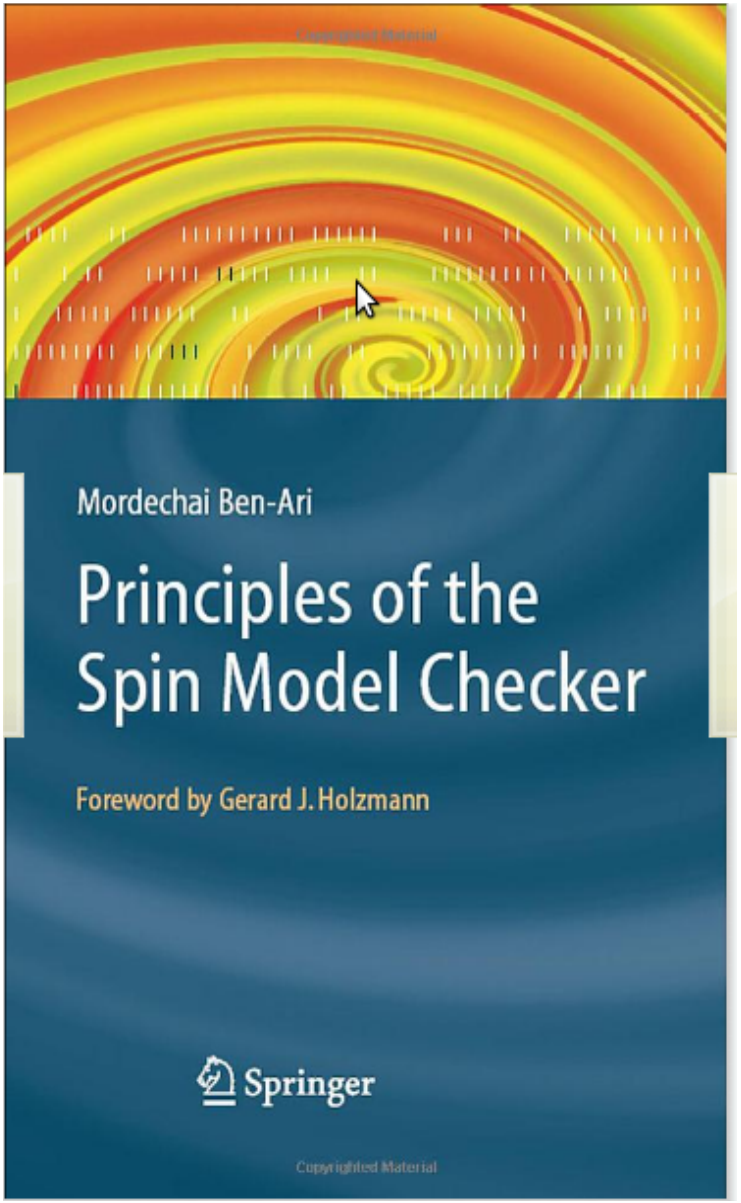
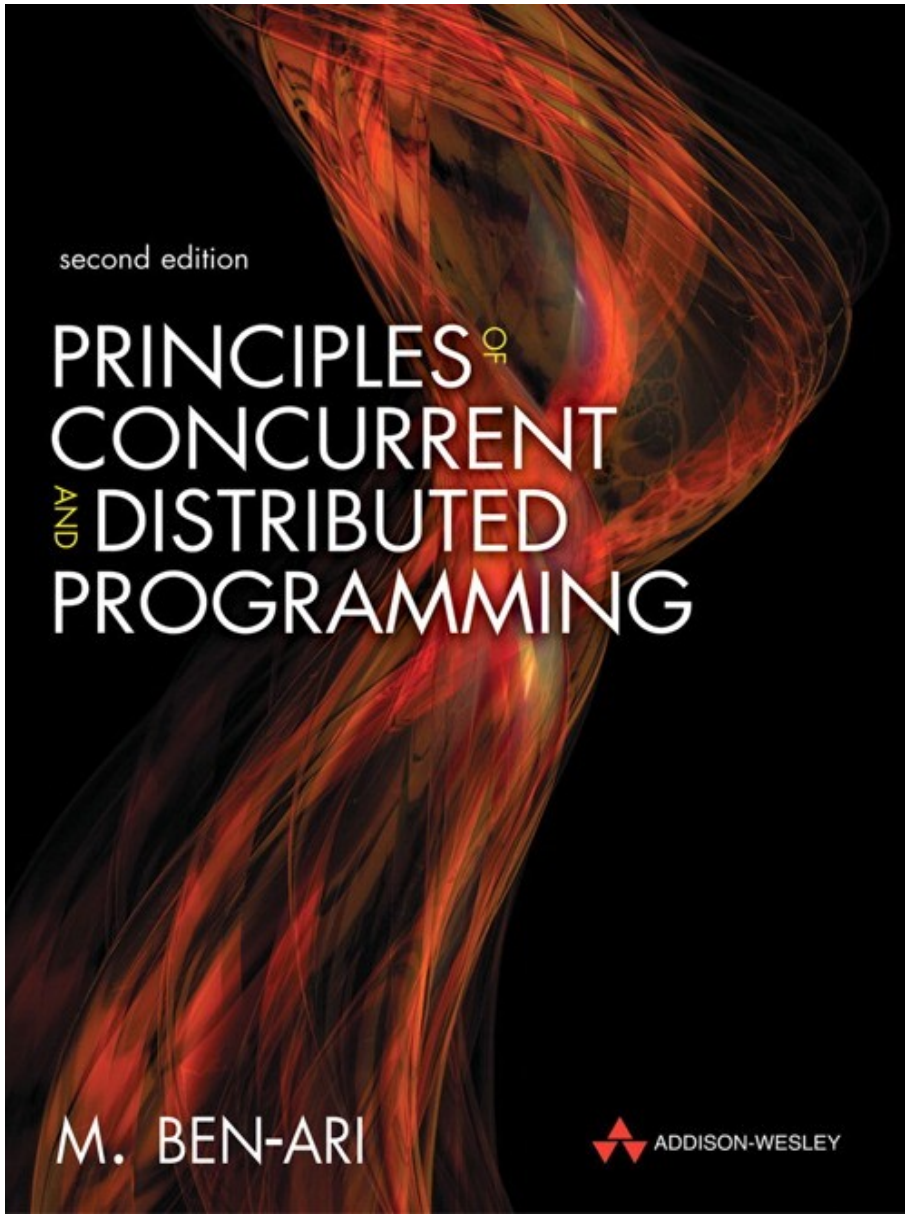


