Chapter 3

A Quick Introduction to SPIN

3.1 Overview of SPIN and PROMELA

SPIN (see `http://netlib.bell-labs.com/netlib/spin/whatispin.html`) allows the simulation of a specification written in the language PROMELA (PROtocol MEta-LAnguage) and the verification of several types of properties, such as model checking of PLTL-formulas (without the next operator X) and verification of state properties, unreachable code, and so on. An overview of the SPIN-tool is given in Figure 3.1. The algorithms that are used by SPIN in order to perform model checking of PLTL-formulas are basically the algorithms that are shown in Section 2.6. For instance, the conversion of a PLTL-formula into a Büchi automaton is performed using the 5 steps that we have described before[1], and the detection of emptiness is done by a nested-depth first search procedure. It is beyond the scope of these lecture notes to

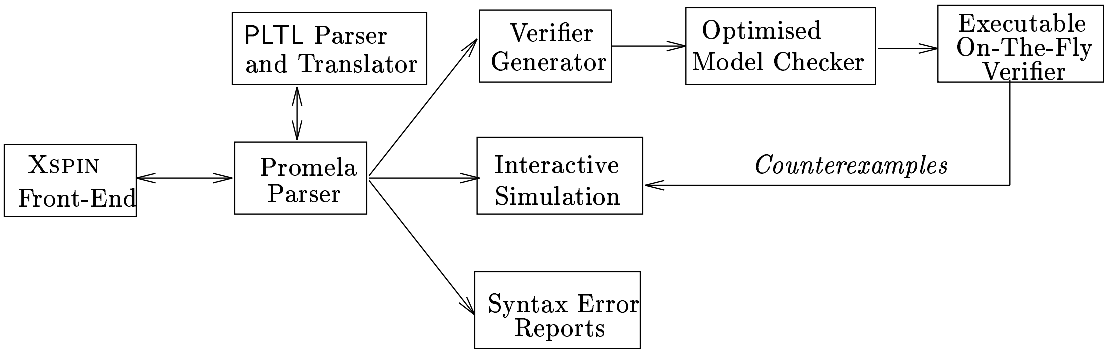

Figure 3.1: The structure of SPIN simulation and verification

give an extensive description of SPIN; there are several overview articles and an introductory book of Holzmann — the developer of SPIN — that cover this (Holzmann, 1991). Instead, we would like to give the reader some insight into how systems can be modeled in PROMELA and how an example verification and simulation is treated using SPIN. For that purpose we use a standard leader election protocol known from the literature. First we give a brief overview of

[1]Although there are some subtle differences (as pointed out to me by Rob Gerth), e.g. the generated automaton by SPIN — called never claim — does accept also finite runs.

the core of the language PROMELA.

A PROMELA specification consists roughly of sequential processes, (local and global) variables, and communication channels used to connect sequential processes. A process, say `P`, is defined as follows

```
proctype P (formal parameters) { local definitions; statements }
```

This construct only defines the process `P`, some variables and constants that are local to `P`, and its behavior, but it does not instantiate the process. Processes are started through the `run` construct that, for instance, appears in the initialization section indicated by `init`. Here, an instance of the processes `P` and `Q` are created and started.

```
init{ run P(actual parameters); run Q(actual parameters); ...... }
```

Statements in PROMELA can be either enabled or blocked. If a statement is blocked, execution of the statement halts at this point until the statement becomes enabled. So, for instance, the statement `(a == b)` is equivalent to `while (a != b) do skip`. PROMELA is strongly influenced by Dijkstra's guarded command language that we have used as an abstract program notation. Statements are separated by semicolons. To summarize the following constructs exist in PROMELA:

- The empty statement (written `skip`). It means 'do nothing'.
- Assignment statement, for instance, `x = 7` means that the variable `x` of type `byte` is assigned the value 7.
- If-statement. An if-statement has the form

  ```
  if :: guard1 -> statements1
     ::   ......
     :: guardn -> statementsn
  fi
  ```

  One of the alternatives whose guard is enabled is selected and the corresponding (non-empty) list of statements is executed. Non-determinism is allowed: if more than one guard is enabled, the selection is made non-deterministically. A special guard `else` is enabled when all other guards are blocked. If there is no else-guard and all other guards are blocked, the if-statement itself is blocked, and the execution halts until one of the guards becomes enabled.
- Do-statement: has the same form as an if-statement, with the only difference that `if` is replaced by `do` and `fi` by `od`. Notice that a do-statement is in principal always an infinite loop, since if all guards are blocked, it waits until one of the guards becomes enabled. A do-statement can therefore only be terminated by the use of `goto` or `break`.
- Send and receive statements. Processes can be interconnected via unidirectional channels of arbitrary (but finite) capacity. Channels of capacity 0 are also allowed. Channels

are specified in the same way as ordinary variables; e.g. `chan c = [5] of byte` defines a channel named `c` of capacity 5 (bytes). A channel can be viewed as a first-in first-out buffer. Sending a message is indicated by an exclamation mark, reception by a question mark. Suppose `c` has a positive capacity. Then `c!2` is enabled if `c` is not fully occupied. Its execution puts the element 2 at the end of the buffer. `c?x` is enabled if `c` is non-empty. Its effect is to remove the head of the buffer `c` and assign its value to the variable `x`. If `c` is unbuffered, i.e. it has size 0, then `c!` (and `c?`) is enabled if there is a corresponding `c?` (and `c!`, respectively) which can be executed simultaneously (and the types of the parameters match). This facility is useful for modeling synchronous communication between processes.

