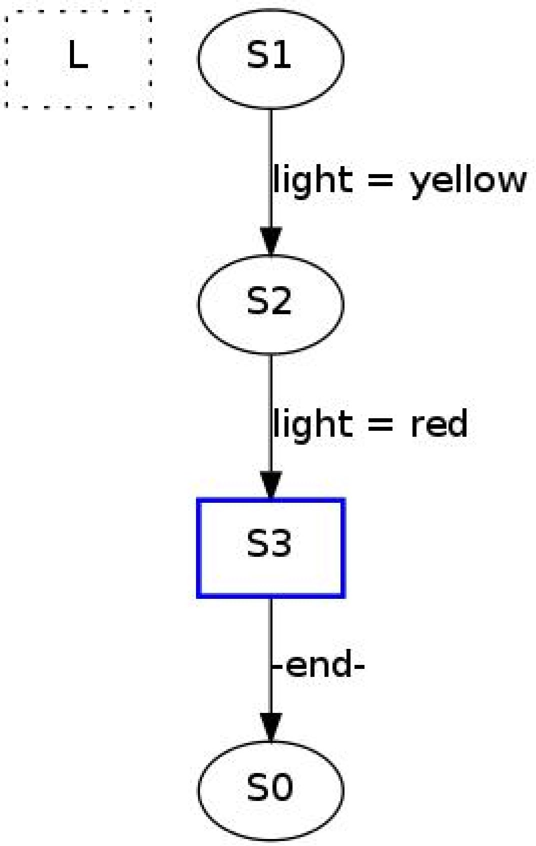
there will be additional (and unnecessary) states in which the value of n is zero.

Operators in Promela

Precedence	Operator	Associativity	Name
14	( )	left	parentheses
14	[ ]	left	array indexing
14	.	left	field selection
13	!	right	logical negation
13	~	right	bitwise complementation
13	++,--	right	increment, decrement
12	*,/,%	left	multiplication, division, modulo
11	+,-	left	addition, subtraction
10	<<,>>	left	left and right bitwise shift
9	<,<=,>,>=	left	arithmetic relational operators
8	==,!=	left	equality, inequality
7	&	left	bitwise and
6	^	left	bitwise exclusive or
5	|	left	bitwise inclusive or
4	&&	left	logical and
3	||	left	logical or
2	(-> :)	right	conditional expression
1	=	right	assignment

Symbolic names

```
#define N 10

#define red 3
#define yellow 2
#define green 1


$ cat -n light_0A.pml
/* light_0A.pml */

mtype = { red, yellow, green };

active proctype L() {

    mtype light = green;

    light = yellow;
    light = red;
}
```



Symbolic names

```
$ cat -n light_0A1.pml
     1 /* light_0A1.pml */
     2
     3 mtype = { red, yellow, green } /* ; <- optional since 6.3.0 */
     4
     5 active proctype L() {
     6
     7     mtype light = green
     8
     9     printf("red = %d, yellow = %d, green = %d\n", red, yellow, green)
    10
    11     printf("light = %e (%d)\n", light, light)
    12
    13     light = yellow
    14     printf("light = %e (%d)\n", light, light)
    15     light = red
    16     printf("light = %e (%d)\n", light, light)
    17 }
$ spin light_0A1.pml
      red = 3, yellow = 2, green = 1
      light = green (1)
      light = yellow (2)
      light = red (3)
1 process created
```

Symbolic names

```
$ cat -n light_0A2.pml
/* light_0A2.pml */

// requires Spin Version 6.4.8 or later

mtype { one, two, three }

mtype:fruit = { apple, pear, banana }

mtype:sizes = { small, medium, large }

proctype recipient(mtype:fruit z; mtype y) {
    atomic {
        printf("z: "); printm(z); printf("\n")
        printf("y: "); printm(y); printf("\n")
    }
}

...
```

Symbolic names

```
...
init {
    mtype numbers
    mtype:fruit snack
    mtype:sizes package

    run recipient(pear, two)

    numbers = one
    snack = pear
    package = large

    printm(numbers)
    printm(snack)
    printm(package)
    printf("\n")
}

$ spin light_0A2.pml
          z:           pear
          y:           two
      one       pear       large
2 processes created
```

Symbolic names

NAME
mtype - for defining symbolic names of numeric constants.

SYNTAX
```
mtype [:name][ = ] { name [, name ]* }
mtype [:name] name [ = mtype_name ]
mtype [:name] name '[' const ']'[ = mtype_name ]
```

DESCRIPTION
An mtype declaration allows for the introduction of symbolic names for constant values. There can be multiple mtype declarations in a verification model. If multiple declarations are given, they are equivalent to a single mtype declaration that contains the concatenation of all separate lists of symbolic names.

... the keyword mtype can be followed by a colon and a name (as in mtype:fruit = { apple, pear }) to indicate a specific subtype. If so, then every use of the mtype keyword must be followed by the same suffix for all variables declared to be of that subtype.

Repetitions

```
$ cat -n light_0B.pml
/* light_0B.pml */

active proctype L() {

    do
    :: skip
    od
}
```

skip ≡ true ≡ (1)

Repetitions

```
$ cat -n ex_1a.pml
     1  // http://spinroot.com/spin/Doc/Exercises.html
     2
     3  init {
     4          byte i  // initialized to 0 by default
     5          do
     6          :: i++  // increment i by one
     7          od
     8  }

$ spin ex_1a.pml
spin: ex_1a.pml:6, Error: value (256->0 (8)) truncated in
assignment
spin: ex_1a.pml:6, Error: value (256->0 (8)) truncated in
assignment
spin: ex_1a.pml:6, Error: value (256->0 (8)) truncated in
assignment
...
<Ctrl-C>
```

Repetitions

```
$ spin -u100 ex_1a.pml
-------------
depth-limit (-u100 steps) reached
#processes: 1
100:    proc  0 (:init::1) ex_1a.pml:5 (state 2)
1 process created

$ spin -p -l -u4 ex_1a.pml
  0:    proc  - (:root:) creates proc  0 (:init:)
  1:    proc  0 (:init::1) ex_1a.pml:6 (state 1)        [i = (i+1)]
                :init:(0):i = 1
  2:    proc  0 (:init::1) ex_1a.pml:8 (state 3)        [.(goto)]
  3:    proc  0 (:init::1) ex_1a.pml:6 (state 1)        [i = (i+1)]
                :init:(0):i = 2
  4:    proc  0 (:init::1) ex_1a.pml:8 (state 3)        [.(goto)]
-------------
depth-limit (-u4 steps) reached
#processes: 1
  4:    proc  0 (:init::1) ex_1a.pml:5 (state 2)
1 process created
```

