VALIDATION MODELS 5

5.1 INTRODUCTION

In Chapter 2 we discussed the five main elements of a protocol definition: a service specification, explicit assumptions about the environment, the protocol vocabulary, format definitions, and procedure rules. Most of these elements can be structured in a hierarchical manner. The service specification, for instance, can be divided into layers, each new layer building upon the ones below it and providing a higher-level service to the user. To realize the service at a given layer, a consistent set of procedure rules must be derived and described in some formal language. The design of a complete and consistent set of procedure rules, however, is one of the hardest problems in protocol design.

A PROTOCOL VALIDATION LANGUAGE

In this chapter we introduce a notation for the specification and verification of procedure rules. Since the focus is on the procedure rules, these specifications provide only a partial description of a protocol. We call such a partial description a protocol *validation model*. The language we use to describe validation models is called PROMELA.

Our aim is to model protocols as succinctly as possible in order to be able to study their structure and to verify their completeness and logical consistency. Doing so, we deliberately abstract from other issues of protocol design, such as message format. A validation model defines the interactions of processes in a distributed system. It does not resolve implementation details. It does not say how a message is to be transmitted, encoded, or stored. By simplifying the problem in this way we can isolate and

concentrate on the hardest part: the design of a complete and consistent set of rules to govern the interactions in a distributed system.

This chapter gives an introduction to the use of PROMELA for specifying system behavior in formal validation models. The next chapter discusses specific methods for defining the precise correctness criteria that can be applied to the validation of these models. A brief reference manual to PROMELA can be found in Appendix C. Chapter 7 gives an example of a serious application of PROMELA in the design of a file transfer protocol. Chapter 12 discusses the design of an interpreter/simulator for the language, and Chapter 13 discusses how this software can be extended with an automated analyzer for PROMELA models.

5.2 PROCESSES, CHANNELS, VARIABLES

We describe procedure rules as formal programs for an abstract model of a distributed system. Of course, we want this model to be as simple as possible, yet sufficiently powerful to represent all types of coordination problems that can occur in distributed systems. As far as descriptive power is concerned, it would suffice to define just one type of object: the finite state machine. The state machine can model all other objects that we may be interested in, including finite variables and message channels (bounded FIFO queues). Though such a model may be sufficient, it is not very convenient to work with. We therefore define validation models directly in terms of three specific types of objects:

- *processes*
- message *channels*
- state *variables*

For the purpose of analysis, each of these objects may be translated into a finite state machine by a simple translation process that is considered in Chapter 8. For now, however, we can pretend to have the luxury of working directly with these higher-level objects. All processes are by definition global objects. Variables and channels represent data that can be either global or local to a process.

5.3 EXECUTABILITY OF STATEMENTS

In PROMELA there is no difference between conditions and statements. Even isolated boolean conditions can be used as statements. The execution of a statement is conditional on its *executability*. All PROMELA statements are either executable or blocked, depending on the current values of variables or the contents of message channels. Executability is the basic means of synchronization. A process can wait for an event to happen by waiting for a statement to become executable. For instance, instead of writing a busy wait loop:

```
while (a != b) skip     /* wait for a == b */
```

we can achieve the same effect in PROMELA with the statement

```
(a == b)
```

The condition can only be executed (passed) if it holds. If the condition does not

hold, execution blocks until it does. Arithmetic and boolean operators in conditions such as these are the same as in C. As we will see below, assignments to variables are always executable.

5.4 VARIABLES AND DATA TYPES

Variables in PROMELA are used to store either global information about the system as a whole or information that is local to one specific process, depending on where the declaration for the variable is placed. A variable can be one of the following six predefined data types:

```
bit, bool, byte, short, int, chan.
```

The first five types in this list are called the basic data types. They are used to specify objects that can hold a single value at a time. The sixth type specifies message channels. A message channel is an object that can store a number of values, grouped in user-defined structures. We discuss the basic data types first. Message channels are discussed separately in Section 5.6.

The declarations

```
bool    flag;
int     state;
byte    msg;
```

define variables that can store integer values in three different ranges. The scope of the variable is global if it is declared outside all process declarations, and local if it is declared within a process declaration. Table 5.1 summarizes the basic data types, sizes, and the corresponding value ranges on DEC/VAX computers.

