1

Sequential Programming in PROMELA

SPIN is a *model checker* – a software tool for verifying models of physical systems, 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 specified; 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. Model checking is challenging and fascinating because one must write a model that describes the system in sufficient detail to represent it faithfully, and yet the model must be sufficiently simple so that the model checker can perform the verification with the available resources (time and memory).

Our goal is to learn how to perform model checking in SPIN. We start with the first stage: learning the PROMELA language that is used for writing models in SPIN. PROMELA is, in effect, a simple programming language, so we will show how to use PROMELA to write sequential programs, and then gradually introduce the constructs used for performing verification and for writing models of real systems.

1.1 A first program in PROMELA

Assignment statements and expressions in PROMELA are written using the syntax of C-like languages. Listing 1.1 is a trivial program that reverses the digits of a three-digit number. Programs in PROMELA are composed of a set of *processes*; here we start with a single process declared by the words **active proctype**. Processes may have parameters, though we shall not use them until much later; even if there are no parameters, the parentheses ()

must appear. The statements of the process are written between the braces { and }. Comments are enclosed between /* and */.

Listing 1.1. Reversing digits

```
1 active proctype P() {
2   int value = 123; /* Try with a byte variable here ... */
3   int reversed;    /* ... and here! */
4   reversed =
5     (value % 10) * 100 +
6     ((value / 10) % 10) * 10 +
7     (value / 100);
8   printf("value = %d, reversed = %d\n", value, reversed)
9 }
```

In the program we declared two variables, value and reversed, of type **int**, the first of which is given an explicit initial value. The value assigned to the variable reversed is computed from the value of the variable value using the division and modulo operators; then, the values of both variables are printed. The **printf** statement is taken from the C language: a quoted string followed by a list of variables; the list of variables should match the *format specifiers* embedded within the string. The specifier for printing integer values is %d, and the string can be terminated by the newline character \n.

If you run a random simulation of this program as described in the next section, SPIN will print:

```
value = 123, reversed = 321
```

Advanced: Single-line comments

The single line comment from // until the end of the line is not normally available in SPIN unless you use a different preprocessor; this is explained in the *man* page for `macros`.

1.2 Random simulation

In simulation mode, SPIN compiles and executes a PROMELA program. Here we discuss random simulation mode; the meaning of "random" in this context will be apparent later.

By convention, we use the extension `pml` for PROMELA files.

To run a random simulation:

jSpin

Select `Open (ctrl-O)` to load a file into the editor window, or `File / New` to create a new file. Edit the program and save the file (`File / Save (ctrl-S)`). A message that the file has been saved appears in the message pane at the bottom of the frame. It is not necessary to explicitly save files as this is automatically done before executing SPIN.

Select `Random (alt-R)`. SPIN will compile and execute the program, and the compile-time errors or the output from the execution will appear in the right pane.

You may wish to perform a syntax check before running the program: select `Check (alt-K)`.

By default, jSPIN displays the state of the program after executing each instruction; for running the first simple programs in this book turn this output off by selecting `Options / Common / Clear all / OK` and then `Options/Save` to save the changes.

Command line

Run:

```
spin filename
```

Output will be printed on standard output and can be redirected or piped.

You can set a limit on the number of steps that will be executed in a simulation run; this will be especially important when we discuss concurrent programs that need not terminate.

jSpin

Select `Settings / Max steps (ctrl-M)` and enter a value.

Command line

Run SPIN with the parameter `-uN`, where `N` is the maximum number of steps.

Advanced: Filtering output in jSpin

JSPIN can be used to filter the output so that only the results of certain **printf** statements are displayed. Change the configuration file option MSC to true. Then only lines beginning with the string "MSC:" will be displayed. This prefix is also used to display process interactions in the *Message Sequence Charts* of XSPIN; see Chapter 12 of *SMC*.

Advanced: Input in Promela

By convention, a PROMELA program does not have input, since it is intended for simulating a *closed* system. That is, if there is a some unit in the environment that could influence the system, it should be modeled as a process. Nevertheless, there is an input channel STDIN connected to standard input that can be useful for running simulations of a single model with different parameters; see the *man* pages.

1.3 Data types

The numeric data types of PROMELA are based upon those of the C compiler used to compile SPIN itself; they are currently as shown in Table 1.1.[1] All effort should be made to model data using types that need as few bits as possible to avoid combinatorial explosion in the number of states during a verification: **short** instead of **int** and **byte** instead of **short**.

Table 1.1. Numeric data types in PROMELA

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

The type **bool** and the values **true** and **false** are syntactic sugar for the type **bit** and the values 1 and 0, respectively.[2] Values of type **bit** and **bool** can be printed only as integer values with specifier %d.

[1] The other data types in PROMELA are: **chan** (Chapter 7), **pid** (Section 3.5), and **mtype** (Section 1.4.2).

[2] *Syntactic sugar* is a term for alternate syntactic constructs that add no additional capabilities to a programming language, but instead are intended to enable more readable programs to be written. It is easier to read a program that contains **bool** done = **false** than **bit** done = 0, although there is no semantic difference whatsoever between them.

Warning

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

PROMELA *does not* have some familiar data types:

- 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`.[3]

Advanced: 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.

Advanced: Unsigned integer type

The type **unsigned** can be used for variables intended to hold unsigned values of a specified number of bits. It is meaningful when compression of the state vector is used. See *SMC* Chapter 3 and the *man* page for `datatypes`.

[3] If floating-point numbers are truly needed you can use them in embedded segments of C code. SPIN can verify a model with embedded C code on the assumption that this code is correct. See Chapter 17 of *SMC*.

1.3.1 Type conversions

There are no explicit type conversions in PROMELA. Arithmetic is *always* performed by first implicitly converting all values to **int**; upon assignment, the value is implicitly converted to the type of the variable. In our first program, if the variable value is declared to be of type **byte**, the program is still correct because the computation is performed on integers and then assigned to the variable reversed of type **int**. If, however, reversed is declared to be of type **byte**, the attempt to assign 321 to that variable will not succeed; the value will be truncated and an error message printed:

```
Error: value (321->65 (8)) truncated in assignment
value = 123, reversed = 65
```

You may be surprised that the error does not cause an exception or the termination of the program and that the truncated value is printed. SPIN leaves it up to you to decide if the truncated value is meaningful or not.

1.4 Operators and expressions

The set of operators in PROMELA, together with their precedence and associativity, is shown in Table 1.2; the operators are almost identical to those in C-like languages. Needless to say, it is not a good idea to try to memorize the table, but rather to use parentheses liberally to clarify precedence and associativity within expressions!

The following rule is central to the design of PROMELA:

> **Warning**
>
> Expressions in PROMELA must be side-effect free.

The reason for the rule is that expressions are used to determine if a statement is executable or not (Section 4.2), so it must be possible to evaluate an expression without side effects.

This requirement leads to several differences between PROMELA and the C language; in PROMELA

- Assignment statements are not expressions.
- The increment and decrement operators (++, --) may only be used as *postfix* operators in an assignment statement like:

```
b++
```


Table 1.2. 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

and not in an expression like the right-hand side of an assignment statement:

```
a = b++
```

- There are no *prefix* increment and decrement operators. (Even if there were, there could be no difference between them and the postfix operators because they cannot be used in expressions.)

1.4.1 Local variables

The scope of a local variable is the entire process in which it is declared.

> **Warning**
>
> It is not necessary to declare variables at the beginning of a process; however, all variable declarations are implicitly moved to the beginning of the process.

This can have weird effects if you are used to the style in the JAVA language of declaring and initializing variables in the middle of a computation; for example:

```
byte a = 1;
. . .
a = 5;
byte b = a+2;
printf("b= %d\n", b);
```

The output is 3. The variable b is implicitly declared *immediately after* the declaration of a; therefore, the expression a+2 uses the initial value 1.

1.4.2 Symbolic names*

If you just need to declare a symbol for a number, a preprocessor macro can be used at the beginning of the program:

```
#define N 10
```

Textual substitution is used when the symbol is encountered:

```
i = j % N;
```

The type **mtype** can be used to give mnemonic names to values (Listing 1.2).[4] The advantage of using **mtype** over a sequence of **#define**'s is that the symbolic values can be printed using the %e format specifier, and they will appear in traces of programs.

Internally, the values of the **mtype** are represented as *positive* byte values, so there can be at most 255 values of the type.

A limitation on **mtype** is that there is only one set of names defined for an entire program; if you add declarations, the new symbols are added to the existing set. Listing 1.3 shows how to add states for traffic signals in which two lights are on simultaneously (as is done in many countries).

Advanced: Printing values of mtype

The **printm** statement can be used to print a value of an **mtype**. See the *man* pages for **mtype** and **printf** for an explanation of when to use this instead of the format specifier %e.

[4] The term is short for *message type* because its original use was to give symbolic names instead of numbers to messages.

Listing 1.2. Symbolic names

```
1  mtype = { red, yellow, green };
2  mtype light = green;
3
4  active proctype P() {
5   do
6   :: if
7      :: light == red -> light = green
8      :: light == yellow -> light = red
9      :: light == green -> light = yellow
10     fi;
11     printf("The light is now %e\n", light)
12  od
13 }
```

Listing 1.3. Adding new symbolic names

```
1  mtype = { red, yellow, green };
2  mtype = { green_and_yellow, yellow_and_red };
3  mtype light = green;
4
5  active proctype P() {
6   do
7   :: if
8      :: light == red -> light = yellow_and_red
9      :: light == yellow_and_red -> light = green
10     :: light == green -> light = green_and_yellow
11     :: light == green_and_yellow -> light = red
12     fi;
13     printf("The light is now %e\n", light)
14  od
15 }
```

1.5 Control statements

While the syntax and semantics of expressions in PROMELA are taken from C-like languages, the control statements are taken from a formalism called *guarded commands* invented by E.W. Dijkstra. This formalism is particularly well suited for expressing nondeterminism and thus is a good match for modeling systems like communication systems that are by nature nondeterministic. We will treat nondeterminism at length in Chapter 8.

There are five control statements: sequence, selection, repetition, jump, and **unless**; the first four are presented here, while **unless** is described in Section 9.4.

The semicolon is the *separator* between statements that are executed in sequence. Semicolons are used as separators in the Pascal language, but most readers are probably more familiar with their use as terminators of statements in C-like languages. Fewer semicolons are needed when they are used as separators, so the code looks cleaner. Don't worry if you use an unnecessary semicolon, as SPIN will rarely complain.

Terminology: When a processor executes a program, a register called a *location counter* maintains the address of the next instruction that can be executed.[5] 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.

1.6 Selection statements

The classic **if**-statement is based upon sequential checking of expressions until one evaluates to true, at which point the associated sequence of statements is executed. The sequential nature can be seen from the use of the keyword **else** in languages like JAVA or C:

[5] Other terms for *location counter* are *program counter (pc)* and *instruction pointer (ip)*.


```
if (expression-1) {
   statement-1-1; statement-1-2; statement-1-3;
}
else if (expression-2) {
   statement-2-1;
}
else {
   statement-3-1; statement-3-2;
}
```

In SPIN there is no semantic meaning to the order of the alternatives; the semantics of the statement merely says that if the expression of an alternative is true, the sequence of statements that follows it *can be* executed. The program in Listing 1.4 contains an **if**-statement that checks the discriminant of a quadratic equation; the three expressions are mutually exclusive and exhaustive, so that exactly one of them will be true whenever the statement is executed. The effect of such a statement is the same as that of a familiar **if**-statement.

Listing 1.4. Discriminant of a quadratic equation

```
1  active proctype P() {
2    int a = 1, b = -4, c = 4;
3    int disc;
4    disc = b * b - 4 * a * c;
5    if
6    :: disc < 0 ->
7       printf("disc = %d: no real roots\n", disc)
8    :: disc == 0 ->
9       printf("disc = %d: duplicate real roots\n", disc)
10   :: disc > 0 ->
11      printf("disc = %d: two real roots\n", disc)
12   fi
13 }
```

An **if**-statement starts with the reserved word **if** and ends with the reserved word **fi** (**if** spelled backward). In between are one or more *alternatives*, each consisting of a double colon, a statement called a *guard*, an arrow,

and a sequence of statements.[6] (Note that no semicolon is required before a double colon or the **fi** because the semicolon is a separator, not a terminator.)

The execution of an **if**-statement begins with the evaluation of the guards; if at least one evaluates to true, the sequence of statements following the arrow corresponding to one of the true guards is executed. When those statements have been executed, the **if**-statement terminates.

Listing 1.5 shows a program for computing the number of days in a month. Compound boolean expressions are used for each guard. The example also shows the **else** guard whose meaning is: if and only if *all* the other guards evaluate to false, the statements following the **else** will be executed.

Listing 1.5. Number of days in a month

```
1  active proctype P() {
2    byte days;
3    byte month = 2;
4    int year = 2000;
5    if
6    :: month == 1 || month == 3 || month == 5 ||
7       month == 7 || month == 8 || month == 10 ||
8       month == 12 ->
9         days = 31
10   :: month == 4 || month == 6 || month == 9 ||
11      month == 11 ->
12        days = 30
13   :: month == 2 && year % 4 == 0 && /* Leap year */
14      (year % 100 != 0 || year % 400 == 0) ->
15        days = 29
16   :: else ->
17        days = 28
18   fi;
19   printf("month = %d, year = %d, days = %d\n",
20          month, year, days)
21 }
```

[6] In *SMC* an alternative is called an *option sequence*, but I prefer the former term.

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.

The next example shows how nondeterminism works. When computing the maximum of two values, it does not matter which is chosen if the two values are equal (Listing 1.6). If two or more guards evaluate to true, the statements associated with either may be executed. In the example, we have used an additional variable, branch, to record which alternative is taken. Run the program a few times and you will see that both values can be printed. This demonstrates the concept of random simulation: Whenever a nondeterministic choice exists, SPIN randomly chooses one of them.

Finally, it is possible that all alternatives could be false. In that case the process blocks until some guard evaluates to true, which can only happen in a concurrent program with more than one process (Chapter 3).

The sequence of statements following a guard can be empty, in which case control leaves the **if**-statement after evaluating the guard. If you are bothered by the empty sequence, you can use **skip**, which is syntactic sugar for a statement (actually, an expression) that always evaluates to true like **true** or (1).

Listing 1.6. Maximum of two values

```
1  active proctype P() {
2    int a = 5, b = 5;
3    int max;
4    int branch;
5    if
6    :: a >= b -> max = a; branch = 1
7    :: b >= a -> max = b; branch = 2
8    fi;
9    printf("The maximum of %d and %d = %d by branch %d\n",
10     a, b, max, branch)
11 }
```

Advanced: Arrows as separators

The arrow symbol is syntactic sugar for a semicolon. Guards are simply PROMELA statements with no special syntax, so, for example, the **if**-statement in Listing 1.4 is equivalent to one written as:

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

The arrow syntax is preferred because it emphasizes the role of the guard in deciding which alternative to execute.

1.6.1 Conditional expressions*

A conditional expression enables you to obtain a value that depends on the result of evaluating a boolean expression:[7]

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

The variable max is assigned the value of a if a > b; otherwise, it is assigned the value of b. The syntax is different from that in C-like languages: An arrow is used instead of a question mark to separate the condition from the two expressions.

> **Warning**
>
> A conditional expression *must* be contained within parentheses. It is a syntax error to write:
>
> ```
> max = a > b -> a : b
> ```

In Listing 1.5, we could slightly simplify the computation of the number of days in February by using a conditional expression:

```
:: month == 2 && year % 4 == 0 ->
   days = (year % 100 != 0 || year % 400 == 0 ->
             29 : 28)
```

Note the difference between the two arrows: the first arrow separates the guard from the assignment statement, while the second arrow is used in the conditional expression.

