AN OVERVIEW OF PROMELA 3

"What we see depends on mainly what we look for."
(Sir John Lubbock, 1834–1913)

In the last chapter we saw that the emphasis in PROMELA models is placed on the coordination and synchronization aspects of a distributed system, and not on its computational aspects. There are some good reasons for this choice. First, the design and verification of correct coordination structures for distributed systems software tends to be much harder in practice than the design of a non-interactive sequential computation, such as the computation of compound interest or square roots. Second, the curious situation exists that the logical *verification* of the interaction in a distributed system, though often computationally expensive, can be done more thoroughly and more reliably today than the verification of even the simplest computational procedure. The specification language we use for systems verification is therefore deliberately designed to encourage the user to abstract from the purely computational aspects of a design, and to focus on the specification of process interaction at the system level.

As a result of this specialization, PROMELA contains many features that are not found in mainstream programming languages. These features are intended to facilitate the construction of high-level models of distributed systems, The language supports, for instance, the specification non-deterministic control structures; it includes primitives for process creation, and a fairly rich set of primitives for interprocess communication. The other side of the coin is that the language also lacks some features that are found in most programming languages, such as functions that return values, expressions with side effects, data and functions pointers, etc. The reason is simple: PROMELA is not a programming language. PROMELA is a language for building verification models.

A verification model differs in at least two important ways from a program

written in a mainstream programming language such as Java or C.

- A verification model represents an *abstraction* of a design that contains only those aspects of a system that are relevant to the properties one wants to verify.
- A verification model often contains things that are typically not part of an implementation. It can, for instance, include worst-case assumptions about the behavior of the *environment* that may interact with the modeled system, and, most importantly, it either explicitly or implicitly contains a specification of *correctness properties*.

Even though it can be attractive to have a single specification that can serve as both a verification model and as an implementation of a system design — verification and implementation have some fundamentally different objectives. A verification model is comparable in its purpose to the prototype or design model that a civil engineer might construct: it serves to prove that the design principles are sound. Design models are normally not expected to be part of the final implementation of a system.

A full system implementation typically contains more information, and far more detail, than a design model. This means that it can be difficult to find automatic procedures for converting design models into system implementations. The reverse, however, is not necessarily true. In Chapter 10 we will explore means for mechanically extracting the main elements of a verification model directly from an implementation, guided by abstraction techniques. Similarly, in Chapter 17 we will discuss the specific constructs that are available in PROMELA to facilitate model extraction tools. These topics, though, should be considered advanced use of the model checker, so we will conveniently ignore them for now.

In the last chapter we gave a bird's-eye view of the language, briefly touching on some of the main language constructs that are available to build verification models. In this chapter we cover the language more thoroughly. We will try to cover all main language features in a systematic way, starting with the most general constructs, and slowly descending into more of the specifics. We restrict ourselves here to the mechanisms that are at our disposal for describing process behavior and process interaction. In the next chapter we will continue the discussion with a description of the various means we have to define correctness claims. After we have covered these basics, we move on in Chapter 5 to discuss methods for exploiting design abstraction techniques as an aid in the control of verification complexity.

First then, our overview of the basic language for specifying the behavior of concurrently executing, and potentially interacting, processes in a distributed system.

TYPES OF OBJECTS

PROMELA derives many of its notational conventions from the C programming language. This includes, for instance, the syntax for boolean and arithmetic operators, for assignment (a single equals) and equality (a double equals), for variable and parameter declarations, variable initialization and comments, and the use of curly braces to indicate the beginning and end of program blocks. But there are also important differences, prompted by the focus in PROMELA on the construction of high-level models of the interactions in distributed systems.

A PROMELA model is constructed from three basic types of objects:

- Processes
- Data objects
- Message channels

Processes are instantiations of `proctype`s, and are used to define behavior. There must be at least one `proctype` declaration in a model, and for the model to be of much use there will normally also be at least one process instantiation.

A `proctype` body consists of zero or more data declarations, and one or more statements. The semantics of statement execution is somewhat special in PROMELA, since it also doubles as the primary mechanism for enforcing process synchronizations. We have seen some of this in the last chapter, and we will return to it in more detail in the section on *executability* (p. 51).

Process types are always declared globally. Data objects and message channels can be declared either globally, that is, outside all process type declarations, or locally, that is, within a process type declaration. Accordingly, there are only two levels of scope in PROMELA: global and process local. It is, for instance, not possible to restrict the scope of a global object to only a subset of the processes, or to restrict the scope of a local object to only part of a `proctype` body.

The next three sections contain a more detailed discussion of each of the three basic types of objects in PROMELA. This is followed by a discussion of PROMELA's rules for executability, and a more comprehensive overview of the primitives in PROMELA for defining flow of control.

PROCESSES

In the last chapter we saw that we can declare and instantiate processes by prefixing a `proctype` declaration with the keyword `active`. There are several ways to instantiate processes in PROMELA. We can create multiple instantiations of a given `proctype` by adding the desired number in square brackets to the `active` prefix, for instance as follows:

```
active [2] proctype you_run()
{
        printf("my pid is: %d\n", _pid)
}
```

Each running process has a unique process instantiation number. These instantiation numbers are always non-negative, and are assigned in order of creation, starting at zero for the first created process. Each process can refer to its own instantiation number via the predefined local variable `_pid`. Simulating the example above, for instance, produces the following output:

```
$ spin you_run.pml
my pid is: 0
        my pid is: 1
2 processes created
```

The two processes that are instantiated here each print the value of their process instantiation number and then terminate. The two lines of output happen to come out in numeric order here, but since process execution is asynchronous, it could just as well have been the opposite. By default, during simulation runs, SPIN arranges for the output of each active process to appear in a different column: the *pid* number is used to set the number of tab stops used to indent each new line of output that is produced by a process.[1]

There is also another way to instantiate new PROMELA processes. Any running process can start other processes by using a predefined operator called `run`. For instance, we could rewrite the last example as follows:

```
proctype you_run(byte x)
{
        printf("x = %d, pid = %d\n", x, _pid)
}

init {
        run you_run(0);
        run you_run(1)
}
```

A disadvantage of this solution is that it often creates one process more than strictly necessary (i.e., the `init` process). For simulation or implementation, the extra process would not matter too much, but in system verification we usually take every possible precaution to keep the system descriptions at a minimum: avoiding all unnecessary elements.

A simulation run of the last model produces the following result:

1. We can now see that the string *hello world* in the last chapter was printed left justified by a happy coincidence. It was because the process executing the statement had *pid* zero. We can suppress the default indentations by invoking `spin` with option `-T` (see p. 513).

```
$ spin you_run2.pml
                x = 1, pid = 2
        x = 0, pid = 1
3 processes created
```

In this version of the `proctype you_run`, we added a parameter of type `byte`. This formal parameter is initialized in the `run` statement, which appears here in the `init` process. This means that when the ''execution'' of a `run` statement results in the creation of a new process, all formal parameters from the target `proctype` declaration are initialized to the values of the corresponding actual parameters that are provided in the `run` statement (i.e., parameter passing is by value).

Parameter values, of course, cannot be passed to the `init` process, or to processes that are instantiated as `active proctype`s. If processes created through the use of an `active` prefix have formal parameters, they are treated as if they were local variables, and they are initialized to zero. This initialization rule matches the rule for all data objects in PROMELA: if no explicit initialization is present, an object is always initialized to zero.

A newly created process may, *but need not*, start executing immediately after it is instantiated. Similarly, the new process may, but need not and generally will not, terminate before the process that created it moves on to its next statement. That is: processes do not behave like functions. Each process, no matter how it is created, defines an asynchronous thread of execution that can interleave its statement executions in arbitrary ways with other processes.