Table 5.1 — Basic Data Types

Name	Size (bits)	Usage	Range
`bit`	1	unsigned	0..1
`bool`	1	unsigned	0..1
`byte`	8	unsigned	0..255
`short`	16	signed	$-2^{15}..2^{15}-1$
`int`	32	signed	$-2^{31}..2^{31}-1$

The names `bit` and `bool` are synonyms for a single bit of information. A `byte` is an unsigned quantity that can store a value between 0 and 255. `short`s and `int`s are signed quantities that differ only in the range of values they can hold.

ARRAYS

Variables can be declared as arrays. For instance,

```
byte state[N]
```

declares an array of `N` bytes that can be accessed in statements such as

```
state[0] = state[3] + 5 * state[3*2/n]
```

where `n` is a constant or a variable declared elsewhere. The index to an array can be any expression that determines a unique integer value. The valid range of indexes is `0 .. N-1`. The effect of the use of an index value outside that range is undefined; most likely it will cause a runtime error.

So far we have seen examples of a variable declaration and of two basic types of statements: boolean conditions and assignments. Declarations and assignments are always *executable.*

5.5 PROCESS TYPES

To execute a process we have to be able to name it, define its type, and instantiate it. Let us first look at the definition and naming of processes. All types of processes that can be instantiated are defined in `proctype` declarations. The following, for instance, declares a process with one local variable named `state`.

```
proctype A() { byte state; state = 3 }
```

The process type is named `A`. The body of the declaration, enclosed in parentheses, consists of local variable or channel declarations and statements. The declaration above contains one local variable declaration and a single statement: an assignment of the value 3 to variable `state`.

The semicolon is a statement *separator* (not a statement terminator, hence there is no semicolon after the last statement). PROMELA defines two different statement separators: an arrow, `->`, and a semicolon, `;`. The two separators are equivalent. The arrow is sometimes used as an informal way to indicate a causal relation between two statements. Consider the following example.

```
byte state = 2;

proctype A() { (state == 1) -> state = 3 }

proctype B() { state = state – 1 }
```

In this example we declared two process types, `A` and `B`. Variable `state` is now a global, initialized to the value 2. Process type `A` contains two statements, separated by an arrow. Process type declaration `B` contains a single statement that decrements the value of the state variable by 1. Since the assignment is always executable, processes of type B can always terminate without delay. Processes of type `A`, however, are delayed until the variable `state` contains the proper value.

THE INITIAL PROCESS

A `proctype` definition only declares process behavior, it does not execute it. Initially, just one process is executed: a process of type `init` which must be declared explicitly in every PROMELA specification. The `init` process is comparable to the function `main()` of a standard C program. The smallest possible PROMELA specification is

```
init { skip }
```

where `skip` is a null statement. Only slightly more complicated is the PROMELA equivalent of the famous ‘‘hello world’’ program from C:

```
init { printf("hello world\n") }
```

More interestingly, however, the initial process can initialize global variables, create message channels, and instantiate processes. An `init` declaration for the two-process system in Section 5.5, for instance, might look as follows.

```
init { run A(); run B() }
```

This `init` process starts two processes, which will run concurrently with the `init` process from then on. In the above case, the `init` process terminates after starting the second process, but it need not do so. `Run` is a unary operator that instantiates a copy of a given process type (for example, `A`). It does not wait for the process to terminate. The `run` statement is executable and returns a positive result only if the process can effectively be instantiated. It is unexecutable and returns zero if this cannot be done, for instance if too many processes are already running. Since PROMELA models finite state systems, the number of processes and message channels is always bounded. The precise value of the bound is hardware-dependent and therefore undefined in PROMELA. The value returned by `run` is a run-time process number, or `pid`. Because `run` is defined as an operator, `run A()` is an expression that can be embedded in other expressions. It would therefore be valid, though perhaps not too useful, to use it in a composite expression such as

```
i = run A() && (run B() || run C())
```

Since communication between processes is defined on named channels, the process numbers (`pids`) are usually irrelevant. There is one important exception that we discuss in Chapter 6, Section 6.7.

