CORRECTNESS REQUIREMENTS 6

6.1 INTRODUCTION

In Chapter 5 we developed a language for modeling behavior in distributed systems. The language is deliberately defined at a high level of abstraction to allow us to focus on design rather than on implementation issues. The programs that can be written in this language are therefore called *validation models*. The details required to convert a validation model into an implementation could presumably be filled in with relatively little effort, perhaps even mechanically. But the primary purpose of PROMELA is validation, not implementation.

To validate a design, we need to be able to specify precisely what it means for a design to be correct. A design can be proven correct only with respect to specific correctness criteria. Three fairly standard criteria were listed in Chapter 2: the absence of deadlocks, livelocks, and improper terminations. It is, for instance, never enough to just ‘‘know’’ that a design is free of deadlocks.

A good design is ***provably*** *free of deadlocks.*

There are many protocol properties that one might be interested in proving for any given design. But the problems we are dealing with are complex. It is not too hard to show that the problem of verifying even the simplest protocol properties, such as absence of deadlock, is PSPACE hard (see Bibliographic Notes), even for a finite state model. In attempting to prove the correctness of a protocol, we have to be aware of these complexity bounds. Since it is our goal to develop a validation methodology that can be applied to protocols of arbitrary size, we need to develop methods for battling the complexity from two different sides:

- □ We need a formalism for specifying correctness requirements that is not so inherently complex that effective analysis for larger models becomes impossible.
- □ We need a method for reducing the complexity of models that are beyond the range of our validation methods.

The second point, reduction, is discussed in Chapters 8 and 11 (see Section 8.9, Generalization of Machines, and Section 11.7, Complexity Management). The first point will be addressed here. It says that the more expressive we make our notation for specifying correctness requirements, the less useful it will be in practice. The set of correctness criteria that can be expressed in PROMELA is therefore chosen carefully. This set is deliberately not restricted to a single all-powerful mechanism. Several independent levels of complexity are supported. The simplest, most frequently used requirements, such as absence of deadlock, are expressed straightforwardly and checked independently of other properties. They can be analyzed mechanically with fast and frugal algorithms even for very large systems. Slightly more complicated types of requirements, such as absence of livelocks, are expressed independently, and carry an independent price-tag in computational expense when validated mechanically. The most sophisticated requirements are inevitably also the most expensive to check. In Chapter 11 we discuss the best known algorithms for the automated validation of each type of correctness requirement and quantify the size of systems that they can validate.

In the next section we give an overview of the types of correctness criteria that can be expressed for PROMELA models. Each of the sections that follow it elaborates one of these properties in detail. It shows the PROMELA language structures that are needed to express each property and gives some examples of its use.

6.2 REASONING ABOUT BEHAVIOR

We formalize correctness criteria as claims about the behavior of a PROMELA validation model. Two general types of claims are then that a given behavior is either

- ○ Inevitable or
- ○ Impossible

Since the number of possible behaviors of any given PROMELA model is finite, however, a claim of either type implicitly defines a complementary and equivalent claim of the other type. It therefore suffices to support just one.

> *All correctness criteria that can be expressed in PROMELA define behaviors that are claimed to be **impossible**.*

To state that a given behavior is inevitable, for instance, we can state that all deviant behaviors are impossible. Similarly, if a correctness assertion states that a condition is invariantly true, the correctness claim states that it is impossible for the assertion to be violated, independent of the system behavior.

Before we can continue, however, we will have to be more precise about the terms we are using. What, for instance, is a behavior, and how can we make claims about it?

The *behavior* of a validation model is defined completely by the set of all the execution sequences it can perform, where an *execution sequence* is simply a finite, ordered set of states. A *state*, in turn, is completely defined by the specification of all values for local and global variables, all control flow points of running processes, and the contents of all message channels. We say that a validation model can *reach* a given state by the execution of PROMELA statements, using the earlier defined semantics of executability. A validation model can also *be placed* in a given state by an assignment of the appropriate values to variables, control flow points and channels.