Repetitions

```
$ spin -p -l -u512 ex_1a.pml
  0:    proc  - (:root:) creates proc  0 (:init:)
  1:    proc  0 (:init::1) ex_1a.pml:6 (state 1)        [i = (i+1)]
                :init:(0):i = 1
  2:    proc  0 (:init::1) ex_1a.pml:8 (state 3)        [.(goto)]
...
509:    proc  0 (:init::1) ex_1a.pml:6 (state 1)        [i = (i+1)]
                :init:(0):i = 255
510:    proc  0 (:init::1) ex_1a.pml:8 (state 3)        [.(goto)]
spin: ex_1a.pml:6, Error: value (256->0 (8)) truncated in assignment
511:    proc  0 (:init::1) ex_1a.pml:6 (state 1)        [i = (i+1)]
                :init:(0):i = 0
512:    proc  0 (:init::1) ex_1a.pml:8 (state 3)        [.(goto)]
-------------
depth-limit (-u512 steps) reached
#processes: 1
512:    proc  0 (:init::1) ex_1a.pml:5 (state 2)
1 process created
```

Repetitions

NAME
do - repetition construct.

SYNTAX
do :: *sequence* [:: *sequence*]* **od**

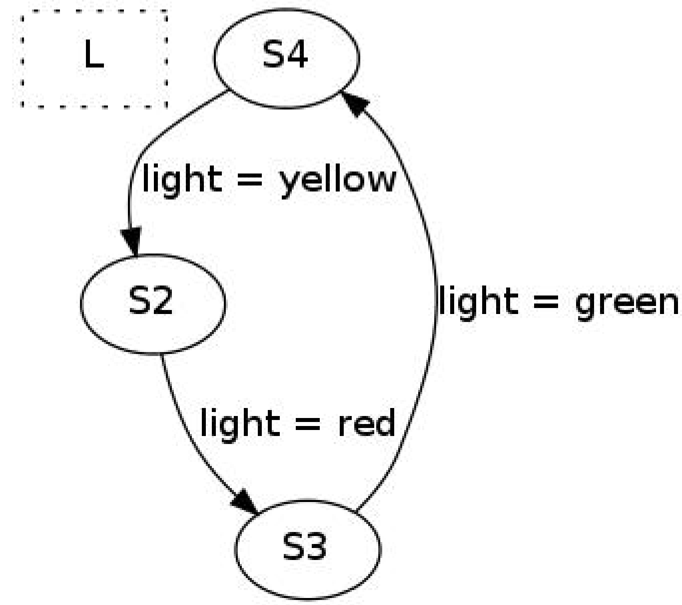
DESCRIPTION
There must be at least one option sequence in each repetition construct.
Each option sequence starts with a double-colon.
The first statement in each sequence is called *guard*.
An option can be selected for execution only when its guard statement is executable.
If more than one guard statement is executable, one of them will be selected non-deterministically.
If none of the guards are executable, the repetition construct as a whole **blocks**.
A repetition construct as a whole **is executable** if and only if at least one of its guards is executable.

Repetitions

```
$ cat -n blocked.pml
     1  active proctype Foo() {
     2      byte x=0
     3
     4      do
     5      :: x == 1    // or just 0 (or just false)
     6      od
     7  }
```

```
$ spin blocked.pml
      timeout
#processes: 1
  0:    proc  0 (Foo:1) blocked.pml:4 (state 2)
1 process created
```

Repetitions




```
$ cat -n light_0C.pml
/* light_0C.pml */

mtype = { red, yellow, green };

active proctype L() {
    mtype light = green;

    do
    :: light = yellow; light = red; light = green
    od
}
```

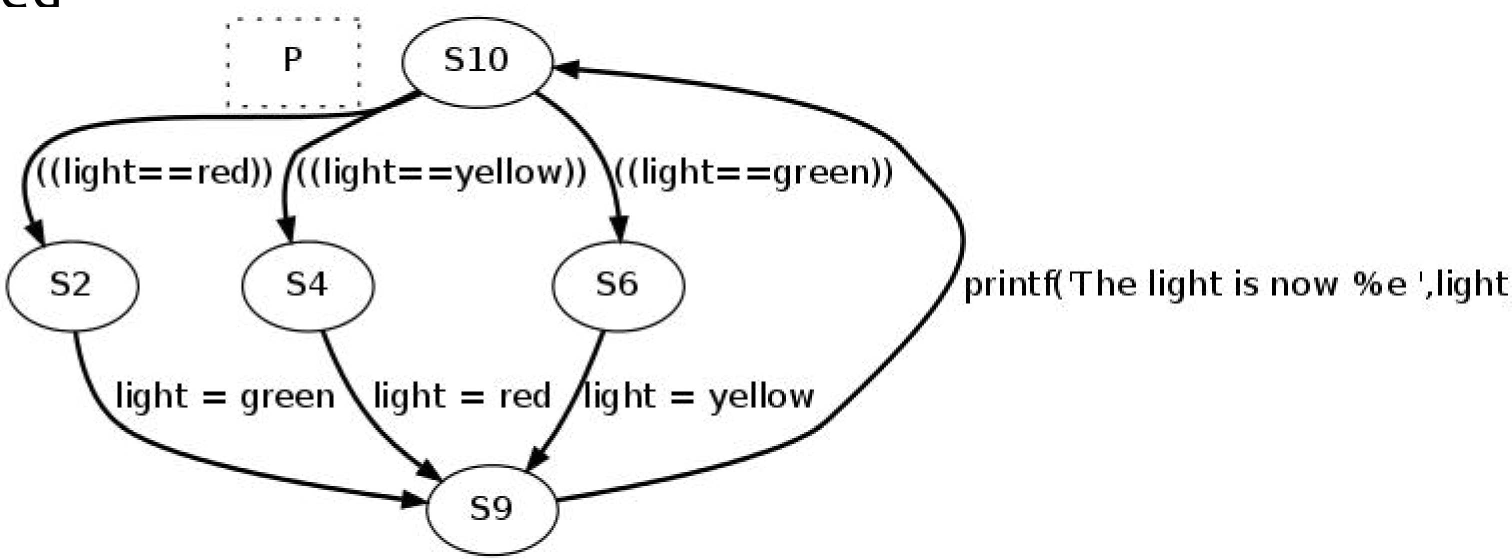
Repetitions

```
$ cat -n light_0C1.pml
     1	/* light_0C1.pml */
     2	
     3	mtype = { red, yellow, green }
     4	
     5	active proctype L() {
     6	    mtype light = green
     7	
     8	    do
     9	    :: light = yellow
    10	       light = red
    11	       light = green
    12	    od
    13	}
$ spin -u10 light_0C1.pml   # -u11, -u12, u13
-------------
depth-limit (-u10 steps) reached
#processes: 1
 10:    proc  0 (L:1) light_0C1.pml:11 (state 3)
1 process created
```