We mentioned in passing that `run` is really an *operator*, and therefore technically what so far we have casually referred to as a `run` ''statement'' is really an *expression*. Technically again, the expression is not ''executed'' but evaluated. The `run` expression is the only type of expression that can have a side effect when it evaluates to non-zero, but not when it evaluates to zero (i.e., when it fails to instantiate a new process). A run expression is also special in the sense that it can contain only one `run` operator and cannot be combined with any other conditionals.

The value of a `run` expression evaluates to zero if no process can be instantiated, otherwise it evaluates to a non-zero value which equals the process instantiation number of the newly created process. Note that the *pid* returned upon successful process instantiation can never itself be zero, because there must be at least one process running to evaluate the expression. Evaluating a `run` expression, then, produces a value of type `pid` (cf. p. 16, 36).

Because `run` is an operator, we can also change the definition of `init` in the last example into the following version, where the process instantiation numbers are stored in local variables.

```
init {  pid p0, p1;

        p0 = run you_run(0);
        p1 = run you_run(1);
        printf("pids: %d and %d\n", p0, p1)
}
```

Simulating the execution of this model produces:

```
$ spin you_run2.pml
        x = 1, pid = 2
pids: 1 and 2
                x = 0, pid = 1
3 processes created
```

Note that the output from the three processes can again appear in any order because of the concurrent nature of the executions.

Finiteness: Why would evaluating a `run` expression ever *fail* to instantiate a new process, and return zero? The reason lies in the fact that a PROMELA model can only define finite systems. Enforcing that restriction helps to guarantee that any correctness property that can be stated in PROMELA is decidable. It is impossible to define a PROMELA model for which the total number of reachable system states can grow to infinity. Data objects can only have a finite range of possible values; there can be only finitely many active processes, finitely many message channels, and every such channel can have only finite capacity. The language does not prescribe a precise bound for all these quantities, other than that there is such a bound and that it is finite. For all currently existing versions of SPIN, the bound on the number of active processes and the bound on the number of message channels is put at 255.

An attempt to ignore these bounds will necessarily fail. For instance, we could try to define the following model:

```
active proctype splurge(int n)
{       pid p;
        printf("%d\n", n);
        p = run splurge(n+1)
}
```

Simulating the execution of this model with SPIN, using the `-T` option to disable the default indentation of `printf` output, produces the following result:

```
$ spin -T splurge.pml
0
1
2
3
...
```

```
252
253
254
spin: too many processes (255 max)
255 processes created
```

The creation of the 256th process fails (note that the process numbering start at zero) and ends the simulation run. But there are more interesting things to discover here, not just about how processes are instantiated, but also about how they can terminate and die. Process termination and process death are two distinct events in PROMELA.

- A process ‘‘terminates’’ when it reaches the end of its code, that is, the closing curly brace at the end of the `proctype` body from which it was instantiated.
- A process can only ‘‘die’’ and be removed as an active process if all processes that were instantiated later than this process have died first.

Processes can *terminate* in any order, but they can only *die* in the reverse order of their creation. When a process reaches the end of its code this only signifies process *termination*, but not process *death*. When a process has terminated, this means that it can no longer execute statements, but will still be counted as an active process in the system. Specifically, the process *pid* number remains associated with this process and cannot be reused for a new process. When a process dies, it is removed from the system and its *pid* can be reused for another process.

This means that each instantiation of the `proctype splurge` in the last example *terminates* immediately after it creates the next process, but none of these processes can *die* until the process creation fails for the first time on the 255th attempt. That last process is the first process that can die and be removed from the system, since it is the most recently created process in the system. Once this happens, its immediate predecessor can die, followed by its predecessor, and all the way back to the first created process in stack order, until the number of active processes drops to zero, and the simulation ends.

PROVIDED CLAUSES

Process execution is normally only guided by the rules of synchronization captured in the statement semantics of `proctype` specifications. It is possible, though, to define additional global constraints on process executions. This can be done with the help of the keyword `provided` which can follow the parameter list of a `proctype` declaration, as illustrated in the following example:

```
bool    toggle = true;          /* global variables   */
short   cnt;                    /* visible to A and B */

active proctype A() provided (toggle == true)
{
L:      cnt++;          /* means: cnt = cnt+1 */
        printf("A: cnt=%d\n", cnt);
        toggle = false; /* yield control to B */
        goto L          /* do it again */
}

active proctype B() provided (toggle == false)
{
L:      cnt--;          /* means: cnt = cnt-1 */
        printf("B: cnt=%d\n", cnt);
        toggle = true;  /* yield control to A */
        goto L
}
```

The `provided` clauses used in this example force the process executions to alternate, producing an infinite stream of output:

```
$ spin toggle.pml | more
A: cnt=1
        B: cnt=0
A: cnt=1
        B: cnt=0
A: cnt=1
        B: cnt=0
A: cnt=1
...
```

A process cannot take any step unless its `provided` clause evaluates to *true*. An absent `provided` clause defaults to the expression *true*, imposing no additional constraints on process execution.

Provided clauses can be used to implement non-standard process scheduling algorithms. This feature can carry a price-tag in system verification, though. The use of `provided` clauses can disable some of SPIN's most powerful search optimization algorithms (cf. Chapter 9).

DATA OBJECTS

There are only two levels of scope in PROMELA models: global and process local. Naturally, within each level of scope, all objects must be declared before they can first be referenced. Because there are no intermediate levels of scope, the scope of a global variable cannot be restricted to just a subset of processes, and the scope of a process local variable cannot be restricted to specific blocks of statements. A local variable can be referenced from its point of declaration to the end of the `proctype` body in which it appears,

Table 3.1 Basic Data Types

Type	Typical Range
bit	0,1
bool	*false*,*true*
byte	0..255
chan	1..255
mtype	1..255
pid	0..255
short	-2^{15} .. $2^{15}-1$
int	-2^{31} .. $2^{31}-1$
unsigned	0 .. $2^{n}-1$

even when it appears in a nested block (i.e., a piece of code enclosed in curly braces). This is illustrated by the following example:

```
init {
        /* x declared in outer block */
        int x;
        {       /* y declared in inner block */
                int y;
                printf("x = %d, y = %d\n", x, y);
                x++;
                y++;
        }
        /* y remains in scope */
        printf("x = %d, y = %d\n", x, y);
}
```

When simulated this model produces the output:

```
$ spin scope.pml
x = 0, y = 0
x = 1, y = 1
1 process created
```

Table 3.1 summarizes the basic data types in PROMELA, and the typical range of values that corresponds to each type on most machines.

The data type `unsigned`, like its counterpart in the C programming language, can be used to declare a quantity that is stored in a user-defined number of bits *n*, with $1 \leq n \leq 32$. With just two exceptions, these data types can store only unsigned values. The two exceptions are `short` and `int`, which can hold either positive or negative values. The precise value ranges of the various types is implementation dependent. For `short`, `int`, and `unsigned`, the effective range matches those of the same types in C programs when compiled

on the same hardware. For `byte`, `chan`, `mtype`, and `pid`, the range matches that of the type `unsigned char` in C programs. The value ranges for `bit` and `bool` are always restricted to two values.

Typical declarations of variables of these basic types include:

```
bit  x, y;           /* two single bits, initially 0   */
bool turn = true;    /* boolean value, initially true  */
byte a[12];          /* all elements initialized to 0  */
chan m;              /* uninitialized message channel  */
mtype n;             /* uninitialized mtype variable   */
short b[4] = 89;     /* all elements initialized to 89 */
int  cnt = 67;       /* integer scalar, initially 67   */
unsigned v : 5;      /* unsigned stored in 5 bits      */
unsigned w : 3 = 5;  /* value range 0..7, initially 5  */
```

Only one-dimensional arrays of variables are supported, although there are indirect ways of defining multidimensional arrays through the use of structure definitions, as we will see shortly. All variables, including arrays, are by default initialized to zero, independent of whether they are global or local to a process.

