Chapter 4. Defining Correctness Claims

> If the odds are a million to one against something occurring, chances are fifty−fifty that it will.
>
> —(Folklore wisdom)

The goal of system verification is to establish what is possible and what is not. Often, this assessment of what is logically possible will be subject to some set of assumptions about the context in which a system executes, such as the possible behavior of external components that the system will be interacting with. When performing logical verification we are especially interested in determining whether design requirements could possibly be violated, not necessarily in how likely or unlikely such violations might be. Dramatic system failures are almost always the result of seemingly unlikely sequences of events: that is precisely why these sequences of events are often overlooked in the design phase. Once we understand how a requirement may be violated, we can reconsider the original design decisions made, and devise alternate strategies that can prevent the error from occurring. Logical correctness, then, is concerned primarily with possibilities, not with probabilities.

Stronger Proof

This restriction to the possible, rather than the probable, has two implications. For one, it can strengthen the proofs of correctness we can achieve with system verification. If the verifier tells us that there is no possible violation of a given requirement, this is a significantly stronger result than the verdict that violating executions have a low probability of occurrence. Secondly, the restriction makes it possible to perform verification more efficiently than if we attempted to consider also the probability of execution scenarios. The results of probability estimates are further undermined by the difficulty one would face in deriving accurate metrics for the probabilities of execution of specific statements in the system. Any errors of judgment made here are magnified in the verification process, limiting the value of the final results.

We should be able to prove the essential logical correctness properties of a distributed system independently of any assumption about the relative speeds of execution of processes, the time it takes to execute specific instructions, or the probability of occurrence of particular types of events, such as the loss of messages on a transmission channel or the failure of external devices.

The proof of correctness of an algorithm is ideally also implementation independent. Specifically, the correctness of the algorithm should not depend on whether it is implemented on a fast machine or on a slow machine. In the verification process, therefore, we should not rely on such assumptions. It is even desirable that we cannot make such statements at all. Not surprisingly, PROMELA effectively makes it impossible to state any correctness requirement that would violate these rules.

The rules we follow here are specific to our area of interest: the verification of distributed systems software. Different rules apply in, for instance, hardware verification. The correctness of a chip may well depend critically, and unavoidably, on signal propagation delays and the speed of individual circuit elements. Signal propagation times and the layout of circuit elements are part of a chip's design and functionality: they cannot be changed independently from it. The correctness of a data communications protocol or a distributed operating system, on the other hand, should never depend on such issues. The speed of execution of a software system is almost guaranteed to change dramatically over the lifetime of the design.

Basic Types of Claims

In distributed systems design it is standard to make a distinction between two types of correctness requirements: safety and liveness. Safety is usually defined as the set of properties that the system may not violate, while liveness is defined as the set of properties that the system must satisfy. Safety, then, defines the bad things that should be avoided, and liveness defines the good things that capture the required functionality of a system. The function of a verification system, however, is not as lofty. It need not, and indeed cannot, determine what is good or bad; it can only help the designer to determine what is possible and what is not.

From the point of view of the verifier there are also two types of correctness claims: claims about reachable or unreachable states and claims about feasible or infeasible executions (i.e., sequences of states). The former are sometimes called state properties and the latter path properties. Paths, or executions, can be either finite or infinite (e.g., cyclic).

A simple type of state property is a system invariant that should hold in every reachable state of the system. A slightly weaker version is a process assertion that should hold only in specific reachable states. State properties can be combined to build path properties. An example of a path property is, for instance, that every visit to a state with property P must eventually be followed by a visit to a state with property Q, without in the interim visiting any state with property R. The verifier SPIN can check both state and path properties, and both can be expressed in the specification language PROMELA.

Some types of properties are so basic that they need not be stated explicitly. SPIN checks them by default. One such property is, for instance, the absence of reachable system deadlock states. The user can, however, modify the semantics of also the built−in checks through a simple form of statement labeling. A deadlock, for instance, is by default considered to be an unintended end state of the system. We can tell the verifier that certain specific end states are intended by placing end−state labels, as we shall see shortly.

Correctness properties are formalized in PROMELA through the use of the following constructs:

- Basic assertions
- End−state labels
- Progress−state labels
- Accept−state labels
- `Never` claims
- Trace assertions

`Never` claims can be written by hand, or they can (often more easily) be automatically generated from logic formulae or from timeline property descriptions. We will discuss each of these constructs in more detail below.

Basic Assertions

Statements of the form

```
assert(expression)
```

are called basic assertions in PROMELA to distinguish them from trace assertions that we will discuss later. The usefulness of assertions of this type was recognized very early on. A mention of it can even be found in the work of John von Neumann (1903−1957). It reads as follows. Of course, the letter C that is used here does not refer to programming language that would be created some 30 years later:

> It may be true, that whenever C actually reaches a certain point in the flow diagram, one or more bound variables will necessarily possess certain specified values, or possess certain properties, or satisfy certain relations with each other. Furthermore, we may, at such a point, indicate the validity of these limitations. For this reason we will denote each area in which the validity of such limitations is being asserted, by a special box, which we call an 'assertion box.'
>
> —Goldstein and von Neumann, 1947

PROMELA basic assertions are always executable, much like assignments, and print or `skip` statements. The execution of this type of statement has no effect provided that the expression that is specified evaluates to the boolean value true, or alternatively to a non−zero integer value. The implied correctness property is that it is never possible for the expression to evaluate to false (or zero). A failing assertion will trigger an error message.