Mordechai Ben-Ari

Principles of the Spin Model Checker

Springer, 2008

ISBN: 978-1-84628-769-5



Mordechai Ben-Ari

Principles of Concurrent and Distributed Programming

Second Edition
Addison-Wesley, 2008

ISBN: 978-0-32131-283-9

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 9

Advanced Topics in Promela

Section 9.1

Specifiers for variables

NAME
hidden - for excluding global data from the state descriptor during verification.

SYNTAX
hidden *typename ivar*

DESCRIPTION
The keyword hidden can be used to prefix the declaration of any global variable to exclude the value of that variable from the definition of the global system state. The addition of this prefix can affect only the verification process, by potentially changing the outcome of state matching operations.

EXAMPLES

```
hidden byte a;
hidden short p[3];
```

NOTES
The prefix should only be used for write-only scratch variables. Alternatively, the predefined write-only scratch variable _ (underscore) can always be used instead of a hidden integer variable.

This mechanism can be used, for instance, to hide variables that are used as pseudo-local variables inside d_step sequences, provided that they are not used anywhere outside that sequence.

NAME
local - prefix on global variable declarations to assert exclusive use by a single process.

SYNTAX
local *typename ivar*

DESCRIPTION
The keyword local can be used to prefix the declaration of any global variable. It persuades the partial order reduction algorithm in the model checker to treat the variable as if it were declared local to a single process, yet by being declared global it can freely be used in LTL formulae and in never claims.

The addition of this prefix can increase the effect of partial order reduction during verification, and lower verification complexity.

