Principles of the Spin Model Checker
Supplementary Material on Spin Version 6

Mordechai (Moti) Ben-Ari
`http://stwww.weizmann.ac.il/g-cs/benari/`

December 20, 2010

In December 2010, Version 6 of SPIN was released; this version introduced new structures into the modeling language PROMELA. This document describes the new structures and shows how they can be used in the examples from *Principles of the Spin Model Checker*. The latest software archive `ps-programs.zip` contains these modified programs.

The new version of SPIN also made changes to the format of its output data. This required some modifications of the JSPIN development environment; the new version can be downloaded from `http://code.google.com/p/jspin/`.

The modifications of PROMELA include: a **for**-statement that replaces the use of a macro; a **select**-statement for nondeterministic selection of values; the scope rules have been changed; formulas in linear temporal logic (LTL) can be included within the PROMELA source; LTL formulas can contain expressions; LTL formulas can use keywords for the temporal operators.

Version 6 also supports multiple **never**-claims and synchronous products of claims but this will not be discussed here.

The notation §n.m refers to **Listing n.m** in *Principles of the Spin Model Checker*.

1 The for-statement

PROMELA uses the nondeterministic guarded command **do** to support loops. When you want a simple counting loop, the following code can used (§1.8):

```
byte i;
i = 1;
do
:: i > 10 -> break
:: else ->
    /* body of the loop */
    i++
od;
```

This code is frequently encapsulated in a pair of macros: `for` and `rof`.

A new control structure implements a counting loop directly:

```
byte i;
for (i : 1..10) {
  /* body of the loop */
}
```

This is equivalent to the following code which is different from that used in §1.8:

```
byte i;
i = 1;
do
:: i<= 10 ->
    /* body of the loop */
    i++
:: else -> break
od;
```

The bounds can be expressions:

```
for (i : 0..N-1)
```

PROMELA differs from many programming languages in that the index variable must be explicitly declared prior to the loop.

Many of the examples (§1.8, §3.5, §6.2, §11.14) were changed to use the **for**-statement.

2 The select-statement

To choose a value nondeterministically, you can use an **if**-statement with true (absent) guards. For example, to choose the next row in the 8-queens problem, we used (encapsulated within an **inline** construct Choose in §11.5):

```
if
:: row = 1
:: row = 2
   ...
:: row = 8
fi
```

This can now be succinctly written as:

```
select(row : 1..8)
```

The bounds can be expressions and the variable must be previously defined.

Since **true** and **false** are just names for the constants 1 and 0, the following is correct:

```
bool b;
select(b : false..true)
```

as a replacement for:

```
bool b;
if :: b = false :: b = true fi
```

The implementation of **select** uses a nondeterministic loop (Section 4.6.2):

```
row = 1;
do
:: row < 8 -> row++
:: break
od
```

3 Scope rules

Previously, **inline** definitions did not create a new scope for variable declarations, although the syntax is implies that it does. Given the definition:

```
inline write(n) {
 byte nsq;
 nsq = n*n;
 printf("n = %d, n squared = %d\n", n, nsq)
}
```

The following is illegal:

```
active proctype P() {
 byte a = 10, b = 12;
 write(a);
 write(b)
}
```

because the variable `nsq` is redeclared by `write(b)` within the scope of the **proctype**. SPIN, however, runs the program as expected, although an error message is given. Now, the program runs without the error message because the scope of `nsq` is limited to the **inline** constructs.

Warning

Do not write **byte** `nsq = n*n;` in the **inline** definition!
The scope rule refers only to the *name* itself; all *variables* within a **proctype** are still collected and placed at the beginning of the **proctype** and initialized *once* when the **proctype** is activated.

Note: §6.3 was and is illegal because an array name cannot be passed to an **inline** definition, although this is not expressly forbidden in the PROMELA reference manual.

4 LTL formulas

Consider an algorithm for solving the critical-section problem:

```
bool csp = false, csq = false;

active proctype P() {
 do
 ::
   ...
   csp = true; /* Enter critical section */
   csp = false; /* Leave critical section */
   ...
 od
}

active proctype Q() { /* Similar */ }
```

To verify mutual exclusion we have to show that the following LTL formula holds:

```
[]!(csp && csq)
```

while to verify absence of starvation the formula is:

```
[]<>csp
```

In SPIN this is done by: (a) writing the LTL formula in a file (or an argument); (b) negating the formula; (c) translating it into a **never**-claim; (d) running a verification with the **never**-claim. In previous versions of SPIN, the user had to negate the formula and then run a command to translate it into a **never**-claim; the **never**-claim was then included as an argument to the verification.[1] Currently, the LTL formula can be written *within* the PROMELA program:

```
bool csp = false, csq = false;

ltl { []!(csp && csq) }
```

SPIN will *automatically* negate the formula and translate it into a **never**-claim when the verification is performed.

To simplify matters even further, several *named* LTL formulas can be included and the choice of the formula to use is made when the verification is done:

```
ltl mutex { []!(csp && csq) }
ltl nostarvation { []<>csp }
```

[1] An environment can simplify the process. For example, JSPIN automatically negates the LTL formula, translates it into a **never**-claim when a button is clicked and then includes the claim in the arguments to the verification.

When there is only one LTL formula in a program, just perform the verification (`spin -a`, `gcc`, `pan`) and the formula will be automatically used by SPIN. When there is more than one (named) LTL formula, you must specify which formula is to be used in a verification.

---

jSpin

Enter one *name* in the text field labeled `LTL formula` and select `LTL name` from the toolbar or the `Convert` menu.

Command line

Specify one *name* when the verification is done:

```
pan -N mutex
```

The inclusion of multiple LTL formulas within a PROMELA program simplifies the configuration management of a project because all the correctness properties are contained within the same file as the source code of the model.

Additional new features concerning LTL formulas:

- Expressions can be used in an internal LTL formula, so it is no longer necessary to define symbols for this purpose: `ltl { [](critical <= 1) }`.
- A remote reference is considered an expression, so `ltl { []!(P@cs && Q@cs) }` can be given as the correctness specification of §5.3 without defining the symbol `mutex`. The formula must appear *after* the **proctype**'s where the labels are defined.
- Temporal operators can be given as keywords:

```
ltl mutex { always !(csp && csq) }
ltl nostarvation { always eventually csp }
```