The semantics of conditional expressions is different from that of **if**-statements. An assignment statement like

[7] In the *man* pages a conditional expression is called cond_expr.


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

is an atomic statement, while the the **if**-statement

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

is not, and interleaving is possible between the guard and the following assignment statement.

1.7 Repetitive statements

There is one repetitive statement in PROMELA, the **do**-statement. The program in Listing 1.7 computes the greatest common denominator (GCD) of two values of type **int** by repeated subtraction of the smaller from the larger. 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.

Listing 1.7. Greatest common denominator

```
1  active proctype P() {
2    int x = 15, y = 20;
3    int a = x, b = y;
4    do
5    :: a > b -> a = a - b
6    :: b > a -> b = b - a
7    :: a == b -> break
8    od;
9    printf("The GCD of %d and %d = %d\n", x, y, a)
10 }
```

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**.

1.7.1 Counting loops

Unfortunately, there are no counting loops in PROMELA similar to the for-statements of C-like languages. The program in Listing 1.8 shows how to implement a counting loop. The control variable `i` is declared and initialized. One alternative of the **do**-statement (line 7) checks if the value of `i` is greater than the upper limit `N`, and if so executes a **break**. Otherwise (**else**), the body of the loop is executed and the control variable incremented (lines 8–10).

Listing 1.8. A counting loop

```
1  #define N 10
2
3  active proctype P() {
4    int sum = 0;
5    byte i = 1;
6    do
7    :: i > N -> break
8    :: else ->
9          sum = sum + i;
10         i++
11   od;
12   printf("The sum of the first %d numbers = %d\n", N, sum)
13 }
```

> **Warning**
>
> Do not forget the **else** in a counting loop! It is not an error of syntax to omit the **else** and so no error message will result. SPIN will nondeterministically choose to execute one of the alternatives whose guard evaluates to true (if any), and the results may be unexpected.

PROMELA contains a macro facility that can be used to make programs more readable (Section 6.3.2). Macros can be written to simulate the control statements of more familiar languages. Personally, I prefer to use PROMELA's guarded-commands syntax, except in the case of of counting loops. The program in Listing 1.9 shows how to implement the counting loop using a macro

to simulate a for-statement. The for macro takes three parameters: the control variable, and the lower and upper limits of the loop. The `rof` macro at the end of the loop takes the control variable as a parameter. The text of these macros is contained in a file `for.h` which must be **include**'d in the program. Be sure that this file is in the same directory as your program. (The definition of these macros is given in Section 6.3.2.)

Listing 1.9. Counting with a for-loop macro

```
1  #include "for.h"
2  #define N 10
3
4  active proctype P() {
5    int sum = 0;
6    for (i, 1, N)
7      sum = sum + i
8    rof (i);
9    printf("The sum of the first %d numbers = %d\n", N, sum)
10 }
```

> **Warning**
>
> This macro declares the loop variable `i`, but does not create a new scope as in JAVA. If you use it more than once with the same variable, you will get an error message that the variable is multiply declared, although no harm is done and you can ignore the message. Alternatively, modify the macro to remove the declaration and declare the variable yourself.

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. See Section 8.1 for a reasonable use of the **goto**-statement.

> **Warning**
>
> 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.

Here is an example showing how a label must be attached to the entire **do**-statement and not to a single alternative:

```
start:
 do
 :: wantP ->
    if
    :: wantQ -> goto start
    :: else -> skip
    fi
 :: else ->
    ...
 od
```

> **Warning**
>
> Labels in PROMELA are not used only as targets of jump statements; they are also used in correctness specifications. See Sections 4.7.2 and 10.4.

2

Verification of Sequential Programs

Although SPIN is designed for verifying models of concurrent and distributed systems, we will introduce verification within the elementary context of sequential programs. This chapter shows how to express correctness specifications using assertions and describes the procedure for carrying out a verification in SPIN.

2.1 Assertions

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

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

The *state space* of a program is the set of states that can *possibly* occur during a computation.[1] In model checking the state space of a program is generated in order to search for a counterexample – if one exists – to the correctness specifications. This section shows how to use *assertions* to express correctness specifications.

Assertions can be placed between any two statements of a program and the model checker will evaluate the assertions as part of its search of the state space. If, during the search, it finds a computation leading to a false assertion,

[1] Section 4.3 elaborates on the meaning of *possibly* in this context.

either the program is incorrect, or the assertion does not properly express a correctness property that holds for the program.

Listing 2.1 shows a program for integer division that works by repeatedly subtracting the divisor from the dividend until what remains is less than the divisor. We have added assertions to the program. The first assertion (line 6) is the *precondition*, an assertion that specifies what must be true in the initial state. Here the precondition states that the dividend is nonnegative and that the divisor is positive. The *postcondition* (lines 20–21) specifies what must be true in any final state of the program. The first line claims that the remainder is nonnegative and less than the divisor, and the second claims that the expected relation holds among the four quantities.

Listing 2.1. Integer division

```
1  active proctype P() {
2    int dividend = 15;
3    int divisor = 4;
4    int quotient, remainder;
5
6    assert (dividend >= 0 && divisor > 0);
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
20   assert (0 <= remainder && remainder < divisor);
21   assert (dividend == quotient * divisor + remainder)
22 }
```


Assertions are statements consisting of the keyword **assert** followed by an expression. When an **assert** statement is executed during a simulation, the expression is evaluated. If it is true, execution proceeds normally to the next statement; if it is false, the program terminates with an error message.

Run a random simulation for this program; it will terminate normally with no error messages, printing:

```
15 divided by 4 = 3, remainder = 3
```

Now, change the initial value of the variable `dividend` to 16 (line 2), change the guard `remainder >= divisor` to `remainder > divisor` (line 11), and rerun the simulation. The assertion in line 20 will evaluate to false and the program will terminate with the error message:

```
spin: line  20 "divide-error.pml", Error: assertion violated
spin: text of failed assertion:
  assert(((0<=remainder)&&(remainder<divisor)))
```

The error message identifies the assertion that evaluated to false. Using the data displayed in JSPIN or running SPIN from the command line with the argument `-l` will quickly show that the assertion is evaluated when both `remainder` and `divisor` equal 4, so that `remainder < divisor` is false.

Listing 2.2 shows another program for integer division. In addition to the precondition and postcondition, we have added an assertion within the loop (lines 11–12); this assertion is evaluated each time the first alternative of the **do**-statement (lines 9–21) is executed. An assertion within a loop is called an *invariant* of the loop because it must remain true as long as the loop body continues to be executed.

Run a verification (as described in the next section) and show that the program is correct.

Advanced: Preconditions in Promela models

Preconditions have little meaning when models are verified in SPIN. They are intended to specify conditions on the input to a program, but PROMELA models rarely, if ever, have input. Instead, initial values are given that trivially satisfy the precondition, as shown in the program in Listing 2.1. In a real system any input to the system will be received in a register or memory cell of finite size, so it can modeled by PROMELA statements that nondeterministically choose values from a limited range (Section 4.6). Postconditions are more meaningful – for models that are intended to terminate – because there may be many different computations that can terminate, and it makes sense to specify properties of final states.

Listing 2.2. Another program for integer division

```
1  active proctype P() {
2    int dividend = 15, divisor = 4;
3    int quotient = 0, remainder = 0;
4    int n = dividend;
5
6    assert (dividend >= 0 && divisor > 0);
7
8    do
9    :: n != 0 ->
10
11     assert (dividend == quotient * divisor + remainder + n);
12     assert (0 <= remainder && remainder < divisor);
13
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;
25   printf("%d divided by %d = %d, remainder = %d\n",
26        dividend, divisor, quotient, remainder);
27
28   assert (0 <= remainder && remainder < divisor);
29   assert (dividend == quotient * divisor + remainder)
30 }
```


Advanced: Deductive verification

An alternative approach to verification is *deduction*. A formal semantics is defined for program constructs and then a formal logic with axioms and inference rules is used to deduce that a program satisfies correctness specifications, expressed, for example, as assertions. The advantage of deductive verification is that it is not limited by the size of the state space because the deduction is done on symbolic formulas; the disadvantage is that it is less amenable to automation and requires mathematical ingenuity.

A deductive verification of the program in Listing 2.2 is given in Section B.4 of *PCDP*; it was partially automated using the verification capabilities of the SPARK system [3].

For an overview of deductive verification, see Chapter 9 of *MLCS*; an advanced textbook is [1].

2.2 Verifying a program in SPIN

Consider the program in Listing 2.3 that has an error in the second alternative (line 5). When a equals b a random simulation is just as likely to take the first alternative of the **if**-statement as the second. In fact, even if we run the simulation repeatedly, it is possible – although unlikely – that the same alternative will always be taken. In other words, no amount of simulation can ever verify that the postcondition is true, because it may become true if one alternative is taken, while it is falsified in the other alternative.

Listing 2.3. Maximum with an error

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

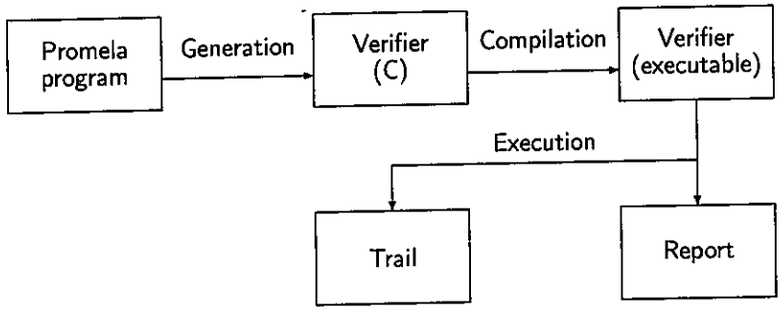
The only way to verify that a program is correct is to systematically check that the correctness specifications hold in *all possible computations*, and that is what model checkers like SPIN are designed to do.

In a deterministic program (with no input), there is only one possible computation, so a single random simulation will suffice to demonstrate the correctness of a program. For a concurrent or nondeterministic program, checking all possible computations involves executing the program and backtracking over each choice of the next statement to execute. One of the ways that SPIN achieves efficiency is by generating an optimized program called a *verifier* for each PROMELA model. Verification in SPIN is a three-step process (Figure 2.1):

- Generate the verifier 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. (The *Trail* shown in the figure is explained in the next section.)

Fortunately, there is no need to examine the C source code of the verifier; you simply perform these three steps within a script, or use JSPIN, which invokes SPIN, the C compiler and the compiled verifier.

Fig. 2.1. The architecture of SPIN



jSpin

Select `Verify`. The commands that are executed are listed in the message pane. The report of the verifier is displayed in the right pane.

Command line

Run SPIN with the argument `-a` to generate the verifier source code:

```
spin -a max.pml
```

Check your directory; you should find files `pan.*` including `pan.c`, which contains the source code of the main program. (The file name pan is historical and is derived from *p*rotocol *an*alyzer.) The next step is to compile this file; for the gcc compiler the command is:

```
gcc -o pan pan.c
```

Finally, run the verifier:

```
pan
```

You may need to enter this command as `./pan` or `.\pan`.

Verify the program in Listing 1.6 for the maximum of two numbers; you should get `errors = 0`. (For now, you can ignore the rest of the output.)

Next, verify the program in Listing 2.3 that contains an error; the report will be:

```
pan: assertion violated
   ( ((a>=b)) ? ((max==a)) : ((max==b)) ) (at depth 0)
pan: wrote max1.pml.trail
(Spin Version 4.2.8 -- 6 January 2007)
Warning: Search not completed
   ...
State-vector 24 byte, depth reached 2, errors: 1
   ...
```

SPIN does not bother to search the entire state space; instead, it stops as soon as one assertion is violated because the existence of one counterexample is usually sufficient to locate an error in the program or the correctness specifications.

Advanced: Continuing past the first error

The argument `-e` to pan causes trails for all errors to be created.
The argument `-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.

2.2.1 Guided simulation

You may hope that your first attempt at verifying a model will succeed; however, this is unrealistically optimistic! Almost invariably it takes a long time to understand the interactions among components of the model, and between the model and its correctness specifications, in order to achieve a successful verification. Thus, a primary task of a model checker is to assist the systems engineer in understanding why a verification has failed.

SPIN supports the analysis of failed verifications by maintaining internal data structures during its search of the state space; these are used to reconstruct a computation that leads to an error. The data required for reconstructing a computation are written into a file called a *trail*. (The name of the file is the same as that of the PROMELA source code file with the additional extension `.trail`.) The trail file is not intended to be read; rather, it is used to reconstruct a computation by running SPIN in *guided simulation mode*.

---

jSpin

After running a verification that has reported errors, select `Trail` .

Command line

After running a verification that has reported errors, run SPIN again with the `-t` argument:

```
spin -t max.pml
```

An examination of the guided simulation for the program in Listing 2.3 will show that the bad computation actually occurs when the alternative with the mistake (line 5) is executed:

```
Starting P with pid 0
  0:    proc  - (:root:) creates proc  0 (P)
0 P     3   b>=a
0 P     5   max = (b+1)
```

As a check of your understanding of assertions, write the postcondition for the program in Listing 1.7 that computes the greatest common denominator of two integer numbers; verify that the program is correct.

2.2.2 Displaying a computation

When examining a computation produced by a random or guided simulation, we need more than the output that results from the print statements.

We need to examine the sequence of statements executed, as well as the state of the computation – the values of the variables and the location counters – after executing each statement. SPIN can print any subset of the following data:

- The statements executed by the processes;
- The values of the global variables;
- The values of the local variables;
- Send instructions executed on a channel (see Chapter 7);
- Receive instructions executed on a channel (see Chapter 7).

To select which data to display:

---

jSpin

Select `Options / Common` and select the data you want displayed. It is simplest to check all of them (`Set all`).
The states are displayed in tabular form in the right pane with a separate entry for each state. An entry contains the statement executed, the process it came from, and the values of the variables in the program.
JSPIN has many options for customizing the display of data during a simulation. You can interactively choose to exclude some variables or statements from the display by entering their identifiers in the text areas that pop up after selecting `Output / Exclude variables` or `Output / Exclude statements`.
The width of the fields used for displaying the values of the variables can be specified by selecting `Output / Variable width`.
To maximize the right pane, select `Output / Maximize` (`alt-M`).

Command line

The following arguments cause SPIN to display the data described above during a simulation: `-p` (statements), `-g` (globals), `-l` (locals), `-s` (send), `-r` (receive).
Each item of data is displayed on a separate line. You can redirect the output into a file and then examine the data using an editor.

3

Concurrency

Many computers are used in *embedded systems*, which are composed of hardware, software, sensors, controllers and displays, and which are intended to be run indefinitely and without continuous supervision. One only has to think of airplanes, medical monitors, and cell phone networks to appreciate the complexity of embedded systems. Invariably, these systems contain several *processes* that must sample sensors and perform computations at roughly the same time; for example, a medical monitor samples heart rate, blood pressure, and temperature, and determines if they are within a predetermined range. Programming for multiprocess systems is called *concurrent programming*.

Frequently, embedded systems contain several *processors*; microprocessors have become so inexpensive that it is feasible to devote a separate processor to each subsystem. Furthermore, many systems like cell phone networks are by nature geographically dispersed, relying on communications networks for passing data between processors. Programming for such systems is called *distributed programming*.

SPIN supports modeling of both concurrent and distributed systems. This chapter is the first of several on writing and verifying concurrent programs; in Chapter 7 we discuss the use of channels to model distributed systems.