The `run` operator can pass parameter values to the new process, for instance as follows:

```
proctype A(byte state; short set)
{       (state == 1) -> state = set
}
init { run A(1, 3) }
```

Only message channels, discussed later, and instances of the five basic data types can be passed as parameters. Arrays and process types cannot be passed.

`Run` can be used in any process to spawn new processes, not just in the initial process. An executing process disappears when it terminates (i.e., reaches the end of the body of its process type declaration), but not before all the processes that it has instantiated (its ‘‘children’’) have terminated first.

Going back to the earlier example, note that using `run` we can create any number of copies of the process types `A` and `B`. If, however, more than one concurrent process is

allowed both to read and write the value of a global variable a well-known set of problems can result (see Bibliographic Notes). Consider, for instance, the following system of two processes, sharing access to the global variable `state`.

```
byte state = 1;

proctype A() { (state == 1) -> state = state + 1 }

proctype B() { (state == 1) -> state = state – 1 }

init { run A(); run B() }
```

If one of the two processes terminates before its competitor has started, the other process will block forever on the initial condition. If both pass the condition simultaneously, both can terminate, but the resulting value of `state` is unpredictable. It can be 0, 1, or 2.

Many solutions to this problem have been considered, ranging from the abolition of global variables to the provision of special machine instructions that can guarantee an indivisible test-and-set sequence on a shared variable. The example below was one of the first solutions published. It is due to the Dutch mathematician T. Dekker. It grants two processes mutually exclusive access to an arbitrary *critical section* in their code by manipulating three global state variables. The first four lines in the PROMELA specification below are C-style macro definitions. The first two macros define `true` to be a constant value equal to 1 and `false` to be a constant 0. Similarly, `Aturn` and `Bturn` are defined as boolean constants.

```
 1 #define true        1
 2 #define false       0
 3 #define Aturn       1
 4 #define Bturn       0
 5
 6 bool x, y, t;
 7
 8 proctype A()
 9 {   x = true;
10     t = Bturn;
11     (y == false || t == Aturn);
12     /* critical section */
13     x = false
14 }
15 proctype B()
16 {   y = true;
17     t = Aturn;
18     (x == false || t == Bturn);
19     /* critical section */
20     y = false
21 }
22 init { run A(); run B() }
```

The conditions on lines 11 and 18 are used to synchronize the processes. They can only be executed if they hold. The algorithm can be executed repeatedly and is

independent of the relative speeds of the two processes.

ATOMIC SEQUENCES

In PROMELA there is another way to avoid the *test-and-set* problem: `atomic` sequences. A sequence of statements enclosed in parentheses prefixed with the keyword `atomic` indicates that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. It is an error if any statement, other than the first one, can block in an atomic sequence. The executing process will abort in that case. Here is the earlier example, rewritten with two atomic sequences.

```
byte state = 1;
proctype A() { atomic { (state == 1) -> state = state + 1 } }
proctype B() { atomic { (state == 1) -> state = state – 1 } }
init { run A(); run B() }
```

In this case the final value of `state` is either 0 or 2, depending on which process executes. The other process will be blocked forever.

Atomic sequences can be an important tool in reducing the complexity of a validation model. An atomic sequence restricts the amount of interleaving that is allowed which can effectively render complex validation models tractable, without loss of generality. The example below illustrates this.

```
proctype nr(short pid, a, b)
{       int res;

atomic  {       res = (a*a+b)/2*a;
                printf("result %d: %d\n", pid, res)
        }
}
init { run nr(1,1,1); run nr(1,2,2); run nr(1,3,2) }
```

The `init` process starts up three copies of the process type `nr`. Each process computes some number and prints it. The manipulations of the variables within these processes are all local and cannot affect the behavior of the other processes. Defining the body of the process as an atomic sequence dramatically reduces the number of cases that would need to be considered in a validation (Chapter 11), without changing the possible behaviors of the processes in any way. It is usually trivial to identify statement sequences that can be rewritten with atomic sequences.

