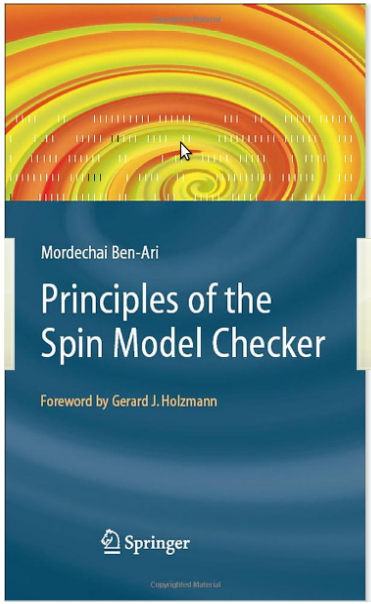
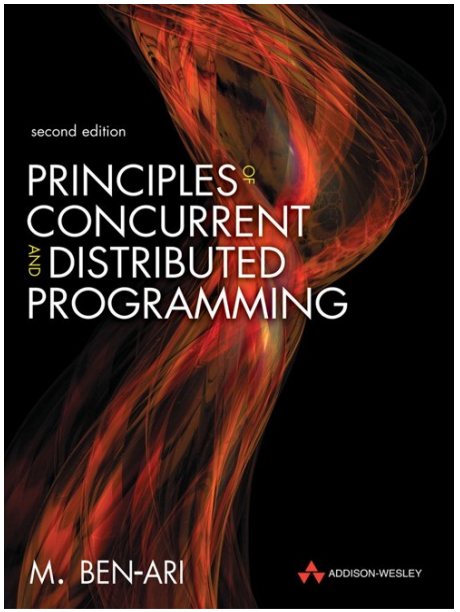


Mordechai Ben-Ari

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 6

Data and Program Structures

To make programs more readable:

- arrays
- type definitions
- macros
- inline declarations

Section 6.1

Arrays

In common with almost all programming languages, Promela includes the *array* – a sequence of data values of the same type whose elements can be accessed by providing an index giving the position of the element within the sequence.

The syntax and semantics of arrays are similar to those of C-like languages; the first position in the array is at index zero and square brackets are used for the indexing operation.

Arrays in Promela are one-dimensional.

Listing 6.1. Computing the sum of the elements of an array of integer type (`listing_6_1.pml`)

```
1
2  active proctype P() {
3    int a[5] = { 0, 10, 20, 30, 40 }
4    int sum = 0, i = 0
5
6    for (i in a) {
7      sum = sum + a[i]
8    }
9    printf("The sum of the numbers = %d\n", sum)
10 }
```

The elements of an array can be initialized by a computation from within a loop:

```
for (j in a) {
  a[j] = j * 10
}
```

perhaps by a nondeterministic expression:

```
for (j in a) {
  if
  :: a[j] = j * 10
  :: a[j] = j + 5
  fi
}
```

An initial value in a declaration is assigned to all the elements of the array:

```
int a[5] = 10
```


Section 6.2

Type definitions

Compound types are defined with **typedef** and are primarily used for defining the structure of messages to be sent over channels:

```
typedef MESSAGE {
  mtype message
  byte source
  byte destination
  bool urgent
}
```

An additional use of the type definitions is to work around the Promela limitation to one-dimensional arrays. A two-dimensional array is declared as an array whose elements are of a type defined by a type definition with a single array field:

```
typedef VECTOR {
  int vector[10]
}
VECTOR matrix[5]
... matrix[3].vector[6] = matrix[4].vector[7]
```


Listing 6.2. Data structure for a sparse array (`sparse_v6.pml`)

```
 3  #define N 4
 4
 5  typedef ENTRY {
 6    byte row
 7    byte col
 8    int value
 9  }
10
11  ENTRY a[N]
12
13  active proctype P() {
14    int i = 0, r = 0, c = 0                          /*      row   col   value  */
15
16    a[0].row = 0; a[0].col = 1; a[0].value = -5      /* 0:    0     1     -5    */
17    a[1].row = 0; a[1].col = 3; a[1].value = 8       /* 1:    0     3      8    */
18    a[2].row = 2; a[2].col = 0; a[2].value = 20      /* 2:    2     0     20    */
19    a[3].row = 3; a[3].col = 3; a[3].value = -3      /* 3:    3     3     -3    */
20
21    for (r in a) {
22      for (c in a) {
23        if
24        :: i == N -> printf("0 ")
25        :: i < N && r == a[i].row && c == a[i].col ->
26             printf("%d ", a[i].value)
27             i++
28        :: else -> printf("0 ")
29        fi
30      }
31      printf("\n")
32    }
33  }
```

The output of this program is:

```
0       -5       0       8
0       0       0       0
20       0       0       0
0       0       0       -3
```

Promela does not support width specifiers as does C.

Section 6.3

The preprocessor

Inclusion of source code is implemented by a compile-time software tool called the *preprocessor*, which is called before the compiler itself is executed.

The preprocessor is also used to conduct text-based macro processing on the source code. *Text-based* means that the preprocessor has no knowledge of the syntax and semantics of the lenguage, but instead treats the source code as pure text.