Of course, not just any arbitrary collection of states is also a valid execution sequence. A finite, ordered set of states is said to be *valid* for a given PROMELA model ***M*** if it satisfies the following two criteria:

- ☐ The first state of the sequence, i.e., the state with ordinal number 1, is the initial system state of ***M***, with all variables initialized to zero, all message channels empty, with only the `init` process active, and set in its initial state.
- ☐ If ***M*** is placed in the state with ordinal number *i*, there is at least one executable statement that can bring it to the state with ordinal number $i+1$.

We will consider two special types of sequences, called terminating and cyclic sequences.

- ☐ An execution sequence is said to be *terminating* if no state occurs more than once in the sequence, and the model ***M*** contains no executable statements when placed in the last state of the sequence.
- ☐ An execution sequence is said to be *cyclic* if all states except the last one are distinct, and the last state of the sequence is equal to one of the earlier states.

Cyclic sequences define potentially infinite executions. All terminating and cyclic execution sequences that can be generated by executing a PROMELA model, together define the *system behavior* of that model. The union of all states included in the system behavior is called *the set of reachable states* of the model.

PROPERTIES OF STATES

Correctness claims for PROMELA models can be built up from simple propositions, where a proposition is a boolean condition on the state of the system. The propositions can refer to all the elements of a system state: local and global variables, control-flow points of arbitrary executing processes, and the contents of message channels. Some of the notation for this was discussed in Chapter 5; the remaining features will be introduced shortly.

The propositions implicitly define a labeling of states. In any given state a proposition is either true or false. Correctness criteria then can be expressed in terms of states, e.g., by defining explicitly in which states a given proposition is required to hold. Some of these requirements can be specified in PROMELA with, for instance, *assertion* statements that are embedded in the model. This mechanism by itself, however, is not sufficient. If more than one proposition is used, we may want to express a correctness requirement as a temporal ordering of propositions, i.e., by specifying the order in which propositions are required to hold (with the truth of one proposition either immediately or eventually following the truth of another). Alternatively, the temporal ordering can define the order in which propositions should never hold. As indicated above, these two alternatives for defining temporal orderings are complementary. Only the second alternative is supported in PROMELA. The formalism to support this is a new language feature called a *temporal claim*.

TEMPORAL CLAIMS

In the formalization of temporal claims we specify an ordering of propositions. It is important to note that the semantics of these *proposition* orderings is different from the semantics of *statement* orderings elsewhere in a PROMELA model. Within `proctype` definitions, a sequential ordering of two statements implies that the second statement is to be executed after the first one terminates. Since we are not allowed to make any assumptions about the relative speeds of concurrently executing processes, the only valid interpretation of the word *after* in the above sentence is *eventually after*. Correctness claims, however, may have to be more specific. In a temporal claim a sequential ordering of two propositions defines an *immediate* consequence. We will show later how other types of temporal relations can also be specified with this mechanism.

The types of correctness requirements we make can be different for terminating and cyclic sequences, as are the algorithms we will ultimately need to check these claims. An important requirement that applies to terminating sequences is, for instance, absence of deadlock. Not all terminating sequences, however, correspond to deadlocks. We will have to be able to express which properties the final state in a sequence must have to make that sequence acceptable as a non-deadlocking terminating sequence. For cyclic sequences, finally, we should be able to express general conditions such as the absence of livelock.

OVERVIEW

The remainder of this chapter is devoted to a more detailed discussion of the formalization of correctness criteria in PROMELA. It will introduce the last few language constructs that specifically deal with validation:

- The `assert()` statement, which can be used to express both local assertions and global system invariants
- Three types of labels that can be used to define a small class of frequently used correctness claims for terminating and cyclic sequences
- The formalization of general temporal claims
- The notation that can be used in assertions and in temporal claims to refer to the control-flow states and the local variables of arbitrary running processes

We begin by taking a closer look at the specification of the correctness properties of states.

6.3 ASSERTIONS

Correctness criteria can often be expressed as boolean conditions that must be satisfied whenever a process reaches a given state. The PROMELA statement

```
assert(condition)
```

is always executable and can be placed anywhere in a PROMELA model. The condition can be an arbitrary boolean expression. If the condition is true, the statement has no effect. The validity of the statement is violated, however, if there is at least one execution sequence in which the condition is false when the `assert` statement becomes executable.