Listing 1.2. Symbolic names

```
$ cat -n mtype.pml
     1	/* Copyright 2007 by Moti Ben-Ari under the GNU GPL;
see readme.txt */
     2	mtype = { red, yellow, green };
     3	
     4	mtype light = green;
     5	
     6	active proctype P() {
     7	    do
     8	    :: if
     9	       :: light == red -> light = green
    10	       :: light == yellow -> light = red
    11	       :: light == green -> light = yellow
    12	       fi;
    13	       printf("The light is now %e\n", light)
    14	    od
    15	}
    16	
```

```
$ spin -u30 mtype.pml
      The light is now yellow
      The light is now red
      The light is now green
      The light is now yellow
      The light is now red
      The light is now green
-------------
depth-limit (-u30 steps) reached
#processes: 1
                light = green
 30:    proc  0 (P:1) mtype.pml:7 (state 10)
1 process created
```



Listing 1.3. Adding new symbolic names

```
$ cat -n mtype1.pml
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

mtype = { red, yellow, green };
mtype = { green_and_yellow, yellow_and_red };

mtype light = green;

active proctype P() {
    do
    ::  if
        :: light == red -> light = yellow_and_red
        :: light == yellow_and_red -> light = green
        :: light == green -> light = green_and_yellow
        :: light == green_and_yellow -> light = red
        fi;
        printf("The light is now %e\n", light)
    od
}
```

```
$ spin -u40 mtype1.pml
      The light is now green_and_yellow
      The light is now red
      The light is now yellow_and_red
      The light is now green
      The light is now green_and_yellow
      The light is now red
      The light is now yellow_and_red
      The light is now green
-------------
depth-limit (-u40 steps) reached
#processes: 1
                light = green
 40:    proc  0 (P:1) mtype1.pml:9 (state 12)
1 process created
```

Listing 1.3. Adding new symbolic names

```
$ cat -n mtype1A.pml
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see
readme.txt */
/* modified */

mtype = { red, yellow, green }
mtype = { green_and_yellow, yellow_and_red }

mtype light = green

active proctype P() {
    do
    ::  if
        :: light == red -> light = yellow_and_red
        :: light == yellow_and_red -> light = green
        :: light == green -> light = green_and_yellow
        :: light == green_and_yellow -> light = red
        fi
        printf("The light is now %e (%d)\n", light, light)
    od
}

```

```
$ spin -u40 mtype1A.pml
      The light is now green_and_yellow (5)
      The light is now red (3)
      The light is now yellow_and_red (4)
      The light is now green (1)
      The light is now green_and_yellow (5)
      The light is now red (3)
      The light is now yellow_and_red (4)
      The light is now green (1)
-------------
depth-limit (-u40 steps) reached
#processes: 1
                light = green
 40:    proc  0 (P:1) mtype1A.pml:10 (state 12)
1 process created
```

Repetitions: **do**

```
$ cat -n counter.pml
     1 byte count
     2
     3 active proctype counter()
     4 {
     5     do
     6     :: count++              /* unconditionally executable */
     7     :: count--              /* unconditionally executable */
     8     :: (count == 0) ->
     9         break
    10     od
    11 }

$ spin counter.pml
1 process created

$ spin counter.pml
spin: counter.pml:6, Error: value (256->0 (8)) truncated in assignment
spin: counter.pml:7, Error: value (-1->255 (8)) truncated in assignment
spin: counter.pml:6, Error: value (256->0 (8)) truncated in assignment
1 process created
```

Repetitions: Spin option `-i` (interactive (random simulation))

```
$ spin -i counter.pml
Select stmnt (proc  0 (counter:1) )
        choice 1: count = (count+1)
        choice 2: count = (count-1)
        choice 3: ((count==0))
Select [0-3]: 1
Select stmnt (proc  0 (counter:1) )
        choice 1: count = (count+1)
        choice 2: count = (count-1)
Select [0-3]: 2
Select stmnt (proc  0 (counter:1) )
        choice 1: count = (count+1)
        choice 2: count = (count-1)
        choice 3: ((count==0))
Select [0-3]: 3
1 process created
```

Repetitions: Spin option -p (print all statements)

```
$ spin -i -p counter.pml
  0:    proc  - (:root:) creates proc  0 (counter)
Select stmnt (proc  0 (counter:1) )
        choice 1: count = (count+1)
        choice 2: count = (count-1)
        choice 3: ((count==0))
Select [0-3]: 1
  1:    proc  0 (counter:1) counter.pml:6 (state 1)     [count = (count+1)]
  2:    proc  0 (counter:1) counter.pml:11 (state 1)    [.(goto)]
Select stmnt (proc  0 (counter:1) )
        choice 1: count = (count+1)
        choice 2: count = (count-1)
Select [0-3]: 2
  3:    proc  0 (counter:1) counter.pml:7 (state 2)     [count = (count-1)]
  4:    proc  0 (counter:1) counter.pml:11 (state 1)    [.(goto)]
Select stmnt (proc  0 (counter:1) )
        choice 1: count = (count+1)
        choice 2: count = (count-1)
        choice 3: ((count==0))
Select [0-3]: 3
  5:    proc  0 (counter:1) counter.pml:8 (state 3)     [((count==0))]
  6:    proc  0 (counter:1) counter.pml:9 (state 4)     [goto :b0]
  6:    proc  0 (counter:1)       terminates
1 process created
```