As also noted in Chapter 2 (p. 18), the trivial model:

```
init { assert(false) }
```

results in the following output when executed:

```
$ spin false.pml
spin: line   1 "false.pml", Error: assertion violated
#processes: 1
  1:   proc 0 (:init:) line 1 "false.pml" (state 1)
1 process created
```

Here SPIN is used in simulation mode. Execution stops at the point in the model where the assertion failure was detected. When the simulation stops, the executing process, with pid zero, is at the internal state numbered one. The simulator will always list the precise state at which execution stops for all processes that

have been initiated but that have not yet died. If we change the expression used in the assertion to true, no output of note will appear, because there are no running processes left when the execution stops with the death of the only process.

```
$ spin true.pml
1 process created
```

An assertion statement is the only type of correctness property in PROMELA that can be checked during simulation runs with SPIN. All other properties discussed in this chapter require SPIN to be run in verification mode to be checked. If SPIN fails to find an assertion violation in any number of simulation runs, this does not mean that the assertions that are embedded in the model that is simulated cannot be violated. Only a verification run with SPIN can establish that result.

Meta Labels

Labels in a PROMELA specification ordinarily serve merely as targets for unconditional `goto` jumps. There are three types of labels, though, that have a special meaning when SPIN is run in verification mode. The labels are used to identify:

- End states
- Progress states
- Accept states

End States:

When a PROMELA model is checked for reachable deadlock states, using SPIN in verification mode, the verifier must be able to distinguish valid system `end` states from invalid ones. By default, the only valid end states, or termination points, are those in which every PROMELA process that was instantiated has reached the end of its code (i.e., the closing curly brace in the corresponding `proctype` body). Not all PROMELA processes, however, are meant to reach the end of their code. Some may very well linger in a known wait state, or they may sit patiently in a loop ready to spring back to action when new input arrives.

To make it clear to the verifier that these alternate end states are also valid, we can define special labels, called `end`–state labels. We have done so, for instance, in Figure 4.1 in process type `Dijkstra`, which models a semaphore with the help of a rendezvous port `sema`. The semaphore guarantees that only one of three user processes can enter its critical section at a time. The `end` label defines that it is not an error if, at the end of an execution sequence, the process has not reached its closing curly brace, but waits at the label. Of course, such a state could still be part of a deadlock state, but if so, it is not caused by this particular process.

Figure 4.1 Labeling End States

```
mtype { p, v };

chan sema = [0] of { mtype };

active proctype Dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
                sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}

active [3] proctype user()
{       do
        :: sema?p;        /* enter */
critical: skip;           /* leave */
          sema!v;
        od
}
```

There can be any number of end−state labels per PROMELA model, provided that all labels that occur within the same `proctype` body remain unique.

To allow the use of more than one end−state label within the same `proctype` body, PROMELA uses the rule that every label name that starts with the three−letter prefix `end` defines an end−state label. The following label names, therefore, are all counted as valid end−state labels: `endme`, `end0`, `end_of_this_part`.

In verification mode, SPIN checks for invalid end states by default, so no special precautions need to be made to intercept these types of errors. If, on the other hand, the user is not interested in hearing about these types of errors, the run−time flag `−E` can be used to suppress these reports. In a similar way, using the run−time flag `−A` we can disable the reporting of assertion violations (e.g., if we are hunting for other types of errors that may appear only later in a verification run). To disable both types of reports for the sample model in Figure 4.1, we would proceed as follows:

```
$ spin -a dijkstra.pml
$ cc -o pan pan.c
$ ./pan -E -A   # add two restrictions
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim            - (none specified)
        assertion violations   - (disabled by -A flag)
        acceptance cycles      - (not selected)
        invalid end states     - (disabled by -E flag)

State-vector 36 byte, depth reached 8, errors: 0
      15 states, stored
       4 states, matched
      19 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573  memory usage  (Mbyte)

unreached in proctype Dijkstra
        line 14, state 10, "-end-"
        (1 of 10 states)
unreached in proctype user
        line 22, state 7, "-end-"
(1 of 7 states)
```

The output is virtually identical to the one we would get if we had not used either the `−A` or the `−E` option, since there are neither invalid end states nor assertion violations in this model. As a reminder of the restrictions used, though, the verifier duly notes in its output that it has in fact disabled both types of reports.

A SPIN generated verifier can also be asked to impose a stricter requirement on end states than the defaults sketched above. If the additional run−time option `−q` is used with the compiled verifier, all processes must have reached a valid end state and all message channels must be empty for a system state to be considered valid. In the normal case, that is without the `−q` option, the requirement on the emptiness of message channels is omitted.

Progress States:

Similar syntax conventions apply to the use of PROMELA `progress` labels. We can use progress labels to mark statements in a PROMELA model that accomplish something desirable, signifying that the executing process is making effective progress, rather than just idling or waiting for other processes to make progress. We can use the verification mode of SPIN to verify that every potentially infinite execution cycle that is permitted by a model passes through at least one of the progress labels in that model. If cycles can be found that do not have this property, the verifier can declare the existence of a non−progress loop, corresponding to possible starvation.

We can, for instance, place a progress label in the `Dijkstra` example from Figure 4.1, as follows:

```
active proctype Dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
progress:       sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

We interpret the successful passing of a semaphore test as progress here and ask the verifier to make sure that in all infinite executions the semaphore process reach the progress label infinitely often.

To run a check for the absence of non−progress cycles we have to compile the verifier with a special option that adds the right type of temporal claim to the model (we will show the details of that claim on p. 93). The check then proceeds as follows:

```
$ spin -a dijkstra_progress.pml
$ cc -DNP -o pan pan.c  # enable non-progress checking
$ ./pan -l              # search for non-progress cycles
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 40 byte, depth reached 18, errors: 0
      27 states, stored (39 visited)
      27 states, matched
      66 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573 memory usage (Mbyte)
```