Variables always have a strictly bounded range of possible values. The variable `w` in the last example, for instance, can only contain values that can be stored in three bits of memory: from zero to seven. A variable of type `short`, similarly, can only contain values that can be stored in sixteen bits of memory (cf. Table 3.1). In general, if a value is assigned to a variable that lies outside its declared domain, the assigned value is automatically truncated. For instance, the assignment

```
byte a = 300;
```

results in the assignment of the value 44 (300%256). When such an assignment is performed during random or guided simulations, SPIN prints an error message to alert the user to the truncation. The warning is not generated during verification runs, to avoid generating large volumes of repetitive output.

As usual, multiple variables of the same type can be grouped behind a single type name, as in:

```
byte a, b[3] = 1, c = 4;
```

In this case, the variable named `a` is, by default, initialized to zero; all elements of array `b` are initialized to one, and variable `c` is initialized to the value four.

Variables of type `mtype` can hold symbolic values that must be introduced with one or more `mtype` declarations. An `mtype` declaration is typically placed at the start of the specification, and merely enumerates the names, for instance, as follows:

```
mtype = { appel, pear, orange, banana };
mtype = { fruit, vegetables, cardboard };

init {
        mtype n = pear; /* initialize n to pear */

        printf("the value of n is ");
        printm(n);
        printf("\n")
}
```

Of course, none of the names specified in an `mtype` declaration can match reserved words from PROMELA, such as `init`, or `short`.

As shown here, there is a special predefined print routine `printm` that can be used to print the symbolic name of an `mtype` variable. There can be multiple `mtype` declarations in a model, but distinct declarations do not declare distinct types. The last model, for instance, is indistinguishable to SPIN from a model with a single `mtype` declaration, containing the concatenation (in reverse order) of the two lists, as in:

```
mtype = { fruit, vegetables, cardboard,
          appel, pear, orange, banana };
```

Because of the restricted value range of the underlying type, no more than 255 symbolic names can be declared in all `mtype` declarations combined. The SPIN parser flags an error if this limit is exceeded.

DATA STRUCTURES

PROMELA has a simple mechanism for introducing new types of record structures of variables. The following example declares two such structures, and uses them to pass a set of data from one process to another in a single, indivisible operation:

```
typedef Field {
        short f = 3;
        byte  g
};

typedef Record {
        byte a[3];
        int fld1;
        Field fld2;
        chan p[3];
        bit b
};

proctype me(Field z) {
        z.g = 12
}
```

```
init {
        Record goo;
        Field  foo;

        run me(foo)
}
```

We have defined two new data types named `Field` and `Record`, respectively. The local variable `goo` in the `init` process is declared to be of type `Record`. As before, all fields in the new data types that are not explicitly initialized (e.g., all fields except `f` in variables of type `Field`) are by default initialized to zero. References to the elements of a structure are written in a dot notation, as in for instance:

```
goo.a[2] = goo.fld2.f + 12
```

A variable of a user-defined type can be passed as a single argument to a new process in `run` statements, as shown in the example, provided that it contains no arrays. So in this case it is valid to pass the variable named `foo` as a parameter to the `run` operator, but using `goo` would trigger an error message from SPIN about the hidden arrays. In the next section we shall see that these structure type names can also be used as a field declarator in channel declarations.

The mechanism for introducing user-defined types allows for an indirect way of declaring multidimensional arrays, even though PROMELA supports only one-dimensional arrays as first class objects. A two-dimensional array can be created, for instance, as follows:

```
typedef Array {
        byte el[4]
};

Array a[4];
```

This creates a data structure of sixteen elements, that can now be referenced as `a[i].el[j]`.

As in C, the indices of an array of `N` elements range from zero to `N-1`.

MESSAGE CHANNELS

Message channels are used to model the exchange of data between processes. They are declared either locally or globally. In the declaration

```
chan qname = [16] of { short, byte, bool }
```

the typename `chan` introduces a channel declaration. In this case, the channel is named `qname`, and it is declared to be capable of storing up to sixteen messages. There can be any finite number of fields per message. In the example, each message is said to consist of three fields: the first is declared to be of type `short`, the second is of type `byte`, and the last is of type `bool`. Each

field must be either a user-defined type, such as `Field` from the last section, or a predefined type from Table 3.1. In particular, it is not possible to use an array as a type declarator in a message field. An indirect way of achieving this effect is again to embed the array into a user-defined type, and to use the type name as the type declarator for the message field. Note also that since the type `chan` appears in Table 3.1, it is always valid to use `chan` itself as a field declarator. We can make good use of this capability to pass channel identifiers from one process to another.

The statement

```
qname!expr1,expr2,expr3
```

sends a message with the values of the three expressions listed to the channel that we just created. The value of each expression is cast to the type of the message field that corresponds with its relative position in the list of message parameters. By default[2] the send statement is only executable if the target channel is not yet full, and otherwise it blocks.

The statement

```
qname?var1,var2,var3
```

retrieves a message from the head of the same buffer and stores the values from the three fields into the corresponding variables.

The receive statement is executable only if the source channel is non-empty.

It is an error to send or receive either more or fewer message fields than were declared for the message channel that is addressed.

An alternative, and equivalent, notation for the send and receive operations is to use the first message field as a message type indication, and to enclose the remaining fields in parentheses, for instance, as follows:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

Some or all of the parameters to a receive operation can be given as constants (e.g., `mtype` symbolic constants) instead of variables:

```
qname?cons1,var2,cons2
```

In this case, an extra condition on the executability of the receive operation is that the value of all message fields specified as constants match the value of the corresponding fields in the message that is to be received. If we want to use the current value of a variable for this purpose, that is, to constrain the

2. This default can be changed with SPIN option `-m` into one where the send statement is always executable, but the message will be lost when an attempt is made to send a message to a full channel.

receive operation to messages that have a matching field, we can use the predefined function `eval`, for instance, as follows:

```
qname?eval(var1),var2,var3
```

In this case, the variable `var1` is evaluated, and its value is used as a constraint on incoming messages, just like a constant. The receive operation is now executable only if a message is available that has a first field with a value that matches the current value of `var1`. If so, the values of `var2` and `var3` are set to the values of the corresponding fields in that message, and the message is removed from channel `qname`.

A simple example of the mechanisms discussed so far is as follows:

```
mtype = { msg0, msg1, ack0, ack1 };

chan    to_sndr = [2] of { mtype };
chan    to_rcvr = [2] of { mtype };

active proctype Sender()
{
again:  to_rcvr!msg1;
        to_sndr?ack1;
        to_rcvr!msg0;
        to_sndr?ack0;
        goto again
}

active proctype Receiver()
{
again:  to_rcvr?msg1;
        to_sndr!ack1;
        to_rcvr?msg0;
        to_sndr!ack0;
        goto again
}
```

The model shown here is a simplified version of the alternating bit protocol as defined by Bartlett, Scantlebury, and Wilkinson [1969]. We will extend it into a more complete version shortly, after we have covered a little bit more of the language.

The declaration

```
mtype = { msg0, msg1, ack0, ack1 };
```

introduces the four types of messages we will consider as symbolic constants.