Other PROMELA constructs that we need for our small case study will be introduced on-the-fly in the sequel.

3.2 Dolev, Klawe and Rodeh's leader election protocol

It is assumed that there are N processes ($N \geqslant 2$) in a ring topology, connected by unbounded channels. A process can only send messages in a clockwise manner. Initially, each process has a unique identifier *ident*, below assumed to be a natural number. The purpose of a leader election protocol is to make sure that exactly one process will become the leader. In addition, it is the idea that the process with the highest identifier should be elected as a leader. Here the concept is that the identifier of a process represents its capabilities and it is desired that the process with the largest capabilities wins the election. (In real life this is also often desired, but usually the outcome of an election does not conform to the latter principle.) In the protocol of Dolev, Klawe and Rodeh (1982) each process in the ring performs the following task:

<u>*active:*</u>
$d := ident$;
do forever
begin
    $send(d)$;
    $receive(e)$;
    if $e = d$ **then stop**; (* process d is the leader *)
    $send(e)$;
    $receive(f)$;
    if $e \geqslant max(d, f)$ **then** $d := e$ **else goto** *relay*;
end

<u>*relay:*</u>
do forever
begin
  $receive(d)$;
  $send(d)$
end

The intuition behind the protocol is as follows. Initially each process is *active*. As long as a process is active it is responsible for a certain process number (kept in variable d). This value may change during the evolvement of the protocol. When a process determines that it does not keep the identity of a leader-in-spe, it becomes relaying. If a process is relaying it passes messages from left to right in a transparent way, that is, without inspecting nor modifying their contents.

Each active process sends its variable d to its clockwise neighbor, and then waits until it receives the value (e) of its nearest active anti-clockwise neighbor. If the process receives its own d, it concludes that it is the only active process left and that d is indeed the identity of the new leader, and terminates. (Notice that the process itself does not have to become the new leader!) In case a different value is received ($e \neq d$), then the process waits for the second message (f) that contains the value d kept by the second nearest active anti-clockwise neighbor. If the value of the nearest active anti-clockwise neighbor is the largest among e, f, and d, then the process updates its local value (i.e. $d := e$), otherwise it becomes relaying. Thus from every set of active neighbours one will become relaying in every round.

To illustrate how the protocol works, an example run of the protocol for 4 processes with identities 3, 4, 27 and 121, respectively, is shown in Figure 3.2. Here, processes are indicates by circles, and communication channels by arrows. The content of a channel is indicated by the label of the arrow, where the semi-colon symbol is used to separate distinct messages. Each circle is labelled by a triple indicating the current value of the local variables d, e and f. A black circle indicates that the process is relaying, otherwise it is active.

3.3 Modeling the leader election protocol in PROMELA

The PROMELA specification consists of three parts: definitions of global variables, description of the behaviour of a process in the ring, and an initialisation section where all processes are started. The behavioural part of a `process` is actually obtained from a straightforward translation of the protocol description given just above. We have added two print comments to facilitate simulation. A process is connected to its anti-clockwise neighbour via channel `in`, and via channel `out` to its clockwise neighbour.

```
proctype process (chan in, out; byte ident)
{ byte d, e, f;

  printf("MSC: %d\n", ident);

activ:
  d = ident;
  do :: true -> out!d;
                in?e;
                if :: (e == d) ->
                        printf("MSC: %d is LEADER\n", d);
                        goto stop  /* process d is leader */
                   :: else -> skip
```