Consider the following example from Chapter 5.

```
byte state = 1;

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

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

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

We could try to claim that when a process of type `A()` completes the value of variable `state` must be 2, and when a process of type `B()` completes it must be 0. This could be expressed as follows.

```
byte state = 1;

proctype A()
{       (state == 1) -> state = state + 1;
        assert(state == 2)
}
proctype B()
{       (state == 1) -> state = state – 1;
        assert(state == 0)
}
init { run A(); run B() }
```

The claims are, of course, false, and an automated validator would demonstrate that quickly.

6.4 SYSTEM INVARIANTS

A more general application of the `assert` statement is to formalize system invariants, i.e., boolean conditions that, if true in the initial system state, remain true in *all* reachable system states, independently of the execution sequence that leads to each specific state. To express this in PROMELA, it suffices to place the system invariant by itself in a separate, monitor process.

```
proctype monitor() { assert(invariant) }
```

Once an instance of the process type `monitor` has been started (the name is immaterial), with a regular `run` statement, it executes independently of the rest of the system. It can decide to evaluate the assertion at any time; its `assert` statement is executable precisely once for every state of the system.

For a simple application of this type of correctness claim, consider the `dijkstra` semaphore validation model, introduced in Chapter 5.

```
#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 mutually exclusive access of user processes to their critical sections. We can modify the user processes as follows, to count the number of processes in the critical section in a global variable `count`.

```
byte count;

proctype user()
{       sema?p;
        count = count+1;
        skip;   /* critical section */
        count = count-1;
        sema!v;
        skip    /* non-critical section */
}
```

The following system invariant can now be used to verify the correct operation of the semaphore:

```
proctype monitor() { assert(count == 0 || count == 1) }
```

An instantiation of the monitor must be included in the initial process, to allow it to perform the correctness check.

```
init
{       atomic {
                run dijkstra(); run monitor();
                run user(); run user(); run user()
        }
}
```

6.5 DEADLOCKS

In a finite state system, all execution sequences either terminate after a finite number of state transitions, or they cycle back to a previously visited state. Not all terminating sequences, however, are necessarily deadlocks. In order to define what a deadlock in a PROMELA model is, we must be able to distinguish the expected, or *proper*, end-states from the unexpected ones. The unexpected end-states will include not just deadlock states, but also many error states that are the result of a logical incompleteness of the protocol specification. The classic example of the latter is the *unspecified reception*.

The final state in a terminating execution sequence must minimally satisfy the following two criteria to be considered a proper end-state:

- ○ Every process that was instantiated has terminated
- ○ All message channels are empty

But not all processes necessarily terminate. It can be perfectly valid, for instance, for server processes to stay alive after user processes terminate. We must be able, therefore, to identify individual process states in `proctype` definitions as proper end-states. In PROMELA this can be done with *end-state* labels. In the semaphore example from Chapter 5, for instance, we can write

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

to define that any process of type `dijkstra` is considered to be in a proper end-state when it is in the state labeled `end`.

If there is more than one proper end-state within a single `proctype` definition, all label-names must still be unique. An end-state label is defined to be any label-name that has a three-character prefix `end`. So it is valid to use variations such as `enddne`, `end0`, `end_war`. We can now revise the first criterion from the definition of a proper end-state:

- ○ Every process that was instantiated has either terminated or has reached a state marked as a proper end-state

Any final state in a terminating execution sequence that does not satisfy the two criteria for proper end-states is automatically classified as an improper end-state. An implicit correctness claim that is made about all validation models will be that the behaviors they define do not include any improper end-states.

Refer to Chapter 14 and Appendix F for some examples of the use of end-state labels in a real validation.

6.6 BAD CYCLES

Two properties of cyclic sequences can be expressed in PROMELA, corresponding to two standard types of correctness requirements. Both properties are based on the explicit marking of states in a validation model. The first property specifies that

- ○ There are no infinite behaviors of only unmarked states

that is, the system cannot infinitely cycle through unmarked states. The marked states are called *progress-states*, and the execution sequences that violate the above correctness claim are called *non-progress cycles*. The second property is the opposite of the first. It is used to specify that

- ○ There are no infinite behaviors that include marked states

Execution sequences that violate this claim are called *livelocks*. We discuss each type of correctness claim in more detail below.