5.6 MESSAGE CHANNELS

Message channels are used to model the transfer of data from one process to another. They are declared either locally or globally, just like variables of the basic data types, using the keyword `chan`. For instance,

```
chan a, b; chan c[3]
```

declares the names `a`, `b`, and `c` as channel identifiers, the last one as an array. A channel declaration can have an initializer field as well:

```
chan a = [16] of { short }
```

initializes channel `a`. The initializer says that the channel can store up to 16 messages of type `short`. Similarly,

```
chan c[3] = [4] of { byte }
```

initializes an array of 3 channels, each with a capacity of 4 message slots, each slot consisting of one message field of type `byte`.

If the messages to be passed by the channel have more than one field, the declaration looks as follows:

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

This time, we have defined a single channel that can store up to 16 messages, each consisting of 4 fields: an 8-bit value, a 32-bit value, a channel name, and another 8-bit value.

The statement

```
qname!expr
```

sends the value of expression `expr` to the channel we just created, that is, it appends the value to the tail of the channel.

```
qname?msg
```

retrieves a message from the head of the channel, and stores it in the variable `msg`. Channels pass messages in first-in first-out order. In the above cases, only a single value is passed through the channel. If multiple values are transferred per message, they are specified in a comma-separated list

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

If more parameters are sent per message than the message channel can store, the redundant parameters are lost. If fewer parameters are sent then the message channel can store, the value of the remaining parameters is undefined. Similarly, if the receive operation tries to retrieve more parameters than are available, the value of the extra parameters is undefined; if it receives fewer than the number of parameters that was sent, the extra information is lost.

By convention, the first message field is often used to specify the message type (a constant). An alternative and equivalent notation for the send and receive operations is therefore to specify the message type, followed by a list of message fields enclosed in parentheses. In general:

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

The send operation is executable only when the channel addressed is not full. The receive operation, similarly, is only executable when the channel is non-empty.

Optionally, some of the arguments in the receive operation can be constants:

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

In this case, a further condition on the executability of the receive operation is that the value of all message fields that are specified as constants match the value of the corresponding fields in the message that is at the head of the channel.

Here is an example that uses some of the mechanisms introduced so far.

```
proctype A(chan q1)
{       chan q2;
        q1?q2;
        q2!123
}
proctype B(chan qforb)
{       int x;
        qforb?x;
        printf("x = %d\n", x)
}
init
{       chan qname[2] = [1] of { chan };
        chan qforb = [1] of { int };
        run A(qname[0]);
        run B(qforb);
        qname[0]!qforb
}
```

Note that channel `qforb` is not declared as an array and therefore it does not need an index in the send operation at the end of the initial process. The value printed by the process of type `B` will be 123.

A predefined unary operator `len(qname)` takes the name of a channel `qname` as an operand and returns the number of messages that it currently holds. Note that if `len` is used as a condition, rather than on the right side of an assignment, it is unexecutable if the channel is empty: it returns a zero result, which by definition means that the statement is temporarily unexecutable.

Send and receive operations cannot be evaluated without potential side-effects. Composite conditions such as

```
(qname?var == 0)
```

or

```
(a > b && qname!123)
```

are therefore invalid in PROMELA. For the receive operation, however, there is an alternative notation, using square brackets around the clause behind the question mark. For instance,

```
qname?[ack,var]
```

is evaluated as a condition and can be combined with other boolean expressions. It returns a positive result (1) if the corresponding receive statement

```
qname?ack,var
```

would be executable, i.e., if there is indeed a message `ack` at the head of the channel. It returns zero otherwise. It has no side-effect; specifically, it does not remove the message from the channel.

Note carefully that in non-atomic sequences of two statements such as

```
(len(qname) > 0) -> qname?msgtype
```

or

```
qname?[msgtype] -> qname?msgtype
```

the second statement is not necessarily executable after the first one has been executed. There may be race conditions if access to the channels is shared between several processes. In both cases, a second process can steal the message just after the current one determined its presence. PROMELA does not, and indeed cannot, prevent the user from writing these specifications. On the contrary, these are precisely the types of problems we want to model in our validation language.