3.1 Interleaving

Consider the program in Listing 3.1 with two processes, `P` and `Q`. A computation of the program can be displayed in a table with one line for each state that forms the computation. The top line is the initial state. The entry for a state shows the values of the variables and the statement that will be

Listing 3.1. Interleaving statements

```
1  byte  n = 0;
2
3  active proctype P() {
4    n = 1;
5    printf("Process P, n = %d\n", n)
6  }
7
8  active proctype Q() {
9    n = 2;
10   printf("Process Q, n = %d\n", n)
11 }
```

executed next, together with the process in which the statement is declared. Here is the table for one computation of the program in Listing 3.1:[1]

Process	Statement	n	Output
P	n = 1	0	
P	**printf**(P)	1	
Q	n = 2	1	P, n = 1
Q	**printf**(Q)	2	
			Q, n = 2

In the initial state, the first statement executed is n = 1 from process P and that leads to the next state in which the value of n is 1; process P prints this value. Then process Q assigns the value 2 to n and prints the value. The program terminates in a state with the value 2 in the global variable n.

However, this is not the only possible computation of the program. There are six possible computations of the program:

1	2	3	4	5	6
n = 1	n = 1	n = 1	n = 2	n = 2	n = 2
printf(P)	n = 2	n = 2	**printf**(Q)	n = 1	n = 1
n = 2	**printf**(P)	**printf**(Q)	n = 1	**printf**(Q)	**printf**(P)
printf(Q)	**printf**(Q)	**printf**(P)	**printf**(P)	**printf**(P)	**printf**(Q)

[1] To save space, the **printf** statements have been abbreviated.

These computations correspond to the following outputs:

```
Process P, n = 1    Process P, n = 2    Process Q, n = 2
Process Q, n = 2    Process Q, n = 2    Process P, n = 1

Process Q, n = 2    Process Q, n = 1    Process P, n = 1
Process P, n = 1    Process P, n = 1    Process Q, n = 1
```

We say that the computations of a program are obtained by *arbitrarily interleaving* of the statements of the processes. If each process p_i were run by itself, a computation of the process would be a sequence of states $(s_i^0, s_i^1, s_i^2, \ldots)$, where state s_i^{j+1} follows state s_i^j if and only if it is obtained by executing the statement at the location counter of p_i in s_i^j.

Consider now a computation obtained by running all processes concurrently. It is a sequence of states $(s^0, s^1, s^2, \ldots)$, where s^{j+1} follows state s^j if and only if it is obtained by executing the statement at the location counter of *some* process in s^j. The word "interleaving" is intended to represent this image of "selecting" a statement from the possible computations of the individual processes and "merging" then into a computation of all the processes of the system.

For the program in Listing 3.1, a state is a triple consisting of the value of n and the location counters of processes P and Q. The computation obtained by executing the processes by themselves can be represented as

```
(0, 4, -) -> (1, 5, -) -> (1, 6, -)
```

for process P, and

```
(0, -, 9) -> (2, -, 10) -> (2, -, 11)
```

for process Q. The third computation above is obtained by interleaving the two separate computations:

```
(0, 4, 9)   ->   "select" from P
(1, 5, 9)   ->   "select" from Q
(2, 5, 10)  ->   "select" from Q
(2, 5, 11)  ->   "select" from P
(2, 6, 11)
```

3.1.1 Displaying a computation

When SPIN simulates a program it creates *one* computation by interleaving the statements of all the processes. SPIN writes a description of the computation on standard output; JSPIN formats this description to make it easier to understand.

jSpin

The output of a simulation is displayed in the right pane; here is the output for computation 4 from page 30:

```
Process Q, n = 2
Process P, n = 1
2 processes created
```

A tabular format of the states of the computation can be displayed. Select `Options / Common / Set all / OK` (see Section 2.2.2). Here is the table for computation 1:

```
0 P     4   n = 1
Process Statement          n
0 P     5   printf('Proces 1
1 Q     9   n = 2          1
1 Q     10  printf('Proces 2
```

The first columns contain the process ID (number) and name; this is followed by the line number and source code of the statement executed (truncated if necessary), and then the values of the variables.

Command line

When SPIN writes the output of a concurrent program, it automatic indents **printf** statements so that it is easy to see which statement comes from which process. The output for computation 4 (page 30) is:

```
            Process Q, n = 2
        Process P, n = 1
2 processes created
```

and the output from computation 6 is:

```
        Process P, n = 1
            Process Q, n = 1
2 processes created
```

The argument -T turns off the automatic indentation.
Section 2.2.2 listed the arguments that enable the display of the statements that are executed and the values of the variables. Here is the output (edited to fit on the page) that is obtained by running with the arguments -p -g. There is a line for each statement executed and a line that displays the value of the variable n *when it changes*.


```
Starting P with pid 0
0: proc  - (:root:) creates proc  0 (P)
Starting Q with pid 1
0: proc  - (:root:) creates proc  1 (Q)
1: proc  1 (Q) line   9 (state 1) [n = 2]
          n = 2
     Process Q, n = 2
2: proc  1 (Q) line  10 (state 2) [printf(Q)]
2: proc  1 (Q)           terminates
3: proc  0 (P) line   4 (state 1) [n = 1]
          n = 1
    Process P, n = 1
4: proc  0 (P) line   5 (state 2) [printf(Q)]
4: proc  0 (P)       terminates
2 processes created
```

3.2 Atomicity

Statements in PROMELA are *atomic*. At each step, the statement pointed to by the location counter of some (arbitrary) process is executed in its entirety. So, for example, in Listing 3.1 it is not possible for the assignment statements to overlap in a way that causes n to receives some value other than 1 or 2.

> **Warning**
>
> Expressions in PROMELA are *statements*.
>
> In an **if**- or **do**-statement it is possible for interleaving to occur between the evaluation of the expression (statement) that is the guard and the execution of the statement after the guard.

In the following example, assume that a is a global variable; you *cannot* infer that division by zero is impossible:

```
if
:: a != 0 ->
     c = b / a
:: else ->
     c = b
fi
```

Between the evaluation of the guard `a != 0` and the execution of the assignment statement `c = b / a`, some other process might have assigned zero to a. In Section 4.4 we will discuss ways of executing the statements of a sequence atomically, and in Section 3.4 we show how to model hardware at a level lower than a complete expression.

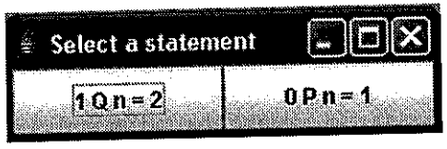
3.3 Interactive simulation

When there are two or more nontrivial processes in a PROMELA program, the number of computations becomes extremely large because every possible interleaving gives rise to a computation. Random simulation tells us almost nothing about the program, except that it works for a few computations. Therefore, verification is essential when one deals with concurrent programs. Verifications are quite likely to find counterexamples and these computations can be displayed as described in Section 2.2.1. However, it is also useful to examine a computation step by step and to manually "select" the next instruction to be executed; in other words, to create a specific interleaving.

With *interactive simulation* a specific computation can be constructed. Before each step that has a *choice point* – either because of nondeterminism within a single process or when a choice of the next statement to execute can be made from several processes – you are presented with the various choices and can interactively choose which one to execute. To run an interactive simulation:

jSpin

Select `Interactive`. The set of choices at each step is displayed in a popup window:



Click on the one you wish to execute. The selections are prefixed by the process ID and name, as the same source statement may appear in several processes. At any time you can close the window to terminate the simulation.

To work with the keyboard: `Tab` moves between the choices and `Space` selects the currently highlighted choice. `Esc` terminates the simulation.

Command line

Execute SPIN with the argument `-i`. Before each step you will be presented with a set of choices:

```
Select a statement
 choice 1: proc  1 (Q) line   9 (state 1) [n = 2]
 choice 2: proc  0 (P) line   4 (state 1) [n = 1]
Select [1-2]:
```

Enter the number of the choice you wish to execute or `q` to terminate the interactive simulation.

> **Warning**
>
> The numbering may not start at one.
>
> Be sure to enter the *choice number* and not the *line number*.

Some of the choices may not be executable. For example, if the value of `x` is 1 and the statement to be executed is a guard `x > 1`, the choice will be marked as unexecutable and you cannot select it.

3.4 Interference between processes

The challenge of writing concurrent programs comes not from interleaving as such, but rather from interference between processes that can cause truly bizarre errors. Consider the program in Listing 3.2 that increments the global variable `n` in each of two processes. Its value is copied into the local variables `temp` when the add operations are performed, and the result is copied back to the global variable in a separate assignment statement. (This models a CPU that performs computation in registers as explained in more detail at the end of this section.) Between the statements of process `P` on lines 5 and 6 it is possible to interleave statements from process `Q`, and similarly for process `Q`. Such interleaving would not be possible if the computation were performed in a single atomic statement `n = n + 1`.

Clearly, we expect that incrementing the variable `n` twice will cause its final value to be 2, whatever the order in which the increment instructions are executed. Surprisingly, this is not true, as can be seen by the computation in Figure 3.1, in which the final value is 1.[2] This results from interference

[2] Since the variable `temp` is used in both processes, we prefix the variable name by its process name to resolve the ambiguity.

Listing 3.2. Interference between two processes

```
1  byte  n = 0;
2
3  active proctype P() {
4    byte temp;
5    temp = n + 1;
6    n = temp;
7    printf("Process P, n = %d\n", n)
8  }
9
10 active proctype Q() {
11   byte temp;
12   temp = n + 1;
13   n = temp;
14   printf("Process Q, n = %d\n", n)
15 }
```

between the two processes. Both copy the same initial value, and the updated value from one process is overwritten by the updated value from the second process. Run a random simulation of the program several times until you get a computation in which 1 is printed twice. To ensure that you understand how the computation is obtained, create it by interactive simulation.

Fig. 3.1. Perfect interleaving

Process	Statement	n	P:temp	Q:temp	Output
P	temp = n + 1	0	0	0	
Q	temp = n + 1	0	1	0	
P	n = temp	0	1	1	
Q	n = temp	1	1	1	
P	printf(P)	1	1	1	
Q	printf(Q)	1	1	1	P, n = 1
					Q, n = 1

Advanced: Modeling a CPU with registers

This program is a simple model of a CPU that performs computation in registers:

```
load   R1, n
add    R1, #1
store  R1, n
```

In the PROMELA program the variable n represents a memory cell and the variables temp represent the register. In a multiprocess system each process has its own copy of the contents of the registers, which is loaded into the CPU registers and saved in memory during a context switch. We have modeled the computation in two statements rather than three, since the add operation is not visible outside the process, so there is no need to model it separately and it can be combined with either the load or the store operation. Alternatively, we could have used three statements and let the SPIN optimization called partial order reduction reduce the state space automatically (Section 10.2).

3.5 Sets of processes

In Listing 3.2 the two processes are identical except for their names. Instead of writing them separately, a set of identical processes can be declared (Listing 3.3). The number in brackets following the keyword **active** (line 3) indicates the number of processes to instantiate.

Listing 3.3. Instantiating two processes

```
1  byte  n = 0;
2
3  active [2] proctype P() {
4    byte temp;
5    temp = n + 1;
6    n = temp;
7    printf("Process P%d, n = %d\n", _pid, n)
8  }
```

How are we to distinguish between the processes? One way is to use the predefined variable **_pid**, which is of a separate type **pid** but actually similar to **byte**. Each time that a process is instantiated, it is assigned a *process identifier* starting with zero. (The maximum number of processes in a SPIN model is 255.) In the program in Listing 3.3, the value of **_pid** is printed so that we can distinguish between the two processes on output (line 7).

An alternate way of instantiating processes from a **proctype** is to use the **run** operator (Listing 3.4). The keyword **run** is followed by the name of a *process type* (lines 13–14), which is indicated by **proctype** without the keyword **active** (line 3); this causes a process of that type to be instantiated. **run** is used to supply initial values to a process: the formal parameters are declared in the process type and are local variables initialized with the values of the actual parameters.

Processes in PROMELA are usually instantiated in a process called **init**, which – if it exists – is always the first process activated and thus the value of **_pid** in this process is 0. In Listing 3.4, the processes are instantiated in an **init** process where they are passed an explicit identifier id, as well as an additional value incr that is used in the assignment statements. The initialization of the global variable has also been moved to the **init** process (line 11), though this would normally be done only for non-trivial initialization code such as the nondeterministic selection of a value (Section 4.6).

By convention, **run** statements are enclosed in an **atomic** sequence to ensure that all processes are instantiated before any of them begins execution (lines 12–15). The meaning of **atomic** will be explained in the next section.

> **Warning**
>
> The formal parameters of a **proctype** are separated with semicolons, not commas.

Advanced: The run operator

run is an *operator*, so **run** `P()` is an expression, not a statement, and it returns a value: the process ID of the process that is instantiated, or zero if the maximum number of processes (255) have already been instantiated.

3.6 Interference revisited

The use of **init** enables us to write a fascinating program for demonstrating interference. The program in Listing 3.5 contains two processes, each of which increments the global variable n ten times.

Listing 3.4. The **init** process

```
1  byte  n;
2
3  proctype P(byte id; byte incr) {
4    byte temp;
5    temp = n + incr;
6    n = temp;
7    printf("Process P%d, n = %d\n", id, n)
8  }
9
10 init {
11   n = 0;
12   atomic {
13     run P(1, 10);
14     run P(2, 15)
15   }
16 }
```

We wish to print the final value of n after the two processes have completed executing their statements, so we need some way to force process **init** to wait for the completion of the other two. This can be done by using the predefined variable **_nr_pr** whose value is the number of processes currently active. The statement in line 17 consisting just of the expression

```
(_nr_pr == 1)
```

causes process **init** to be blocked until the expression evaluates to true, which occurs when the number of active processes is equal to 1, namely, the process **init** itself. (The use of expressions for blocking execution will be discussed in Section 4.2.)

Consider now two computations. In the first the computation is performed without interference: One process executes *all* its statements in sequence, followed by *all* the statements of the second process. It is easy to see that the final value of n is 20. The second computation is performed by "perfect" interleaving: The computation is created by *alternately* "selecting" one statement from each process, generalizing the computation shown in Figure 3.1. It is not difficult to see that each pair of updates of the variable n increments its value by 1, and the final value is 10.

Intuitively, it seems as if "perfect" interleaving represents the maximum "amount of interference" possible, and for many years I taught that the final

Listing 3.5. Counting with interference

```
1  #include "for.h"
2  byte  n = 0;
3
4  proctype P() {
5    byte temp;
6    for (i, 1, 10)
7      temp = n + 1;
8      n = temp
9    rof (i)
10 }
11
12 init {
13   atomic {
14     run P();
15     run P()
16   }
17   (_nr_pr == 1) ->
18     printf("The value is %d\n", n)
19 }
```

value of n must be between 10 and 20. It came somewhat of a shock when I discovered that the final value can be as low as 2 (see [6])! Try to find the computation yourself; if you can't, we will show in Section 3.8 how verification can be used to discover it.

3.7 Deterministic sequences of statements*

As noted in Section 3.2, each statement of PROMELA (including expressions) is executed atomically. Thus, `n = n + 1` is executed atomically, so to model a CPU with an accumulator we wrote a program with local variables and two assignment statements (Listing 3.2).

To model atomic statements that are more complex than a single assignment statement, we can specify that a sequence of statements is to be executed atomically. There are two ways of creating atomic sequences of statements: **d_step** (short for deterministic step) and **atomic**. Listing 3.6 shows the use of **d_step** to ensure that the two statements in the processes in Listing 3.2 are executed atomically (lines 5–8, 14–17), resulting in a program that is equivalent to one with the two-statement sequences replaced by `n = n + 1`. The following computation shows that executing the program prints the expected result:

Process	Statement	n	P:temp	Q:temp	Output
P	`temp = n + 1; n = temp`	0	0	0	
Q	`temp = n + 1; n = temp`	1	1	0	
P	`printf("P")`	2	1	2	
Q	`printf("Q")`	2	1	2	`P, n = 2`
					`Q, n = 2`

In this example, **atomic** can be used instead of **d_step**. The difference between the two will be explained in Section 4.4.