NON-PROGRESS CYCLES

To claim the absence of non-progress cycles, we must be able to define the system states within the PROMELA model that denote progress. These progress states are defined much like end-state labels.

A progress-state label marks a state that *must* be executed for the protocol to make progress. An example can be the incrementing of a sequence number, or the delivery of data to a receiver. In the semaphore example we can label the successful passing of a semaphore test as ‘‘progress.’’ Simply by marking it as a progress state we can express the correctness criterion that the passing of the semaphore guard cannot be postponed infinitely long, e.g., by an infinite execution cycle that does not pass the progress state.

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

An automated validator can readily confirm that indeed this claim cannot be violated. If more than one state carries a progress-state label, variations with a common prefix are again valid: `progress0`, `progressisslow`, and so on.

LIVELOCKS

Suppose we wanted to express the opposite of a progress condition, e.g., we want to formalize that something cannot happen infinitely often. We can express properties like this with the third, and last, class of special PROMELA labels. In addition to the end-state, and progress-state labels introduced earlier, we now define acceptance-state labels. An *acceptance-state label* is any label starting with the character sequence ‘‘accept.’’ It marks a state that *may not* be part of a sequence of states that can be repeated infinitely often.

For example, if we replace the progress-state label in `proctype dijkstra()` with an acceptance-state label

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

we claim that it is impossible to cycle through a series of `p` and `v` operations. We know, of course, that this claim is false. We can either prove that it is false manually, or we can use an automated validator to provide a counter-example.

Again, all variations, such as `acceptor`, `acceptable`, and `accept_yo`, are allowed. In principle, we could make the best use of an acceptance state if we could use it to express complete behaviors that are required to be impossible, rather than only the absence of a designated state in all cycles. We will do precisely that by using the labels to define the acceptance states of special claim automata that model error behaviors. (This also explains the choice of the term ''accept.'') These automata express general temporal claims.

6.7 TEMPORAL CLAIMS

Up to this point we have talked about the specification of correctness criteria with assertions and with three special types of labels that can be used to mark end states, progress states, and acceptance states. Powerful types of correctness criteria can already be expressed with these tools, yet so far our only option is to add them to `proctype` definitions. Suppose that, within this framework, we want to express the temporal claim ''every state in which property ***P*** is true is followed by a state in which property ***Q*** is true.'' We noted before that two different interpretations of the term ''followed by'' are possible, depending on whether the two states must *immediately* or *eventually* follow each other. It is basic to the semantics of PROMELA that no assumptions whatsoever can be made about the relative timing of process executions. This means that, so far, the only legitimate interpretation of the above term is that two steps ''eventually'' follow each other (including ''immediately'' as a special case). That, however, leaves us with the problem to express the other types of properties. For this we need a different type of validation primitive.

Temporal claims define temporal orderings of *properties* of states. To express the requirement that ''every state in which property ***P*** is true is followed by a state in which property ***Q*** is true,'' we could write

```
P -> Q
```

But, it's not quite that simple. There are two snags.

- Since all our correctness criteria are based on properties that are claimed to be *impossible*, the temporal claims we use must also express *orderings* of properties that are impossible.
- The temporal claims are defined on *complete* execution sequences (terminating or

cyclic). Even if a prefix of the sequence is irrelevant, it must still be represented as a trivially-true sequence of propositions.

The requirement above should be expressed, therefore, as

```
never { do :: skip :: break od -> P -> !Q }
```

that is, independent of the initial sequence of events, it is impossible for a state in which property ***P*** is true to be followed by a state in which property ***Q*** is false. The claim is *matched*, and the corresponding correctness property thereby violated, if and when the claim body terminates.

The PROMELA notation for a temporal claim is

```
never  { ... }
```

where the dots contain the details of the claim.