We have used a label, named `again` in each `proctype` and a `goto` statement, with the usual semantics. We talk in more detail about control-flow constructs towards the end of this chapter. The first ten steps of a simulation run with the model above generate the following output.

```
$ spin -c -u10 alternatingbit.pml
proc 0 = Sender
proc 1 = Receiver
q   0   1
  1   to_rcvr!msg1
  1   .   to_rcvr?msg1
  2   .   to_sndr!ack1
  2   to_sndr?ack1
  1   to_rcvr!msg0
  1   .   to_rcvr?msg0
  2   .   to_sndr!ack0
  2   to_sndr?ack0
-------------
depth-limit (-u10 steps) reached
-------------
...
```

We have used the SPIN option `-c` to generate a columnated display of just the send and receive operations, which in many cases gives us just the right type of information about process interaction patterns. Every channel and every process is assigned an identifying instantiation number. Each column in the display above corresponds to a process number as before. Each row (line) of output also contains the instantiation number of the channel that is addressed in the left margin.

We have also used the SPIN option `-u10` to limit the maximum number of steps that will be executed in the simulation to ten.

There are many more operations in PROMELA that may be performed on message channels. We will review the most important operations here.

The predefined function `len(qname)` returns the number of messages that is currently stored in channel `qname`. Some shorthands for the most common uses of this function are: `empty(qname)`, `nempty(qname)`, `full(qname)`, and `nfull(qname)` with the obvious connotations.

In some cases we may want to test whether a send or receive operation would be executable, without actually executing the operation. To do so, we can transform each of the channel operations into a side effect free expression. It is, for instance, not valid to say:

```
(a > b && qname?msg0)             /* not valid */
```

or

```
(len(qname) == 0 && qname!msg0)   /* not valid */
```

because these expressions cannot be evaluated without side effects, or more to the point, because send and receive operations do not qualify as expressions (they are i/o statements).

To state a condition that should evaluate to *true* when both `(a > b)` and the first message in channel `qname` is of type `msg0`, we can, however, write in

PROMELA:

```
(a > b && qname?[msg0])          /* valid */
```

The expression `qname?[msg0]` is *true* precisely when the receive statement `qname?msg0` would be executable at the same point in the execution, but the actual receive is not executed, only its precondition is evaluated. Any receive statement can be turned into a side effect free expression in a similar way, by placing square brackets around the list of message parameters. The channel contents remain undisturbed by the evaluation of such expressions.

CHANNEL POLL OPERATIONS

It is also possible to limit the effect of a receive statement to just the copying of parameter values from message fields, without removing the message from the channel. These operations are called *channel poll* operations. Any receive statement can be turned into a poll operation by placing angle brackets around its list of parameters. For instance, assuming that we have declared a channel `q` with two message fields of type `int`, the receive statement

```
q?<eval(y),x>
```

where `x` and `y` are variables, is executable only if channel `q` contains at least one message *and* if the first field in that message has a value that is equal to the current value of variable `y`. When the statement is executed the value of the second field in the incoming message is copied into variable `x`, but the message itself is not removed from the channel.

SORTED SEND AND RANDOM RECEIVE

Two other types of send and receive statements are used less frequently: sorted send and random receive. A sorted send operation is written with two, instead of one, exclamation marks, as follows:

```
qname!!msg0
```

A sorted send operation inserts a message into the channel's buffer in numerical, rather than in FIFO, order. For instance, if a process sends the numbers from one to ten into a channel in random order, but using the sorted send operation, the channel automatically sorts them, and stores them in numerical order.

When a sorted send operation is executed, the existing contents of the target channel is scanned from the first message towards the last, and the new message is inserted immediately before the first message that follows it in numerical order. To determine the numerical order, all message fields are taken into account and are interpreted as integer values.

The counterpart of the sorted send operation is the random receive. It is written with two, instead of one, question marks:

```
qname??msg0
```

A random receive operation is executable if it is executable for *any* message that is currently buffered in a message channel (instead of being restricted to a match on the first message in the channel). In effect, the random receive operation as implemented in SPIN will always return the *first* message in the channel buffer that matches, so the term ''random receive'' is a bit of a misnomer.

Normal send and receive operations can freely be combined with sorted send and random receive operations. As a small example, if we consider the channel with the sorted list of integers from one to ten, a normal receive operation can only retrieve the first message, which will be the smallest value one. A random receive operation on the same channel would succeed for any of the values from one to ten: the message need not be at the head of the queue. Of course, a random receive operation only makes sense if at least one of the parameters is a constant, and not a variable. (Note that the value of a variable is not evaluated to a constant unless forced with an `eval` function.)

RENDEZVOUS COMMUNICATION

So far we have talked about asynchronous communication between processes via message channels that are declared for instance as

```
chan qname = [N] of { byte }
```

where `N` is a positive constant that defines the maximum number of messages that can be stored in the channel. A logical extension is to allow for the declaration

```
chan port = [0] of { byte }
```

to define a rendezvous port. The channel capacity is now zero, that is, the channel `port` can pass, but cannot store messages. Message interactions via such rendezvous ports are by definition synchronous. Consider the following example:

```
mtype = { msgtype };

chan name = [0] of { mtype, byte };

active proctype A()
{       name!msgtype(124);
        name!msgtype(121)
}

active proctype B()
{       byte state;
        name?msgtype(state)
}
```

Channel `name` is a rendezvous port. The two processes synchronously execute their first statement: a handshake on message `msgtype` and a transfer

of the value 124 from process A into local variable state in process B. The second statement in process A is unexecutable (it blocks), because there is no matching receive operation for it in process B.

If the channel name is defined with a non-zero buffer capacity, the behavior is different. If the buffer size is at least two, the process of type A can complete its execution, before its peer even starts. If the buffer size is one, the sequence of events is as follows. The process of type A can complete its first send action, but it blocks on the second, because the channel is now filled to capacity. The process of type B can then retrieve the first message and terminate. At this point A becomes executable again and also terminates, leaving its second message as a residual in the channel.

Rendezvous communication is binary: only two processes, a sender and a receiver, can meet in a rendezvous handshake.

Message parameters are always passed by value in PROMELA. This still leaves open the possibility to pass the *value* of a locally declared and instantiated message channel from one process to another. The value stored in a variable of type chan is nothing other than the channel identity that is needed to address the channel in send and receive operations. Even though we cannot send the name of the variable in which a channel identity is stored, we can send the identity itself as a value, and thereby make even a local channel accessible to other processes. When the process that declares and instantiates a channel dies, though, the corresponding channel object disappears, and any attempt to access it from another process fails (causing an error that can be caught in verification mode).

As an example, consider the following model:

```
mtype = { msgtype };

chan glob = [0] of { chan };

active proctype A()
{       chan loc = [0] of { mtype, byte };
        glob!loc;
        loc?msgtype(121)
}

active proctype B()
{       chan who;
        glob?who;
        who!msgtype(121)
}
```

There are two channels in this model, declared and instantiated in two different levels of scope. The channel named glob is initially visible to both processes. The channel named loc is initially only visible to the process that contains its declaration. Process A sends the value of its local channel variable

to process B via the global channel, and thereby makes it available to that process for further communications. Process B now transmits a message of the proper type via a rendezvous handshake on that channel and both processes can terminate. When process A dies, channel loc is destroyed and any further attempts to use it will cause an error.

RULES FOR EXECUTABILITY

The definition of PROMELA centers on its semantics of *executability*, which provides the basic means in the language for modeling process synchronizations. Depending on the system state, any statement in a SPIN model is either *executable* or *blocked*. We have already seen four basic types of statements in PROMELA: print statements, assignments, i/o statements, and expression statements. A curiosity in PROMELA is indeed that expressions can be used as if they were statements in any context. They are ‘‘executable’’ (i.e., passable) if and only if they evaluate to the boolean value *true*, or equivalently to a non-zero integer value. The semantics rules of PROMELA further state that print statements and assignments are always unconditionally executable. If a process reaches a point in its code where it has no executable statements left to execute, it simply blocks.

