Chapter 7. PROMELA Semantics

"The whole is often more than the sum of its parts."

—(Aristotle, Metaphysica, 10f–1045a, ca. 330 B.C.)

As we have seen in the earlier chapters, a SPIN model can be used to specify the behavior of collections of asynchronously executing processes in a distributed system. By simulating the execution of a SPIN model we can in principle generate a large directed graph of all reachable system states. Each node in that graph represents a possible state of the model, and each edge represents a single possible execution step by one of the processes. PROMELA is defined in such a way that we know up–front that this graph will always be finite. There can, for instance, be no more than a finite number of processes and message channels, there is a preset bound on the number of messages that can be stored in each channel, and each variable has a preset and finite range of possible values that it can attain during an execution. So, in principle, the complete graph can always be built and analyzed in a finite amount of time.

Basic correctness claims in PROMELA can be interpreted as statements about the presence or absence of specific types of nodes or edges in the global reachability graph. More sophisticated temporal logic properties can be interpreted to express claims about the presence or absence of certain types of subgraphs, or paths, in the reachability graph. The global reachability graph can itself also be interpreted as a formal object: a finite automaton, as defined in Appendix A (p. 553).

The structure of the reachability graph is determined by the semantics of PROMELA. In effect, the PROMELA semantics rules define how the global reachability graph for any given PROMELA model is to be generated. In this chapter we give operational semantics of PROMELA in precisely those terms. The operational semantics definitions should allow us to derive in detail what the structure of the global reachability graph is for any given SPIN model.

Transition Relation

Every PROMELA `proctype` defines a finite state automaton, (S, s_0, L, T, F), as defined in Chapter 6. The set of states of this automaton S corresponds to the possible points of control within the `proctype`. Transition relation T defines the flow of control. The transition label set L links each transition in T with a specific basic statement that defines the executability and the effect of that transition. The set of final states F, finally, is defined with the help of PROMELA end−state, accept−state, and progress−state labels. A precise description of how set F is defined for safety and for liveness properties can be found in Appendix A.

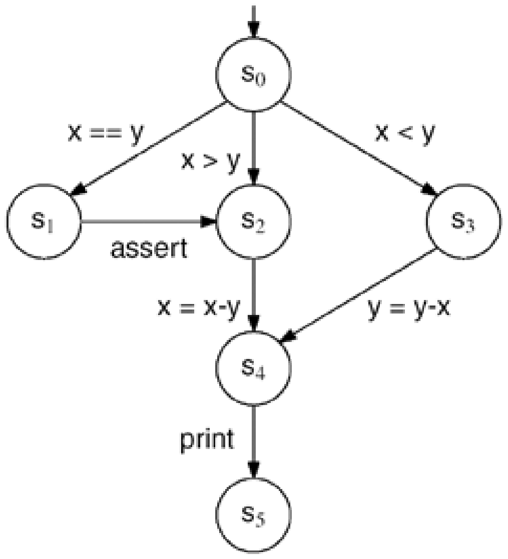
Conveniently, the set of basic statements in PROMELA is very small. It contains just six elements: assignments, assertions, print statements, send or receive statements, and PROMELA's expression statement (cf. p. 51). All other language elements of PROMELA serve only to specify the possible flow of control in a process execution, that is, they help to specify the details of transition relation T. As one small example, note that `goto` is not a basic statement in PROMELA. The `goto` statement, much like the semicolon, merely defines control−flow.

As a small example of how PROMELA definitions translate into automata structures, consider the PROMELA model shown in Figure 7.1, which corresponds to the automaton structure shown in Figure 7.2. The presence of the `goto` achieves that the execution of the assertion statement leads to control state s_2, instead of s_4. Thereby it changes the target state of a transition, but it does not in itself add any transitions. In other words, the `goto` effects a change in transition relation T, but it does not, and cannot, appear in label set L.