Repetitions: Spin option -g (print all global variables)

```
$ spin -i -p -g counter.pml
  0:    proc  - (:root:) creates proc  0 (counter)
Select stmnt (proc  0 (counter:1) )
        choice 1: count = (count+1)
        choice 2: count = (count-1)
        choice 3: ((count==0))
Select [0-3]: 1
  1:    proc  0 (counter:1) counter.pml:6 (state 1)     [count = (count+1)]
                count = 1
  2:    proc  0 (counter:1) counter.pml:11 (state 1)    [.(goto)]
Select stmnt (proc  0 (counter:1) )
        choice 1: count = (count+1)
        choice 2: count = (count-1)
Select [0-3]: 2
  3:    proc  0 (counter:1) counter.pml:7 (state 2)     [count = (count-1)]
                count = 0
  4:    proc  0 (counter:1) counter.pml:11 (state 1)    [.(goto)]
Select stmnt (proc  0 (counter:1) )
        choice 1: count = (count+1)
        choice 2: count = (count-1)
        choice 3: ((count==0))
Select [0-3]: 3
  5:    proc  0 (counter:1) counter.pml:8 (state 3)     [((count==0))]
  6:    proc  0 (counter:1) counter.pml:9 (state 4)     [goto :b0]
  6:    proc  0 (counter:1)       terminates
1 process created
```

Repetitions: to force termination

```
$ cat -n counter1.pml
     1 byte count
     2
     3 active proctype counter()
     4 {
     5     do
     6     ::  count != 0 ->
     7             if
     8             :: count++
     9             :: count--
    10             fi
    11     ::  else ->
    12             break
    13     od
    14 }

$ spin counter1.pml
1 process created
```

¿Cómo funciona este programa?

¿Cuántas iteraciones se hacen?

Repetitions: to force termination

```
$ spin -i -p -g counter1.pml
  0:    proc  - (:root:) creates proc  0 (counter)
Select stmnt (proc  0 (counter:1) )
        choice 2: (else)
  1:    proc  0 (counter:1) counter1.pml:5 (state 8)    [DO]
  2:    proc  0 (counter:1) counter1.pml:12 (state 7)   [goto :b0]
  2:    proc  0 (counter:1)       terminates
1 process created
```

An else condition statement is executable if and only if no other statement within the same process is executable at the same local control state

¿Qué sucede aquí?

```
$ cat -n counter2.pml
     1	byte count = 4
     2	
     3	active proctype counter()
     4	{
     5	    do
     6	    ::  printf("count = %d\n", count)    /* is always executable! */
     7	        count != 0 ->
     8	            if
     9	            :: count++
    10	            :: count--
    11	            fi
    12	    ::  else ->
    13	            break
    14	    od
    15	}

$ spin counter2.pml
      count = 4
      count = 5
      count = 4
      count = 3
      count = 2
      count = 1
      count = 0
      timeout
#processes: 1
                count = 0
 51:    proc  0 (counter:1) counter2.pml:7 (state 2)
1 process created
```

¿Qué sucede aquí?

```
$ cat -n counter3.pml
     1	byte count = 4
     2	
     3	active proctype counter()
     4	{
     5		do
     6		::	count != 0 ->
     7			printf("count = %d\n", count)
     8			if
     9			:: count++
    10			:: count--
    11			fi
    12		::	else ->
    13			break
    14		od
    15	}

$ spin counter3.pml
      count = 4
      count = 5
      count = 4
      count = 3
      count = 2
      count = 1
1 process created
```

1.5 Control statements

The control statements are taken from a formalism called **guarded commands** invented by E.W. Dijkstra.

There are five control statements:

- sequence,
- selection,
- repetition,
- jump, and
- **`unless`**.

The semicolon is the ***separator*** between statements that are executed in sequence.

When a processor executes a program, a register called a ***location counter*** (*program counter – pc*, *instruction counter – ic*) maintains the address of the next instruction that can be executed. An address of an instruction is called a ***control point***. For example, in PROMELA the sequence of statements

```
x = y + 2;
z = x * y;
printf("x = %d, z = %d\n", x, z)
```

has three control points, one before each statement, and the location counter of a process can be at any one of them.

Listing 1.4. Discriminant of a quadratic equation

```
$ cat -n if1.pml
     1 /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
     2
     3 active proctype P() {
     4     int a = 1, b = -4, c = 4;
     5     int disc;
     6     disc = b * b - 4 * a * c;
     7     if
     8     :: disc < 0  -> printf("disc = %d: no real roots\n", disc)
     9     :: disc == 0 -> printf("disc = %d: duplicate real roots\n", disc)
    10     :: disc > 0  -> printf("disc = %d: two real roots\n", disc)
    11     fi
    12 }

$ spin if1.pml
      disc = 0: duplicate real roots
1 process created
```

Listing 1.5. Number of days in a month

```
$ cat -n if2.pml
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

active proctype P() {
    byte days;
    byte month = 2;
    int year = 2000;
    if
    :: month == 1 || month == 3  || month == 5 || month == 7 ||
       month == 8 || month == 10 || month == 12 ->
           days = 31
    :: month == 4 || month == 6  || month == 9 || month == 11 ->
           days = 30
    :: month == 2 && year % 4 == 0 && /* Leap year */
       (year % 100 != 0 || year % 400 == 0) ->
           days = 29
    :: else ->
           days = 28
    fi;
    printf("month = %d, year = %d, days = %d\n", month, year, days)
}
```

```
$ spin if2.pml
      month = 2, year = 2000, days = 29
1 process created
```

Warning

The **else** guard is not the same as a guard consisting of the constant **true**. The latter can ***always*** be selected even if there are other guards that evaluate to true, while the former is only selected if all other guards evaluate to false.

Listing 1.6. Maximum of two values

```
$ cat -n max.pml
     1 /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
     2
     3 active proctype P() {
     4     int a = 5, b = 5;
     5     int max;
     6     int branch;
     7     if
     8     :: a >= b -> max = a; branch = 1;
     9     :: b >= a -> max = b; branch = 2;
    10     fi;
    11     printf("The maximum of %d and %d = %d by branch %d\n",
    12             a, b, max, branch);
    13 }

$ spin max.pml
      The maximum of 5 and 5 = 5 by branch 1
1 process created

$ spin max.pml
      The maximum of 5 and 5 = 5 by branch 2
1 process created
```

Advanced: Arrows as separators

```
...
if
:: disc < 0  ; printf(...)
:: disc == 0 ; printf(...)
:: disc > 0  ; printf(...)
fi
...
```

1.6.1 Conditional expressions

A conditional expression enables you to obtain a value that depends on the result of evaluating a boolean expression:

```
max = (a > b -> a : b)
```
(*)

The value `max` is assigned the value of `a` if `a > b`; otherwise, it is assigned the value of `b`. A conditional expression *must* be contained within parentheses.