For instance, instead of writing a busy wait loop

```
while (a != b)  /* while is not a keyword in Promela  */
        skip;   /* do nothing, while waiting for a==b */
```

we achieve the same effect in PROMELA with the single statement

```
(a == b);       /* block until a equals b */
```

The same effect could be obtained in PROMELA with constructions such as

```
L:      /* dubious */
        if
        :: (a == b) -> skip
        :: else -> goto L
        fi
```

or

```
do      /* also dubious */
:: (a == b) -> break
:: else -> skip
od
```

but this is always less efficient, and is frowned upon by PROMELA natives. (We will cover selection and repetition structures in more detail starting at p. 56.)

We saw earlier that expressions in PROMELA must be side effect free. The reason will be clear: a blocking expression statement may have to be evaluated many times over before it becomes executable, and if each evaluation could have a side effect, chaos would result. There is one exception to the rule. An

expression that contains the `run` operator we discussed earlier can have a side effect, and it is therefore subject to some syntactic restrictions. The main restriction is that there can be only one `run` operator in an expression, and if it appears it cannot be combined with any other operators. This, of course, still allows us to use a `run` statement as a potentially blocking expression. We can indicate this effect more explicitly if instead of writing

```
run you_run(0);          /* potentially blocking */
```

without change of meaning, we write

```
(run you_run(0)) ->      /* potentially blocking */
```

Consider, for instance, what the effect is if we use such a `run` expression in the following model, as a variation on the model we saw on p. 39.

```
active proctype new_splurge(int n)
{
        printf("%d\n", n);
        run new_splurge(n+1)
}
```

As before, because of the bound on the number of processes that can be running simultanesously, the 255th attempt to instantiate a new process will fail. The failure causes the `run` expression to evaluate to zero, and thereby it permanently blocks the process. The blocked process can now not reach the end of its code and it therefore cannot terminate or die. As a result, none of its predecessors can die either. The system of 255 processes comes to a grinding halt with 254 processes terminated but blocked in their attempt to die, and one process blocked in its attempt to start a new process.

If the evaluation of the `run` expression returns zero, execution blocks, but no side effects have occurred, so there is again no danger of repeated side effects in consecutive tests for executability. If the evaluation returns non-zero, there is a side effect as the execution of the statement completes, but the statement as a whole cannot block now. It would decidedly be dubious if compound conditions could be built with `run` operators. For instance,

```
run you_run(0) && run you_run(1)        /* not valid */
```

would block if both processes could not be instantiated, but it would not reveal whether one process was created or none at all. Similarly,

```
run you_run(0) || run you_run(1)        /* not valid */
```

would block if both attempts to instantiate a process fail, but if successful would not reveal which of the two processes was created.

ASSIGNMENTS AND EXPRESSIONS

As in C, the assignments

```
c = c + 1; c = c - 1    /* valid */
```

can be abbreviated to

```
c++; c--                /* valid */
```

but, unlike in C,

```
b = c++
```

is not a valid assignment in PROMELA, because the right-hand side operand is not a side effect free expression. There is no equivalent to the shorthands

```
--c; ++c                /* not valid */
```

in PROMELA, because assignment statements such as

```
c = c-1; c = c+1        /* valid */
```

when taken as a unit are not equivalent to expressions in PROMELA. With these constraints, a statement such as `--c` is always indistinguishable from `c--`, which is supported.

In assignments such as

```
variable = expression
```

the values of all operands used in the expression on the right-hand side of the assignment operator are first cast to signed integers, before any of the operands are applied. The operator precedence rules from C determine the order of evaluation, as reproduced in Table 3.2. After the evaluation of the right-hand side expression completes, and before the assignment takes place, the value produced is cast to the type of the target variable. If the right-hand side yields a value outside the range of the target type, truncation of the assigned value can result. In simulation mode SPIN issues a warning when this occurs; in verification mode, however, this type of truncation is not intercepted.

It is also possible to use C-style conditional expressions in any context where expressions are allowed. The syntax, however, is slightly different from the one used in C. Where in C one would write

```
expr1 ? expr2 : expr3           /* not valid */
```

one writes in PROMELA

```
(expr1 -> expr2 : expr3)        /* valid */
```

The arrow symbol is used here to avoid possible confusion with the question mark from PROMELA receive operations. The value of the conditional expression is equal to the value of `expr2` if and only if `expr1` evaluates to *true* and otherwise it equals the value of `expr3`. PROMELA conditional expressions must be surrounded by parentheses (round braces) to avoid misinterpretation of the arrow as a statement separator.

Table 3.2 Operator Precedence, High to Low

Operators	Associativity	Comment
() [] .	left to right	parentheses, array brackets
! ~ ++ --	right to left	negation, complement, increment, decrement
* / %	left to right	multiplication, division, modulo
+ -	left to right	addition, subtraction
<< >>	left to right	left and right shift
< <= > >=	left to right	relational operators
== !=	left to right	equal, unequal
&	left to right	bitwise and
^	left to right	bitwise exclusive or
|	left to right	bitwise or
&&	left to right	logical and
||	left to right	logical or
-> :	right to left	conditional expression operators
=	right to left	assignment (lowest precedence)

CONTROL FLOW: COMPOUND STATEMENTS

So far, we have mainly focused on the basic statements of PROMELA, and the way in which they can be combined to model process behavior. The main types of statements we have mentioned so far are: print and assignment statements, expressions, and send and receive statements.

We saw that `run` is an operator, which makes a statement such as `run sender()` an expression. Similarly, `skip` is not a statement but an expression: it is equivalent to `(1)` or *true*.

There are five types of compound statements in PROMELA:

- Atomic sequences
- Deterministic steps
- Selections
- Repetitions
- Escape sequences

Another control flow structuring mechanism is available through the definition of macros and PROMELA inline functions. We discuss these constructs in the remaining subsections of this chapter.

ATOMIC SEQUENCES

The simplest compound statement is the atomic sequence. A simple example of an atomic sequence is, for instance:

```
atomic {        /* swap the values of a and b */
        tmp = b;
        b = a;
        a = tmp
}
```

In the example, the values of two variables `a` and `b` are swapped in a sequence of statement executions that is defined to be uninterruptable. That is, in the interleaving of process executions, no other process can execute statements from the moment that the first statement of this sequence begins to execute until the last one has completed.

It is often useful to use `atomic` sequences to initialize a series of processes in such a way that none of them can start executing statements until the initialization of all of them has been completed:

```
init {
        atomic {
                run A(1,2);
                run B(2,3)
}       }
```

Atomic sequences may be non-deterministic. If, however, any statement inside an atomic sequence is found to be unexecutable (i.e., blocks the execution), the atomic chain is broken and another process can take over control. When the blocking statement becomes executable later, control can non-deterministically return to the process, and the atomic execution of the sequence resumes as if it had not been interrupted.