```
unreached in proctype Dijkstra
       line 14, state 10, "-end-"
       (1 of 10 states)

unreached in proctype user
        line 22, state 7, "-end-"
        (1 of 7 states)
```

At the start of the output we see that the search for non−progress cycles was enabled, and that a `never` claim was used. The latter may be surprising, since our model did not contain such a claim. `Never` claims are usually user−defined, or derived from logic formulae. A claim can also be generated by SPIN internally, though, to support a predefined check. In this case, the claim was generated by SPIN and automatically inserted into the model to define a check for the non−progress property. The insertion of the claim was triggered by the use of compiler directive `−DNP`. We will cover the purpose and use of user−defined `never` claims in more detail later in this chapter.

The output from the verifier also tells us that a check for both assertion violations and for non−progress cycles was performed. The error count is zero, which means that no assertion violations or non−progress cycles were found. We can conclude that the model from Figure 4.1 permits no infinite executions that do not contain infinitely many semaphore `P` operations.

By enabling the search for non−progress properties (a liveness property), we automatically disabled the search for invalid end states (a safety property). It is also worth noting that, compared to our last check, the number of reachable states has almost doubled. When we discuss the search algorithms that are used in SPIN for checking the various types of properties (Chapter 8, p. 167) we will see what the cause is.

If more than one state carries a progress label, variations with a common prefix are again valid, such as `progress0`, or `progression`.

As a very simple example of what a non−progress cycle might look like, consider the following contrived model with two processes. [1]

> [1] Note that the two `proctype` bodies are equal. We could, therefore, also have defined the same behavior with two instantiations of a single `proctype`.

```
byte x = 2;

active proctype A()
{       do
        :: x = 3 - x
        od
}

active proctype B()
{       do
        :: x = 3 - x
        od
}
```

Clearly, the two processes will cause the value of the global variable `x` to alternate between 2 and 1, ad infinitum. No progress labels were used, so every cycle is guaranteed to be a non−progress cycle.

We perform the check for non−progress cycles as before:

```
$ spin -a fair.pml
$ cc -DNP -o pan pan.c  # enable non-progress checking
$ ./pan -l              # search for non-progress cycles
pan: non-progress cycle (at depth 2)
pan: wrote fair.pml.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim            +
        assertion violations   + (if within scope of claim)
        non-progress cycles    + (fairness disabled)
        invalid end states     - (disabled by never claim)

State-vector 24 byte, depth reached 7, errors: 1
       3 states, stored (5 visited)
       4 states, matched
       9 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573 memory usage (Mbyte)
```

As expected, a non−progress cycle is found. Recall that when the verifier finds an error it writes a complete trace fpr the error execution into a file called `pan.trail`. Using that file, we can reproduce the error trail with the help of SPIN's guided simulation option, for instance, as follows:

```
$ spin -t -p fair.pml
spin: couldn't find claim (ignored)
  2: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
  4: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
  <<<<<START OF CYCLE>>>>>
  6: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
  8: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
spin: trail ends after 8 steps
#processes: 2

                x = 2
  8: proc 1 (B) line 11 "fair.pml" (state 2)
  8: proc 0 (A) line 5  "fair.pml" (state 2)
2 processes created
```

The warning that SPIN could not locate the claim for this error trail is innocent: the claim we used to state that non−progress cycles cannot exist was predefined. (This claim turns out to be false for this model.) The steps

that do not appear in the numbered output (steps 1, 3, 5, and 7) are the steps that were executed by the hidden claim automaton.

Fair Cycles

The counterexample shows an infinite execution of the process of type `B` alone, without participation of any other process in the system. Given the fact that SPIN does not allow us to make any assumptions about the relative speeds of execution of processes, the special–case where the process of type `A` pauses indefinitely is allowed, and so the counterexample is valid. Still, there may well be cases where we would be interested in the existence of property violations under more realistic fairness assumptions. One such assumption is the finite progress assumption. It says that any process than can execute a statement will eventually proceed with that execution.

There are two variations of this assumption. The stricter version states that if a process reaches a point where it has an executable statement, and the executability of that statement never changes, it will eventually proceed by executing the statement. A more general version states that if the process reaches a point where it has a statement that becomes executable infinitely often, it will eventually proceed by executing the statement. The first version is commonly referred to as weak fairness and the second as strong fairness. In our example enforcing weak fairness in the search for non–progress cycles would rule out the counterexample that is reported in the default search. We can enforce the weak fairness rule as follows during the verification:

```
$ ./pan -l -f
pan: non-progress cycle (at depth 8)
pan: wrote fair.pml.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim           +
        assertion violations  + (if within scope of claim)
        non-progress cycles   + (fairness enabled)
        invalid end states    - (disabled by never claim)

State-vector 24 byte, depth reached 15, errors: 1
       4 states, stored (12 visited)
       9 states, matched
      21 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573 memory usage (Mbyte)
```

The new cycle that is reported here should now be consistent with the weak fairness finite progress assumption. A quick look at the new counterexample can confirm this.

```
$ spin -t -p fair.pml
spin: couldn't find claim (ignored)
  2:  proc  1 (B) line 12  "fair.pml" (state 1) [x = (3-x)]
  4:  proc  1 (B) line 12  "fair.pml" (state 1) [x = (3-x)]
```

```
  6:  proc  1 (B) line 12  "fair.pml" (state 1) [x = (3-x)]
  8:  proc  0 (A) line  6  "fair.pml" (state 1) [x = (3-x)]
  <<<<<START OF CYCLE>>>>>
 10:  proc  1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
 12:  proc  1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
 14:  proc  1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]

 16:  proc  0 (A) line  6 "fair.pml" (state 1) [x = (3-x)]
spin: trail ends after 16 steps
#processes: 2
                x = 2
 16:  proc  1 (B) line 11 "fair.pml" (state 2)
 16:  proc  0 (A) line  5 "fair.pml" (state 2)
2 processes created
```

As another experiment, we could now add a progress label into one (but not both) of the two `proctypes`, for instance, as follows:

```
active proctype B()
{
        do
        :: x = 3 - x; progress: skip
        od
}
```

The process of type `B` will now alternate between a progress state and a non−progress state, and in principle it could pause forever in its non−progress state at the start of the loop. Think for a moment what you would expect the verifier to find now with the different options we have discussed for finding non−progress cycles, with and without the weak fairness option. Perform the experiment and see if it matches your understanding.[2]

> [2] If all is well, you should be able to confirm with this experiment that only non−fair non−progress cycles exist.

Of course, the enforcement of the additional fairness constraint adds to the computational complexity of the verification problem. It is not very noticeable in this small example, but in general, the amount of work that the verifier has to do can increase by a factor N, where N is the number of active processes. The cost would be even higher for enforcing a strong fairness assumption (increasing to a factor of N^2). In practice, this type of overhead would limit the use of a strong fairness option to only the simplest of test cases. Where needed, arbitrary variations of finite progress and fairness assumptions can always be expressed with the help of temporal logic formula, or directly as ω−regular properties with the help of PROMELA `never` claims.

Note also that the notion of fairness that is used in SPIN applies to process scheduling decisions only: it does not apply to the resolution of non−deterministic choices inside processes. Where needed, other types of fairness can be defined with logic formula. We will return to this in Chapter 6.

We will talk more about the implementation of the weak fairness option in SPIN in Chapter 8 (p. 182).

Accept States:

The last type of label, the accept−state label, is normally reserved for use in `never` claims, which are often mechanically generated from logic formulae. We discuss `never` claims in the next section. Although this is rarely done, accept−state labels can also be used elsewhere in a PROMELA model, and do not require the presence of a `never` claim.

By marking a state with any label that starts with the prefix `accept` we can ask the verifier to find all cycles that do pass through at least one of those labels.

Like progress−state labels, accept−state labels have no meaning in simulation runs: they are only interpreted when SPIN is used in verification mode.

The implicit correctness claim that is expressed by the presence of an accept−state label is that there should not exist any executions that can pass through an accept−state label infinitely often.

To allow for the use of more than one accept−state label in a single `proctype` or `never` claim, the name can again be extended. For instance, the following variants are all valid: `accept`, `acceptance`, `accepting`.

The observations we made above about the use of fairness assumptions apply equally to non−progress cycles and acceptance cycles. Try, for instance, to replace the progress label in the last example with an accept label, and confirm that the verifier can find both fair and non−fair acceptance cycles in the resulting model. The verification steps to be performed are as follows:

```
$ spin −a fair_accept.pml
$ cc −o pan pan.c       # note: no −DNP is used
$ ./pan −a              # all acceptance cycles
...
$ ./pan −a −f           # fair acceptance cycles only
...
```

Never Claims

Up to this point we have talked about the specification of correctness criteria with assertion statements and with meta labels. Powerful types of correctness criteria can already be expressed with these tools, yet so far our only option is to add them into the individual `proctype` declarations. We cannot easily express the claim, "every system state in which property `p` is true eventually leads to a system state in which property `q` is true." The reason we cannot check this property with the mechanisms we have described so far is that we have no way yet of defining a check that would be performed at every single execution step of the system. Note that we cannot make assumptions about the relative speeds of processes, which means that in between any two statement executions any standard PROMELA process can pause for an arbitrary number of steps taken by other system processes. PROMELA `never` claims are meant to give us the needed capability for defining more precise checks.

In a nutshell: a `never` claim is normally used to specify either finite or infinite system behavior that should never occur.

How a Never Claim Works

Consider the following execution of the little pots model from the Chapter 3 (p. 64).

```
$ spin -c pots.pml | more
proc 0 = pots
proc 1 = subscriber
q   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 twenty−four system execution steps here. Eleven statements were executed by the pots server and thirteen were executed by the subscriber process. Because this is a distributed system, not only the specific sequence of statements executed in each process, is important, but also the way in which these statement executions are interleaved in time to form a system execution.

A `never` claim gives us the capability to check system properties just before and just after each statement execution, no matter which process performs them. A `never` claim can define precisely which properties we would like to check at each step. The simplest type of claim would be to check a single property `p` at each and every step, to make sure that it never fails. It is easy to check system invariants in this way.

A claim to check such an invariant property could be written in several ways. Originally, a `never` claim was only meant to match behavior that should never occur. That is, the verification system could flag it as an error if the full behavior specified in the claim could be matched by any feasible system execution. The simplest way to write a `never` claim that checks for the invariance of the system property `p`. Then would be as follows:

```
never {
        do
        :: !p -> break
        :: else
        od
}
```

The claim process is executed at each step of the system. As soon as property `p` is found to be false, the claim process breaks from its loop and terminates, thereby indicating that the error behavior occurred. As long as `p` remains true, though, the claim remains in its initial state, and all is well.

It is easy to abuse the properties of a `never` claim a little by writing more intuitive versions of the checker. For instance, we accomplish the same effect with the following version of the `never` claim:

```
never {
       do
       :: assert(p)
       od
}
```

The loop now contains the relevant assertion that the property `p` is always true. The assertion statement is executed over and over. It is executed for the first time in the initial system state. After that point it is executed again immediately following each system execution step that can be performed. If the property is violated in the initial system state, the claim exits immediately, reporting the error before any system step was taken. Clearly, both the expression statement `!p` and the assertion `assert(p)` are always side effect free, so neither version of the claim can enable system behavior that would not be possible in the claim's absence.

For the property we are considering, a simple system invariant, we could also get away with an alternative specification without using `never` claims. We could, for instance, express the property also by adding an extra PROMELA process that acts as a monitor on system executions, as follows:

```
active proctype monitor()
{
        atomic { !p -> assert(false) }
}
```

The monitor could initiate the execution of the `atomic` sequence at any point in the system execution where `p` is found to be false. This means that if there exists any reachable system state where the invariant property `p` is violated, the monitor could execute in precisely that state and flag a violation. If the behavior is possible this means that the verifier will find it and report it, so in this case the monitor process could solve the problem. This is not necessarily the case once we move to slightly more complex temporal properties.

Consider, for instance, the property:

> Every system state in which `p` is true eventually leads to a system state in which `q` is true, and in the interim `p` remains true.