EXAMPLES

```
local byte a;
local short p[3];
```

NOTES
If a variable marked as local is in fact accessed by more than one process, the partial order reduction may become invalid and the result of a verification incomplete. Such violations are not detected by the verifier.

NAME
xr , xs - for defining channel assertions.

SYNTAX
xr *name* [, *name*]*
xs *name* [, *name*]*

DESCRIPTION
Channel assertions such as

```
xr q1;
xs q2;
```

can only appear within a proctype declaration. The channel assertions are only valid if there can be at most one single instantiation of the proctype in which they appear.

The first type of assertion, xr, states that the executing process has exclusive read-access to the channel that is specified. That is, it is asserted to be the only process in the system (determined by its process instantiation number) that can receive messages from the channel.

The second type of assertion, xs, states that the process has exclusive write-access to the channel that is specified. That is, it is asserted to be the only process that can send messages to the channel.

Channel assertions have no effect in simulation runs. With the information that is provided in channel assertions, the partial order reduction algorithm that is normally used during verification, though, can optimize the search and achieve significantly greater reductions.

...

...

Any test on the contents or length of a channel referenced in a channel assertion, including receive poll operations, counts as both a read and a write access of that channel. If such access conflicts with a channel assertion, it is flagged as an error during the verification. If the error is reported, this means that the additional reductions that were applied may be invalid.

The only channel poll operations that are consistent with the use of channel assertions are nempty (consistent with the use of xr), and nfull (consistent with the use of xs). Their predefined negations empty and full have no similar benefit. The grammar prevents circumvention of the type rules by attempting constructions such as !nempty(q), or !full(q).

Summarizing: If a channel-name appears in an xs (xr) channel assertion, messages may be sent to (received from) the corresponding channel by only the process that contains the assertion, and that process can only use send (receive) operations, or the predefined operator nfull (nempty). All other types of access will generate run-time errors from the verifier.

...

...

EXAMPLES

```
    chan q = [2] of { byte };
    chan r = [2] of { byte };

    active proctype S()
    {    xs q;
         xr r;

         do
         :: q!12
         :: r?0 -> break
         od
    }
    active proctype R()
    {    xr q;
         xs r;

         do
         :: q?12
         :: r!0 -> break
         od
    }
```

...

...

NOTES
Channel assertions do not work for rendezvous channels.

For channel arrays, a channel assertion on any element of the array is applied to all elements.

In some cases, the check for compliance with the declared access patterns is too strict. This can happen, for instance, when a channel name is used as a parameter in a run statement, which is counted as both a read and a write access.

Another example of an unintended violation of a channel assertion can occur when a single process can be instantiated with different process instantiation numbers, depending on the precise moment that the process is instantiated in a run. In cases such as these, the checks on the validity of the channel assertions can be suppressed, while maintaining the reductions they allow. To do so, the verifier pan.c can be compiled with directive -DXUSAFE . Use with caution.

NAME
show - to allow for tracking of the access to specific variables in message sequence charts.

SYNTAX
show *typename name*

DESCRIPTION
This keyword has no semantic content. It only serves to determine which variables should be tracked and included in message sequence chart displays in the *Xspin* tool. Updates of the value of all variables that are declared with this prefix are maintained visually, in a separate process line, in these message sequence charts.

NOTES
The use of this prefix only affects the information that *Xspin* includes in message sequence charts, and the information that *Spin* includes in Postscript versions of message sequence charts under *Spin* option -M .

Section 9.2

Predefined variables

NAME

_ - a predefined, global, write-only, integer variable.

SYNTAX

```
_
```

DESCRIPTION

The underscore symbol _ refers to a global, predefined, write-only, integer variable that can be used to store scratch values. It is an error to attempt to use or reference the value of this variable in any context.

EXAMPLES

The following example uses a do -loop to flush the contents of a channel with two message fields of arbitrary type, while ignoring the values of the retrieved messages:

```
do
:: q?_,_
:: empty(q) -> break
od
```

NAME
_pid - a predefined, local, read-only variable of type pid that stores the instantiation number of the executing process.