Figure 7.1 Sample PROMELA Model

```
active proctype not_euclid(int x, y)
{
        if
        :: (x >  y) -> L: x = x - y
        :: (x <  y) -> y = y - x
        :: (x == y) -> assert(x!=y); goto L
        fi;
        printf(";%d\n", x)
}
```

Figure 7.2. Transition Relation for the Model in Figure 7.1



Two points are especially worth noting here. First, language elements such as `if`, `goto`, the statement separators semicolon and arrow, and similarly also `do`, `break`, `unless`, `atomic`, and `d_step`, cannot appear as labels on transitions: only the six basic types of statements in PROMELA can appear in set L.

Second, note that expression statements do appear as first−class transition labels in the automaton, and they are from that point of view indistinguishable from the other types of basic statements. In PROMELA every basic statement has a precondition that defines when it is executable, and an effect that defines what happens when it is executed. We explore many of these issues in more detail in the remainder of this chapter.

Operational Model

Our operation model centers on the specification of a semantics engine which determines how a given PROMELA model defines system executions, including the rules that apply to the interleaved execution of process actions. The semantics engine operates on abstract objects that correspond to asynchronous processes, variables, and message channels. We give formal definitions of these abstract objects first. We also define the concept of a global system state and a state transition, corresponding, respectively, to nodes and edges in a global reachability graph. We skip the definition of more basic terms, such as sets, identifiers, integers, and booleans.

Definition 7.1 (Variable)

A variable is a tuple `(name,scope,domain,inival,curval)` where

`name` is an identifier that is unique within the given `scope`,

`scope` is either global or local to a specific process.

`domain` is a finite set of integers.

`inival`, the initial value, is an integer from the given `domain`, and

`curval`, the current value, is also an integer from the given `domain`.

We will refer to the elements of a tuple with a dot notation. For instance, if `v` is a variable, then `v.scope` is its scope.

The `scope` of a variable is either global, including all processes, or it is restricted to one specific process (to be defined below). The type of a variable trivially determines its `domain`. For instance, a variable of type `bit` has domain {0,1}.

Definition 7.2 (Message)

A message is an ordered set of variables (Def. 7.1).

Definition 7.3 (Message Channel)

A channel is a tuple `(ch_id,nslots,contents)` where

`ch_id` is a positive integer that uniquely identifies the channel,

`nslots` is an integer, and

`contents` is an ordered set of messages (Def. 7.2) with maximum cardinality `nslots`.

Note that the definition of a channel does not contain a `scope`, like the definition of a variable. A PROMELA channel always has global scope. It can be created either globally or locally, by an active process, but its method of creation does not affect its scope. Every channel is in principle accessible to every active process, by knowledge of channel instantiation number `ch_id`. The variable that holds this channel instantiation number can have a local scope, but not the channel itself.

Definition 7.4 (Process)

A process is a tuple

`(pid,lvars,lstates,initial,curstate,trans)` where

`pid` is a positive integer that uniquely identifies the process,

`lvars` is a finite set of local variables (Def. 7.1), each with a `scope`

that is restricted to the process with instantiation number `pid`.

`lstates` is a finite set of integers (see below),

`initial` and `curstate` are elements of set `lstates`, and

`trans` is a finite set of transitions (Def. 7.5) on `lstates`.

The `pid` value is the process instantiation number which uniquely identifies a running process within the system. In the initial state of a newly created process `curstate=initial`, and all elements of `lvars` have `curval=inival`.

We refer to the elements of set `lstates` as the local states of a process. The integer values serve to uniquely identify each state within the set, but hold no more information.

Definition 7.5 (Transition)

A transition in process `P` is defined by a tuple

`(tr_id,source,target,cond,effect,prty,rv)` where

`tr_id` is a non–negative integer,

`source` and `target` are elements from set `P.lstates` (i.e., integers),

`cond` is a boolean condition on the global system state (Def. 7.6),

`effect` is a function that modifies the global system state (Def. 7.6),

`prty` and `rv` are integers.

As we shall see later, the integers `prty` and `rv` are used inside `cond` and `effect` definitions to enforce the semantics of `unless` constructs and rendezvous operations, respectively.