In SPIN verifications, we are not interested in system executions that satisfy this property, but in the executions that can violate it. This means that we want to know if it is possible for first `p` to become true in an execution and thereafter `q` either to remain false forever, or `p` to become false before `q` becomes true. This behavior should never happen.

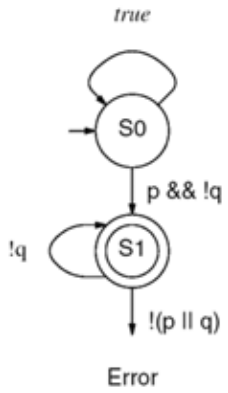
Note that a violation of the property where `q` remains false forever can only be demonstrated with an infinite execution. Since we are dealing with finite state spaces, every infinite execution is necessarily a cyclic execution. We cannot use simple process assertions to capture such (liveness) errors. We can capture precisely the right type of check, though, with the following `never` claim.

```
never {
S0:     do
        :: p && !q -> break
        :: true
        od;
S1:
accept:
        do
        :: !q
        :: !(p || q) -> break
        od
}
```

This type of claim is most easily written as a formula in Linear Temporal Logic (LTL), as illustrated in Figure 4.2. [3] We will explore the link with LTL in more detail in the next section, and we will explore the use of LTL itself in greater detail in Chapter 6.

> [3] We follow the usual conventions by indicating the initial state of every automaton with a short arrow, and by marking the accepting states with a double circle (more details on this will follow in Chapter 6).

Figure 4.2. Control Flow Structure for LTL Property ¬ □ (p ⟶ (p U q)) (LTL, or Linear Temporal Logic, is discussed in Chapter 6)



Claims like this are only interpreted when SPIN is run in verification mode. They are ignored in SPIN simulation runs. The first thing that will happen in a verification is a claim step, to make sure that the `never` claim can perform its first check on the initial system state itself.

Execution of the claim starts at the statement labeled `S0`, which contains a loop with two options for execution. The second option has just a single statement, the expression statement true. Clearly, this option is always executable and has no effect when executed. It leaves the claim in its initial state. The first option is only executable when expression `p` evaluates to true and expression `q` evaluates to false. If this is the case, we have found a match for the antecedent of the property: the truth of `p` but not yet the truth of also `q`. Execution will continue at the statement labeled `accept` As can also be seen in Figure 4.2, the interpretation of a `break` statement, like a `goto` statement, does not really take up an execution step. It merely acts as a special type of statement separator that overrides the default allocation of a successor state. The `break` that appears here defines which statement is to be executed after the second guard in the first cycle is successfully passed.

> (Can be skipped on a first reading.) In principle, the order in which options are placed in a selection or repetition structure is irrelevant: it does not change the semantics of the model. The verifier will consider all possible executions of the model. Although this is strictly seen immaterial, when it has a choice, the verifier will try the various options for execution in the order in which they are listed in the specification. The fact that we have placed the option that can jump to the accept state first in the list of options for state `S0` is therefore no accident: it can help the verifier to find the accepting runs faster.

Clearly, the claim cannot get stuck in its first state, since it always has at least the true guard to execute. Executing the true guard in effect amounts to the claim ignoring the system for one step, deferring any checks until later.

In the `accept` state of the claim there are again two options for execution. The only way for the claim to remain in this state is for expression `q` to remain false forever. If this is possible, it would constitute a violation of our property. For this reason we have marked the state with an `accept` label. The violation can now be caught by the verifier as an acceptance cycle. The only other possible way in which the property can be violated is when `p` becomes false before `q` becomes true. This type of violation is caught by forcing the termination of the `never` claim. Termination of the claim is interpreted as a full match of the behavior specified; behavior that is never supposed to happen.

What happens when both `p` and `q` are found to be true while the claim executes in its `accept` state? Neither of the two options for continued execution are executable: the claim gets stuck. Curiously, this is not an error

but a desirable condition. If the claim cannot match the system execution in its next step, then this means that the undesirable behavior that is captured in the `never` claim cannot fully be matched in a system execution. All is well; the claim stops tracking the system along this execution path and no error is reported. For the verifier to be of use here, we should be able to guarantee that the verifier effectively matches the behavior expressed by the claim against not just one but all possible system executions, so that if any such execution can match, it will be found. Clearly, in simulation mode SPIN cannot give any such guarantee, but in verification mode it can, and it does.

There is one other important observation to be made about the sample `never` claim above. Note that when `p` is true and `q` is false in state `S0`, the claim has two possible options for execution: both guards in the loop are executable. This form of non−determinism is critically important here. If the guard that is now expressed as true would be replaced with else, the claim would require only that the first time in a system execution that expression `p` becomes true should lead to a state where `q` is also true. It would not place any requirements on any future events that would make `p` true after the requirement is met the first time. The non−determinism makes it possible to make a far stronger statement: every time that `p` becomes true should lead to a state where also `q` is true.

In the sample `never` claims we have given here, the statements we used were always side effect free. In fact, apart from general control−flow constructs, we have used only two types of statements: condition statements and assertions. This is of course no accident. A `never` claim is meant to monitor or track system behavior and not to contribute to it. The declaration or the manipulation of variables or message channels, or generally the execution of any statement that could have a side effect on the state of data objects used in the system, is therefore never allowed inside `never` claims.

A nice property of `never` claims is that they can be used not just to define correctness properties, but also to restrict the search of the verifier to a user−defined subset, or slice, of the system. The verifier in effect always eliminates all behavior that cannot be matched by the claim. This is safe to do in verification because the claim is used to specify only invalid behavior, which means that behavior that is eliminated is by definition (of the claim) behavior that is irrelevant to the verification attempt. As a very simple example of this, note that a very simple claim such as

```
never { do :: p od }
```

can be used to restrict a verification run to all states where the condition `p` evaluates to true. If `p` is in fact a system invariant, this results in no reduction of work, but otherwise the state space that is searched is reduced by this claim. Of course, if condition `p` turns out to be false in the initial system state, the state space will be completely empty, which is unlikely to be useful.