SYNTAX
_pid

DESCRIPTION
Process instantiation numbers begin at zero for the first process created and count up for every new process added. The first process, with instantiation number zero, is always created by the system. Processes are created in order of declaration in the model. In the initial system state only process are created for active proctype declarations, and for an init declaration, if present. There must be at least one active proctype or init declaration in the model.

When a process terminates, it can only die and make its _pid number available for the creation of another process, if and when it has the highest _pid number in the system. This means that processes can only die in the reverse order of their creation (in stack order).

The value of the process instantiation number for a process that is created with the run operator is returned by that operator.

Instantiation numbers can be referred to locally by the executing process, through the predefined local _pid variable, and globally in never claims through remote references.

It is an error to attempt to assign a new value to this variable.

...

...

EXAMPLES

The following example shows a way to discover the _pid number of a process, and gives a possible use for a process instantiation number in a remote reference inside a never claim.

```
active [3] proctype A()
{
    printf("this is process: %d\n", _pid);
L:  printf("it terminates after two steps\n")
}

never {
    do
    :: A[0]@L -> break
    od
}
```

The remote reference in the claim automaton checks whether the process with instantiation number zero has reached the statement that was marked with the label L . As soon as it does, the claim automaton reaches its end state by executing the break statement, and reports a match. The three processes that are instantiated in the active proctype declaration can execute in any order, so it is quite possible for the processes with instantiation numbers one and two to terminate before the first process reaches label L .

NOTES

A never claim, if present, is internally also represented by the verifier as a running process. This claim process has no visible instantiation number, and therefore cannot be referred to from within the model. From the user's point of view, the process instantiation numbers are independent of the use of a never claim.

NAME
_nr_pr - a predefined, global, read-only, integer variable.

SYNTAX
_nr_pr

DESCRIPTION
The predefined, global, read-only variable _nr_pr records the number of processes that are currently running (i.e., active processes). It is an error to attempt to assign a value to this variable in any context.

EXAMPLES
The variable can be used to delay a parent process until all of the child processes that it created have terminated. The following example illustrates this type of use:

```
    proctype child()
    {
        printf("child %d\n", _pid)
    }

    active proctype parent()
    {
        do
        :: (_nr_pr == 1) ->
                run child()
        od
    }

...
```

...

The use of the precondition on the creation of a new child process in the parent process guarantees that each child process will have process instantiation number one: one higher than the parent process. There can never be more than two processes running simultaneously in this system. Without the condition, a new child process could be created before the last one terminates and dies. This means that, in principle, an infinite number of processes could result. The verifier puts the limit on the number of processes that can effectively be created at 256, so in practice, if this was attempted, the 256th attempt to create a child process would fail, and the run statement from this example would then block.

Another way to use the predefined variable is to wait for a newly started process to terminate, for instance as follows:

```
pid n;
n = _nr_pr;
run child();
(n == _nr_pr);
```

In this case we first store the current value of the number of running processes in a variable n, then start a new process, which will increase the value of _nr_pr. Then we wait until the value drops back to the original value stored in n, indicating that the newly started process (and all processes started since we assigned n) terminated.

Section 9.3

Priority

NAME

`priority` - setting and using process priorities

SYNTAX

`active [ '[' `*const***` ']' ] proctype `***name***` ( [ `***decl_lst***` ] ) priority`**
const **`{ `***sequence***` }`**

`run `*name***` ( [ `***arg_lst***` ] ) priority `***const***

`get_priority(`*expr***`)`** *(Version 6.2)*

`set_priority(`*expr***`, `***expr***`)`** *(Version 6.2)*

DESCRIPTION

In versions of Spin older than 6.2 the use of process priorities was restricted to random simulations to determine the probability that specific processes are scheduled for execution.

Starting with Spin Version 6.2, process priorities have stronger semantics, used in both simulation and verification. In this mode, a process can only execute statements if it is (one of) the highest priority process(es) in the system. Priorities can be queried and manipulated with the two new function calls **`get_priority`** and **`set_priority`**.

The new behavior can be disabled with option `spin -o6 ...`.