An assignment statement like (*) is an atomic statement, while the **`if`**-statement is not:

```
if
:: a >= b -> max = a
:: b >= a -> max = b
fi
```

Instruction Cycle with Interrupts (from W. Stallings)

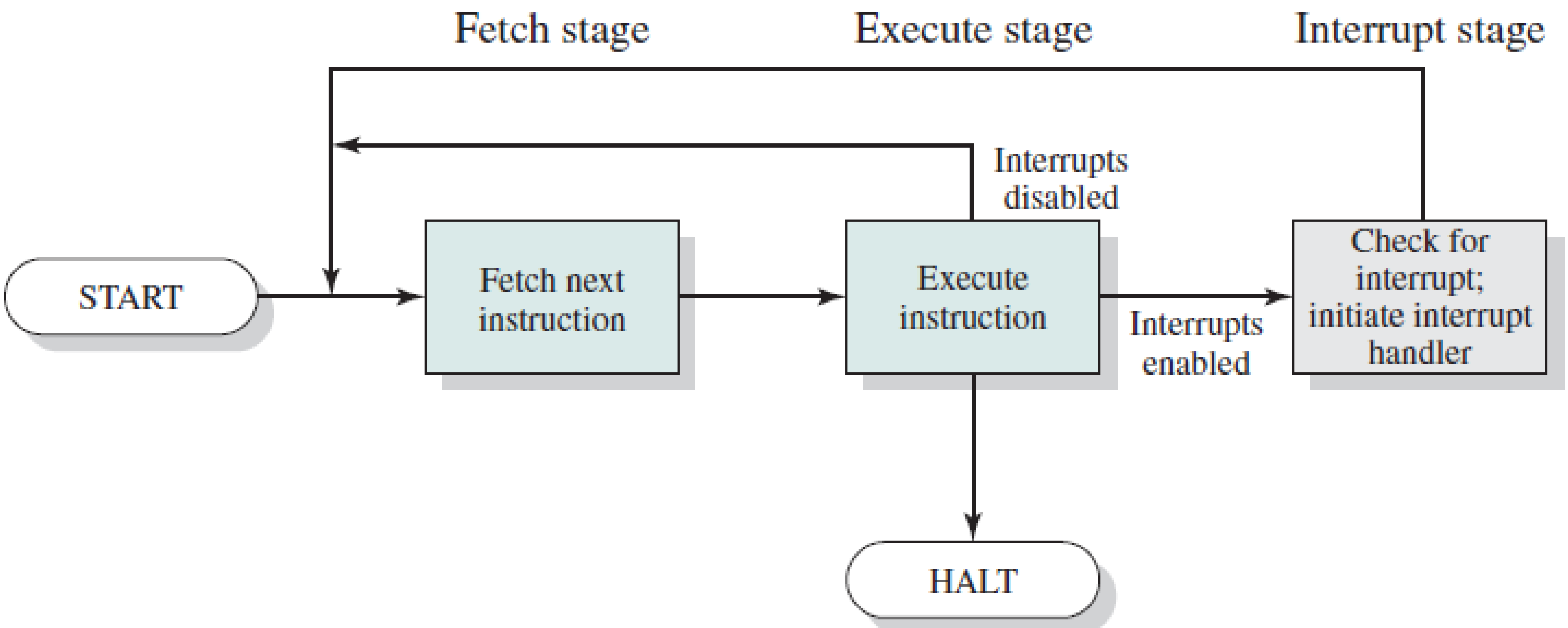


Figure 1.7 Instruction Cycle with Interrupts

1.7 Repetitive statements

There is one repetitive statement in Promela, the **`do`**-statement.

The syntax of the **`do`**-statement is the same as that of the **`if`**-statement, except that the keywords are **`do`** and **`od`**.

The semantics is similar, consisting of the evaluation of the guards, followed by the execution of the sequence of statements following one of the true guards. For a **`do`**-statement, completion of the sequence of statements causes the execution to return to the beginning of the **`do`**-statement and the evaluation of the guards is begun again.

Termination of a loop is accomplished by **`break`**, which is not a statement but rather an indication that control passes from the current location to the statement following the **`od`**.

Listing 1.7. Greatest common denominator

```
$ cat -n gcd.pml
     1 /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
     2
     3 active proctype P() {
     4     int x = 15, y = 20;
     5     int a, b;
     6     a = x; b = y;
     7     do
     8     :: a > b -> a = a - b
     9     :: b > a -> b = b - a
    10     :: a == b -> break
    11     od;
    12     printf("The GCD of %d and %d = %d\n", x, y, a);
    13     assert (x % a == 0 && y % a == 0)
    14 }

$ spin gcd.pml
      The GCD of 15 and 20 = 5
1 process created
```

Listing 1.8. A counting loop

```
$ cat -n counting.pml
     1 /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
     2
     3 active proctype P() {
     4     int N = 10;
     5     int sum = 0;
     6
     7     byte i;
     8     i = 1;
     9     do
    10     :: i > N -> break
    11     :: else ->
    12         sum = sum + i;
    13         i++
    14     od;
    15     printf("The sum of the first %d numbers = %d\n", N, sum);
    16 }

$ spin counting.pml
      The sum of the first 10 numbers = 55
1 process created
```

Listing 1.9. Counting with a for-loop macro

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

#include "for.h"
active proctype P() {
    int N = 10;
    int sum = 0;

    for (i, 1, N)
        sum = sum + i
    rof (i);
    printf("The sum of the first %d numbers = %d\n", N, sum);
}

$ spin for.pml
      The sum of the first 10 numbers = 55
1 process created
```

Macros for counting loops

```
$ cat for.h
/* Copyright (C) 2006 M. Ben-Ari. See copyright.txt */
/* Macros for for-loop */
#define for(I,low,high) byte I; I = low ; do :: ( I > high ) -> break :: else ->
#define rof(I) ; I++ od

$
```

Since Spin version 6

NAME
for - deterministic iteration statement.

SYNTAX
for '(' varref ':' expr '..' expr ')' '{' *sequence* '}'
for '(' varref in array ')' '{' *sequence* '}'
for '(' varref in channel ')' '{' *sequence* '}'

DESCRIPTION
for statements are internally converted into the corresponding Promela code, with the first statement issued being an assignment statement. This means that for statements are always executable (the guard statement is an assignment), independent of what the guard of the sequence in the body of the **for** is. Execution could of course still block inside the body of the **for** statement.

for example #1 (... . . ...)