RENDEZVOUS COMMUNICATION

So far we have talked about asynchronous communication between processes via message channels created in statements such as

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

where `N` is a positive constant that defines the buffer size. Using a channel size of zero, as in

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

defines a rendezvous port that can only pass, and not store, single-byte messages. Message interactions via such rendezvous ports are synchronous, by definition. Consider the following example:

```
#define msgtype 33

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

byte name;

proctype A()
{       name!msgtype(124);
        name!msgtype(121)
}
proctype B()
{       byte state;
        name?msgtype(state)
}
init
{       atomic { run A(); run B() }
}
```

The two `run` statements are placed in an atomic sequence to enforce that the two processes start simultaneously. Of course, they need not terminate simultaneously, and they need not have run to completion before the atomic sequence terminates. Channel `name` is a global rendezvous port. The two processes synchronously execute their first statement: a handshake on message `msgtype` and a transfer of the value 124 to local variable `state`. The second statement in process `A` is unexecutable, because there is no matching receive operation 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 terminates, leaving its last message as a residual in the channel.

Synchronous ports can be declared as arrays, just like asynchronous channels. Rendezvous communication is binary: only two processes, a sender and a receiver, can be synchronized in this way. We will see an example of a way to exploit this to build a semaphore below. But first, let us introduce a few more control flow structures.

5.7 CONTROL FLOW

Between the lines, we have already introduced three ways of defining control flow: concatenation of statements within a process, parallel execution of processes, and atomic sequences. There are three other control flow constructs in PROMELA to be discussed:

- Case selection
- Repetition
- Unconditional jumps

CASE SELECTION

The simplest construct is the selection structure. Using the relative values of two variables `a` and `b` to choose between two options, for instance, we can write

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

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

In the example above the guards are mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected at random. If all guards are unexecutable, the process blocks until at least one of them can be selected. There is no restriction on the type of statement that can be used as a guard. The following example uses input statements:

```
#define a 1
#define b 2

chan ch = [1] of { byte };

proctype A() { ch!a }
proctype B() { ch!b }
proctype C()
{       if
        :: ch?a
        :: ch?b
        fi
}
init { atomic { run A(); run B(); run C() } }
```

This example defines three processes and one channel. The first option in the selection structure of the process of type `C` is executable if the channel contains a message `a`, where `a` is a constant with value 1, as defined in a macro definition at the start of the program. The second option is executable if it contains a message `b`, where `b` is a constant. Which message will be available depends on the relative speeds of the processes.

A process of the following type either increments or decrements the value of variable `count` once.

```
byte count;

proctype counter()
{       if
        :: count = count + 1
        :: count = count – 1
        fi
}
```

REPETITION

A logical extension of the selection structure is the repetition structure. We can modify the above program to obtain a cyclic program that randomly increments or decrements the variable.

```
byte count;

proctype counter()
{       do
        :: count = count + 1
        :: count = count – 1
        :: (count == 0) -> break
        od
}
```

Only one option can be selected for execution at a time. After the option completes, the execution of the structure is repeated. The normal way to terminate the repetition structure is with a `break` statement. In the example, the loop can be broken when the

count reaches zero. It need not terminate since the other two options remain executable. To force termination, we could modify the program as follows:

```
proctype counter()
{       do
        :: (count != 0) ->
                if
                :: count = count + 1
                :: count = count – 1
                fi
        :: (count == 0) -> break
        od
}
```

JUMPS

Another way to break the loop is with an unconditional jump: the infamous `goto` statement. This is illustrated in the following implementation of Euclid's algorithm for finding the greatest common divisor of two positive numbers:

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

The `goto` in this example jumps to a label named `done`. A label can only appear before a statement. Above, we want to jump to the end of the program. In this case a dummy statement `skip` is useful: it is a place holder that is always executable and has no effect. The `goto` statement itself is always executable.

5.8 EXAMPLES

The following example specifies a filter that receives messages from a channel `in` and divides them over two channels `large` and `small` depending on the values attached. The constant `N` is defined to be 128, and `size` is defined to be 16 in two macro definitions.

```
#define N    128
#define size  16

chan in    = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };
```

```
proctype split()
{       short cargo;

        do
        :: in?cargo ->
                if
                :: (cargo >= N) -> large!cargo
                :: (cargo <  N) -> small!cargo
                fi
        od
}
init {  run split() }
```

A process type that merges the two streams back into one, most likely in a different order, and writes it back to the channel `in` could be specified as

```
proctype merge()
{       short cargo;

        do
        ::      if
                :: large?cargo
                :: small?cargo
                fi;
                in!cargo
        od
}
```