A process priority can be specified either with an optional parameter to a **`run`** operator, or as a suffix to an **`active proctype`** declaration. The optional **`priority`** field follows the closing brace of the parameter list in a **`proctype`** declaration.

The default execution priority for a process is 1. Higher numbers indicate higher priorities. In the original (pre Version 6.2) semantics, a priority 10 process is 10 times more likely to execute in a simulation than a priority 1 process. In the new (post Version 6.2) semantics, a priority 10 process always takes priority over a priority 1 process, unless it is blocked. Only the highest priority enabled process can execute statements. Process priorities must be in the range 1..255.

The priority specified in an **`active proctype`** declaration affects all processes that are initiated through the **`active`** prefix, but no others. A process instantiated with a **`run`** statement is always assigned the priority that is explicitly or implicitly specified there (overriding the priority that may be specified in the **`proctype`** declaration for that process).

The predefined local process variable:

```
_priority
```

holds the current priority of the executing process.

Two new predefined functions can be used to query and change process priorities. They are:

`get_priority(p)` which returns the priority of process `p`
`set_priority(p,c)` which sets priority of process `p` to `c`

The parameters `p` and `c` in these examples can be arbitrary Promela expressions.

EXAMPLES

```
run name(...) priority 3
active proctype name() priority 12 { sequence }
```

If both a `priority` clause and a `provided` clause are specified, the `priority` clause should appear first, for instance:

```
active proctype name() priority 5 provided (a < b) {...}
```

A more complete example:

```
 1    byte cnt
 2
 3    active proctype medium() priority 5
 4    {
 5        set_priority(0, 8)
 6        printf("medium %d - pid %d, pr %d, pr1 %d\n",
 7                _priority,
 8                _pid,
 9                get_priority(_pid),
10                get_priority(0))
11        cnt++
12    }
13
14    active proctype high() priority 10
15    {
16        _priority = 9
17        printf("high %d\n", _priority)
18        cnt++
19    }
20
...
```

```
    ...

21    active proctype low() priority 1
22    {
23        /*
24         * this process can only execute if/when it is the
25         * highest priority process with executable statements
26         */
27
28        assert(_priority == 1 && cnt == 2);
29        printf("low %d\n", _priority);
30    }
```

```
$ spin priority_example_1.pml
          high 9
  4: setting priority of proc 0 to 8
      medium 8 - pid 0, pr 8, pr1 8
               low 1
3 processes created
```

NAME

`provided` - for setting a global constraint on process execution.

SYNTAX

`proctype` *name* **`( [`** *decl_lst* **`] ) provided (`** *expr* **`) {`** *sequence* **`}`**

DESCRIPTION

Any **`proctype`** declaration can be suffixed by an optional **`provided`** clause to constrain its execution to those global system states for which the corresponding expression evaluates to true. The **`provided`** clause has the effect of labeling all statements in the proctype declaration with an additional, user-defined executability constraint.

EXAMPLES

The declaration:

```
byte a,b
active proctype A() provided (a > b)
{
      ...
}
```

makes the execution of all instances of **proctype** A conditional on the truth of the expression **(a>b)** , which is, for instance, not true in the initial system state. The expression can contain global references, or references to the process's **_pid** , but no references to any local variables or parameters.

If both a **priority** clause and a **provided** clause are specified, the **priority** clause should come first.

```
active proctype name() priority 2 provided (a > b)
{
     ...
}
```

Section 9.4

Modeling exceptions

NAME
unless - to define exception handling routines.

SYNTAX
stmnt unless *stmnt*

DESCRIPTION
Similar to the repetition and selection constructs, the unless construct is not really a statement, but a method to define the structure of the underlying automaton and to distinguish between higher and lower priority of transitions within a single process. The construct can appear anywhere a basic Promela statement can appear.

The first statement, generally defined as a block or sequence of basic statements, is called the main sequence. The second statement is called the escape sequence. The guard of either sequence can be either a single statement, or it can be an if , do , or lower level unless construct with multiple guards and options for execution.