The Link With LTL

`Never` claims provide a powerful mechanism for expressing properties of distributed systems. Admittedly, though, it can be hard to come up with correct formalizations of system properties by directly encoding them into `never` claims. Fortunately, there are easier ways to do this. One such method is to use SPIN's built–in translator from formulae in linear temporal logic (LTL) to `never` claims. The last claim, for instance, corresponds to the LTL formula

```
![](p -> (p U q))
```

We can generate the `never` claim from this formula with the command:

```
$ spin -f '![](p -> (p U q))'
never {    /* ![](p-> (p U q)) */
T0_init:
        if
         :: (! ((q)) && (p)) -> goto accept_S4
         :: (1) -> goto T0_init
         fi;
 accept_S4:
         if
         :: (! ((q))) -> goto accept_S4
         :: (! ((p)) && ! ((q))) -> goto accept_all
         fi;
 accept_all:
        skip
}
```

Another method is to use a little graphical tool, called the timeline editor, to convert timeline descriptions into `never` claims. In many cases, temporal logic formulae are simpler to understand and use than `never` claims, but they are also strictly less expressive. Timeline properties, in turn, are simpler to understand and use than temporal logic formulae, but again less expressive than these. Fortunately, almost all properties of interest can be expressed as either timeline properties or temporal logic formulae. We will discuss temporal logic in more detail in Chapter 6, and the various methods to generate `never` claims mechanically from either formulae or timelines in, respectively, Chapter 6 (p. 127) and Chapter 13 (p. 283).

Trace Assertions

Like never claims, a trace assertion does not specify new behavior, but instead expresses a correctness requirement on existing behavior in the remainder of the system. A trace assertion expresses properties of message channels, and in particular it formalizes statements about valid or invalid sequences of operations that processes can perform on message channels.

All channel names referenced in a trace assertion must be globally declared, and all message fields must be globally known constants or mtype symbolic constants. A simple example of a trace assertion is as follows:

```
trace {
       do
       :: q1!a; q2?b
       od
}
```

In this example, the assertion specifies the correctness requirement that send operations on channel q1 alternate with receive operations on channel q2, and furthermore that all send operations on q1 are exclusively messages of type a, and all receive operations on channel q2 are exclusively messages of type b.

Trace assertions apply only to send and receive operations on message channels. They can be used to specify a specific relative order in which these types of operations must always be performed. Only the channel names that are specified in the assertion are considered to be within the scope of the check. All send and receive operations on other channels are ignored. The trace assertion trivially defines an automaton that can step through a finite number of control states while it monitors a system execution. The automaton only changes state when a send or a receive operation is executed that is within its scope. If an operation is executed that is within scope, but that cannot be matched by a transition of the trace automaton, the verifier will immediately report an error.

If at least one send (receive) operation on a channel appears in a trace assertion, then all send (receive) operations on that channel are subject to the check.

As with never claims, there are some restrictions on the types of statements that can appear in a trace assertion. Apart from control−flow constructs, a trace assertion may contain only simple send and receive operations. It cannot contain variations such as random receive, sorted send, or channel poll operations. No data objects can be declared or referred to inside a trace assertion.

If a message field must be matched in a send or receive operation that appears inside a trace assertion, it must be specified with a standard mtype name or with a constant value. Don't care values for specific message fields can be specified with the predefined write−only variable _ (i.e., the underscore symbol).

Sends and receives that appear in an event trace are called monitored events. These events do not generate new behavior, but they are required to match send or receive events on the same channels in the model, with matching message parameters. A send or receive event occurs when a send or a receive statement is executed in the system, that is, an event that occurs during a state transition.

On rendezvous channels, a rendezvous handshake consists of an offer (the send half of the handshake) and an accept (the receive half of the handshake). Even traces can only capture the occurrence of the receive part of a handshake, not of the send part. The reason is that the send (the offer) can be made many times before it results in a successful accept.

A `trace` assertion can contain end−state, progress−state, and accept−state labels with the usual interpretation. There are, however, a few important differences between `trace` assertions and `never` claims:

- Unlike `never` claims, `trace` assertions must always be deterministic.
- A `trace` assertion can match event occurrences that occur in the transitions between system states, whereas a `never` claim matches propositional values on system states only.
- A `trace` assertion monitors only a subset of the events in a system: only those of the types that are mentioned in the `trace` (i.e., the monitored events). A `never` claim, on the other hand, looks at all global system states that are reached, and must be able to match the state assignments in the system for every state reached.

A `trace` assertion, just like a `never` claim, has a current state, but it only executes transitions if a monitored event occurs in the system. Unlike `never` claims, `trace` assertions do not execute synchronously with the system; they only execute when events of interest occur.

Note that receive events on rendezvous channels can be monitored with `trace` assertions, but not with `never` claims.

Notrace

Sometimes it is desirable to specify precisely the opposite of a `trace` assertion: a sequence of events that should not occur in an execution. For this purpose the keyword `notrace` is also supported, though it is only rarely used. A `notrace` assertion is violated if the event sequence that is specified, subject to the same rules as `trace` assertions, is matched completely. The assertion is considered to be matched completely when either an end−state label is reached inside the `notrace` sequence, or the closing curly brace of that sequence is reached.

Predefined Variables and Functions

Some predefined variables and functions can be especially useful in `trace` assertions and `never` claims.

There are only four predefined variables in PROMELA. They are:

```
_
np_
_pid
_last
```

We have discussed the `_pid` variable before (p. 16, 36), as well as the global write−only variable _ (p. 92). These two variables can be used freely in any `proctype` declaration.

But we have not encountered the two variables `np_` and `_last` before. These two variables are meant to be used only inside `trace` assertions or `never` claims.