Temporal claims can be primed with progress-state or acceptance-state labels to catch more types of errors than just a complete match of a terminating behavior. The `never` claims can be expressed, for instance, as special finite state machines that cycle through an acceptance state if the undesirable behavior is recognized.

Suppose we wanted to express the temporal property that `condition1` can never remain true infinitely long. To catch violations of this property, we must find a representation in a temporal claim of all behaviors where `condition1` may be false initially, becomes true eventually, and remains true. It is expressed as follows

```
never {
        do
        :: skip
        :: condition1 -> break
        od;
accept: do
        :: condition1
        od
}
```

This (non-terminating) claim is *matched*, and the corresponding correctness property violated, if and when the acceptance cycle is detected. The tricky part is to remember the inclusion of the `skip` (a condition that is always true) in the first loop. Note that sequences where the truth value of `condition1` first changes from true to false a few times are permitted by the claim.

The claim itself is simply a finite state machine, with a proposition defined in every state. For every state transition elsewhere in the validation model, i.e., by the execution of a PROMELA statement, the claim machine must change its state and move from one proposition to the next. To match a temporal claim, at every state in a sequence of states the proposition at corresponding state in the claim machine must be true. If we write erroneously, for instance

```
never {
        do
        :: skip
        :: condition1 -> break
        od;
accept:
        condition1
}
```

the claim contains just two state transitions after `condition1` becomes true. This claim is completely matched if there is at least one execution sequence in which `condition1` holds in two subsequent states.

The finite state machines specified in the claims above contain three states each: the initial state, the state labeled `accept` and the normal end state. The first, correct, version of the claim specifies that it would be an error (a livelock) if the machine can stay in the second state infinitely long. The second version specified that it would be an error if the third (terminal) state is reachable.

SPECIFYING TEMPORAL CLAIMS

The body of a temporal claim is defined just like PROMELA `proctype` bodies. This means that all control flow structures, such as `if-fi` selections, `do-od` repetitions, and `goto` jumps, are allowed. There is, however, one important difference:

Every statement inside a temporal claim is (interpreted as) a condition.

Specifically, this means that the statements inside temporal claims should be free of side-effects. For reference, the PROMELA statements with side-effects are: assignments, assertions, sends, receives, and `printf` statements.

Temporal claims are used to express system behaviors that are considered undesirable or illegal. We say that a temporal claim is *matched* if the undesirable behavior can be realized, and thus our correctness claim can be violated.

The most useful application of temporal claims is in combination with acceptance labels. There are then two ways to *match* a temporal claim, depending on whether the undesirable behavior defines terminating or cyclic execution sequences.

- □ For a terminating execution sequence, a temporal claim is matched only when it can terminate (reaches the closing curly brace) That is, the claim can be violated if the closing curly brace of the PROMELA body of the claim is reachable.
- □ For a cyclic execution sequence, the claim is matched only when an explicit acceptance cycle exists. The acceptance labels within temporal claims are user defined, there are no defaults. This means that in the absence of acceptance labels no cyclic behavior can be matched by a temporal claim. It also means that to check a cyclic temporal claim, acceptance labels should only occur within the claim and not elsewhere in the PROMELA code.

`Never` claims, used in combination with acceptance-state labels, can express also the absence of non-progress cycles. The claims are therefore more general than progress-state labels. The expense (complexity) of finding non-progress cycles directly with progress-state labels, however, is smaller than the expense of the validation of a claim that specifies the same property.

To get the full benefit of temporal claims, we must be able to refer to the control-flow states and the variable values of running processes. As an example, consider the following version of the alternating bit protocol.

```
/* alternating bit - version with message loss */

#define MAX 3

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

chan        sender  =[1] of { byte };
chan        receiver=[1] of { byte };

proctype Sender()
{   byte any;
again:
    do
    :: receiver!msg1;
            if
            :: sender?ack1 -> break
            :: sender?any /* lost */
            :: timeout    /* retransmit */
            fi
    od;
    do
    :: receiver!msg0;
            if
            :: sender?ack0 -> break
            :: sender?any /* lost */
            :: timeout    /* retransmit */
            fi
    od;
    goto again
}

proctype Receiver()
{   byte any;
again:
    do
    :: receiver?msg1 -> sender!ack1; break
    :: receiver?msg0 -> sender!ack0
    :: receiver?any /* lost */
    od;
P0:
    do
    :: receiver?msg0 -> sender!ack0; break
    :: receiver?msg1 -> sender!ack1
    :: receiver?any /* lost */
    od;
P1:
    goto again
}

init { atomic { run Sender(); run Receiver() } }
```

Processes `Receiver` and `Sender` communicate via message channels named `receiver` and `sender`. Each channel can hold one message of type `byte`.