Listing 3.6. Deterministic step

```
1  byte  n = 0;
2
3  active proctype P() {
4    byte temp;
5    d_step {
6      temp = n + 1;
7      n = temp
8    }
9    printf("Process P, n = %d\n", n)
10 }
11
12 active proctype Q() {
13   byte temp;
14   d_step {
15     temp = n + 1;
16     n = temp
17   }
18   printf("Process Q, n = %d\n", n)
19 }
```

There are many synchronization primitives such as *test-and-set* and *exchange* that are based upon the atomic execution of a sequence of statements. These are explained in Section 3.10 of *PCDP*, and implementations in PROMELA are given in the software archive for that book.

3.8 Verification with assertions

Consider again the program in Listing 3.5 that increments a global variable by 10 in each of two processes. We claimed that there is a computation whose output is 2. How can we check this? We can run random simulations until it occurs, and that is how I was first made aware of the existence of such a computation: A student in a computer lab executed the program again and again using a concurrency simulator and called me over when the output was 9, a result totally at odds with my intuition. Eventually, I discovered the principle behind the computation, as well as the fact that such a computation can result in an output of 2.

With SPIN the computation can be obtained automatically by adding the assertion

```
assert (n > 2)
```

at the end of the program and running a verification. You can be excused if this assertion looks weird: We want to prove that the variable n can have the value 2, but instead we assert that its value is *greater than* 2. What SPIN does is to search the state space looking for *counterexamples*, that is, computations that are in error. If the assertion given were `n >= 2`, SPIN would report a successful verification with no errors because, in fact, the final value of n really is greater than or equal to 2. By asserting the false formula `n > 2`, SPIN will find a computation in which `n > 2` false, that is, a computation for which its negation `n <= 2` is true.

Running a verification with SPIN results in the error message:

```
pan: assertion violated (n>2) (at depth 89)
```

As described in Section 2.2, a guided simulation can now be run with the trail in order to examine the computation that caused the assertion to be falsified. The following computation is taken with very little editing directly from the JSPIN display; to save space, the program was run with an upper limit of 5 for the loop:


```
Process Statement        P(1):temp  P(2):temp  n
2 P     7   temp = n
1 P     7   temp = n     0
2 P     8   n = (temp+1) 0          0
2 P     7   temp = n     0          0          1
2 P     8   n = (temp+1) 0          1          1
2 P     7   temp = n     0          1          2
2 P     8   n = (temp+1) 0          2          2
2 P     7   temp = n     0          2          3
2 P     8   n = (temp+1) 0          3          3
1 P     8   n = (temp+1) 0          3          4
2 P     7   temp = n     0          3          1
1 P     7   temp = n     0          1          1
1 P     8   n = (temp+1) 1          1          1
1 P     7   temp = n     1          1          2
1 P     8   n = (temp+1) 2          1          2
1 P     7   temp = n     2          1          3
1 P     8   n = (temp+1) 3          1          3
1 P     7   temp = n     3          1          4
1 P     8   n = (temp+1) 4          1          4
2 P     8   n = (temp+1) 4          1          5
0 :init 16  _nr_pr==1    4          1          2
```

The value of `temp` in `P2` is reset to `1` and remains `1` so that a final addition ignores the value `4` already set in `n`.

jSpin

The computation of a simulation will be displayed in the right pane. Select `Output / Save output` to write the contents of the display to a file which you can edit as necessary. For this display the width of the variable fields was set to 10 and the loop variables `i` of the two processes were excluded from the display, as were the statements that access `i`. See Section 2.2.2 for instructions on how to do this.

Command line

The output of a simulation can be redirected to a file:

```
pan -t count-verif.pml > count-verif.out
```

3.9 The critical section problem

This section begins the presentation of the verification of correctness properties of concurrent systems. To that end we pose the *critical section problem*, which is the archetypal problem in concurrent programming. We will not attempt to give the motivation of this problem, nor a comprehensive set of solutions, as these can be found in textbooks on concurrency like *PCDP*. Let us just state the specification of the problem:

> A system consists of two or more concurrently executing processes. The statements of each process are divided into *critical* and *noncritical* sections that are repeatedly executed one after the other. A process may halt in its noncritical section, but not in its critical section. Design an algorithm for ensuring that the following specifications hold:
>
> **Mutual exclusion**
>
> At most one process is executing its critical section at any time.
>
> **Absence of deadlock**
>
> It is impossible to reach a state in which *some processes* are trying to enter their critical sections, but *no process* is successful.
>
> **Absence of starvation**
>
> If *any process* is trying to execute its critical section, then eventually *that process* is successful.

The program in Listing 3.7 shows an attempt at solving the critical section problem for two processes. Each process executes a nonterminating **do**-statement (lines 4–9, 13–18), alternating between the critical and the noncritical sections which are represented by **printf** statements. The phrase "a process tries to execute its critical section" means that the process has finished executing its noncritical section and its location counter is at the following statement (lines 6, 15). Note that the possibility of halting in the noncritical section has not been modeled (see Section 5.9.2).

The variables `wantP` and `wantQ` are used to signal that a process is accessing its critical section; a process sets its variable to true before the critical section (lines 6, 15) and back to false afterwards (lines 8, 17). Of course, since a process never checks the value of the variable associated with the other process, it is trivial to find a computation in which they are both in their critical sections, indicated by both location counters pointing to the print statements representing the critical section (lines 7, 16). Thus the program in Listing 3.8 is trivially incorrect, but it serves to introduce the structure of a solution to the problem.

In a sequential program a postcondition must be true when the program terminates, and, similarly, an invariant must be true whenever the program

Listing 3.7. Incorrect solution for the critical section problem

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: printf("Noncritical section P\n");
6       wantP = true;
7       printf("Critical section P\n");
8       wantP = false
9    od
10 }
11
12 active proctype Q() {
13   do
14   :: printf("Noncritical section Q\n");
15      wantQ = true;
16      printf("Critical section Q\n");
17      wantQ = false
18   od
19 }
```

evaluates it. However, in a concurrent program, we generally need correctness specifications that consider the global state of *all* the processes in the program. For example, to specify that two processes cannot be in their critical sections at the same time, the specification must refer to control points in both processes.

One way of doing this is to introduce a new variable (`critical`) that is not part of the algorithm but is only used for verification (Listing 3.8). Such a variable is called a *ghost variable*. The variable is incremented before executing a critical section (lines 8, 20) and decremented afterwards (lines 11, 23); clearly, if there exists a state in some computation in which both location counters are at the print statements representing the critical section (lines 9, 21), then in that state the value of `critical` is greater than one.[3]

Running a verification uncovers a state in which the value of the variable `critical` is 2 when one of the **assert** statements is executed, indicating that mutual exclusion has been violated:

[3] This property can also be specified and verified without ghost variables as shown in Section 5.7.

Listing 3.8. Verifying mutual exclusion

```
1  bool wantP = false, wantQ = false;
2  byte critical = 0;
3
4  active proctype P() {
5    do
6    :: printf("Noncritical section P\n");
7       wantP = true;
8       critical++;
9       printf("Critical section P\n");
10      assert (critical <= 1);
11      critical--;
12      wantP = false
13   od
14 }
15
16 active proctype Q() {
17   do
18   :: printf("Noncritical section Q\n");
19      wantQ = true;
20      critical++;
21      printf("Critical section Q\n");
22      assert (critical <= 1);
23      critical--;
24      wantQ = false
25   od
26 }
```

```
spin: line  23 "cs.pml", Error: assertion violated
spin: text of failed assertion: assert((critical<=1))
#processes: 2
Process Statement          critical   wantP      wantQ
1 Q                        2          1          1
0 P                        2          1          1
```

In the next chapter we will explore how synchronization between processes can be achieved in order to solve the critical section problem.

4

Synchronization

PROMELA does not have synchronization primitives such as semaphores, locks, and monitors that you may have encountered. Instead, you model primitives by building on the concept of the *executability* of statements. The architecture of a computer system constrains the design of synchronization mechanisms: In this chapter, we present synchronization mechanisms appropriate for models of shared memory systems, while in Chapter 7 we will discuss channels that are used to model synchronization by communication in distributed systems that lack shared memory.

4.1 Synchronization by blocking

The program in Listing 3.8 that attempted to solve the critical section problem was trivially incorrect because, while each process set a variable indicating its intention to enter its critical section, these variables were not read by the other process. A simple-minded way to try to remedy this difficulty is to write a loop before the entry to the critical section, checking the value of the variable associated with the *other* process (Listing 4.1). This is called *busy-waiting* because the loops in lines 7–10 and 20–23 perform no useful computation; they just evaluate expressions repeatedly until they become true.

While busy-waiting is an acceptable model for some systems – for example, for a multiprocessor with a large number of processors that can afford to have some of them "waste" cycles waiting for an event to occur – normally, computer systems are based upon *blocking* a process so that its processor can be assigned to another process.

Synchronization by blocking will be familiar to anyone with experience in operating systems that implement *multitasking*, the sharing of a single processor among a set of processes. It is multitasking that enables us to perform

Listing 4.1. Synchronization by busy-waiting

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: printf("Noncritical section P\n");
6       wantP = true;
7       do
8       :: !wantQ -> break
9       :: else -> skip
10      od;
11      printf("Critical section P\n");
12      wantP = false
13   od
14 }
15
16 active proctype Q() {
17   do
18   :: printf("Noncritical section Q\n");
19      wantQ = true;
20      do
21      :: !wantP -> break
22      :: else -> skip
23      od;
24      printf("Critical section Q\n");
25      wantQ = false
26   od
27 }
```

activities in parallel, like scrolling through one web page while another is being downloaded. Processes executing concurrently must be synchronized if they access common resources; for example, only one process may be allowed to update the display at any time. Synchronization in multitasking systems is implemented by having a process execute an operation that causes it to become blocked, thus enabling another process to run. Continuing our example, *if* one process is assigned permission to use a resource like the display, *then* other processes that need the display will block themselves until the first process releases it. This shows that blocking is frequently *conditional*.

In the program in Listing 4.1, we would like process P to block itself *until* wantQ becomes false and process Q to block itself *until* wantP becomes false.

We have already encountered a blocking statement in PROMELA, the **if**-statement. Recall (Section 1.6) that an **if**-statement contains a set of alternatives that start with expressions called guards.[1] An alternative is *executable* if its guard evaluates to **true** (or 1, which is the same). The choice of the alternative to execute is made nondeterministically among the executable alternatives. If no guards evaluate to true, the **if**-statement itself is not executable. Similarly, in a **do**-statement, if the guards of all alternatives evaluate to false, the statement is not executable and the process is blocked.

Let us replace the **do**-statements that implement busy-waiting in Listing 4.1 by **do**-statements with a single alternative that can block. In process P replace lines 7–10 by:

```
do
:: !wantQ -> break
od
```

and in process Q replace lines 20–23 by:

```
do
:: !wantP -> break
od
```

Consider now an attempt to execute the **do**-statement in process P. The guard !wantQ will be evaluated: if its value is true (because the value of the variable wantQ is false), the computation will execute the **break** and exit the **do**-statement. If, on the other hand, the guard !wantQ is false, the process is blocked at the **do**-statement.

To say that a process is blocked means that in simulation mode SPIN will not choose the next statement to execute from that process. In verification mode it means that SPIN will not continue the search for a counterexample from this state by looking for states that can be reached by executing a statement from the process. Hopefully, a subsequent execution of statements from other processes will *unblock* the blocked process, enabling it to continue executing in simulation mode, and in verification mode, enabling the verifier to search for states reachable by executing a statement from the process. If process P blocks because wantQ is true, eventually, process Q will execute line 25, setting wantQ to false and unblocking process P.

Check this behavior by running an interactive simulation of the program. Execute statements of the program until a state is reached in which P is

[1] Any statement, not just an expression, can be a guard, but expressions alone will be used in the following discussion.

blocked because wantQ is true; in this state you will not be allowed to choose to execute a statement from process P. Now choose to execute statements from Q until the statement wantQ = **false** is executed, enabling the execution of the a statement from P.

4.2 Executability of statements

> **Warning**
>
> The concept presented in this section is likely to be unfamiliar even to experienced programmers. Please read it carefully!

There is something rather strange about the construct:

```
do
:: !wantQ -> break
od
```

Either wantQ is false and the **break** causes the loop to be left, or it is true and the process blocks; when it is unblocked the process can leave the loop. In no case is there any "looping," so the **do**-statement is superfluous. In PROMELA it is possible to block on a simple statement, not just on a compound statement. Lines 7–10 in Listing 4.1 can be replaced by a single statement that is the expression !wantQ, and similarly, lines 20–23 by !wantP. An expression statement is *executable* if and only if it evaluates to true, in this case if the value of wantQ is false.

Listing 4.2 shows the program for the critical section problem written as it should be in PROMELA, with expressions alone used for blocking processes.[2] Again, we suggest that you run an interactive simulation of the program in order to experience the phenomenon of blocking on an expression.

The concept of executability holds for *every* statement in PROMELA. In the *man* page for each statement in PROMELA there is a section that specifies the conditions for the statement to be executable. Assignment statements and **printf** statements are always executable, so executability is primarily meaningful for expressions that can evaluate to true or false, including those that appear as guards in compound statements. The conditions for executability are also important in the definitions of channel operations (Chapter 7).

[2] This is the *third attempt* at solving the critical section problem in Chapter 3 of *PCDP*.

Listing 4.2. Synchronization with deadlock

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: printf("Noncritical section P\n");
6       wantP = true;
7       !wantQ;
8       printf("Critical section P\n");
9       wantP = false
10   od
11 }
12
13 active proctype Q() {
14   do
15   :: printf("Noncritical section Q\n");
16      wantQ = true;
17      !wantP;
18      printf("Critical section Q\n");
19      wantQ = false
20   od
21 }
```

4.3 State transition diagrams

Recall (Section 2.1) that a *state* of a program is a set of values of the variables and the location counters, and consider a program with two processes p and q that have s_p and s_q statements, respectively, and two variables x and y that range over v_x and v_y values, respectively. The number of possible states that can appear in computations of the program is

$$s_p \cdot s_q \cdot v_x \cdot v_y.$$

For example, the program in Listing 4.3 has $3 \cdot 3 \cdot 2 \cdot 2 = 36$ possible states.

However, not every possible state is *reachable* from the initial state during a computation of the program. In particular, a solution to the critical section problem is correct only if there are possible states that are *not* reachable, namely, states where the location counters of both processes are in their critical sections, thus falsifying the requirement of mutual exclusion.

Listing 4.3. Abbreviated solution for the critical section problem

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do :: wantP = true;
5          !wantQ;
6          wantP = false
7    od
8  }
9
10 active proctype Q() {
11   do :: wantQ = true;
12         !wantP;
13         wantQ = false
14   od
15 }
```

In principle, the set S of reachable states of a program is easily constructed:

1. Let $S = \{s_0\}$, where s_0 is the initial state; mark s_0 as *unexplored*.
2. For each unexplored state $s \in S$, let t be a state that results from executing an executable statement in state s; if $t \notin S$, add t to S and mark it unexplored. If no such states exist, mark s as *explored*.
3. Terminate when all states in S are marked explored.