The predefined variable `np_` holds the boolean value false in all system states where at least one running process is currently at a control−flow state that was marked with a `progress` label. That is, the value of variable `np_` tells whether the system is currently in a progress or a non−progress state. We can use this variable easily to build a `never` claim that can detect the existence of non−progress cycles, for instance, as follows:

```
never { /* non-progress cycle detector */
        do
        :: true
        :: np_ -> break
        od;
accept:
        do
        :: np_
        od
}
```

After a finite prefix of arbitrary length, optionally passing through any finite number of non−progress states, the claim automaton moves from its initial state to the final accepting state, where it can only remain if there exists at least one infinitely long execution sequence that never traverses any more progress states.

The true purpose of the `np_` variable is not in the definition of this claim, since this precise claim is used automatically when SPIN's default search for non−progress cycles is invoked. The availability of the variable makes it possible to include the non−progress property into other more complex types of temporal properties. A standard application, for instance, would be to search for non−progress cycles while at the same time enforcing non−standard fairness constraints.

The predefined global variable `_last` holds the instantiation number of the process that performed the last step in a system execution sequence. Its value is not part of the system state unless it is explicitly used in a specification. Its initial value is zero.

The use of the following three predefined functions is restricted to `never` claims:

```
pc_value(pid)
enabled(pid)
procname[pid]@label
```

The first of these functions,

```
pc_value(pid)
```

returns the current control state of the process with instantiation number `pid` or zero if no such process exists. The number returned is always a non−negative integer and it corresponds to the internal state number that SPIN tracks as the equivalent of a program−counter in a running process.

In the following example, one process will print out its internal state number for three consecutive steps, and a second blocks until the first process reaches at least a state that is numbered higher than two.

```
active proctype A()
{
        printf("%d\n", pc_value(_pid));
        printf("%d\n", pc_value(_pid));
        printf("%d\n", pc_value(_pid));
}

active proctype B()
{
        (pc_value(0) > 2);
        printf("ok\n")
}
```

If we simulate the execution of this model, the following is one of two possible runs:

```
$ spin pcval.pml
spin: Warning, using pc_value() outside never claim
      1
      2
         ok
      3
2 processes created
```

SPIN's warning reminds us that the use of this predefined function is intended to be restricted to `never` claims.

The function

```
enabled(pid)
```

tells whether the process with instantiation number `pid` has at least one statement that is executable in its current state. The following example illustrates the use of `_last` and `enabled()`:

```
/* It is not possible for the process with pid 1
* to remain enabled without ever executing.
*/
never {
accept:
       do
       :: _last != 1 && enabled(1)
       od
}
```

The last predefined function

```
procname[pid]@label
```

returns a nonzero value only if the next statement that can be executed in the process with instantiation number `pid` is the statement that was marked with label `label` in `proctype procname` It is an error if the process referred to with the `pid` number is not an instantiation of the specified `proctype`.

The following example shows how one might employ this type of remote referencing inside a `never` claim:

```
/*
* Processes 1 and 2 cannot enter their
* critical sections at the same time.
*/
never {
        do
        :: user[1]@critical && user[2]@critical ->
                break   /* implicitly accepting */
        :: else /* repeat */
        od
}
```

If there is only one instantiation of a given `proctype` in the system, the process identify is really superfluous, and (in SPIN version 4.0 and higher) the reference from the last example can in that case be given as

```
user@critical.
```

If, nonetheless, there turns out to be more than one instantiation of the `proctype`, this type of reference selects an arbitrary one of them. The SPIN simulator issues a warning if it encounters a case like this; the verifier does not.

Remote Referencing

In SPIN version 4.0 and higher (cf. Chapters 10 and 17), another type of reference is also supported. The additional type of reference bypasses the standard scope rules of PROMELA by making it possible for any process, and also the `never` claim, to refer to the current value of local variables from other processes. This capability should be used with caution, since it conflicts with the assumptions about scope rules that are made in SPIN's partial order reduction strategy (see Chapter 9).

The syntax is the same as for remote label references, with the replacement of the "@" symbol for a single colon ":". For instance, if we wanted to refer to the variable `count` in the process of type `Dijkstra` in the example on page 77, we could do so with the syntax

```
Dijkstra[0]:count.
```

If there is only one instantiation of `proctype Dijkstra`, we can again use the shorter version

```
Dijkstra:count,
```

following the same rules as before.

Path Quantification

We will discuss the verification algorithms used in SPIN in more detail in later chapters, but we can already note a few important features of the approach taken. All correctness properties that can be verified with the SPIN system can be interpreted as formal claims that certain types of behavior are, or are not, possible.

- An assertion statement formalizes the claim that it is impossible for the given expression to evaluate to false.
- An end label states that it is impossible for the system to terminate without all active processes having either terminated or stopped at one of the specially labeled end states.
- A progress label states that it is impossible for the system to execute forever without also passing through at least one of the specially labeled progress states infinitely often.
- An accept label states that it is impossible for the system to execute forever while also passing through at least one of the specially labeled accept states infinitely often.
- A `never` claim states that it is impossible for the system to exhibit, either infinite or finite, behavior that completely matches the behavior that is formalized by the claim.
- A trace assertion, finally, states that it is impossible for the system to deviate from the behavior that is formalized.

In all cases, the verification performed by SPIN is designed to prove the user wrong: SPIN will try its best to find a counterexample to at least one of the formal claims that is stated. So, in a way SPIN never tries to prove the correctness of a specification. It tries to do the opposite.

Hunting for counterexamples, rather than direct proof, has advantages. Specifically, it can allow the verifier to employ a more efficient search procedure. If, for instance, the error behavior is formalized in a `never` claim, SPIN can restrict its search for counterexamples to the behaviors that match the claim. `never` claims, then, in a way act as a restriction on the search space. If the error behavior is indeed impossible, as the user claims, the verifier may have very little work to do. If the error behavior is almost impossible, it may have to do a little more work, but still not necessarily as much as when it has to search the entire space of all possible behaviors. This only happens in the worst case.