Note carefully that without atomic sequences, in two subsequent statements such as

```
nfull(qname) -> qname!msg0
```

or

```
qname?[msg0] -> qname?msg0
```

the second statement is not necessarily executable after the first one is executed. There may be race conditions when access to the channels is shared between several processes. In the first example, another process can send a message to the channel just after this process determined that it was not full. In the second example, another process can steal away the message just after our process determined its presence. On the other, it would be redundant to write

```
atomic { qname?[msg0] -> qname?msg0 }
```

since this is equivalent to the single statement

```
qname?msg0
```

DETERMINISTIC STEPS

Another way to define an indivisible sequence of actions is to use the `d_step` statement. In the above case, for instance, we could also have written:

```
d_step {  /* swap the values of a and b */
        tmp = b;
        b = a;
        a = tmp
}
```

Unlike an atomic sequence, a `d_step` sequence is always executed as if it were a single statement: it is intended to provide a means for defining new types of primitive statements in PROMELA. This restricts the use of `d_step` sequences in several ways, compared to atomic sequences:

- The execution of a `d_step` sequence is always deterministic. If non-determinism is encountered in a `d_step` sequence, it is resolved in a fixed way, for example, by executing the first true guard in each non-deterministic selection or repetition structure. The precise way in which the non-determinism inside `d_step` sequences is resolved is undefined.
- No `goto` jumps into or out of `d_step` sequences are permitted: they will be flagged as errors by the SPIN parser.
- The execution of a `d_step` sequence may not be interrupted by blocking statements. It is an error if any statement other than the first one (the *guard* statement) in a `d_step` sequence is found to be unexecutable.

None of the above three restrictions apply to atomic sequences. This means that the keyword `d_step` can always be replaced with the keyword `atomic`, but not vice versa. It is safe to embed `d_step` sequences inside atomic sequences, but the reverse is not allowed.

SELECTION

Using the relative values of two variables `a` and `b` we can define a choice between the execution of two different options with a selection structure, as follows:

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

The selection structure above contains two execution sequences, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its first statement, that is, the first statement that follows the double colon, is executable. The first statement is therefore called the *guard* of the option sequence.

In the last example the guards are mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected nondeterministically. If all guards are unexecutable the process will block until at least one of them can be selected. There is no restriction on the type of statements that can be used as a guard: it may include sends or receives, assignments, `printf`, `skip`, etc. The rules of executability determine in each case what the semantics of the complete selection structure will be. The following example, for instance, illustrates the use of send statements as guards in a selection.

```
mtype = { a, b };

chan ch = [1] of { mtype };

active proctype A() { ch?a }

active proctype B() { ch?b }

active proctype C()
{       if
        :: ch!a
        :: ch!b
        fi
}
```

The example defines three processes and one channel. The first option in the selection structure of the process of type `C` is executable if channel `ch` is non-full, a condition that is satisfied in the initial state. Since both guards are executable, the process of type `C` can arbitrarily pick one, and execute it, depositing a message in channel `ch`. The process of type `A` can execute its sole statement if the message sent was an `a`, where `a` is a symbolic constant defined in the `mtype` declaration at the start of the model. Its peer process of type `B` can execute its sole statement if the message is of type `b`, where, similarly, `b` is a symbolic constant.

If we switch all send statements for receive statements, and vice versa, we also get a valid PROMELA model. This time, the choice in `C` is forced by the message that gets sent into the channel, which in turn depends on the unknown relative speeds of execution of the processes of type `A` and `B`. In both versions of the model, one of the three running processes hangs at the end of system execution, and will fail to terminate.

A process of the following type either increments or decrements the value of variable `count`. Because assignments are always executable, the choice made here is truly a non-deterministic one that is independent of the initial value of the variable `count`.

```
byte count;     /* initial value defaults to zero */

active proctype counter()
{
        if
        :: count++
        :: count--
        fi
}
```

REPETITION

We can modify the last model to obtain a cyclic program that randomly changes the value of the variable up or down by replacing the selection structure with a repetition.

```
byte count;

active proctype counter()
{
        do
        :: count++
        :: count--
        :: (count == 0) -> break
        od
}
```

As before, only one option can be selected for execution at a time. After the option completes, the execution of the repetition structure is repeated. The normal way to terminate the repetition structure is with a `break` statement. In the example, the loop can be broken only when the count reaches zero. Note, however, that it need not terminate since the other two options always remain executable. To force termination we could modify the program as follows:

```
active proctype counter()
{
        do
        :: (count != 0) ->
                if
                :: count++
                :: count--
                fi
        :: (count == 0) -> break
        od
}
```

A special type of statement that is useful in selection and repetition structures is the `else` statement. An `else` statement becomes executable only if no other statement within the same process, at the same control-flow point, is executable. We could try to use it in two places in the example, as follows:

```
active proctype counter()
{
        do
        :: (count != 0) ->
                if
                :: count++
                :: count--
                :: else
                fi
        :: else -> break
        od
}
```

The first `else`, inside the nested selection structure, can never become executable though, and is therefore redundant (both alternative guards of the selection are assignments, which are always executable). The second use of the `else`, however, becomes executable exactly when `!(count != 0)` or `(count == 0)`, and therefore preserves the option to break from the loop.

There is also an alternative way to exit the do-loop, without using a `break` statement: the infamous `goto`. This is illustrated in the following PROMELA implementation of Euclid's algorithm for finding the greatest common divisor of two non-zero, positive numbers.

```
proctype Euclid(int x, y)
{
        do
        :: (x >  y) -> x = x - y
        :: (x <  y) -> y = y - x
        :: (x == y) -> goto done
        od;

done:
        printf("answer: %d\n", x)
}

init { run Euclid(36, 12) }
```

Simulating the execution of this model, with the numbers given, yields:

```
$ spin euclid.pml
        answer: 12
2 processes created
```

The `goto` in this example jumps to a label named `done`. Multiple labels may be used to label the same statement, but at least one statement is required. If, for instance, we wanted to omit the `printf` statement behind the label, we must replace it with a dummy `skip`. Like a `skip`, a `goto` statement is always executable and has no other effect than to change the control-flow point of the process that executes it.

With these extra constructs, we can now also define a slightly more complete

description of the alternating bit protocol (cf. p. 46).

```
mtype = { msg, ack };

chan    to_sndr = [2] of { mtype, bit };
chan    to_rcvr = [2] of { mtype, bit };

active proctype Sender()
{       bool seq_out, seq_in;
        do
        :: to_rcvr!msg(seq_out) ->
                to_sndr?ack(seq_in);
                if
                :: seq_in == seq_out ->
                        seq_out = 1 - seq_out;
                :: else
                fi
        od
}

active proctype Receiver()
{       bool seq_in;
        do
        :: to_rcvr?msg(seq_in) ->
                to_sndr!ack(seq_in)
        :: timeout ->   /* recover from msg loss */
                to_sndr!ack(seq_in)
        od
}
```

The sender transmits messages of type `msg` to the receiver, and then waits for an acknowledgement of type `ack` with a matching sequence number. If an acknowledgement with the wrong sequence number comes back, the sender retransmits the message. The receiver can timeout while waiting for a new message to arrive, and will then retransmit its last acknowledgement.

The semantics of PROMELA's `timeout` statement is very similar to that of the `else` statement we saw earlier. A `timeout` is defined at the system level, though, and an `else` statement is defined at the process level. `timeout` is a predefined global variable that becomes *true* if and only if there are no executable statements at all in any of the currently running processes. The primary purpose of `timeout` is to allow us to model recovery actions from potential deadlock states. Note carefully that `timeout` is a predefined variable and not a function: it takes no parameters, and in particular it is not possible to specify a numeric argument with a specific timebound after which the timeout should become executable. The reason is that the types of properties we would like to prove for PROMELA models must be fully independent of all absolute and relative timing considerations. The relative speeds of processes is a fundamentally unknown and unknowable quantity in an asynchronous system.

ESCAPE SEQUENCES

The last type of compound structure to be discussed is the `unless` statement. This type of statement is used less frequently, but it requires a little more explanation. It is safe to skip this section on a first reading.