```
$ cat -n for6A.pml
     1 int i
     2
     3 active proctype for_example() {
     4
     5     for (i : 1 .. 10) {
     6         printf("i = %d\n", i)
     7     }
     8 }

$ spin for6A.pml
      i = 1
      i = 2
      i = 3
      i = 4
      i = 5
      i = 6
      i = 7
      i = 8
      i = 9
      i = 10
1 process created
```

for example #1 (do ... od equivalence)

```
$ cat -n for6A1.pml
     1 active proctype for_example() {
     2     int i=1;
     3
     4     do
     5     :: i <= 10 -> printf("i = %d\n", i); i++
     6     :: else -> break
     7     od
     8 }

$ spin for6A1.pml
      i = 1
      i = 2
      i = 3
      i = 4
      i = 5
      i = 6
      i = 7
      i = 8
      i = 9
      i = 10
1 process created
```

for example #2 (... in ...)

```
$ cat -n for6B.pml
     1 int a[10], i
     2
     3 active proctype for_example() {
     4
     5     for (i in a) {  /* the index values of the array are generated */
     6         printf("i = %d, a[%d] = %d\n", i, i, a[i])
     7     }
     8 }

$ spin for6B.pml
      i = 0, a[0] = 0
      i = 1, a[1] = 0
      i = 2, a[2] = 0
      i = 3, a[3] = 0
      i = 4, a[4] = 0
      i = 5, a[5] = 0
      i = 6, a[6] = 0
      i = 7, a[7] = 0
      i = 8, a[8] = 0
      i = 9, a[9] = 0
1 process created
```

for example #2 (do . . . od equivalence)

```
$ cat -n for6B1.pml
     1 active proctype for_example() {
     2     int i=0;
     3
     4     do
     5     :: i < 10 -> printf("i = %d\n", i); i++
     6     :: else -> break
     7     od
     8 }

$ spin for6B1.pml
      i = 0
      i = 1
      i = 2
      i = 3
      i = 4
      i = 5
      i = 6
      i = 7
      i = 8
      i = 9
1 process created
```

Since Spin version 6

NAME
select - non-deterministic value selection.

SYNTAX
```
select '(' varref ':' expr '..' expr ')'
```

DESCRIPTION
select statements are internally converted into the corresponding Promela code, with the first statement issued being an assignment statement. This means that **select** statements are always executable (the guard statement is an assignment), but can take several steps to execute. More precisely, if there are N values in the range to choose from, then the **select** statement can take between 1 and N steps to arrive at the non-deterministically chosen value.

Caution 1: Because the **select** is implemented with a non-deterministic do-loop (making it possible to use expressions for the ranges that are evaluated at run-time), you will not get very random behavior in simulation runs. Note that each time through the loop, a random simulation will give equal odds to selecting the end of the loop or its continuation, until the upper-limit is reached. That means that values close to the start are much more likely to be picked in simulation runs than values close to the end. The behavior in verification runs is of course guaranteed to be correct, with all possible choices being verified.

select example

```
$ cat -n select6A.pml
     1 active proctype select_example() {
     2     int i
     3
     4     do
     5     :: i == 4 -> break
     6     :: else -> select (i : 1 .. 4)
     7                printf("i = %d\n", i)
     8     od
     9 }

$ spin select6A.pml
      i = 1
      i = 1
      i = 1
      i = 1
      i = 1
      i = 4
1 process created
```

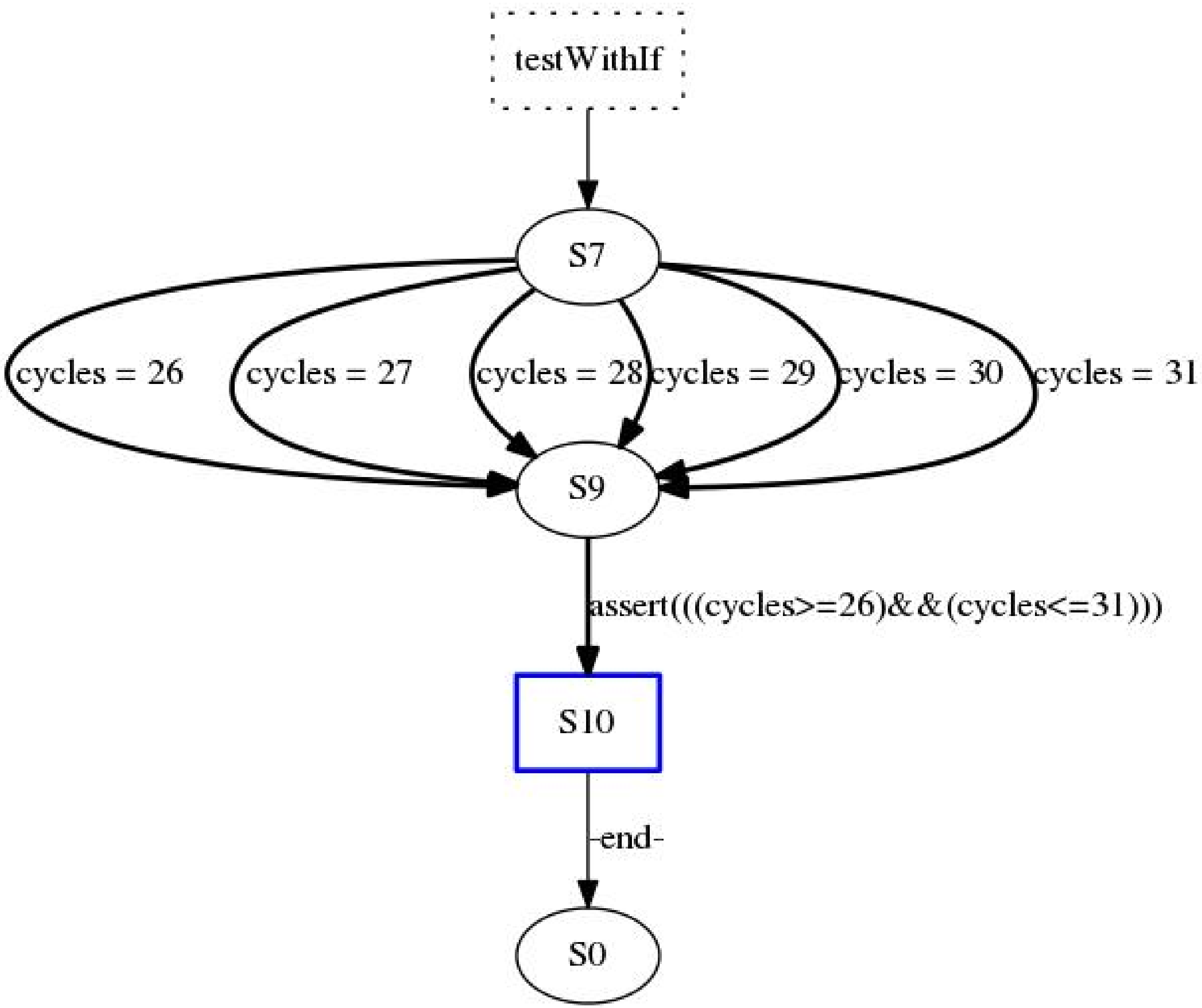
select example (expanded Promela code)