Formalities

Let E be the complete set of all possible ω–runs of the system. Given a correctness property ø formalized as an LTL property, we say that the system satisfies ø if and only if all runs in E do. We can express this as

Equation 4.1

$$(\mathrm{E} \models \phi) \leftrightarrow \forall \sigma, (\sigma \in \mathrm{E} \rightarrow \sigma \models \phi).$$

SPIN, however, does not attempt to prove this directly. We can only use SPIN in an attempt to disprove the claim by trying to find a counterexample that shows that ¬ ø is satisfied for at least one run. That is, instead of proving [4.1] directly, SPIN tries to show the opposite

Equation 4.2

$$\neg (\mathrm{E} \models \phi) \leftrightarrow \exists \sigma, (\sigma \in \mathrm{E} \wedge \neg (\sigma \models \phi))$$

where of course ¬ (σ|= ø) means that (σ|= ¬ ø).

If the attempt to show that the right–hand side of [4.2] holds succeeds, we have shown that ¬ (E|= ø), and therefore that the left–hand side of [4.1] does not hold: the system does not satisfy the property.

If the attempt to find a counterexample fails, we have shown that the left–hand side of [4.2] does not hold, and therefore that the left–hand side of [4.1] must hold: the system satisfies the property.

This all looks fairly straightforward, but note carefully that the step from [4.1] to [4.2] cannot be achieved by the mere negation of property ø. If, for instance, instead of property ø we try to prove with the same procedure that ¬ ø is satisfied, then [4.1] becomes

Equation 4.3

$$(\mathrm{E} \models \neg \phi) \leftrightarrow \forall \sigma, (\sigma \in \mathrm{E} \rightarrow (\sigma \models \neg \phi)).$$

which is the same as

Equation 4.4

$$(\mathrm{E} \models \neg \phi) \leftrightarrow \forall \sigma, (\sigma \in \mathrm{E} \rightarrow \neg (\sigma \models \phi)).$$

Note that the right−hand sides of [4.2] and [4.4] differ. The logical negation of the right−hand side of [4.1] is the right−hand side of [4.2], and not the right−hand side of [4.4]. In other words, (E|= ¬ ø)is not the same as ¬ (E|= ø). The first equation does logically imply the second, but the reverse is not necessarily true:

Equation 4.5

$$(E \models \neg \phi) \rightarrow \neg (E \models \phi).$$

It is not too hard to come up with examples where we have simultaneously:

Equation 4.6

$$\neg (E \models \neg \phi) \wedge \neg (E \models \phi).$$

For instance, if ø is the property that some variable x is never equal to zero, then ¬ ø states that x is zero at least once. It is quite possible to construct a model that has at least one run where variable x eventually becomes zero (providing a counterexample to ø), and also at least one run where x never becomes zero (providing a counterexample to ¬ ø).

The following simple system has precisely that property.

```
byte x = 1;

init {

       do
       :: x = 0
       :: x = 2
       od
}

#define p      (x != 0)

#ifdef PHI

never {    /* []p */
accept:
        do
        :: (p)
        od
}

#else

never {     /* ![]p */
        do
        :: !p -> break
        :: true
        od
}
```

```
#endif
```

Both the property `[]p` and its negation `![]p` will produce counterexamples for this model, as is quickly demonstrated by SPIN:

```
$ spin -DPHI -a prop.pml
$ cc -o pan pan.c
$ ./pan -a
pan: acceptance cycle (at depth 2)
pan: wrote prop.pml.trail
...
```

We can replay this first error trail with the guided simulation option, as before.

```
$ spin -t -p -v -DPHI prop.pml
1: proc - (:never:) line 18 "prop.pml" (state 1) [(x!=0)]
Never claim moves to line 18 [(x!=0)]
2: proc 0  (:init:) line  7 "prop.pml" (state 2) [x = 2]
 <<<<<START OF CYCLE>>>>>

3: proc - (:never:) line 18 "prop.pml" (state 1) [(x!=0)]
4: proc 0 (:init:)  line 7  "prop.pml" (state 2) [x = 2]
5: proc - (:never:) line 18 "prop.pml" (state 1) [(x!=0)]
spin: trail ends after 5 steps
...
```

And, for the negated version of the claim:

```
$ spin -DNOTPHI -a prop.pml
$ cc -o pan pan.c
$ ./pan
pan: claim violated! (at depth 3)
pan: wrote prop.pml.trail
...
$ spin -t -p -v -DNOTPHI prop.pml
1: proc - (:never:) line 27 "prop.pml" (state 3) [(1)]
Never claim moves to line 27  [(1)]
2: proc 0 (:init:) line   6 "prop.pml" (state 1) [x = 0]
3: proc - (:never:) line 26 "prop.pml" (state 1) [!(x!=0)]
Never claim moves to line 26 [!(x!=0)]
spin: trail ends after 3 steps
...
```

Specifically, therefore, if a SPIN verification run shows that a given system `S` fails to satisfy property ø, we cannot conclude that the same system `S` will satisfy the inverse property ¬ ø.

Finding Out More

This concludes our overview of the various ways for expressing correctness requirements that are supported in PROMELA. More information about the derivation of `never` claims from Linear Temporal Logic formulae can be found in Chapter 6. More about the specification of requirements for a SPIN verification in a more intuitive, graphical format can be found in Chapter 13 where we discuss the timeline editing tool.

We have seen in this chapter that there are many different ways in which correctness requirements can be expressed. Be reassured, though, that the most easily understood mechanism for this purpose is also the most commonly used: the simple use of assertions.

The terms safety and liveness were first systematically discussed by Leslie Lamport, cf. Lamport [1983], and see also Alpern and Schneider [1985,1987]. An excellent introduction to the formalization of correctness properties for distributed systems can be found in Manna and Pnueli [1995]. Many other textbooks contain excellent introductions to this material; see, for instance, Berard et al. [2001], Clarke et al. [2000], or Huth and Ryan [2000].

A more detailed treatment of the differences between various ways of specifying, for instance, system invariants in PROMELA can be found in Ruys [2001].