The syntax of an escape sequence is as follows:

```
{ P } unless { E }
```

where the letters `P` and `E` represent arbitrary PROMELA fragments. Execution of the `unless` statement begins with the execution of statements from `P`. Before each statement execution in `P` the executability of the first statement of `E` is checked, using the normal PROMELA semantics of executability. Execution of statements from `P` proceeds only while the first statement of `E` remains unexecutable. The first time that this 'guard of the escape sequence' is found to be executable, control changes to it, and execution continues as defined for `E`. Individual statement executions remain indivisible, so control can only change from inside `P` to the start of `E` in between individual statement executions. If the guard of the escape sequence does not become executable during the execution of `P`, then it is skipped entirely when `P` terminates.

An example of the use of escape sequences is:

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

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

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

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

An example application of an escape sequence is shown in Figure 3.1. Shown

here is a somewhat naive model of the behavior of a *pots* (*p*lain *o*ld *t*elephone *s*ervice) system (cf. Chapter 14, p. 299).

There are two processes in this system, a subscriber and the *pots* server. The subscriber process follows a strict regimen. After going offhook it always waits for a dial tone, and it always sends a number to be connected to when the dial tone message is received. After that it waits to receive either a busy or a ring tone. On seeing a busy tone, our idealized subscriber process hangs up and tries the call again. On seeing a ring tone, it either waits for the signal that the call is connected, or it impatiently hangs up. When connected, it waits for notification from the *pots* server that the remote party has disconnected the call, but if this does not come, it can timeout and terminate the call anyway.

The model of the subscriber behavior is fairly standard, requiring no unusual control-flow constructs. We can be more creative in modeling the *pots* server. The server process starts in its *idle* state by waiting for a subscriber to send an offhook signal together with the channel via which it wants to communicate with the server for this session. The server always complies by sending a dial tone, and then waits for the number to be dialed. Once the number has been received, either a busy tone or a ring tone is chosen, matching the subscriber's expectations at this point in the call. A ring tone is followed by a connected signal, and after this the server process proceeds to the *zombie* state where it waits for the subscriber to hangup the phone, possibly, but not necessarily sending a `hungup` message first. Note that the `skip` and the `goto zombie` statements lead to the same next state in this case (meaning that the `goto` is really redundant here).

Note that we have not included any treatment for a subscriber `hangup` message in this main flow of the *pots* behavior. The reason is that we would like to model the fact that the behavior of the *pots* server can be interrupted at *any* point in this flow if a `hangup` message should unexpectedly arrive. Similarly, if the *pots* server gets stuck at any point in its flow, it should be possible to define a timeout option, without spelling out that very same option at any point in the main flow where the server could possibly get stuck. The escape clause of the `unless` construct spells out the two conditions under which the main flow should be aborted, and gives the actions that must be taken in each case. After a `hangup`, the server simply returns to its *idle* state, since it knows that the subscriber is back onhook. After a *timeout*, it moves to the *zombie* state.

A fragment of the output for a SPIN simulation run for this system follows. The run can in principle be continued *ad infinitum*, so it is prudent to filter the output from SPIN through a utility like `more`. The first two full executions, starting and ending with both processes in their *idle* state, look as follows:

```
mtype = { offhook, dialtone, number, ringing,
        busy, connected, hangup, hungup };

chan line = [0] of { mtype, chan };

active proctype pots()
{       chan who;
idle:   line?offhook,who;
        {       who!dialtone;
                who?number;
                if
                :: who!busy; goto zombie
                :: who!ringing ->
                        who!connected;
                        if
                        :: who!hungup; goto zombie
                        :: skip
                        fi
                fi
        } unless
        {       if
                :: who?hangup -> goto idle
                :: timeout -> goto zombie
                fi
        }
zombie: who?hangup; goto idle
}

active proctype subscriber()
{       chan me = [0] of { mtype };
idle:   line!offhook,me;
        me?dialtone;
        me!number;
        if
        :: me?busy
        :: me?ringing ->
                if
                :: me?connected;
                        if
                        :: me?hungup
                        :: timeout
                        fi
                :: skip
                fi
        fi;
        me!hangup; goto idle
}
```

Figure 3.1 Simple Model of a Telephone System

```
$ spin -c pots.pml | more
proc 0 = pots
proc 1 = subscriber
q\p   0   1
  2   .   line!offhook,1
  2   line?offhook,1
  1   who!dialtone
  1   .   me?dialtone
  1   .   me!number
  1   who?number
  1   who!ringing
  1   .   me?ringing
  1   who!connected
  1   .   me?connected
timeout
  1   .   me!hangup
  1   who?hangup
  2   .   line!offhook,1
  2   line?offhook,1
  1   who!dialtone
  1   .   me?dialtone
  1   .   me!number

  1   who?number
  1   who!ringing
  1   .   me?ringing
  1   .   me!hangup
  1   who?hangup
```

There are no surprises here. The model, though, cannot properly be called a verification model just yet. For that, we would have to add some statement of the requirements or properties that we would like this model to satisfy. We may well ask, for instance, if it is possible for the server to get stuck permanently in the *zombie* state. Only a verification run can give the answer to such questions.

INLINE DEFINITIONS

Some motivation for and examples of the use of PROMELA `inline`'s was already given in the last chapter. The PROMELA `inline` is meant to provide some of the structuring mechanism of a traditional procedure call, without introducing any overhead during the verification process. The PROMELA parser replaces each point of invocation of an `inline` with the text of the `inline` body. If any parameters are used, their actual values from the call will textually replace the formal place holders that are used inside the definition of the `inline` body. That is, there is no concept of value passing with `inline`'s. The parameter names used inside the definition are mere stand ins for the names provided at the place of call. A small example can clarify the working and intent of this mechanism, as follows:

```
inline example(x, y) {
        y = a;
        x = b;
        assert(x)
}
init {
        int a, b;

        example(a,b)
}
```

In this example we have defined an `inline` named `example` and we gave it two parameters. The parameters do not have a type associated with them. They could in fact be replaced in a call with variables of any type that matches the use of the names inside the `inline` body.

At the point of invocation the names of two variables are provided as actual parameters. The parser treats this code as if we had written the following specification instead:

```
init {
        int a, b;

        b = a;
        a = b;
        assert(a)
}
```

This version of the model is obtained by inserting the body of the `inline` at the point of call, while textually replacing every occurrence of the name `x` with the name `a` and every occurrence of `y` with `b`, as stipulated by the parameter list at the point of invocation.

We could have achieved the same effect by defining a C-style macro, as follows:

```
#define example(x, y) \
        y = a;  \
        x = b;  \
        assert(x)

init {
        int a, b;

        example(a,b)
}
```

For a small `inline` function the difference is not that remarkable, but for larger pieces of code the macro method can quickly become unwieldy. There is one other benefit to the use of an `inline` compared to a macro definition. When we simulate (or verify) the version of the example using the `inline` definition of `example`, we see the following output:

```
$ spin inline.pml
spin: line   4 "inline", Error: assertion violated
spin: text of failed assertion: assert(a)
#processes: 1
  3:    proc  0 (:init:) line   4 "inline" (state 3)
1 process created
```

Not surprisingly, the assertion is violated. The line number pointed at by SPIN is the location of the `assert` statement inside the `inline` body, as one would expect. If, however, we try to do the same with the version using a macro, we see this result:

```
$ spin macro.pml
spin: line   9 "macro", Error: assertion violated
spin: text of failed assertion: assert(a)
#processes: 1
  3:    proc  0 (:init:) line   9 "macro" (state 3)
1 process created
```

The same assertion violation is reported, but the line number reference now gives the point of invocation of the macro, rather than the location of the failing assertion. Finding the source of an error by searching through possibly complex macro definitions can be challenging, which makes the use of PROMELA `inline`s preferable in most cases.