With the following modification to the `init` process, the split and merge processes can busily perform their duties forever.

```
init
{       in!345; in!12; in!6777; in!32; in!0;
        run split(); run merge()
}
```

As a final example, consider the following implementation of a Dijkstra semaphore, using binary rendezvous communication.

```
#define p       0
#define v       1

chan sema = [0] of { bit };

proctype dijkstra()
{       do
        :: sema!p -> sema?v
        od
}
```

```
proctype user()
{       sema?p;
        /* critical section */
        sema!v
        /* non-critical section */
}
init
{       atomic {
                run dijkstra();
                run user(); run user(); run user()
        }
}
```

The semaphore guarantees that only one user process can enter its critical section at a time. In the example, each user process accesses its critical section only once. If repeated access could be requested, the semaphore would not necessarily prevent one process from monopolizing access to the critical section.

5.9 MODELING PROCEDURES AND RECURSION

Procedures, even recursive ones, can be modeled as processes. The return value can be passed back to the calling process via a global variable or via a message. The following program illustrates this.

```
proctype fact(int n; chan p)
{       int result;

        if
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
                chan child = [1] of { int };
                run fact(n-1, child);
                child?result;
                p!n*result
        fi
}
init
{       int result;
        chan child = [1] of { int };

        run fact(7, child);
        child?result;
        printf("result: %d\n", result)
}
```

The process `fact(n, p)` recursively calculates the factorial of `n`, communicating the result to its parent process via channel `p`.

5.10 MESSAGE-TYPE DEFINITIONS

We have seen how constants can be defined using `C`-style macros. As a mild form of syntactic sugar, PROMELA allows for message type definitions of the form

```
mtype = { ack, nak, err, next, accept }
```

The definition is equivalent to the following sequence of macro definitions.

```
#define ack     1
#define nak     2
#define err     3
#define next    4
#define accept  5
```

A formal message-type definition is the preferred way of specifying the message types since it defers any decision on the specific values to be used. At the same time, it makes the names of the constants, rather than the values, available to an implementation, which can improve error reporting. There can be only one message-type definition per specification.

5.11 MODELING TIMEOUTS

We have already discussed two types of statements with a predefined meaning in PROMELA: `skip` and `break`. Another predefined statement is `timeout`. The `timeout` statement allows a process to abort the waiting for a condition that can no longer become true, for example, an input from an empty channel. The timeout provides an escape from a hang state. It can be considered an artificial, predefined condition that becomes true only when no other statements in the distributed system are executable. Note that it carries no value: it does not specify a timeout interval, but a timeout possibility. We deliberately abstract from absolute timing considerations, which is crucial in validation work, and we do not specify how the timeout should be implemented. A simple example is the following process that sends a reset message to a channel named `guard` whenever the system comes to a standstill.

```
proctype watchdog()
{       do
        :: timeout -> guard!reset
        od
}
```

The timeout, as defined here, does not model errors caused by premature timeouts in a real system. If this is required, it can be achieved by redefining the keyword in a macro, for instance as follows.

```
#define timeout 1       /* always enabled, arbitrary delay */
```

More examples are given in Chapter 7.

STATEMENT TYPES

With the exception of `assert` statements and temporal claim primitives (see Chapter 6) we have now discussed all basic types of statements defined in PROMELA:

- Assignments and conditions
- Selections and repetitions
- Send and receive
- `Goto` and `break` statements
- `Timeout`

Note that `run` and `len` are not statements but unary operators that can be used in assignments and conditions.

The `skip` statement was introduced as a filler to satisfy syntax requirements. It is not formally part of the language but a *pseudo-statement*, a synonym for another statement with the same effect. Trivially, `skip` is equivalent to the condition `(1)`; it is always executable and has no effect.