The executability of all basic statements in the main sequence is constrained to the non-executability of all guard statements of the escape sequence. If and when one of the guard statements of the escape sequence becomes executable, execution proceeds with the remainder of the escape sequence and does not return to the main sequence. If all guards of the escape sequence remain unexecutable throughout the execution of the main sequence, the escape sequence as a whole is skipped.

The effect of the escape sequence is distributed to all the basic statements inside the main sequence, including those that are contained inside atomic sequences. If a d_step sequence is included, though, the escape affects only its guard statement (that is, the first statement) of the sequence, and not the remaining statements inside the d_step . A d_step is always equivalent to a single statement that can only be executed in its entirety from start to finish.

...

...

As noted, the guard statement of an unless construct can itself be a selection or a repetition construct, allowing for a non-deterministic selection of a specific executable escape. Following the semantics model, the guard statements of an escape sequence are assigned a higher priority than the basic statements from the main sequence.

Unless constructs may be nested. In that case, the guard statements from each unless statement take higher priority than those from the statements that are enclosed. This priority rule can be reversed, giving the highest priority to the most deeply nested unless escapes, by using Spin run-time option -J . This option is called -J because it enforces a priority rule that matches the evaluation order of nested catch statements in Java programs.

Promela unless statements are meant to facilitate the modeling of error handling methods in implementation level languages.

EXAMPLES
Consider the following unless statement:

```
    { B1; B2; B3 } unless { C1; C2 }
```

where the parts inside the curly braces are arbitrary Promela fragments. Execution of this unless statement begins with the execution of B1 . Before each statement execution in the sequence B1;B2;B3 , the executability of the first statement, or guard , of fragment C1 is checked using the normal Promela semantics of executability. Execution of statements from B1;B2;B3 proceeds only while the guard statement of C1 remains unexecutable. The first instant that this guard of the escape sequence is found to be executable, control changes to it, and execution continues as defined for C1;C2 . Individual statement executions remain indivisible, so control can only change from inside B1;B2;B3 to the start of C1 in between individual statement executions. If the guard of the escape sequence does not become executable during the execution of B1;B2;B3 , it is skipped when B3 terminates.

...

...

Another example of the use of unless is:

```
A;
do
:: b1 -> B1
:: b2 -> B2
...
od unless { c -> C };
D
```

The curly braces around the main or the escape sequence may be deleted if there can be no confusion about which statements belong to those sequences. In the example, condition c acts as a watchdog on the repetition construct from the main sequence. Note that this is not necessarily equivalent to the construct:

```
A;
do
:: b1 -> B1
:: b2 -> B2
...
:: c -> break
od;
C; D
```

if B1 or B2 are non-empty. In the first version of the example, execution of the iteration can be interrupted at any point inside each option sequence. In the second version, execution can only be interrupted at the start of the option sequences.

...

```
...
```

```
NOTES
In the presence of rendezvous operations, the precise effect of an unless construct can be
hard to assess. See the semantics definition for details on resolving apparent semantic
conflicts.
```

Section 9.5

Reading from standard input

NAME
STDIN - predefined read-only channel, for use in simulation.

SYNTAX
chan STDIN; STDIN?*var*

DESCRIPTION
During simulation runs, it is sometimes useful to be able to connect Spin to other programs that can produce useful input, or directly to the standard input stream to read input from the terminal or from a file.

...

...

EXAMPLES

A sample use of this feature is this model of a word count program:

```
    chan STDIN;   /* no channel initialization */

    init {
            int c, nl, nw, nc;
            bool inword = false;
            do
            :: STDIN?c ->
                    if
                    :: c == -1 ->   break   /* EOF */
                    :: c == '\en' -> nc++; nl++
                    :: else ->      nc++
                    fi;
                    if
                    :: c == ' ' || c == '\et' || c == '\en' ->
                            inword = false
                    :: else ->
                            if
                            :: !inword ->
                                    nw++; inword = true
                            :: else /* do nothing */
                    fi       fi
            od;
            printf("%d\t%d\t%d\n", nl, nw, nc)
    }
```

...

```
...
```

NOTES

The STDIN channel can be used only in simulations. The name has no special meaning in verification. A verification for the example model would report an attempt to receive data from an unitialized channel.