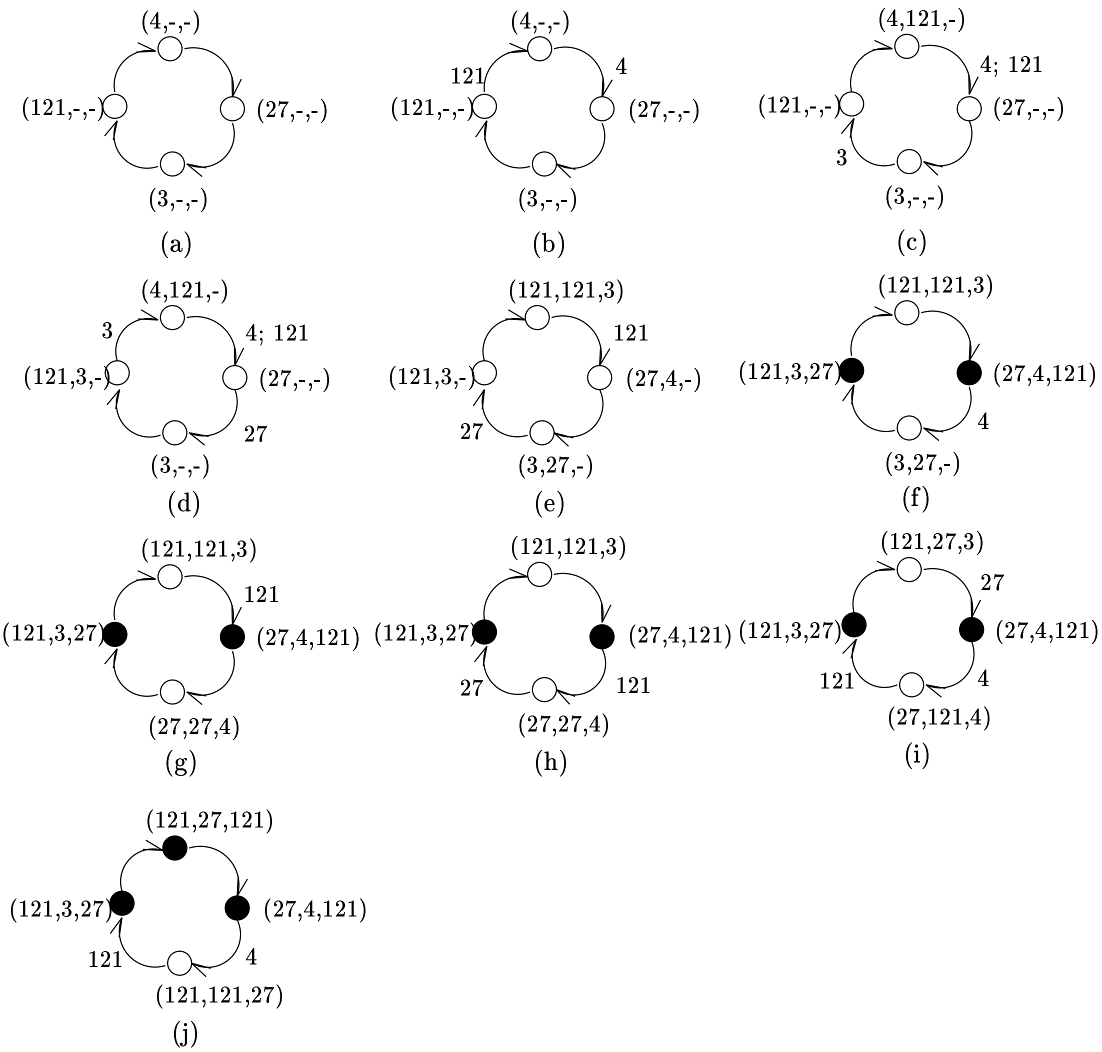

Figure 3.2: Example run of the leader election protocol

```
                fi;
                out!e;
                in?f;
                if :: (e >= d) && (e >= f) -> d = e
                   :: else -> goto relay
                fi
  od;

relay:
end:
  do :: in?d -> out!d
  od;

stop:
```

```
  skip
}
```

A process can be in three states: `activ` (we do not use "active", since this is a reserved keyword in PROMELA), `relay`, and `stop`. A process can only reach the stop state if it has recognised a new leader. A process that does not recognise this leader will end in the relay state. In order to avoid the generation of an invalid endstate (a state that is neither the end of a program body nor labeled with `end:`) we label the relay state as an end-state. If one were to verify the above PROMELA specification without this endlabel, SPIN would find an invalid end-state and report this as an error.

Notice that, for instance, the statement `in?e` is enabled only if the channel `in` is non-empty. If the channel is empty, the statement is blocked and the execution of the process is suspended until the statement becomes enabled. Therefore `in?e` should be read as: wait until a message arrives in channel `in` and then store its value in the variable `e`.

The PROMELA specification starts with the definition of the global variables:

```
#define N       5               /* number of processes */
#define I       3               /* process with smallest identifier */
#define L       10              /* size of buffer (>= 2*N) */

chan q[N] = [L] of { byte };   /* N channels of capacity L each */
```

The constant `I` is used for initialisation purposes only; it indicates the process that will obtain the smallest `ident`. The capacity `L` of connecting channels is taken to be at least $2N$ since each process sends at most two messages in each round of the election. Since messages in the leader election protocol carry only a single parameter (a process identifier), the channels `q[0]` through `q[N-1]` can contain items of type `byte`.