To help find out what really happens with parameter substitution in `inline` functions and preprocessing macros, option `-I` causes SPIN to generate a version of the source text that shows the result of all macro-processing and inlining on `proctype` bodies. It can be an invaluable source of information in determining the cause of subtle problems with preprocessing. The two versions of our sample program, the first using an `inline` definition and the second using a macro, produce the following results:

```
$ spin -I inline.pml
proctype :init:()
{
    {
      b = a;
      a = b;
      assert(a);
    };
}

$ spin -I macro.pml
proctype :init:()
{
   b = a;
   a = b;
   assert(a);
}
```

Note that the version of the model that is generated with the `-I` option is not

itself a complete model. No variable declarations are included, and some of the names used for `proctype`s and labels are the internally assigned names used by SPIN (using, for instance, `:init:` instead of `init`). The `proctype` body text, though, shows the result of all preprocessing.

There is not much difference in the output for the two versions, except that the use of the `inline` function creates a non-atomic sequence (the part enclosed in curly braces), where the macro definition does not. There is no difference in behavior.

When using `inline` definitions, it is good to keep the scope rules of PROMELA in mind. Because PROMELA only knows two levels of scope for variables, global and process local, there is no subscope for `inline` bodies. This means that an attempt to declare a local scratch variable, such as this:

```
inline thisworks(x) {
        int y;

        y = x;
        printf("%d\n", y)
}

init {
        int a;
        a = 34;
        thisworks(a)
}
```

produces the following, after inlining is performed:

```
init {
        int a;
        a = 34;

        int y;
        y = a;
        printf("%d\n", y)
}
```

This works because variable declarations can appear anywhere in a PROMELA model, with their scope extending from the point of declaration to the closing curly brace of the surrounding `proctype` or `init` body. This means that the variable `y` remains in scope, also after the point of invocation of the `inline`. It would therefore be valid, though certainly confusing, to write

```
inline thisworks2(x) {
        int y;

        y = x;
        printf("%d\n", y)
}
```

```
init {
        int a;
        a = 34;
        thisworks(a);
        y = 0
}
```

that is, to access the variable `y` outside the `inline` body in which it was declared.

READING INPUT

On an initial introduction to PROMELA it may strike one as odd that there is a generic output statement to communicate information to the user in the form of the `printf`, but there is no matching `scanf` statement to read information from the input. The reason is that we want verification models to be *closed* to their environment. A model must always contain *all* the information that could possibly be required to verify its properties. It would be rather clumsy, for instance, if the model checker would have to be stopped dead in its tracks each time it needed to read information from the user's keyboard.

Outputs, like `printf`, are harmless in this context, since they generate no new information that can affect future behavior of the executing process, but the executing of an input statement like `scanf` can cause the modification of variable values that can impact future behavior. If input is required, its source must always be represented in the model. The input can then be captured with the available primitives in PROMELA, such as sends and receives.

In one minor instance we deviate from this rather strict standard. When SPIN is used in *simulation mode*, there is a way to read characters interactively from a user-defined input. To enable this feature, it suffices to declare a channel of the reserved type `STDIN` in a PROMELA model. There is only one message field available on this predefined channel, and it is of type `int`. The model in Figure 3.2 shows a simple word count program as an example.

We can simulate the execution of this model (but not verify it) by invoking SPIN as follows, feeding the source text for the model itself as input.

```
$ spin wc.pml < wc.pml
27      85      699
1 process created
```

PROMELA supports a small number of other special purpose keywords that can be used to fine-tune verification models for optimal performance of the verifiers that can be generated by SPIN. We mention the most important of these here. (This section can safely be skipped on a first reading.)

SPECIAL FEATURES

The verifiers that can be generated by SPIN by default apply a partial order reduction algorithm that tries to minimize the amount of work done to prove

```
chan STDIN;
int c, nl, nw, nc;

init {
        bool inword = false;

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

Figure 3.2 Word Count Program Using STDIN Feature

system properties. The performance of this algorithm can be improved, sometimes very substantially, if the user provides some hints about the usage of data objects. For instance, if it is known that some of the message channels are only used to receive messages from a single source process, the user can record this knowledge in a channel assertion.

In the example shown in Figure 3.3, for instance, the number of states that has to be searched by the verifier is reduced by 16 percent if the lines containing the keywords `xr` and `xs` are included. (The two keywords are acronyms for *exclusive read* access and *exclusive write* access, respectively.) These

```
mtype = { msg, ack, nak };

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

active proctype S()
{       byte s = 1;

        xs q;   /* assert that only S sends to chan q */
        xr r;   /* and only S receives from chan r */

        do
        :: q!msg(s);
                if
                :: r?ack; s++
                :: r?nak
                fi
        od
}
active proctype R()
{       byte ns, s;

        xs r;   /* only R sends messages to chan r */
        xr q;   /* only R retrieves messages from chan q */

        do
        :: q?msg(ns);
                if
                :: (ns == s+1) -> s = ns; r!ack
                :: else -> r!nak
                fi
        od

}
```

Figure 3.3 Using Channel Assertions

statements are called *channel assertions*.

The statements are called assertions because the validity of the claims they make about channel usage can, and will, be checked during verifications. If, for instance, it is possible for a process to send messages to a channel that was claimed to be non-shared by another process, then the verifier can always detect this and it can flag a channel assertion violation. The violation of a channel assertion in effect means that the additional reduction that is based on its presence is invalid. The correct counter-measure is to then remove the channel assertion.

The reduction method used in SPIN (more fully explained in Chapter 9) can also take advantage of the fact that the access to local variables cannot be shared between processes. If, however, the verification model contains a globally declared variable that the user knows to be non-shared, the keyword `local` can be used as a prefix to the variable declaration. For instance, in the last example we could have declared the variable `ns` from `proctype R` as a global variable, without incurring a penalty for this change from the partial order reduction algorithm, by declaring it globally as:

```
local byte ns;
```

The use of this prefix allows the verifier to treat all access to this variable as if it were access to a process local variable. Other than for channel assertions, though, the verifier does not check if the use of the prefix is unwarranted.

Another case that one occasionally runs into is when a variable is used only as a scratch variable, for temporary use, say, deep inside a `d_step` or an `atomic` sequence. In that case, it can be beneficial to tell the verifier that the variable has no permanent state information and should not be stored as part of the global state-descriptor for the modeled system. We can do so by using the prefix `hidden`. The variable must again be declared globally, for instance, as:

```
hidden int t;
```

In the following PROMELA fragment the variable `t` is used as a temporary variable that stores no relevant information that must be preserved outside the `d_step` sequence in which it is used:

```
d_step {        /* swap the values of a an b */
        t = a;
        a = b;
        b = t
}
```

As with the use of the `local` prefix, the verifier takes the information on good faith and does not check if the use of the `hidden` keyword is unwarranted. If a hidden variable does contain relevant state information, the search performed by the verifier will be incomplete and the results of the search become unreliable.

There is a third, and last, type of prefix that can be used with variable declarations in special cases. The use of the prefix `show` on a variable declaration, as in

```
show byte cnt;
```

tells SPIN's graphical user interface XSPIN that any value changes of this variable should be visualized in the message sequence charts that it can generate. We will discuss this interface in more detail in Chapter 12.

The `show` prefix can be used on both global and local variable declarations.

FINDING OUT MORE

This concludes our overview of the main features of the PROMELA specification language. A few more seldomly used constructs were only mentioned in passing here, but are discussed in greater detail in the manual pages that are included in Chapters 16 and 17. More examples of PROMELA models are included in Chapters 14 and 15. A definition of the operational semantics for PROMELA can be found in Chapter 7.

Alternate introductions to the language can be found in, for instance, Ruys [2001] and Holzmann [1991]. Several other tutorial-style introductions to the language can also be found on the SPIN Web site (see Appendix D).