Definition 7.6 (System State)

A global system state is a tuple of the form

`(gvars,procs,chans,exclusive,handshake,timeout,else, stutter)` where

`gvars` is a finite set of variables (Def. 7.1) with global scope,

`procs` is a finite set of processes (Def. 7.4),

`chans` is a finite set of message channels (Def. 7.3),

`exclusive`, and `handshake` are integers,

`timeout`, `else`, and `stutter` are booleans.

In the initial system state all processes (Def. 7.4) are in their initial state, all global variables (Def. 7.1) have `curval=inival`, all message channels (Def. 7.3) have `contents={}` (i.e., empty), `exclusive` and `handshake` are zero, and the booleans `timeout` and `else` and `stutter` all have the initial value false.

Definitions 7.1 to 7.6 capture the minimal information that is needed to define the semantics of the PROMELA language in terms of an operational model, with processes defined as transition systems (i.e., automata). A small number of predefined integer "system variables" that are manipulated by the semantics engine appear in these definitions:

- `prty`, which is used to enforce the semantics of the `unless` construct,
- `rv` and `handshake`, to enforce the semantics of rendezvous operations,
- `exclusive`, to enforce the semantics of `atomic` and `d_step` sequences,
- `stutter`, to enforce the stutter extension rule (cf. p. 130), and `timeout` and `else`, to enforce the semantics of the matching PROMELA statements.

In the next section we define the semantics engine. With the help of this definition it should be possible to resolve any question about the interpretation of PROMELA constructs independently from the implementation of SPIN.

Operational Model, Semantics Engine

The semantics engine executes a SPIN model in a step by step manner. In each step, one executable basic statement is selected. To determine if a statement is executable or not, one of the conditions that must be evaluated is the corresponding executability clause, as described in the PROMELA manual pages that start on p. 363. If more than one statement is executable, any one of them can be selected. The semantics definitions deliberately do not specify (or restrict) how the selection of a statement from a set of simultaneously executable statements should be done. The selection could, for instance, be random. By leaving this decision open, we in effect specify that the correctness of every SPIN model should be independent of the selection criterion that is used.

For the selected statement, the effect clause from the statement is applied, as described in the PROMELA manual pages for that statement, and the control state of the process that executes the statement is updated. The semantics engine continues executing statements until no executable statements remain, which happens if either the number of processes drops to zero, or when the remaining processes reach a system deadlock state.

The semantics engine executes the system, at least conceptually, in a stepwise manner: selecting and executing one basic statement at a time. At the highest level of abstraction, the behavior of this engine can be defined as follows:

Let `E` be a set of pairs (`p`,`t`), with `p` a process, and `t` a transition. Let `executable(s)` be a function, yet to be defined, that returns a set of such pairs, one for each executable transition in system state `s`. The semantics engine then performs as shown in Figure 7.3.

Figure 7.3 PROMELA Semantics Engine

```
while ((E = executable(s)) != {})
{
    for some (p,t) from E
    {     s' = apply(t.effect, s)

         if  (handshake == 0)
         {     s = s'
               p.curstate = t.target
         }  else
         {      /* try to complete rv handshake */
               E' = executable(s')
               /* if E' is {}, s is unchanged */

               for some (p',t') from E'
               {    s = apply(t'.effect, s')
                    p.curstate = t.target
                    p'.curstate = t'.target
               }
               handshake = 0
    }   }
}
while (stutter) { s = s } /* 'stutter' extension */
```

As long as there are executable transitions (corresponding to the basic statements of PROMELA), the semantics engine repeatedly selects one of them at random and executes it.

The function `apply()` applies the effect of the selected transition to the system state, possibly modifying system and local variables, the contents of channels, or even the values of the reserved variables `exclusive` and `handshake`, as defined in the effect clauses from `atomic` or rendezvous `send` operations, respectively. If no rendezvous offer was made (line 6), the global state change takes effect by an update of the system state (line 7), and the current state of the process that executed the transition is updated (line 8).

If a rendezvous offer was made in the last transition, it cannot result in a global state change unless the offer can also be accepted. On line 11 the transitions that have now become executable are selected. The definition of the function `executable()` below guarantees that this set can only contain accepting transitions for the given offer. If there are none, the global state change is declined, and execution proceeds with the selection of a new executable candidate transition from the original set `E`. If the offer can be matched, the global state change takes effect (line 15). In both processes, the current control state is now updated from source to target state (lines 16 and 17).