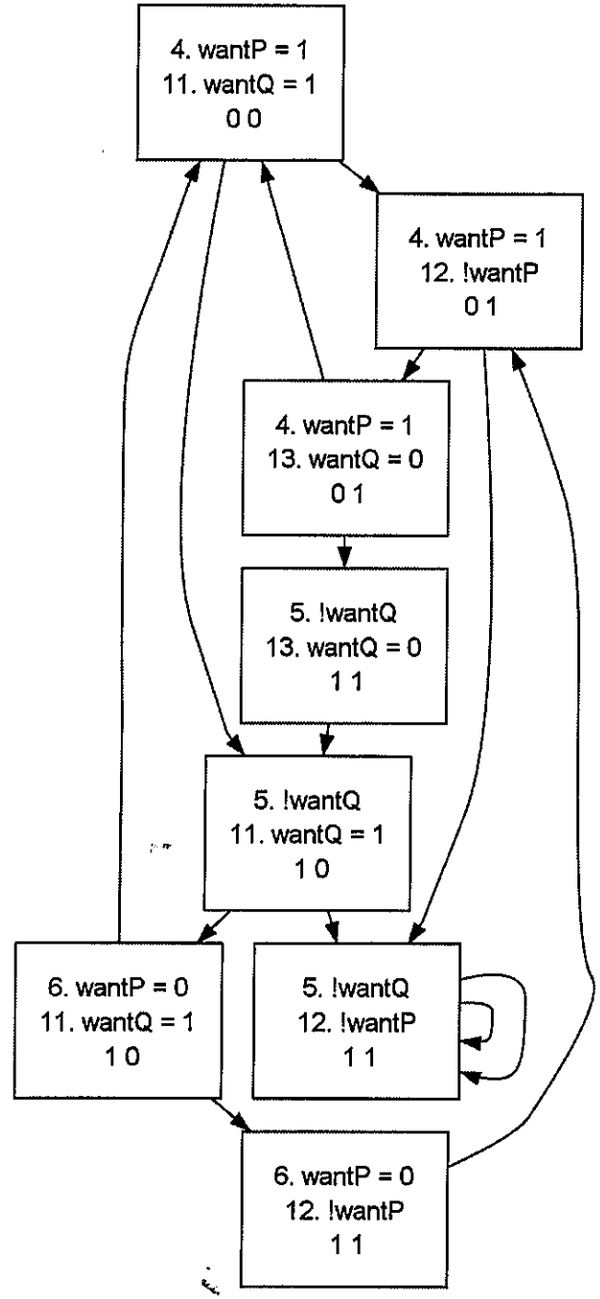
The reachable states of a concurrent program can be visualized as a connected directed graph called a *state transition diagram*. The nodes of the diagram are the reachable states and an edge exists from state s to state t if and only if there is a statement whose execution in s leads to t.

Figure 4.1 is the state transition diagram of the program in Listing 4.3.[3] Each node is labeled by the location counters for processes P and Q, followed by the values of the variables wantP and wantQ. To facilitate reading the diagram, the value of a location counter is given together with the source code at that control point.

The node that represents the initial state is at the top of the figure and the other nodes and the edges are constructed as described above. Most states have two outgoing edges because the next statement could be executed either

[3] The relation of Listing 4.3 to Listing 4.2 is explained at the end of this section.

Fig. 4.1. State diagram for the program in Listing 4.3



from process P or from process Q. However, the fourth state from the top labeled (5. !wantQ, 13. wantQ=0, 1, 1) has only one outgoing edge, because the statement 5. !wantQ in process P is not executable when the value of wantQ is true (= 1).[4]

The number of reachable states (8) is *much* less than the number of possible states (36).

The program in Listing 4.3 is an abbreviated version of the program in Listing 4.2.[5] The **printf** statements representing the critical and noncritical sections have been removed to obtain a more concise diagram. A **printf** statement is always executable and does not change the variables of the program, so if a state exists with a location counter before a print statement, there also exists a state with the location counter after the statement and with the same values for the variables. The same correctness specifications will thus be provable whether the print statements appear or not.

Consider the mutual exclusion property for the program in Listing 4.2. It holds if and only no state (8. printf(P), 18. printf(Q), x, y) is reachable for arbitrary x and y. Therefore, mutual exclusion holds if and only there is no state (9. wantP=0, 19. wantQ=0, x, y). Clearly, then, mutual exclusion holds if and only if, in the abbreviated program, a state of the form (6. wantP=0, 13. wantQ=0, x, y) is not reachable. A quick glance at the diagram in Figure 4.1 shows that no such state exists, so mutual exclusion must hold.

The program is not free from deadlock. The state (5. !wantQ, 12. !wantP, 1, 1) is reachable and in that state both processes are trying to enter their critical sections, but neither can succeed.

4.4 Atomic sequences of statements

It is quite difficult to come up with a fully correct solution to the critical section problem just using expressions and assignment statements.[6] However, easy solutions to the problem can be given if the system can execute sequences of these statements atomically.

[4] The state (5. !wantQ, 11. !wantP, 1, 1) near the bottom of the diagram is a state from which no transition is possible and should have no outgoing edges; the two edges curving back to the same state are an artifact of the way SPIN represents such a state.

[5] The layout of the program is slightly different from our usual style because of the requirements of the SPINSPIDER tool used to generate the state transition diagram (see Appendix A.3.)

[6] See, for example, Dekker's algorithm and Peterson's algorithm, Algorithms 3.10 and 3.13 in *PCDP*.

The program in Listing 4.4 contains a potentially blocking expression and an assignment statement as one atomic sequence of statements (lines 6–9, 18–21). This ensures that once process P has checked that !wantQ is true, it is not possible for process Q to set wantQ to true before P sets wantP to true.

Listing 4.4. Atomic sequences of statements

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: printf("Noncritical section P\n");
6       atomic {
7         !wantQ;
8         wantP = true
9       }
10      printf("Critical section P\n");
11      wantP = false
12   od
13 }
14
15 active proctype Q() {
16   do
17   :: printf("Noncritical section Q\n");
18      atomic {
19        !wantP;
20        wantQ = true
21      }
22      printf("Critical section Q\n");
23      wantQ = false
24   od
25 }
```

In Listing 4.4, the potentially blocking statements (the expression !wantQ at line 7 and the expression !wantP at line 19) are at the beginning of the atomic sequences. Therefore, the atomic sequence may be blocked from executing, but once it starts executing, both statements are executed without interference from the other process.

Verify that this program fulfils the correctness requirements of mutual exclusion and of absence of deadlock. However, starvation is possible; the techniques for verifying this property and for finding a counterexample are presented in Chapter 5.

4.4.1 d_step and atomic*

In Section 3.7 we mentioned that there are two constructs in PROMELA for specifying that a sequence of statements must be executed atomically: **d_step** and **atomic**.

The advantage of **d_step** is that it is extremely efficient because the statements of the sequence are executed or verified as a single step in a fully deterministic manner. However, there are three limitations on **d_step**:

- Except for the first statement in the sequence (the guard), statements cannot block.
- It is illegal to jump into the sequence or out of it using **goto** or **break**.
- Nondeterminism is always resolved by choosing the first true alternative in a guarded command. For example, if a equals b in the following code, the value of branch will always equal 1:

```
d_step {
  if
  :: a >= b -> max = a; branch = 1
  :: b >= a -> max = b; branch = 2
  fi
}
```

d_step is usually reserved for fragments of sequential code, while **atomic** is preferred for implementing synchronization primitives.

Consider the program in Listing 4.5, which models an unreliable component for relaying data. Process Source generates values for input and the Process Destination prints the values of output. Process Relay transfers data from the Source to the Destination. The **atomic** sequence (lines 13–20) waits until the variable input has data and then waits until the variable output is empty (modeled by zero); then, it nondeterministically either transfers the value from input to output or it ignores the data. The program works as expected and repeated random simulations print out different subsequences of the input sequence.

If **atomic** is replaced by **d_step**, two problems occur. First, since nondeterminism is resolved deterministically in favor of the first alternative, no data are ever dropped at line 18, and the output sequence is always the same

Listing 4.5. Unreliable relay

```
1  #include "for.h"
2  byte input, output;
3
4  active proctype Source() {
5    for (i, 1, 10)
6      input == 0; /* Wait until empty */
7      input = i
8    rof (i)
9  }
10
11 active proctype Relay() {
12   do
13   :: atomic {
14        input != 0;
15        output == 0;
16        if
17        :: output = input
18        :: skip /* Drop input data */
19        fi
20     }
21     input = 0
22   od
23 }
24
25 active proctype Destination() {
26   do
27   :: output != 0; /* Wait until full */
28      printf("Output = %d\n", output);
29      output = 0
30   od
31 }
```

as the input sequence. Second, it is not legal to block at line 15 which is within the **d_step** sequence; this can be modeled, for example, by using a for-loop with an upper bound less than ten instead of the nonterminating **do**-statement in Destination.

An unreliable relay can also be modeled using channels (Chapter 7):

```
active proctype Relay() {
  byte i;
  do
  :: atomic {
       input ? i;
       if
       :: output ! i
       :: skip
       fi
     }
  od
}
```

Again, changing **atomic** to **d_step** cancels the nondeterministic selection of an alternative and can cause an error at the output statement output ! i if the channel is full or if a rendezvous channel is used and the process Destination is not ready.

4.5 Semaphores

Atomic sequences of statements can be used to model synchronization primitives such as semaphores. The most widely known construct for synchronizing concurrent programs is the *semaphore*. Here is a simple definition of a semaphore using concepts of PROMELA:

> A semaphore sem is a variable of type **byte** (nonnegative integers). There are two *atomic* operations defined for a semaphore:
> - wait(sem): The operation is executable when the value of sem is positive; executing the operation decrements the value of sem.
> - signal(sem): The operation is always executable; executing the operation increments the value of sem.

Listing 4.6 shows a solution to the critical section problem using semaphores. The wait operation is implemented using **atomic** to ensure that the value of sem is decremented only when it is positive (lines 6–9, 18–21). The signal operation needs no special implementation because the assignment statement

is atomic (lines 11, 23). Verify that the program fulfils the correctness properties of mutual exclusion and absence of deadlock.

Listing 4.6. The critical section problem with semaphores

```
1  byte sem = 1;
2
3  active proctype P() {
4    do
5    :: printf("Noncritical section P\n");
6       atomic {    /* wait(sem) */
7         sem > 0;
8         sem--
9       }
10      printf("Critical section P\n");
11      sem++       /* signal(sem) */
12   od
13 }
14
15 active proctype Q() {
16   do
17   :: printf("Noncritical section Q\n");
18      atomic {    /* wait(sem) */
19        sem > 0;
20        sem--
21      }
22      printf("Critical section Q\n");
23      sem++       /* signal(sem) */
24   od
25 }
```

Advanced: Fairness of semaphores

The subject of semaphores is more complex than this simple example indicates. The difficulties arise when we try to define the *fairness* of the semaphore operations. Even when a verification is performed with weak fairness enabled (see Section 5.5), a computation for starvation is found because process Q can enter its critical section repeatedly while P does not.
In this computation, the only process that executes is process Q, which repeatedly executes its entire loop from line 16 to 24. The computation is weakly fair because P is enabled infinitely often (after Q executes sem++ at line 23); the nonfair computation simply chooses not to execute the atomic statement from P when it is enabled.
The `signal` operation is usually defined to unblock one of the processes blocked on the semaphore (if any) as part of its atomic operation. A *strong semaphore* implements the set of blocked processes as a FIFO (first in-first out) queue; this is easy to model in PROMELA using channels (Chapter 7). The `signal` operation of a *weak semaphore* unblocks an arbitrary element of the set; weak semaphores are harder to model in PROMELA (see Exercise 6.15 of *PCDP*).

4.6 Nondeterminism in models of concurrent systems

Consider, for example, a communications system that must be able to receive and process an *arbitrary* stream of messages of several different types. A natural approach to modeling this requirement is to generate the messages stream by using a *random number* generator. If there are n message types $m_0, m_1, \ldots, m_{n-1}$, each message in the stream is obtained by generating a random number in the range 0 to $n-1$.

However, this approach is flawed. While a random number generator can be used to obtain a random computation (and this is precisely what SPIN does in random simulation mode), for verification *all* computations must be checked, not just those that happen to be generated randomly.

By design, SPIN does not contain constructs for modeling probability or for specifying that an event must occur with a certain probability. The intended use of model checking is to detect errors that occur under complex scenarios that are unlikely to be discovered during system testing. "In a well-designed system, erroneous behavior should be impossible, not just improbable" [*SMC*, p. 454].[7]

[7] See also the discussion on p. 570 of *SMC*.

In SPIN, nondeterminism is used to model arbitrary values of data: whenever a value – such as a message type in a stream – is needed, a nondeterministic choice is made among all values in the range.

4.6.1 Generating values nondeterministically

Suppose that we want to model a client-server system in which the client *nondeterministically* chooses which request to make; we can use an **if**-statement whose guards are always true:[8]

```
active proctype Client() {
  if
  :: true -> request = 1
  :: true -> request = 2
  fi;
  /* Wait for service */
  if
  :: true -> request = 1
  :: true -> request = 2
  fi;
  /* Wait for service */
}
```

The code can be shortened by doing away with the expressions **true** which serve no purpose. Instead, the assignment statements themselves – which are always executable – can be used as guards:[9]

```
active proctype Client() {
  if
  :: request = 1
  :: request = 2
  fi;
  /* Wait for service */
  if
  :: request = 1
  :: request = 2
  fi;
  /* Wait for service */
}
```

[8] You may want to study the program in Section 4.7.2 before reading this section.
[9] The arrows are also not needed; they are just syntactic sugar for semicolons that are separators, but since there is only one statement in each alternative there is nothing to separate (Section 1.6).

In a random simulation, SPIN randomly chooses which alternative of an **if**-statement to execute; here, the choice is between both alternatives of the statement since they are both executable. In a verification, SPIN chooses the first alternative and searches for a counterexample; if one is not found, it *backtracks* and continues the search from the state that results from choosing the second alternative.

Of course, it doesn't make sense to model a client that generates only two requests. A **do**-statement can be used to model a client that generates an unending stream of requests in an arbitrary order:

```
active proctype Client() {
  do
  :: request = 1;
     /* Wait for service */
  :: request = 2;
     /* Wait for service */
  od
}
```

In Chapter 7 we will discuss some of the correctness properties that can be checked for client-server systems.

4.6.2 Generating from an arbitrary range*

We have shown how to model nondeterministically generated values from a small range:

```
byte number;
if
:: number = 1
:: number = 2
:: number = 3
:: number = 4
fi
```

As the range of values gets larger, it becomes inconvenient to write alternatives for each value. The following PROMELA code shows how to choose nondeterministically values from an arbitrary range, in this case from 0 to 9:


```
#define LOW 0
#define HIGH 9
  byte number = LOW;
  do
  :: number < HIGH -> number++
  :: break
  od
```

As long as the value of number is less than HIGH, both alternatives are executable and SPIN can choose either one. If it chooses the first one, the value of number is incremented; if it chooses the second, the loop is left and the current value of number is used in the subsequent code. It follows that the final value of number can be any value within the range.

To check that 9 is, in fact, a possible value of number, add the assertion

```
assert (number != HIGH)
```

after the **do**-statement and run a verification. You will get a counterexample and this computation can be examined by running a guided simulation.

Do not put any faith in the uniformity of the probability distribution of the "random numbers" generated using this technique. Assuming that SPIN chooses uniformly between alternatives in the **do**-statement, the first value 0 has a probability of $1/2$ while the last value 9 has a probability of 2^{-10}. Nondeterminism is used to generate arbitrary computations for verification, not random numbers for a faithful simulation.

4.7 Termination of processes

4.7.1 Deadlock

Unfortunately, the program in Listing 4.2 is not a correct solution of the critical section problem. The processes of the program consist of loops with no **goto** or **break** statements, so the program should never terminate.[10] If you run several random simulations of the program, you will likely encounter a computation in which execution terminates with the output `timeout`. This means that *no* statements are executable, a condition called *deadlock*.[11]

It is quite easy to construct the computation that leads to deadlock. Simply execute statements in perfect interleaving (one statement alternately from each process); both `wantP` and `wantQ` are set to true (lines 6, 16) and

[10] When running a simulation in SPIN, you normally limit of the number of statements that can be executed by using the argument `-uN` (see Section 1.2).

[11] `timeout` is discussed further in Section 8.1.1.

then both processes are blocked waiting for the other one to set its variable to false (lines 7, 17).

An attempt at verification will discover an error called an *invalid end state*:

```
pan: invalid end state (at depth 8)
```

By default, a process that does terminate must do so after executing its last instruction, otherwise it is said to be in an invalid end state. This error is checked for regardless of any other correctness specifications. This default behavior can be overridden as described in the next subsection.

4.7.2 End states*

Consider the program in Listing 4.7. There are two server processes supplying different services and one client process that requests service 1 and then service 2. The client process indicates which service it needs by setting the variable `request` to the number of the service; it then blocks waiting for the expression `request == 0` to become true. The guard of the alternative in one of the server processes is now true, so it can provide the service (represented by a **printf** statement). Then, the server resets `request` to zero to indicate that the service is complete; this unblocks the client.

This PROMELA program is a reasonable model of a very simple *client-server system*.[12] However, if you simulate or verify it in SPIN, you will receive an error message that there is an invalid end state. The reason is that while the client executes a finite number of statements and then terminates, the servers are always blocked at the guard of the **do**-statement waiting for it to become executable. Now, this is acceptable behavior because servers should wait indefinitely and be ready to supply a service whenever it is needed. Since the server cannot know how many requests it will receive, it is unreasonable to require termination of a process modeling a server.

You can indicate that a control point within a process is to be considered a valid end point even though it is not the last statement of the process by prefixing it with a label that begins with end:

```
active proctype Server1() {
endserver:
  do
  :: request == 1 -> . . .
  od
}
```

[12] Client-server systems are presented more systematically in Chapter 7.

Listing 4.7. A client-server program with end states

```
1  byte request = 0;
2
3  active proctype Server1() {
4    do
5    :: request == 1 ->
6         printf("Service 1\n");
7         request = 0
8    od
9  }
10
11 active proctype Server2() {
12   do
13   :: request == 2 ->
14        printf("Service 2\n");
15        request = 0
16   od
17 }
18
19 active proctype Client() {
20   request = 1;
21   request == 0;
22   request = 2;
23   request == 0
24 }
```

Add end labels to both server processes in the program in Listing 4.7 and show that a verification no longer reports an invalid end state.

An alternate way of ignoring end states is to ask SPIN to refrain from reporting invalid end states during a verification:

jSpin

Select `Options / Pan` and add the argument `-E`. Be sure to remove the argument when it is no longer needed.

Command line

Add the argument `-E` to the pan command for running the verifier.

The program in Listing 4.2 *does* fulfil the requirement of mutual exclusion; this can be shown using the technique described in Listing 3.8: counting the number of processes in their critical sections and asserting that the value of the variable is less than or equal to 1. If you run a verification, be sure to turn off checking of invalid end states as described above.

4.7.3 The order of process termination*

A process *terminates* when it has reached the end of its code, but it is considered to be an active process until it *dies*. SPIN manages process allocation in the LIFO (last in-first out) order of a stack, so a process can die only if it is the most recent process that was created. Usually, the distinction between process termination and death is not an issue, but it can sometimes explain why a program does not end as expected.

Process termination and death are demonstrated by the program in Listing 4.8. The two servers each perform one service and then terminate, incrementing the variable `finished` that counts the number of processes that have terminated. Since processes created by **active proctype** are instantiated in the order written, the two server processes do not die until the client process finds `finished == 2` and terminates.

The output is just as we expect it to be:

```
11:    proc  2 (Client) terminates
11:    proc  1 (Server2) terminates
11:    proc  0 (Server1) terminates
```

Suppose now that we change line 21 to `finished == 3` so that the client process does not terminate. By the LIFO rule, the server processes will not terminate, and the simulation goes into a state called `timeout` in which no process is at an executable statement:

```
timeout
#processes: 3
2 Client                2          0
1 Server                2          0
0 Server                2          0
```

All three processes are still active, though none are executable.

Listing 4.8. Client-server termination

```
1  byte request = 0;
2  byte finished = 0;
3
4  active proctype Server1() {
5    request == 1;
6    request = 0:
7    finished++
8  }
9
10 active proctype Server2() {
11   request == 2;
12   request = 0;
13   finished++
14 }
15
16 active proctype Client() {
17   request = 1:
18   request == 0;
19   request = 2;
20   request == 0;
21   finished == 2:
22 }
```

Next, move the process `Client` so that it appears *before* the server processes in the source code. Now, the server processes are created after the client process so they can terminate without waiting for the client process, which is blocked, hopelessly waiting for `finished` to receive the value 3:

```
timeout
#processes: 1
0 Client                2          0
```

5

Verification with Temporal Logic

In Sections 3.8 and 3.9 we showed how to use assertions to specify and verify correctness properties of concurrent programs written in PROMELA. However, assertions are not sufficient to specify and verify most correctness properties of models. This chapter presents *linear temporal logic (LTL)*, which is the formal logic used for verification in SPIN.[1] We start with an informal description of correctness properties more advanced than assertions. This is followed by an introduction to the syntax and semantics of LTL, an explanation of how to specify correctness properties in LTL, and a description of the techniques for using SPIN to verify that an LTL formula holds for a model. Section 5.9 gives an overview of more advanced ways of expressing properties in temporal logic. For a definitive treatment of LTL, see [16, 17].

5.1 Beyond assertions

Assertions are limited in the properties that they can specify because they are attached to specific control points in the processes. For example, in order to verify that mutual exclusion holds for a solution to the critical section problem, we inserted the following code at the control points representing the critical section in *each* process:

```
critical++;
assert (critical <= 1);
critical--;
```

[1] There are many forms of temporal logic; one, *computational tree logic (CTL)*, is also used extensively in verification [8]. Since SPIN limits itself to supporting LTL, the use of the term "temporal logic" in this book refers to LTL.

Usually, however, it is necessary or at least more convenient to express a correctness property as a global property of the system that is not associated with specific control points. Here are several examples of such properties:

- **Mutual exclusion**
  Mutual exclusion can be expressed as a global invariant:
  In every state of every computation, `critical <= 1`.
- **Absence of deadlock (invalid end states)**
  A PROMELA program is said to deadlock if it enters an invalid end state (Sections 4.7.1–4.7.2); this can be expressed as a global invariant:
  In every state of every computation, if no statements are executable, the location counter of each process must be at the end of the process or at a statement labeled end.
  This correctness property is checked automatically by SPIN.
- **Array index bounds**
  Let a be an array, let `LEN` be the length of the array, and let `i` be a variable used to index the array. An important global invariant is:
  In every state of every computation, `0 <= i <= LEN-1`.
  This formula could be added as an assertion after every statement that assigns a new value to `i`, but it is easier to specify that it holds in every state. This avoids errors caused if you forget to attach an assertion to one of the relevant statements.
- **Quantity invariant**
  In distributed algorithms called *token-passing algorithms,* mutual exclusion is achieved by passing a *token* – an explicit representation of the permission to enter the critical section – among the processes (see Sections 10.6 and 10.7 of *PCDP*). A global invariant that must hold in such algorithms is:
  In every state of every computation, there is at most one token in existence.

Furthermore, there are some correctness properties that simply cannot be expressed using assertions, because the properties cannot be checked by evaluating an expression in a *single state* of a computation. For example, in the critical section problem the following two properties are expressed as relations between two states of the computation: a state s in which processes are trying to enter their critical sections, and a state t in which a process does enter it. t may occur thousands of states later in the computation than s:

- **Absence of deadlock**[2]
  In every state of every computation, if some processes are trying to enter their critical sections, eventually some process *does so.*
- **Absence of starvation**
  In every state of every computation, if a process tries to enter its critical section, eventually that process *does so.*

A correctness specification like the ones given in this section is expressed in SPIN by a finite automaton called a *never claim* that is is executed together with the finite automaton that represents the PROMELA program. Specifying a correctness property directly as a never claim is difficult; instead, a formula written in linear temporal logic is translated by SPIN into a never claim, which is then used for verification. A brief introduction to never claims is given in Section 10.3, but for most purposes you need not concern yourself with never claims and can work entirely with LTL formulas.

The next section presents LTL as a formal logic. This is followed by sections describing how to express correctness properties of models in LTL and how to carry out verifications in SPIN.

5.2 Introduction to linear temporal logic

5.2.1 The syntax of LTL

LTL is based upon the propositional calculus; formulas of the propositional calculus are composed from atomic propositions (denoted by letters $p, q, \ldots$) and the operators:

Operator	Math	SPIN
not	$\neg$	!
and	$\wedge$	&&
or	$\vee$	||
implies	$\rightarrow$	->
equivalent	$\leftrightarrow$	<->

We have given both the usual mathematical symbols and the syntax used for writing formulas in SPIN; here is a formula written in both notations:

$$(p \wedge \neg q) \rightarrow (p \vee \neg q), \qquad \texttt{(p \&\& !q) -> (p || !q)}.$$

[2] The deadlock (invalid end states) described previously states that the computation cannot continue; this specification of deadlock states that the computation can continue, but processes cannot enter their critical sections.

A formula of LTL is built from atomic propositions and from operators that include the operators of the propositional calculus as well as temporal operators. The atomic propositions of LTL are described in the next subsection. The temporal operators are:

Operator	Math	SPIN
always	$\Box$	[]
eventually	$\Diamond$	<>
until	$\mathcal{U}$	U

The $\Box$ and $\Diamond$ operators are unary and the $\mathcal{U}$ operator is binary. Temporal and propositional operators combine freely, so the following formula (given in both mathematical and PROMELA notation) is syntactically correct:

$$\Box((p \wedge q) \rightarrow r\,\mathcal{U}(p \vee r)), \qquad \texttt{[]((p \&\& q) -> r U (p || r))}.$$

Read this as:

Always, (p and q) implies that r holds until (p or r) holds.

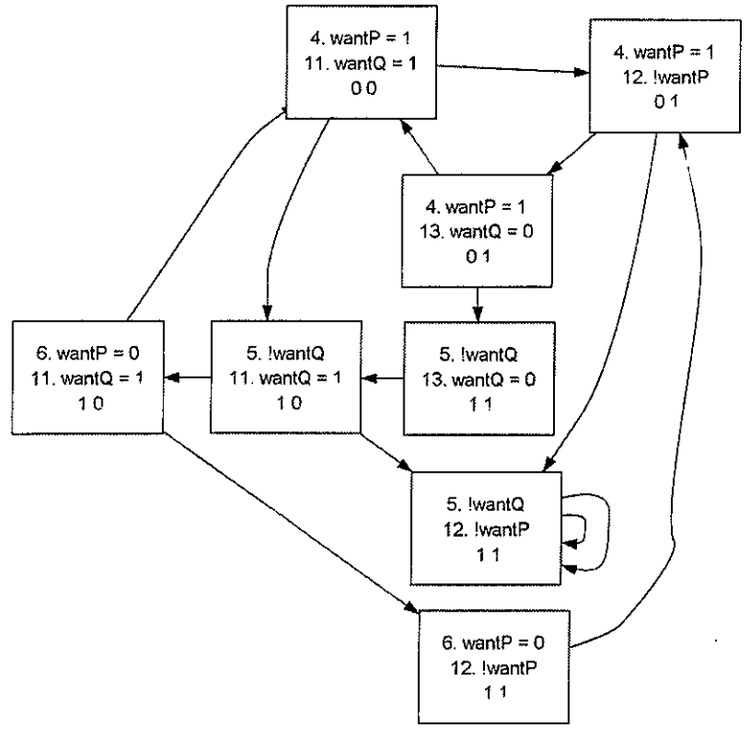
5.2.2 The semantics of LTL

The semantics, the meaning, of a syntactically correct formula is defined by giving it an *interpretation*: an assignment of truth values, T (true) or F (false), to its atomic propositions and the extension of the assignment to an interpretation of the entire formula according to the rules for the operators. For the propositional calculus these are given by the familiar *truth tables*, where A and B are any formulas:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

For temporal logic, the semantics of a formula is given in terms of computations and the states of a computation. The atomic propositions of temporal logic are boolean expressions that can be evaluated in a single state *independently* of a computation. For example, let *critical* be the value of the variable `critical` in a program for the critical section problem; the expression $critical \leq 1$ is an atomic proposition because it can be assigned a truth value in a state s just by checking the value of the variable `critical` in s. Similarly, if *csp* is a boolean expression that is true if and only if the location counter

Fig. 5.1. State diagram for the third attempt



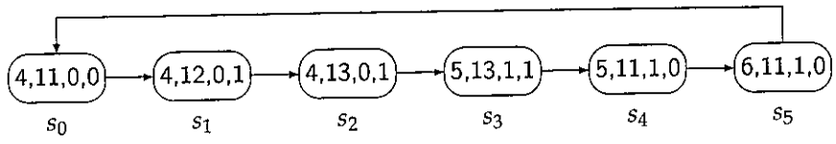
of process P is at the control point corresponding to the critical section of the process, then *csp* is an atomic proposition because it can be evaluated in any single state.

Atomic propositions can be combined using the operators of the propositional calculus; such formulas can also be evaluated just by checking values in a single state. For example, if *csq* is similar to *csp* but for process Q, the expression $\neg(csp \wedge csq)$ specifies that mutual exclusion holds *in the state in which it is evaluated.*

Consider again the program for the critical section problem in Listing 4.3 and its state diagram in Figure 4.1 (repeated in Figure 5.1). A computation of the program is an *infinite* sequence of states that starts in the initial state (`4. wantP=1, 11. wantQ=1, 0, 0`), and continues by taking legal transitions, which are the ones shown in the diagram. For example, here are the first few states of a computation:

s_0 = (`4. wantP=1, 11. wantQ=1, 0, 0`) ⟶
s_1 = (`4. wantP=1, 12. !wantP, 0, 1`) ⟶
s_2 = (`4. wantP=1, 13. wantQ=0, 0, 1`) ⟶
s_3 = (`5. !wantQ, 13. wantQ=0, 1, 1`) ⟶
s_4 = (`5. !wantQ, 11. wantQ=1, 1, 0`) ⟶
s_5 = (`6. wantP=0, 11. wantQ=1, 1, 0`) ⟶
s_6 = (`4. wantP=1, 11. wantQ=1, 0, 0`)

A computation, an infinite sequence of states, is obtained by repeating the same transitions indefinitely. Since the last state is the same as the first, the infinite sequence of states can be *finitely presented* by identifying the first and last states; that is, instead of creating a new state s_6, we create a transition from s_5 to s_0:[3]



Process P is in its critical section if its location counter is at line 6 and process Q is in its critical section if its location counter is at line 13. Let *csp* and *csq* be atomic propositions representing these properties. Clearly, *for the computation shown above*, the formula $\neg(csp \wedge csq)$ that expresses the correctness property of mutual exclusion is true in *all* its states.

We have shown that this formula is true for one specific computation, but since there are no states in which process P is at line 6 and process Q is at line 13, we can generalize and claim that the following statement is true:

> The formula $\neg(csp \wedge csq)$ is true in every state of every computation.

Let us now show how to express this property in LTL and how to verify that the property holds for the program in Listing 4.3.

[3] In this diagram, the values of the location counters are indicated by line numbers without the source code.

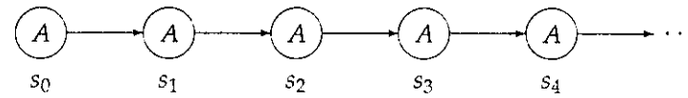
5.3 Safety properties

5.3.1 Expressing safety properties in LTL

> Let A be an LTL formula and let $\tau = (s_0, s_1, s_2, \ldots)$ be a computation. Then $\Box A$, read *always* A, is true in state s_i if and only if A is true *for all* s_j in τ such that $j \geq i$.

The operator is reflexive so if $\Box A$ is true in a state s, then A must also be true in s. The formula $\Box A$ is called a *safety property* because it specifies that the computation is safe in that nothing "bad" ever happens, or equivalently, that the only things that happen are "good."

We can draw a diagram of a computation, labeling each state s_i with A if A is true in s_i and with $\neg A$ if A is false in s_i. If the following diagram is extended indefinitely with all states labeled A, then $\Box A$ is true in s_0:



The correctness property of mutual exclusion can be expressed by the LTL formula $\Box\neg(csp \wedge csq)$. This is a safety property because it is true if something "bad" – $csp \wedge csq$, meaning the two processes in their critical section – never happens. Equivalently, the only states the computation enters are "good" ones in which $\neg(csp \wedge csq)$ is true.

Important: In linear temporal logic, each formula implicitly refers to *all* computations of a model. Therefore, when a correctness property for a model is specified by an LTL formula, it means that the property holds if the formula is true in *all* computations of the model. If SPIN finds even a single counterexample – a computation in which the formula is false – the correctness property does not hold for the model.

5.3.2 Expressing safety properties in PROMELA

For the program in Listing 4.3 we verified that mutual exclusion holds by writing **assert** statements in each critical section. It is also possible to express this property in LTL. As before, we declare a variable `critical`, incrementing it at the beginning of each critical section and decrementing it at the end:

```
active proctype P() {
 do
 :: wantP = true;
    !wantQ;
    critical++;
    critical--;
    wantP = false;
 od
}

/* Similarly for process Q */
```

Since the critical section in the abbreviated program is not explicitly written, the statement `critical--` immediately follows the statement `critical++`, and the critical section is the control point between them.

The symbol `mutex` is defined to represent an expression that is true if and only if mutual exclusion holds:

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

Mutual exclusion can now be specified in PROMELA by the LTL formula:

```
[]mutex
```

Alternatively, we could define two variables `csp` and `csq` of boolean type, and set these variables to indicate when a process is in its critical section:

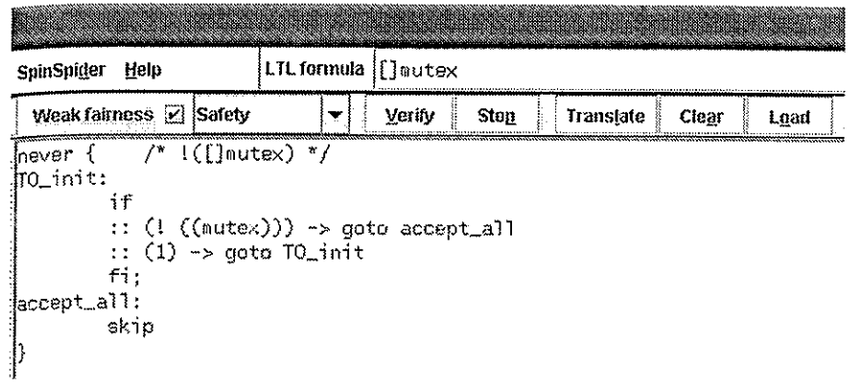
```
active proctype P() {
 do
 :: wantP = true;
    !wantQ;
    csp = true;
    csp = false;
    wantP = false;
 od
}

/* Similarly, for process Q */
```

The LTL formula expressing mutual exclusion is now

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

A third way of expressing this property that does not use ghost variables is described in Section 5.7.

Fig. 5.2. Verifying a safety property in JSPIN



5.3.3 Verifying safety properties in SPIN

This section shows how to verify safety properties in SPIN.

jSpin

The upper right corner of the JSPIN display contains the interface that is used to enter LTL formulas and to perform verifications (Figure 5.2). Write the LTL formula in the text field provided and select `Translate`. The formula is saved in a file with the same name as the PROMELA source file and with extension `prp`; then SPIN is called to translate the formula into a never claim, which is displayed in the right pane (though you may ignore it) and saved in a file with extension `ltl`.

Ensure that `Safety` is selected for the verification mode and select `Verify`. SPIN will perform the verification and display the result in the right pane. In this case, no errors are found so we have proved that mutual exclusion holds for this program.

There are three other buttons related to LTL formulas in the JSPIN interface. `Clear` clears the field for the LTL formula and ensures that subsequent verification runs will not use the contents of the field. `Load` brings up a file chooser to load an LTL formula from a `prp` file. `Stop` terminates a verification if it is taking too much time.

Command line

To verify a safety property, first add the *negation* of the LTL formula with the `-f` argument to the SPIN command that generates the veri-

fier. Then, compile the verifier with the -DSAFETY argument so that it is optimized for checking safety properties. Finally, run pan as usual:[4]

```
spin -a -f "![]mutex" third-safety.pml
gcc -DSAFETY -o pan pan.c
pan
```

As expected, no errors are reported.
Since the program will probably change more frequently than the correctness specification, you can save the LTL formula

```
![]mutex
```

in a one-line file `safety.prp` and include the file during the generation of the verifier using the -F argument:

```
spin -a -F safety.prp third-safety.pml
```

Alternatively, the translation of the LTL formula to a never claim can be saved in a file and this file included in the generation of the verifier using the -N argument:

```
spin -a -f "![]mutex"  > safety.ltl
spin -a -N safety.ltl third-safety.pml
```

If the verification seems to be taking too much time, you can terminate it as you would terminate any program (`ctrl-C`).

Warning

Atomic propositions in an LTL formula must be identifiers starting with a lower-case letter.

Furthermore, they must be boolean variables or defined as symbols for boolean-valued expressions.

Warning

Section 10.3 explains why the correctness specification must be negated. This is done automatically in JSPIN, but if you run SPIN from the command line, be sure to do it yourself.

[4] You may need to use single quotes instead of double quotes.

Warning

SPIN runs in a separate process that is forked from the process that runs JSPIN, so it is possible to terminate JSPIN without terminating SPIN. In that case, your computer may start to run slowly if SPIN is executing a long verification or a simulation of an infinite loop, and you will have to terminate SPIN manually.

In Windows this is done by pressing `ctrl-alt-del` to bring up the `Task Manager`, then selecting `Task List` and `Processes` and selecting `End Process` for each occurrence of `spin.exe` or `pan.exe`.

5.4 Liveness properties

Let A be a formula of LTL and let $\tau = (s_0, s_1, s_2, \ldots)$ be a computation. Then $\Diamond A$, read *eventually* A, is true in state s_i if and only if A is true *for some* s_j in τ such that $j \geq i$.

The operator is reflexive, so if A is true in a state s, then so is $\Diamond A$. The formula $\Diamond A$ is called a *liveness property* because it specifies that something "good" eventually happens in the computation.

If *csp* is the atomic proposition that is true in a state if process P is in its critical section, then $\Diamond csp$ holds if and only if process P eventually enters its critical section.[5]

It is essential that correctness specifications contain liveness properties because a safety property is vacuously satisfied by an empty program that does nothing! For example, a solution to the critical section problem in which neither process tries to enter its critical section trivially fulfils the correctness properties of mutual exclusion and absence of deadlock:

```
start:
 do
 :: printf("Noncritical section\n");
    goto start;
    wantP = true; /* Try to enter the critical section */
    printf("Critical section\n")
 od
```

[5] A better way of specifying absence of starvation is presented in Section 5.9.2.

5.4.1 Expressing liveness properties in SPIN

Listing 5.1 shows a program for the critical section problem.[6] We leave it to the reader to verify that both mutual exclusion and absence of deadlock hold. Unfortunately, this program is not fully correct because starvation may occur, that is, there is a computation in which process P never enters its critical section:

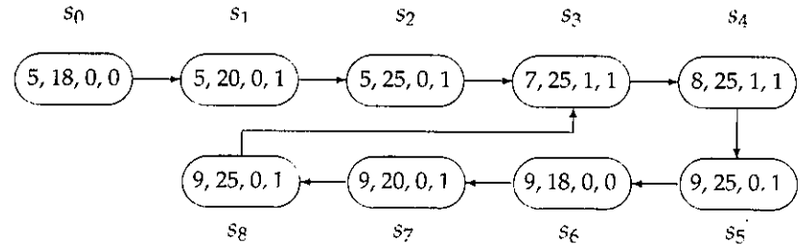
Listing 5.1. Critical section with starvation

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: wantP = true;
6       do
7       :: wantQ ->
8            wantP = false;
9            wantP = true
10      :: else -> break
11      od;
12      wantP = false
13   od
14 }
15
16 active proctype Q() {
17   do
18   :: wantQ = true;
19      do
20      :: wantP ->
21           wantQ = false;
22           wantQ = true
23      :: else -> break
24      od;
25      wantQ = false
26   od
27 }
```

[6] This is the *fourth attempt* described in Section 3.8 of *PCDP*.

s_0 = (5. wantP=1, 18. wantQ=1, 0, 0) ⟶
s_1 = (5. wantP=1, 20. wantP, 0, 1) ⟶
s_2 = (5. wantP=1, 25. wantQ=0, 0, 1) ⟶
s_3 = (7. wantQ, 25. wantQ=0, 1, 1) ⟶
s_4 = (8. wantP=0. 25. wantQ=0. 1, 1) ⟶
s_5 = (9. wantP=1, 25. wantQ=0, 0, 1) ⟶
s_6 = (9. wantP=1, 18. wantQ=1, 0, 0) ⟶
s_7 = (9. wantP=1, 20. wantP, 0, 1) ⟶
s_8 = (9. wantP=1, 25. wantQ=0, 0, 1) ⟶
s_9 = (7. wantQ, 25. wantQ=0, 1, 1)

Since state s_9 is the same as state s_3, they can be identified and the sequence of states extended to an infinite computation:



The critical section of process P (line 12) does not appear in any state of this computation, demonstrating that absence of starvation does not hold for this program.

5.4.2 Verifying liveness properties in SPIN

Add the statements

```
csp = true;
csp = false;
```

between lines 11 and 12 of the program in Listing 5.1; then the LTL formula <>csp expresses absence of starvation for process P. The verification of the temporal formula is carried out in a manner similar to that of the safety property, except that it must be performed in a mode called searching for *acceptance cycles* (Section 10.3.2). *Weak fairness*, explained in Section 5.5, must also be specified when this program is verified.

jSpin

Select Acceptance instead of Safety from the pulldown menu, and ensure that the box labeled Weak fairness is checked. Select Verify.

Command line

Run the verifier with the -a (*a*cceptance) argument and the -f (weak *f*airness) argument:[7]

```
spin -a -f "!<>csp" fourth-liveness.pml
gcc -o pan pan.c
pan -a -f
```

Liveness does not hold for this program; the error message is

```
pan: acceptance cycle (at depth 14)
```

For safety properties, a counterexample consists of one state where the formula is false, but for a liveness property, a counterexample is an infinite computation in which something good – in this case, csp becomes true – never happens. To produce the counterexample, run a guided simulation with the trail. The output from jSPIN is:

```
Process Statement          wantQ      wantP
1 Q     wantQ = 1
1 Q     wantQ = 0          1
0 P     wantP = 1          0
1 Q     wantQ = 1          1          0
1 Q     wantP              1          1
0 P     wantQ              1          1
<<<<<START OF CYCLE>>>>>
1 Q     wantQ = 0          1          1
1 Q     wantQ = 1          1          0
1 Q     wantP              1          1
0 P     wantP = 0          1          1
1 Q     wantQ = 0          0          1
1 Q     wantQ = 1          0          0
0 P     wantP = 1          0          1
0 P     wantQ              1          1
1 Q     wantQ = 0          1          1
0 P     wantP = 0          1          0
1 Q     wantQ = 1          0          0
1 Q     wantQ = 0          0          1
0 P     wantP = 1          0          0
1 Q     wantQ = 1          1          0
1 Q     wantP              1          1
0 P     wantQ              1          1
spin: trail ends after 50 steps
```

The line START OF CYCLE indicates that the subsequent states form a cycle that can be repeated indefinitely. Since a variable appears in the SPIN output only when it is assigned to, the absence of a value for csp means that the variable has never been assigned to and hence that starvation occurs in this computation.

Advanced: Finding the shortest counterexamples

SPIN did not find the *shortest* counterexample. That is because SPIN performs a depth-first search of the state diagram and stops with the first counterexample it finds. The -i and -I arguments to pan can be used to perform an iterated search for shorter counterexamples; see pages 24–25 of *SMC* for details.

5.5 Fairness

Consider again the program for the critical section problem in Listing 5.1. Is the following computation a counterexample for the property of absence of starvation?

$$
\begin{aligned}
s_0 &= \texttt{(5. wantP=1, 18. wantQ=1, 0, 0)} \longrightarrow \\
s_1 &= \texttt{(5. wantP=1, 20. wantP, 0, 1)} \longrightarrow \\
s_2 &= \texttt{(5. wantP=1, 25. wantQ=0, 0, 1)} \longrightarrow \\
s_3 &= \texttt{(5. wantP=1, 18. wantQ=1, 0, 0)}
\end{aligned}
$$

State s_3 is identical to s_0, so an infinite computation can be composed from just the three states s_0, s_1, s_2. In this computation, process Q enters its critical section repeatedly, while process P never executes any of its statements. The computation *is* a counterexample to a claim that <>csp is true, but it is unsatisfactory because it doesn't give process P a "fair" chance to try to enter its critical section. This concept can be formalized by the following definition:[8]

> A computation is *weakly fair* if and only if the following condition holds: if a statement is *always* executable, then it is *eventually* executed as part of the computation.

[7] Ensure that the -DSAFETY argument is not used in the compilation.

[8] There is also a concept called *strong fairness*; see Section 2.7 of *PCDP*.

The computation described above is not weakly fair: Although like all assignment statements, `5. wantP = true` is always executable, it is never executed in the computation. As we have shown, absence of starvation does not, in fact, hold for the program in Listing 5.1, but it seems reasonable to require that only fair computations be considered as counterexamples.

jSpin

Ensure that the box labeled `Weak fairness` is checked before selecting `Verify`. (This is the default.)

Command line

Add the argument `-f` (in addition to the argument `-a`) when executing the verifier pan.

Warning

Restricting verification to computations that are weakly fair requires a lot of memory. By default, SPIN limits the number of processes to two in a verification with fairness; if there are more processes, you need to compile the verifier with a higher value for the parameter `-DNFAIR=n`.

We conclude this section with an example of a program whose properties depend critically on fairness (Listing 5.2). The assignment in process `Q` is always enabled, so in a weakly fair computation it will eventually be executed, causing the loop in process `P` to terminate. If weak fairness is *not* specified, there is a nonterminating computation in which the **do**-statement is executed indefinitely. Thus the correctness property "the program always terminates" holds if and only if computations are required to be weakly fair.