We have already seen the use of the preprocessor to include a file:

```
#include "semaphore.h"
```

and to declare a symbol:

```
#define N 4
```


Declaring a symbol does not use memory because the definition is simply substituted for the symbol before Spin simulates the program or generates the code for the verifier.

#define is also used for declaring symbols for expressions used in correctness specifications:

```
#define mutex (critical <=1)
```

Subsection 6.3.1

Condition compilation

The preprocessor can be used to implement *conditional compilation*, which enables the compile-time parameterization of a program. Suppose that a model is to be verified under several different assumptions, for example, under three different priority schemes. The following preprocessor code enables different expressions to be used for the variable `currentPriority` simply by defining one of the symbols `VerOne`, `VerTwo`, or `VerThree` with **`#define`** at the beginning of the program:

```
#define VerThree
...
#ifdef VerOne
  CurrentPriority = (p1 > p2 -> p1 : p2);
#endif
#ifdef VerTwo
  CurrentPriority = PMAX;
#endif
#ifdef VerThree
  CurrentPriority = PMIN;
#endif
```


Symbols can also be defined using the **`-D`** argument on the Spin command; so, for example,

```
spin -DVerTwo pri.pml
```

would run Spin on this program with the symbol **`VerTwo`** defined and thus `currentPriority` set to `PMAX`.

NAME

Macros and include files – preprocessing support

SYNTAX

```
#define name token-string
#define name (arg, ..., arg)  token-string
#ifdef name
#ifndef name
#if constant-expression
#else
#endif
#undef name
#include "filename"
```

Subsection 6.3.2

Macros

`#define` is not limited to simple textual substitution of a string for a single symbol; it can be used to create macros that can improve the readability of Promela programs. For example:

```
#define Find1(m,t,id) \
    do :: i < MAXTIME-t -> \
        if \
        :: atomic { !m[i] -> m[i] = id; i++; break }\
        :: else -> i++ \
        fi \
    od
```

The backslash character denotes that the substitution text for the macro is continued on the next line. Alternatively, you can define the entire macro on one line.

Advanced: Debugging and changing the preprocessor

To debug macros, run Spin with the argument **`-I`** (*show result of inlining and preprocessing*); this will write to standard output the results of performing the preprocessing.

You can use an alternate preprocessor by calling Spin with the argument **`-P`** (**`-Pxxx`** - *use xxx for preprocessing*). This argument is also useful for giving the full path of the preprocessor if Spin cannot find it.

Section 6.4

Inline

Although Promela does not have functions or procedures for structuring code, it can be convenient to group statements together so that they can appear in several places in a program.

This is done using the **inline** construct, which gives a name to a sequence of statements.

Listing 6.3. Printing an array with **inline** (listing_6_3.pml)

```
 1  #define N 5
 2
 3  inline write(ar) {
 4      d_step {
 5          for (i in ar) {
 6              printf("%d ", ar[i])
 7          }
 8          printf("\n")
 9      }
10  }
11
12  active proctype P() {
13    int a[N] = 0, i
14
15    write(a)
16    for (i in a) {
17        a[i] = i
18    }
19    write(a)
20  }
```



```
$ spin listing_6_3.pml
spin: listing_6_3.pml:15, Error: missing array index for 'a' saw
'')' = 41'
spin: listing_6_3.pml:19, Error: missing array index for 'a' saw
'')' = 41'
      0       0       0       0       0
      0       1       2       3       4
1 process created
```

Is the variable a scalar or array? Check it with an option -I.

In this case, it's just warning message.

No new scope is created for an inline sequence!

Any variables declared within the sequence are equivalent to local variables declared directly within the **proctype** where the sequence is called.


```
$ spin -I listing_6_3.pml
spin: listing_6_3.pml:15, Error: missing array index for 'a'   saw '')' = 41'
spin: listing_6_3.pml:19, Error: missing array index for 'a'   saw '')' = 41'
proctype P()
{
    {
      d_step {
         i = 0;
         do
         ::
            ((i<=4));
            printf('%d ',a[i]);
            i = (i+1);

         ::
            else;
            goto :b0;

         od;
:b0:
         printf('\\n');
       };
    };
...
```

```
...
   i = 0;
   do
   ::
      ((i<=4));
      a[i] = i;
      i = (i+1);

   ::
      else;
      goto :b1;

   od;
:b1:
    {
      d_step {
         i = 0;
         do
         ::
            ((i<=4));
            printf('%d ',a[i]);
            i = (i+1);

         ::
...
```



```
...

            else;
            goto :b2;

         od;
:b2:
         printf('\\n');
      };
   };
}
```


`inline` is useful for initializing data structures. The readability of the program for sparse arrays in Listing 6.2 can be improved by declaring an **`inline`** for the initialization of the entries:

```
inline initEntry(I, R, C, V) {
  a[I].row = R
  a[I].col = C
  a[I].value = V
}
```

The statements for initializing the values of the array are now much easier to write and read:

```
initEntry(0, 0, 1, -5)
initEntry(1, 0, 3, 8)
initEntry(2, 2, 0, 20)
initEntry(3, 3, 3, -3)
```


Advanced: inline vs. macros

The **`inline`** construct in Spin is almost identical to the macro construct, but its syntax is more "friendly" because there is no need to use continuation characters.

In addition, errors will be reported with the line number within the **`inline`** construct, rather than with the line of the call.