5.12 LYNCH´s PROTOCOL REVISITED

Now that we have a language, let us try to describe the example protocol from Chapter 2. The version below is based on Figures 2.1 and 2.3, with the trial extension for accepting messages and for initializing the data transfer discussed in Section 2.4.

```
mtype = { ack, nak, err, next, accept }

proctype transfer(chan in, out, chin, chout)
{       byte o, i;

        in?next(o);
        do
        :: chin?nak(i) -> out!accept(i); chout!ack(o)
        :: chin?ack(i) -> out!accept(i); in?next(o); chout!ack(o)
        :: chin?err(i) -> chout!nak(o)
        od
}
init
{       chan AtoB = [1] of { byte, byte };
        chan BtoA = [1] of { byte, byte };
        chan Ain  = [2] of { byte, byte };
        chan Bin  = [2] of { byte, byte };
        chan Aout = [2] of { byte, byte };
        chan Bout = [2] of { byte, byte };

        atomic {
                run transfer(Ain, Aout, AtoB, BtoA);
                run transfer(Bin, Bout, BtoA, AtoB)
        };
        AtoB!err(0)
}
```

The channels `Ain` and `Bin` are to be filled with token messages of type `next` and arbitrary values (e.g., ASCII character values) by unspecified background processes: the users of the transfer service. Similarly, these user processes can read received data from the channels `Aout` and `Bout`. The processes are initialized in an atomic statement and started with the dummy `err` message.

As a last example, below is a listing in PROMELA of a viciously complex procedure to calculate Ackermann's function, which is defined recursively as

```
ack(0,b) = b+1
ack(a,0) = ack(a-1, 1)
```

```
ack(a,b) = ack(a-1, ack(a,b-1))
```

The PROMELA version is as follows.

```
/***** Ackermann's function *****/

proctype ack(short a, b; chan ch1)
{       chan ch2 = [1] of { short };
        short ans;

        if
        :: (a == 0) ->
                ans = b+1
        :: (a != 0) ->
                if
                :: (b == 0) ->
                        run ack(a-1, 1, ch2)
                :: (b != 0) ->
                        run ack(a, b-1, ch2);
                        ch2?ans;
                        run ack(a-1, ans, ch2)
                fi;
                ch2?ans
        fi;
        ch1!ans
}
init
{       chan ch = [1] of { short };
        short ans;

        run ack(3, 3, ch);
        ch?ans;
        printf("ack(3,3) = %d\n", ans);
        assert(0)       /* a forced stop, (Chapter 6) */
}
```

Seems simple enough? It takes 2433 process instantiations to produce the answer. The answer, by the way, is 61.

5.13 SUMMARY

We have introduced a notation for describing protocol procedure rules in a specification and modeling language named PROMELA. In this chapter we have discussed PROMELA features for describing system behavior only. In the next chapter we discuss the remaining language features that are specifically related to the specification of correctness criteria.

The validation modeling language has several unusual features that make it suited for modeling distributed systems. All communication between processes takes place via either messages or shared variables. Synchronous and asynchronous communication are modeled as special cases of a general message-passing mechanism.

Every statement in PROMELA can potentially model delay: it is either executable or not, in most cases depending on the state of the environment of the running process.

Process interaction and process coordination are thus at the very basis of the language. The semantics of the language make a mapping from the flow chart language used in the first part of the book to PROMELA programs straightforward. It is probably good to keep in mind that PROMELA is a modeling language, not a programming language. There are no abstract data types, and only a few basic types of variables. A validation model is an abstraction of a protocol implementation. The abstraction maintains the essentials of the process interaction so that it can be studied in isolation. It suppresses implementation and programming detail. An overview of the language can be found in Appendix C.

In the next chapters we find good use for the language. In Chapter 7 it is used in the design of a file transfer protocol. In Part IV we show how to develop the software for analyzing protocol models written in PROMELA.

EXERCISES

5-1. Assume the statement run A() is unexecutable, for instance because there were too many processes running. Can you say if b = run A() is executable? □

5-2. If the statement (qname?var == 0) were allowed in PROMELA, what would its effect be? Hint: consider the side-effects of the receive operation. □

5-3. Revise the two programs from Section 5.6 to incorporate the use of messages of type `eot` to signify the end of an input stream. □

5-4. Rewrite the declaration for process types `fact()` and `ack()` to use a global variable instead of messages to communicate the result of the calculation from a child process to its parent. □

5-5. Rewrite the `fact()` program to return the n-th Fibonacci number, `f(n) = f(n-1) + f(n-2)`, instead of a factorial. By definition, `f(0) = 0` and `f(1) = 1`. □