To verify liveness properties with SPIN, we must be able to treat finite executions as special cases of infinite executions. The standard way of doing so is to define a stutter extension of finite executions, where the final state is repeated ad infinitum. The engine in Figure 7.3 uses the system variable `stutter` to determine if the stuttering rule is in effect (line 22). Only the verification system can change this variable.

Changes in the value of this particular system variable are not covered by the semantics of PROMELA proper, but they are determined by the verification algorithms that are used for checking, for instance, ω–regular properties of PROMELA models. Note that the `stutter` variable is only used to render a formal judgment on the semantics of a given model; it is not part of the semantics definition itself. More specific notes on the verification of PROMELA models follow at the end of this chapter.

A key part of the semantics is in the definition of what precisely constitutes an executable transition. One part will be clear: for transition `t` to be executable in the current system state, its executability clause `t.cond` must be satisfied. But there is more, as illustrated by the specification of function `executable()` in Figure 7.4. To avoid confusion, the reserved state variables `timeout`, `else`, and `exclusive` are set in bold in the figure. These variables are the only ones that can be modified within this function as part of the selection process.

Figure 7.4 Specification of Procedure `executable()`

```
1 Set
2 executable(State s)
3 {     new Set E
4       new Set e
5
6       E = {}
7       timeout = false
8 AllProcs:
9       for each active process p
10      {   if (exclusive == 0
11          or  exclusive == p.pid)
12          {   for u from high to low   /* priority */
13              {   e = {};  else = false
14 OneProc:         for each transition t in p.trans
15                  {   if (t.source == p.curstate
16                      and t.prty == u
17                      and (handshake == 0
18                      or   handshake == t.rv)
19                      and  eval(t.cond) == true)
20                      {     add (p,t) to set e
21                  }   }
22                if (e != {})
23                {   add all elements of e to E
24                    break /* on to next process */
25                } else if (else == false)
26                {   else = true
27                    goto OneProc
28                } /* or else lower the priority */
29        }   }  }
30
31        if (E == {} and exclusive != 0)
32        {   exclusive = 0
33            goto AllProcs
34        }
35        if (E == {} and timeout == false)
36        {   timeout = true
37            goto AllProcs
38        }
39
40        return E     /* executable transitions */
41 }
```

For a transition to be added to the set of executable transitions it has to pass a number of tests.

- The test on line 10–11 checks the value of the reserved system variable `exclusive`. By default it is zero, and the semantics engine itself never changes the value to non−zero. Any transition that is part of an `atomic` sequence sets `exclusive` to the value of `p.pid`, to make sure that the sequence is not interrupted by other processes, unless the sequence itself blocks. In the latter case the semantics engine restores the defaults (line 32).
- The test on line 16 checks the priority level, set on line 12. Within each process, the semantics engine selects the highest priority transitions that are executable. Note that priorities can affect the selection of transitions within a process, not between processes. Priorities are defined in PROMELA with the `unless` construct.
- The test on line 15 matches the source state of the transition in the labeled transition system with the current state of the process, selected on line 9.
- The test on lines 17−18 makes sure that either no rendezvous offer is outstanding, or, if one is, that the transition being considered can accept the offer on the corresponding rendezvous port.
- The test on line 19, finally, checks whether the executability condition for the transition itself is satisfied.

If no transitions are found to be executable with the default value false for system variable `else`, the transitions of the current process are checked again, this time with `else` equal to true (lines 26–27). If no transitions are executable in any process, the value of system variable `timeout` is changed to true and the entire selection process is repeated (lines 32–35). The new value of `timeout` sticks for just one step (line 7), but it can cause any number of transitions in any number of processes to become executable in the current global system state. The syntax of PROMELA prohibits the use of both `else` and `timeout` within a single condition statement.

Note again that the semantics engine does not establish the validity or invalidity of correctness requirements, as the judgement of what is correct system behavior is formally outside the definition of PROMELA semantics proper.