```
$ cat -n select6A1.pml
     1 active proctype select_example() {
     2     int i
     3
     4     do
     5     :: i == 4 -> break
     6     :: else -> i = 1
                      do
                      :: i < 4 -> i++
                      :: break              /* always executable */
                      od
     7                printf("i = %d\n", i)
     8     od
     9 }

$ spin select6A1.pml
      i = 1
      i = 2
      i = 1
      < 4 veces más >
      i = 3
      i = 2
      i = 1
      i = 2
      i = 4
1 process created
```

http://stackoverflow.com/questions/22664165/select-statement-in-promela-much-slower-than-the-equivalent-if-statement

```
$ cat -n testWithIf.pml
     1 int cycles
     2
     3 active proctype testWithIf() {
     4     if
     5     ::  cycles = 26
     6     ::  cycles = 27
     7     ::  cycles = 28
     8     ::  cycles = 29
     9     ::  cycles = 30
    10     ::  cycles = 31
    11     fi
    12
    13     assert(cycles >= 26 && cycles <= 31)
    14 }

$ spin -a testWithIf.pml

$ gcc -o pan pan.c

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

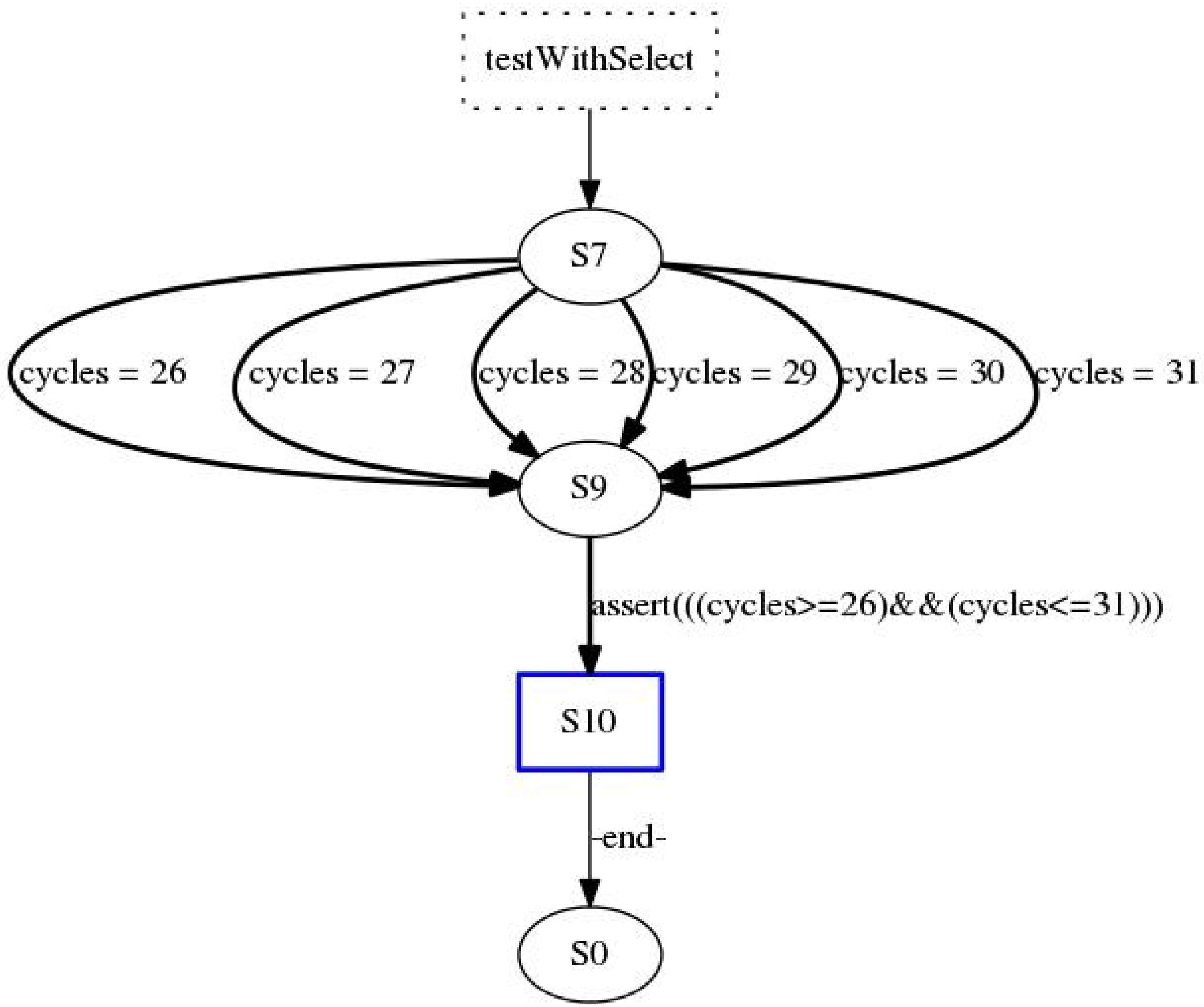
http://stackoverflow.com/questions/22664165/select-statement-in-promela-much-slower-than-the-equivalent-if-statement