Message loss is modeled explicitly in the sender and the receiver processes with a clause that can steal an incoming message before it is processed (lines 15, 23, 36, and 42). We may want to express the claim ‘‘it is always true that when the sender transmits a message, the receiver will eventually accept it.’’ Our first job is again to find the corresponding undesirable property that can be expressed in a temporal claim. To be able to specify this, however, we need to be able to refer to states inside the sender and receiver processes. The required notation is used in the following formalization of the claim.

```
never {
        do
        :: skip /* allow any time delay */
        :: receiver?[msg0] -> goto accept0
        :: receiver?[msg1] -> goto accept1
        od;
accept0:
        do
        :: !Receiver[2]:P0
        od;
accept1:
        do
        :: !Receiver[2]:P1
        od
}
```

The claim above is a four-state machine: the inital state, the two states that were labeled, and the normal end state. At least one of three conditions must be true in the initial system state. The claim remains in this state as long as channel `receiver` is empty. If it contains a message[1] `msg0` or `msg1` it will change state to either `accept0` or `accept1`, depending on the message that was matched. Once the transition to, for instance, state `accept0` has been made, the claim can only remain in that state if the receiver process will never accept a message with the same sequence number, i.e., if the receiver process never passes the state labeled `P0`.

There can be many instantiations of the process type `Receiver` so we need some way of specifying exactly which particular instantiation we mean when we refer to the state of a process. This is the only time that we need to be able to refer to the *instantiation number* or the `pid` of a process. The `pid` of a process is the number that is returned by the `run` operator, when a process is instantiated. The `pid`s are assigned in the order in which processes are started, but they may be recycled when processes die. The initial process always has pid zero, and its number is never recycled. A `pid` can usually easily be inferred from the program text. Since the receiver process is the second process that is instantiated in this system, its `pid` is two. We can refer to the receiver process unambiguously as `Receiver[2]`. The condition that the receiver is currently in the state labeled `P0` is expressed as `Receiver[2]:P0`. The condition is false whenever the second process that was instantiated is in any state other than at

1. The notation for a side-effect free inspection of the contents of a message channel was introduced in Chapter 5 on page 98.

label `P0` of process type `Receiver`.

The notation `Receiver[2]:P0` is a special case of a more general construct which allows temporal claims to refer to internal conditions of the asynchronous processes defined within a PROMELA model. A reference to the current value of local variable `any` in the receiver process, for instance, is written `Receiver[2].any`. It can be used in arbitrary expressions, such as

```
assert(Receiver[2].any < 0)
```

In process references, a colon is used to refer to labels (i.e., control flow states) and a period is used to refer to local variables. In the first case, the value returned is a boolean. In the second case it is the integer value of the variable specified.

The example claim can be proven for the alternating bit protocol as specified, using the default semantics of `timeout`. This means that the claim is matched provided that the retransmission timers behave as intended: there will be no retransmission unless a message was lost. To check also the rather perverse case where timeouts can fire at any time, independently of message loss, we could add the macro definition

```
#define timeout skip
```

Now the temporal claim can be violated. Counter-examples are easily produced with an automated validator, such as the one discussed in Chapters 11-14.

6.8 SUMMARY

In this chapter we have introduced all the remaining language features of the validation modeling language PROMELA. All are directly related to the specification of the correctness requirements of a model. They are

- Assertion statements
- End-state, progress-state, and acceptance-state labels
- The temporal claim `never`
- The notation for referring to the control-flow states and local variable values of running processes within assertions and temporal claims

The order in which we have introduced the tools for expressing correctness properties corresponds roughly to an increasing level of sophistication in performing validations.