Interpreting PROMELA Models

The basic objects that are manipulated by the semantics engine are, of course, intended to correspond to the basic objects of a PROMELA model. Much of the language merely provides a convenient mechanism for dealing with the underlying objects. In the PROMELA reference manual in Chapter 16, some language constructs are defined as meta−terms, syntactic sugar that is translated into PROMELA proper by SPIN's preprocessor. Other language elements deal with the mechanism for declaring and instantiating variables, processes, and message channels. The control−flow constructs, finally, provide a convenient high−level means for defining transition relations on processes. An `if` statement, for instance, defines how multiple transitions can exit from the same local process state. The semantics engine does not have to know anything about control−flow constructs such as `if`, `do`, `break`, and `goto`; as shown, it merely deals with local states and transitions.

Some PROMELA constructs, such as assignments and message passing operations, cannot be translated away. The semantics model is defined in such a way that these primitive constructs correspond directly to the transitions of the underlying state machines. We call these PROMELA constructs basic statements, and there are surprisingly few of them in the language. The language reference manual defines the transition elements for each basic statement that is part of the language.

Three Examples

Consider the following PROMELA model.

```
chan x = [0] of { bit };
chan y = [0] of { bit };
active proctype A() { x?0 unless y!0 }
active proctype B() { y?0 unless x!0 }
```

Only one of two possible rendezvous handshakes can take place. Do the semantics rules tell us which one? If so, can the same rules also resolve the following, very similar, situation?

```
chan x = [0] of { bit };
chan y = [0] of { bit };
active proctype A() { x!0 unless y!0 }
active proctype B() { y?0 unless x?0 }
```

And, finally, what should we expect to happen in the following case?

```
chan x = [0] of { bit };
chan y = [0] of { bit };
active proctype A() { x!0 unless y?0 }
active proctype B() { y!0 unless x?0 }
```

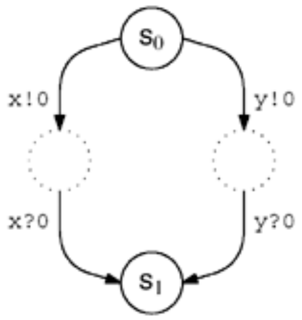
Each of these cases can be hard to resolve without guidance from a semantics definition. The semantics rules for handling rendezvous communication and for handling `unless` statements seem to conflict here. This is what we know.

- The definition of `unless` states that the statement that precedes the `unless` keyword has a lower execution priority than the statement that follows it. These priorities must be used to resolve executability conflicts between the two transitions within each process.
- Rendezvous handshakes occur in two parts: the send statement constitutes a rendezvous offer, which can succeed if it is matched by a receive operation on the same channel in the immediately following execution step by the other process. To make the offer, the send statement must be executable by the rules of the semantics engine, and to accept the offer the matching receive operation must be executable.
- The effect clause of the rendezvous send operation states that the value of reserved variable `handshake` is set to the value of the channel instantiation number `ch_id` for the channel used. Lines 17–18 in Figure 7.4 then imply that no statement can now be executed, unless it has the `rv` parameter on that transition set to the same value, which is only the case for receive operations that target the same channel. A global state transition in the main execution loop of the semantics engine can only take place for rendezvous operations if the offer can be accepted.

We are now ready to resolve the semantics questions.

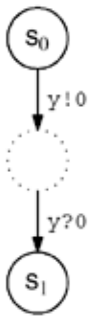
In the first example, according to the priority rule enforced by the `unless` operator, two statements are executable in the initial state: `x!0` and `y!0`. Either one could be selected for execution. If the first is executed, we enter a rendezvous offer, with `handshake` set to the `ch_id` of channel `x`. In the intermediate global state `s'` then reached, only one statement can be added to set `E'`, namely `x?0`. The final successor state has `handshake` == 0 with both processes in their final state. Alternatively, `y!0` could be selected for execution, with an analogous result. The resulting state space structure is illustrated in Figure 7.5. For convenience, we have included the intermediate states where rendezvous offers are in progress. If a rendezvous offer cannot be accepted, the search algorithm will not actually store the intermediate state in the state space. Similarly, if the offer is accepted, the transition from state s_0 to s_1 is equivalent to an atomic step.