http://stackoverflow.com/questions/22664165/select-statement-in-promela-much-slower-than-the-equivalent-if-statement

```
$ cat -n testWithSelect.pml
     1 int cycles
     2
     3 active proctype testWithSelect() {
     4     select(cycles: 26 .. 31)
     5
     6     assert(cycles >= 26 && cycles <= 31)
     7 }

$ spin -a testWithSelect.pml

$ gcc -o pan pan.c

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

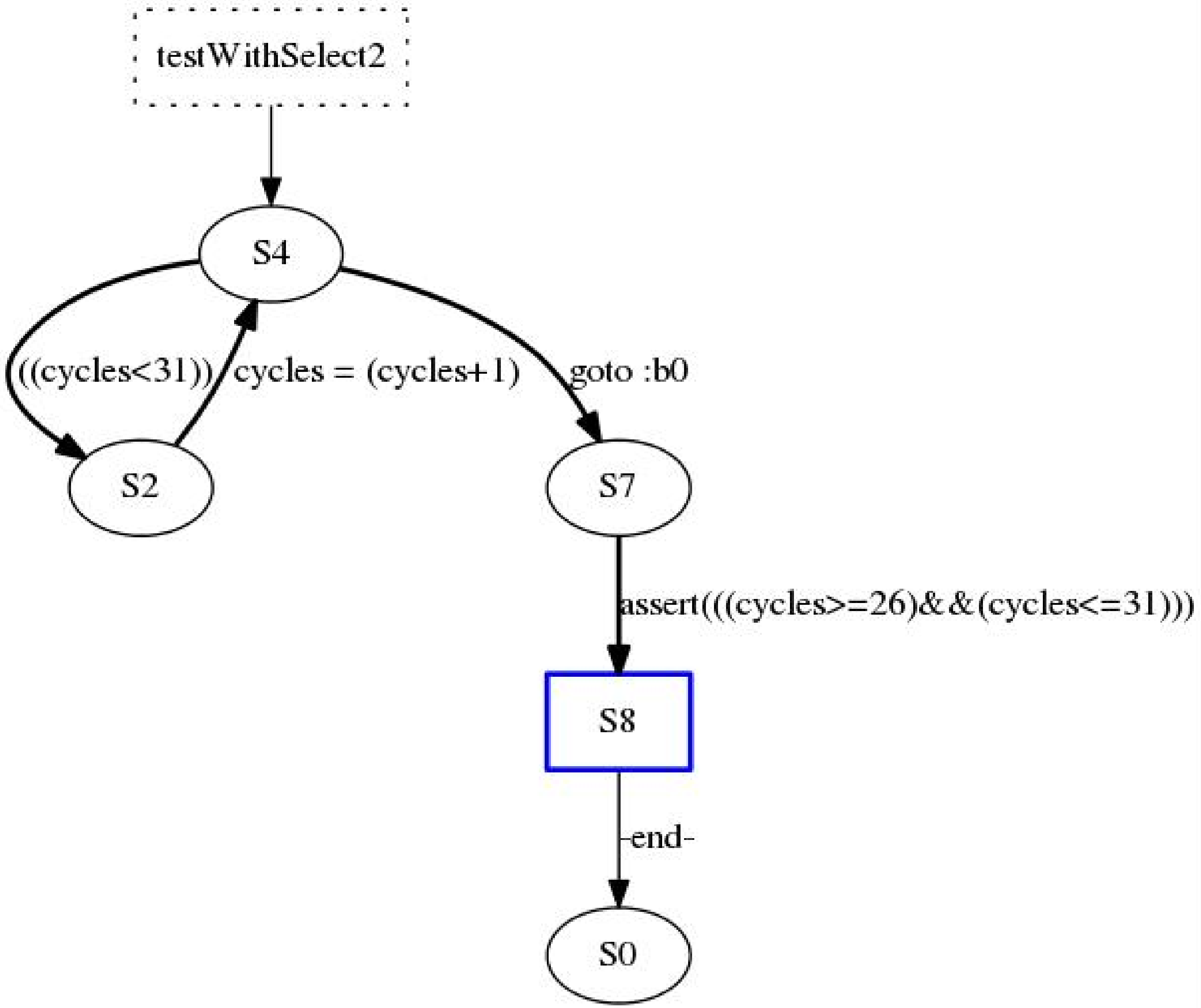
http://stackoverflow.com/questions/22664165/select-statement-in-promela-much-slower-than-the-equivalent-if-statement



http://stackoverflow.com/questions/22664165/select-statement-in-promela-much-slower-than-the-equivalent-if-statement

```
$ cat -n testWithSelect2.pml
     1	int cycles = 26
     2	
     3	active proctype testWithSelect2() {
     4	    do
     5	    :: cycles < 31 -> cycles++
     6	    :: break                        /* always executable */
     7	    od
     8	
     9	    assert(cycles >= 26 && cycles <= 31)
    10	}

$ spin -a testWithSelect2.pml

$ gcc -o pan pan.c

$ ./pan -D | dot -Tjpg -o testWithSelect2.jpg
```

http://stackoverflow.com/questions/22664165/select-statement-in-promela-much-slower-than-the-equivalent-if-statement



1.8 Jump statements

Promela contains a **goto**-statement that causes control to jump to a label, which is an identifier followed by a (single) colon. **goto** can be used instead of **break** to exit a loop:

```
  do
  :: i > N -> goto exitloop
  :: else -> ...
  od
exitloop:
  printf(...)
```

though normally the **break** is preferred since it is more structured and doesn't require a label.

There is no control point at the beginning of an alternative in an **if**- or **do**-statement, so it is a syntax error to place a label in front of a guard. Instead, there is a "joint" control point for all alternatives at the beginning of the statement.

```
$ cat -n testWithJump_error.pml
     1 int i = 0
     2
     3 active proctype testWithJump() {
     4     do
     5     :: maybelast: i > 4 -> goto exitloop
     6     :: else -> printf("%d",i)
     7                i++
     8     od
     9   exitloop:
    10     printf("%d\n",i)
    11 }
$ spin testWithJump_error.pml
error: (testWithJump_error.pml:5) label maybelast placed incorrectly
=====> instead of
do (or if)
:: ...
:: Label: statement
od (of fi)
=====> use
Label: do (or if)
:: ...
:: statement
od (or fi)
      0       1       2       3       4       5
1 process created
```

```
$ cat -n testWithJump.pml
     1 int i = 0
     2
     3 active proctype testWithJump() {
     4   checking:
     5     do
     6     :: i > 4 -> goto exitloop
     7     :: else -> printf("%d",i)
     8                i++
     9     od
    10   exitloop:
    11     printf("%d\n",i)
    12 }

$ spin testWithJump.pml
      0      1      2      3      4      5
1 process created
```