5.6 Duality

The operators $\Box$ and $\Diamond$ are *dual* in a manner similar to the duality expressed by deMorgan's laws:

$$\neg(p \wedge q) \equiv (\neg p \vee \neg q), \quad \neg(p \vee q) \equiv (\neg p \wedge \neg q).$$

Passing a negation through a unary temporal operator changes the operator to the other one:

$$\neg \Box p \equiv \Diamond \neg p, \quad \neg \Diamond p \equiv \Box \neg p.$$

Listing 5.2. Termination under weak fairness

```
1  int n = 0;
2  bool flag = false;
3
4  active proctype P() {
5    do
6    :: flag -> break
7    :: else -> n = 1 - n
8    od
9  }
10
11 active proctype Q() {
12   flag = true
13 }
```

Since double negations cancel out, duality can be used to simplify formulas with temporal operators. Let *good* and *bad* be atomic propositions such that *good* is equivalent to $\neg$*bad*. Then we have the following equivalences:

$$\neg \Box good \equiv \Diamond \neg good \equiv \Diamond \neg\neg bad \equiv \Diamond bad,$$
$$\neg \Diamond good \equiv \Box \neg good \equiv \Box \neg\neg bad \equiv \Box bad.$$

These make sense when read out loud: if it is false that something good is always true, then eventually something bad must happen; if it is false that something good eventually happens, then something bad is always true.

It is important to get used to reasoning with the duality of the temporal operators because negations of correctness specifications are at the foundation of model checking (Section 10.3).

5.7 Verifying correctness without ghost variables*

We have used ghost variables like `critical` and `csp` as proxies for control points in a PROMELA program. While this causes no problems in the small programs shown in the book, when modeling large systems you will want to keep the number of variables as small as possible. Ghost variables also unnecessarily complicate graphical representations of the state transition diagrams that are generated by the SPINSPIDER tool (Appendix A.3).

PROMELA supports *remote references* that can be used to refer to control points in correctness specifications, either directly within never claims or in

LTL formulas. For example, in a program for the critical section problem, we can replace the ghost variables by defining labels cs at the control points corresponding to the critical sections of the two processes and then defining a symbol that expresses mutual exclusion using remote references:

```
#define mutex !(P@cs && Q@cs)

active proctype P() {
  do
  :: wantP = true;
     !wantQ;
cs:  wantP = false;
  od
}

/* Similarly for process Q */
```

The expression P@cs returns a nonzero value if and only if the location counter of process P is at the control point labeled by cs. Mutual exclusion holds only if both P@cs and Q@cs cannot be true at the same time, expressed as []mutex. A verification run shows that this formula does indeed hold.

It is also possible to refer to the value of a local variable of a process using the syntax process:variable.

> **Warning**
>
> A remote reference is not a *symbol* so it cannot appear directly within an LTL formula. It can appear in a boolean expression for which a symbol is defined as shown above.

5.8 Modeling a noncritical section*

One of the correctness properties of the critical section problem is that a process be able to enter its critical section infinitely often even if another process fails in its *noncritical* section. This can be modeled in PROMELA by including a nondeterministic if-statement in a process that is allowed to fail.

The program in Listing 5.3 is a solution to the critical section problem that achieves mutual exclusion.[9] This can be checked by verifying the safety property shown in Section 5.7: define the symbol mutex as !(P@cs && Q@cs) and verfiy []mutex.

[9] This is the *first attempt* described in Chapter 3 of *PCDP*.

Lines 5–8 model the noncritical section: P can nondeterministically choose to do nothing (line 6) or to fail by blocking until **false** becomes true, which, of course, will never occur (line 7).

The program in Listing 5.3 is not a correct solution to the critical section problem, because if process P fails in its noncritical section (by blocking at line 7), process Q will eventually become blocked indefinitely waiting for turn == 2 to become true (line 16).

Listing 5.3. Modeling failure in the noncritical section

```
1  byte  turn = 1;
2
3  active proctype P() {
4    do
5    :: if
6       :: true
7       :: true -> false
8       fi;
9       turn == 1;
10 cs:  turn = 2
11   od
12 }
13
14 active proctype Q() {
15   do
16   :: turn == 2;
17 cs:  turn = 1
18   od
19 }
```

Now add an **if**-statement like the one in lines 5–8 to one of the processes of a correct solution to the critical section problem: Dekker's algorithm (Algorithm 3.10 of *PCDP*) or Peterson's algorithm (Listing 5.4 at the end of this chapter). Define the symbol live as Q@cs and verify the absence of starvation: []<>live. Process P fails only when wantP is false, so process Q can continue entering its critical section infinitely often because the expression at line 17 always evaluates to true regardless of the value of the variable last.

5.9 Advanced temporal specifications*

The temporal operators $\Box$ and $\Diamond$ can be applied to any formula of LTL, so that $\Box\Diamond\Diamond A$ and $\Diamond\Box\Diamond(A \wedge \Box B)$ are syntactically correct. It is beyond the scope of this book to present the deductive theory of LTL: axioms, rules of inferences, and theorems relating to properties of formulas such as associativity and commutivity (see *MLCS*, Chapter 12). We just mention two results:

- A formula with sequences of consecutive occurences of the operators $\Box$ or $\Diamond$ is equivalent to one in which the sequences are collapsed to a single occurrence of the operator. For example, $\Box\Box\Diamond\Diamond A$ is equivalent to $\Box\Diamond A$.
- A formula with any sequence of alternate occurences of the operators $\Box$ and $\Diamond$ is equivalent to one in which the sequence is collapsed into one of the two-operator sequences $\Box\Diamond$ or $\Diamond\Box$. For example, $\Diamond\Box\Diamond A$ is equivalent to $\Box\Diamond A$.

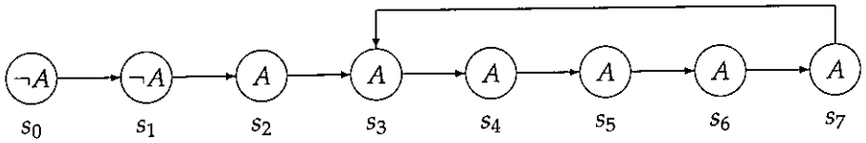
Thus, any sequence of unary temporal operators can be collapsed into a sequence of one or two operators.

The next two subsections describe the use of two-operator sequences in formulas expressing commonly used correctness specifications. This is followed by two subsections on the binary temporal operator $\mathcal{U}$ and a final subsection on the *next* operator, which is rarely used in SPIN.

Temporal logic formulas with more than two or three operators are difficult to understand. To help write correctness specifications in temporal logic a set of *patterns* has been developed at Kansas State University. The patterns are classified by properties such as *precedence* and *existence*, as well as by scope such as *before* and *between*. Formulas are given not just for the linear temporal logic used in SPIN but also for other logics used in verification. The address of the website of this project is given in Appendix B.

5.9.1 Latching

The formula $\Diamond\Box A$ expresses a *latching* property: A may not be true initially in a computation, but eventually it becomes true and remains true:

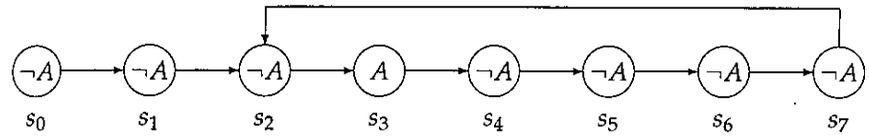


The formula $\Diamond\Box A$ is true in s_0: Although A is not true in s_0 or s_1, it becomes true in s_2 and remains true in all subsequent states of the computation.

Latching is important because it is unusual for a property to be true initially and always; rather, some statements must be executed to make the property true, although once it becomes true, the property remains true. Latching can also express properties that relate to exceptional situations. For example, suppose that a multiprocessor system is designed so that if a processor fails it automatically sets its variables to zero. Then for the program in Listing 5.1, we could claim $\Diamond fails_Q \rightarrow \Diamond\Box\neg wantQ$, that is, if ever the processor executing process Q fails, the value of wantQ is latched to false.[10] From this we can deduce that process P will not be starved even if Q fails because eventually the guard wantQ in line 7 will always be false and the **else**-alternative in line 10 can be taken.

5.9.2 Infinitely often

The formula $\Box\Diamond A$ expresses the property that A is true *infinitely often*: A need not always be true, but at *any* state in the computation s, A will be true in s or in some state that comes after s:



It is easy to see that A is true in the states $s_3, s_9, s_{15}, \ldots$, so at any state s_i, A is true in one of the states $s_i, s_{i+1}, s_{i+2}, s_{i+3}, s_{i+4}, s_{i+5}$.

For solutions to the critical section problem, liveness means not just that a process can enter its critical section, but that it can enter its critical section repeatedly. This can be modeled in PROMELA as follows. First, after setting a variable that indicates that P is in its critical section, we immediately reset it to indicate that P has left its critical section:

```
active proctype P() {
  do
  :: /* Try to enter critical section */
     csp = true;
     csp = false;
     /* Leave critical section */
  od
}
```

[10] wantQ can be considered to "belong" to process Q because it is only assigned to in Q, while process P only reads its value.

Then – if the algorithm is free from starvation – we can verify the program for the temporal formula `[]<>csp`.

5.9.3 Precedence

The operators $\Box$ and $\Diamond$ are unary and cannot express properties that relate two points in time, such as the *precedence* property that requires that A become true before B becomes true. This can be expressed with the binary operator $\mathcal{U}$ called *until* and written `U` in SPIN:

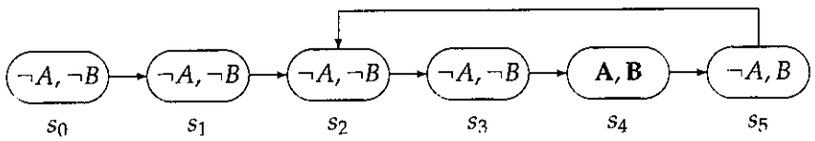
$$\neg B \,\mathcal{U}\, A.$$

Read this as: B remains false until A becomes true. More formally:

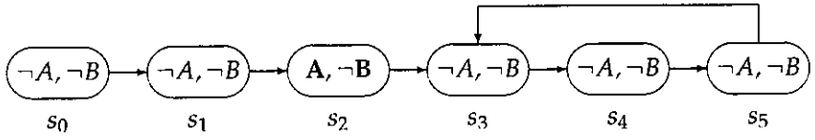
> $p\,\mathcal{U}\,q$ is true in state s_i of a computation τ if and only if there is some state s_k in τ with $k \geq i$, such that q is true in s_k, and for all s_j in τ such that $i \leq j < k$, p is true in s_j.

If q is already true in s_i, the second requirement is vacuous.

The formula $\neg B\,\mathcal{U}\,A$ is true in s_0 of the following computation because B remains false as long as A does; only in s_4, when A becomes true, does B also become true:



Note that B need not be true in s_4, because we are only interested in specifying that it remain false until A becomes true. In fact, B can be false throughout the entire computation, and the truth of A beyond its first true occurrence is irrelevant; it follows that $\neg B\,\mathcal{U}\,A$ is true in s_0 of the following computation:



The operator $\mathcal{U}$ is called the *strong until* operator, because the subformula to the right of $\mathcal{U}$ is required to become true eventually. In fact $\Diamond q$ can be defined as $true\,\mathcal{U}\,q$. Since *true* is trivially true, $true\,\mathcal{U}\,q$ is true if and only if q eventually becomes true.

There is a *weak until* operator $\mathcal{W}$ that does not require that the right subformula eventually become true. The two operators are related as follows:

$$p\,\mathcal{U}\,q \equiv p\,\mathcal{W}\,q \wedge \Diamond q, \qquad p\,\mathcal{W}\,q \equiv p\,\mathcal{U}\,q \vee \Box p.$$

Warning

SPIN does not have the weak until operator $\mathcal{W}$.

Advanced: The V operator

SPIN has an operator `V` that is defined so that `pVq` is equivalent to `!( (!p) U (!q) )`. The operator `V` is *not* the same as $\mathcal{W}$; if it were, the corresponding formula would be `!( (!q) U (!p && !q) )`.

5.9.4 Overtaking

We will demonstrate the use of the $\mathcal{U}$ operator to specify *one-bounded overtaking* in Peterson's algorithm (Listing 5.4), a correct solution to the critical section problem. One-bounded overtaking means that if process `P` tries to enter its critical section, process `Q` can enter its critical section at most once before `P` does.

Let us define the symbols:

```
#define   ptry   P@try
#define   qcs    Q@cs
#define   pcs    P@cs
```

If process `P` is not in its critical section, it is *not* true that `csq` is false, and it is certainly not true that `csq` remains false until `P` enters its critical section. First, process `Q` may currently be in its critical section, but even if it isn't, it may *overtake* process `P` and enter its critical section first.

One-bounded overtaking is expressed by the LTL formula [17, p. 265]:

```
[]( ptry -> ( !qcs U ( qcs U (!qcs U pcs) ) ) )
```

A nested *until* formula of this form expresses the property that a sequence of intervals must satisfy successive subformula. The formula above expresses the property that, *always*, if process `P` is trying to enter its critical section (`ptry` is true), the computation must start with the following sequence of intervals: (a) process `Q` is not in its critical section (`!qcs`); (b) process `Q` is in its critical section (`qcs`); (c) again, process `Q` is not in its critical section (`!qcs`); and finally (d) process `P` is in its critical section (`pcs`).

According to the definition of the $\mathcal{U}$ operator, the intervals may be empty, but the correctness of this property ensures that there cannot be two *separate* intervals where `qcs` is true before the state where `pcs` becomes true.

Run a verification of the program in Listing 5.4 for this formula and show that one-bounded overtaking holds.

Listing 5.4. Peterson's algorithm

```
1  bool  wantP, wantQ;
2  byte  last = 1;
3
4  active proctype P() {
5    do
6    ::  wantP = true;
7        last = 1;
8  try:  (wantQ == false) || (last == 2);
9  cs:   wantP = false
10   od
11 }
12
13 active proctype Q() {
14   do
15   ::  wantQ = true;
16       last = 2;
17 try:  (wantP == false) || (last == 1);
18 cs:   wantQ = false
19   od
20 }
```

Warning

The operator U is defined in SPIN to be left-associative, while the operator $\mathcal{U}$ is defined to be right-associative in [17, p. 48], the definitive reference on linear temporal logic. Right-associativity is more natural because it corresponds to a sequence of intervals, as explained in [17].

To avoid confusion, use parentheses liberally!

Advanced: Bounded overtaking with weak until

Normally, one-bounded overtaking would be verified using a formula with the weak until operator $\mathcal{W}$ rather than $\mathcal{U}$. This is because in the critical section problem a process need not attempt to execute its critical section, so csq may never become true. Since $\mathcal{W}$ is not implemented in SPIN, we have used $\mathcal{U}$; the verification still works because the algorithm in Listing 5.4 does not model a process that remains in its noncritical section.

5.9.5 Next

Temporal logic contains an additional unary operator $\mathcal{X}$, called *next* and written X in SPIN. $\mathcal{X}A$ is true in a state s_i of a computation if A is true in the following state s_{i+1}. The operator is of limited usefulness for two reasons. First, the usual model of a concurrent or distributed system abstracts away from the concept of time. For example, in a client-server system, we want to specify that a client process *eventually* receives a service from a server process, but it doesn't really matter if that occurs in the next state or ten states later. The modeling of real-time systems in which time does matter is discussed in the case studies in Sections 11.3 and 11.4.

The second reason for avoiding $\mathcal{X}$ is that without this operator temporal logic formulas are *stutter invariant*. This means that any correctness specification that is true in a computation remains true if duplicate consecutive states are removed to form a more concise computation. The algorithms in SPIN are more efficient if stutter-invariant specifications are used. In fact you may have noticed the following message from SPIN:

```
warning: for p.o. reduction to be valid
  the never claim must be stutter-invariant
(never claims generated from LTL formulae
  are stutter-invariant)
```

`p.o. reduction` refers to *partial order reduction*, which is one of the main optimizations that SPIN is able to perform (Section 10.2.) As the last lines of the message note, if you limit yourself to writing correctness specifications in LTL, you need not worry about affecting this optimization.

For more on these topics see *SMC* Chapter 6.