Figure 7.5. State Space Structure for First and Third Example



In the second example, only one statement is executable in the initial system state: `y!0`, and only the corresponding handshake can take place. The resulting state space structure is illustrated in Figure 7.6.

Figure 7.6. State Space Structure for Second Example



In the third example, the first two statements considered, at the highest priority (line 12, Figure 7.3), are both unexecutable. One priority level lower, though, two statements become executable: `x!0` and `y!0`, and the resulting two system executions are again analogous to those from the first example, as illustrated in Figure 7.5.

A few quick checks with SPIN can confirm that indeed the basic executions we derived here are the only ones that can occur.

Verification

The addition of a verification option does not affect the semantics of a PROMELA model as it is defined here. Note, for instance, that the semantics engine does not include any special mention or interpretation of valid end states, accepting states, non–progress states, or assertions, and it does not include a definition for the semantics of `never` claims or `trace` assertions. The reason is that these language elements have no formal semantics within the model: they cannot be used to define any part of the behavior of a model.

Assertion statements, special labels, `never` claims, and `trace` assertions are used for making meta statements about the semantics of a model. How such meta statements are to be interpreted is defined in a verifier, as part of the verification algorithm.

When a verifier checks for safety properties it is interested, for instance, in cases where an `assert` statement can fail, or in the presence of executions that violate the requirements for proper termination (e.g., with all processes in a valid `end` state, and all message channels empty). In this case, the predefined system variable `stutter`, used in the definition of the semantics engine on line 22 in Figure 7.3, is set to false, and any mechanism can be in principle used to generate the executions of the system, in search of the violations.

When the verifier checks for liveness properties, it is interested in the presence of infinite executions that either contain finitely many traversals of user–defined `progress` states, or infinitely many traversals of user–defined `accept` states. The predefined system variable `stutter` is set to true in this case, and, again, any mechanism can be used to generate the infinite executions, as long as it conforms to the semantics as defined before. We discuss the algorithms that SPIN uses to solve these problems in Chapters 8 and 9. The definition of final states in product automata is further detailed in Appendix A.

The Never Claim

For purposes of verification, it is not necessary that indeed all finite or infinite executions that comply with the formal semantics are inspected by the verifier. In fact, the verifiers that are generated by SPIN make every effort to avoid inspecting all possible executions. Instead, they try to concentrate their efforts on a small set of executions that suffices to produce possible counterexamples to the correctness properties. The use of `never` claims plays an important role here. A `never` claim does not define new semantics, but is used to identify which part of the existing semantics can violate an independently stated correctness criterion.

The interpretation of a `never` claim by the verifier in the context of the semantics engine is as follows. Note that the purpose of the claim is to suppress the inspection of executions that could not possibly lead to a counterexample. To accomplish this, the verifier tries to reject some valid executions as soon as possible. The decision whether an execution should be rejected or continued can happen in two places: at line 2 of the semantics engine, and at line 22 (Figure 7.3), as illustrated in Figure 7.7.

Figure 7.7 Claim Stutter

```
  1 while ((E = executable(s)) != {})
 *2 {   if (check_fails()) Stop;
  3     for some (p,t) from E
  . . .
 21 }
*22 while (stutter) { s = s; if (check_fails()) Stop; }
```

SPIN implements the decision from line 22 by checking at the end of a finite execution if the `never` claim automaton can execute at least one more transition. Repeated stutter steps can then still lead to a counterexample. When the claim is generated from an `LTL` formula, all its transitions are `condition` statements, formalizing atomic propositions on the global system state. Only infinite executions that are consistent with the formal semantics of the model and with the constraint expressed by the `never` claim can now be generated.

With or without a constraint provided by a `never` claim, a verifier hunting for violations of liveness properties can check infinite executions for the presence of counterexamples to a correctness property. The method that the verifier uses to find and report those infinite executions is discussed in Chapter 8.