In the initial stages of a design, a user is unlikely to use more than assertions and perhaps end-state labels. In the final stages of a design, when all initial flaws have been corrected and a more precise qualitative assessment of the design can be made, validations with explicit temporal claims may be developed. In many cases this level of sophistication in a validation is never required and all the necessary properties can be established without it.

It is almost impossible to manually verify correctness requirements such as the ones we have discussed. The behavior of even very simple protocol systems is of a staggering complexity that no designer can be expected to assess accurately. Tools are needed not only to express the correctness requirements of a protocol design, but also to verify them reliably. In the next chapters we show how we can use a language such as PROMELA for systematic protocol design and how automated systems can be developed to support efficient validations of correctness claims.

EXERCISES

6-1. The assertion used to verify mutual exclusion in the semaphore example was formalized as a global system invariant. Find another place for the assertion to perform the same check without an extra monitor process. □

6-2. Find the temporal claim `never` that expresses the same correctness requirement as the ‘‘ordinary’’ `proctype` definition

```
proctype monitor() { (invariant) -> assert(invariant) }
```

Explain the difference in semantics of the semicolon (or arrow). □

6-3. Consider Dekker’s algorithm from Chapter 5. Make a new PROMELA model, where the processes repeatedly access their critical sections. Express the correctness requirement that no two processes can be in their critical sections simultaneously, with

- ○ An assertion
- ○ A system invariant
- ○ End-state labels
- ○ A temporal claim `never` (combined with acceptance-state labels)

End-state labels can be used in this problem, for instance, by forcing the system into an improper end-state when the exclusion is violated. You are allowed to introduce extra global ‘‘state’’ variables, e.g., to count the number of processes inside the critical section. □

6-4. Repeat Exercise 6-3, but this time do not use any global variables in the assertions or in the temporal claim. Define extra ‘‘ordinary’’ labels to define control flow points in the program bodies that can be monitored in assertions and claims. □

6-5. For the model from Exercise 6-4, express the correctness requirement that no single process can monopolize access to the critical section. Express the claim that within finite time after access is attempted, access is granted. Use

- ○ Progress-state labels
- ○ Acceptance-state labels
- ○ A temporal claim `never` combined with acceptance-state labels

(Three different solutions.) Add variables and labels as needed. □

6-6. All ‘‘linear-time propositional temporal logic formulas’’ (see Bibliographic Notes) can be expressed in PROMELA with acceptance-state labels and temporal claims. To illustrate this, find PROMELA representations of the following requirements

- ○ Eventually, proposition ***p*** always remains true
- ○ ***p*** is always true until ***q*** becomes true
- ○ Eventually, ***p*** is always true until ***q*** becomes true

Can you discover a pattern in the modeling of these formulas into PROMELA temporal claims? Could it be automated? □

6-7. Find a way translate arbitrary temporal claims back into temporal logic formulas. □

6-8. In temporal claims, consider if the following construction is equivalent to `assert(0)`

```
accept:do :: skip od
```

□

BIBLIOGRAPHIC NOTES

The definition of terminating and cyclic execution sequences to reason about the behavior of a distributed system is described in Owicki and Lamport [1982], Manna and Wolper [1984], and Snepscheut [1985]. The formalization of correctness requirements in a notation that is based on temporal logic formulas was first explored in Pnueli [1977]. It was soon also applied to the study of concurrent systems. Early applications are the French validation system Cesar, Queille [1982], and Clarke's model checking system, Clarke [1982], Browne, Clarke, Dill, and Mishra [1986].
Formally, the temporal claims defined in this chapter describe nondeterministic Büchi automata, a special class of the $\forall$-automata described in Manna and Pnueli [1987]. In Wolper [1981] it was shown that Büchi automata have the expressive power of an extended type of temporal logic formulas.

PSPACE hardness is a measure for the complexity of algorithms. Informally, a problem that is PSPACE hard is known to have no efficient solution. For a detailed discussion see Garey and Johnson [1979].