5-6. Rewrite your program for generating Fibonacci numbers to reduce the number of processes that is required. (Hint: make the program singly recursive; every process creates no more than one child.) □

5-7. Extend the model of Lynch's protocol with two user processes that use the transfer service. □

5-8. Extend the same program with a process type modeling a faulty transmission channel between the two users. □

5-9. Write a PROMELA program that performs a *bubble* sort on the elements of a channel that is initialized with messages of type `int`, each carrying a value. A bubble sort is done by scanning through a list of numbers repeatedly, swapping any pair of adjacent numbers that are out of order. □

5-10. Write a PROMELA program that sorts integers by building a binary tree of processes. Each process holds one integer in the sequence. It has one parent process and up to two children processes, left and right. The integers enter the sorter via the process at the root of the tree. All processes follow the same discipline. If the next integer received is larger than the one held by the receiver, it is routed to the left. If it is smaller, it is routed to the right. Children processes are created only when necessary. When the last integer has been processed, the values stored in the tree must be retrieved in the right order and

printed. □

5-11. With distributed processes, it is relatively easy to design resource managers that can, for instance, provide user programs mutually exclusive access to devices or services. Write a sample printer server that ‘‘owns’’ a `display` channel and allows processes to submit a sequence of messages to it (e.g., anything up to an `eot`) in fragments, without the possibility of interruption by other processes. □

5-12. Rewrite the example using the Dijkstra semaphore with rendezvous communication into a solution using asynchronous communication between the monitor process and the user processes. □

5-13. Consider in detail why the C data types `real` or `double` are not defined in PROMELA. □

5-14. (Paul Haahr) Many processors use ‘‘interrupt priority levels’’ to ensure that some devices get handled before others (e.g., disks are usually treated as more urgent that keyboards). The current priority level is stored in a special CPU register, usually a 3-bit or 4-bit integer. During normal operation, the priority level is zero. Each hardware device that can interrupt the CPU is assigned a priority level. When a hardware interrupt occurs, if the processor is currently running at a level less than that of the device, the processor starts running the appropriate interrupt handler; if not, the device waits until the priority drops and then interrupts. When the interrupt handler starts, the priority level is set to the device’s priority. It is reset to the previous level when the handler terminates. The processor can also set the priority level independently of the interrupt handlers, e.g., with an instruction `spl(x)`. This can be used to prevent an interrupt handler from being interrupted in the middle—when its data structures are not necessarily in a proper state—by another interrupt that will use the same structures. It is also used by operating systems to assure mutual exclusion. For example, if a device (say a disk) interrupts at level six, the device driver that runs the disk has to set the priority level temporarily to six before using data structures that may be altered by a disk interrupt. Model the interrupt priority scheme in PROMELA for three processes, modeling the behavior of a CPU, a disk process and a terminal process. (Hint: use an array to model the stack of priority levels.) □

5-15. Modify the validation model for Lynch’s protocol to model the possibility of transmission errors. □

BIBLIOGRAPHIC NOTES

PROMELA is an extension of a smaller language named `Argos` that was developed in 1983 for protocol validation, e.g., Holzmann [1985]. The syntax of PROMELA expressions, declarations, and assignments is loosely based on the language C, Kernighan and Ritchie [1978]. The language was influenced significantly by the ‘‘guarded command languages’’ of E.W. Dijkstra [1975] and C.A.R. Hoare [1978]. There are, however, important differences. Dijkstra’s language had no primitives for process interaction. Hoare’s language was based exclusively on synchronous communication. Also in Hoare’s language, the type of statements that could appear in the guards of an option was restricted. The semantics of the selection and cycling statements in PROMELA is also rather different from other guarded command languages: the statements are not aborted when all guards are false but they block, thus providing the required synchronization.

The mutual exclusion (or ‘‘critical section’’) problem, referred to briefly in this

chapter, has been studied for many years. The following intriguing series of articles documents some of the improvements that have been made: Dijkstra [1965], Knuth [1966], deBruyn [1967], Dijkstra [1968], Eisenberg and McGuire [1972], Lamport [1974, 1976, 1986]. More elaborate discussions can also be found in Bredt [1970] or Holzmann [1979].