In order to create the ring topology and to instantiate the processes in the ring, the `init` construct is used. This part of the PROMELA specification is, for instance, as follows:

```
init {
  byte i;
  atomic {
        i = 1;
        do :: i <= N -> run process (q[i-1], q[i%N], (N+I-i)%N+1);
                        i = i+1
           :: i > N -> break
        od
 }
}
```

where `%N` denotes the modulo N-operator. A process is started with some actual parameters by the `run process ()` statement. The first parameter of a process is the incoming channel, the second parameter is the outgoing channel, and the third parameter is its identity. There are, of course, different mechanisms to assign identifiers to processes, and we simply selected

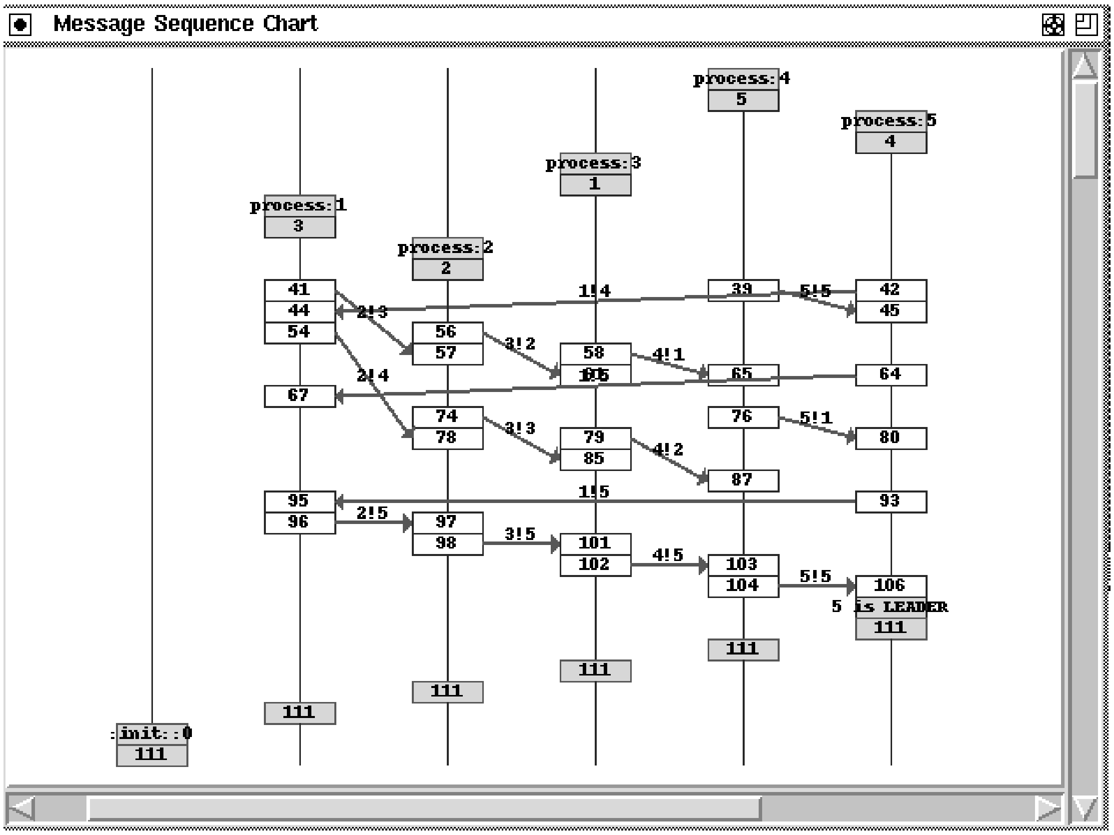

Figure 3.3: Visualisation of a simulation run in SPIN

one. It is, however, assumed that each process gets a unique identifier. This is essential for the correctness of the algorithm of Dolev et. al. In order to allow all processes to start at the same time we use the `atomic` construct. This prevents some process starts to already execute its process body, as long as other processes have not yet been created.

Notice that we have fixed one particular assignment of process identities to processes. The results we will obtain by simulation and verification are valid only for this assignment. In order to increase confidence in the correctness of the protocol, other – in fact, all possible — assignments need to be checked. Another possibility is to write a PROMELA fragment that assigns process identifiers randomly.

The PROMELA specification ontained in this way can now be verified, and no unreachable code or deadlocks are obtained. In order to obtain insight into the operation of the protocol a (random, guided, or interactive) simulation can be carried out. This produces, for instance, the result shown in Figure 3.3. Here, the behaviour of each process, including the root process (leftmost line), is represented along a vertical line.

3.4 Model checking the PROMELA specification

The most important property that a leader election protocol must possess is that it does not allow more than one process to become a leader. This is expressed in PLTL as:

$$\mathsf{G}\,\neg\,[\#leaders \geqslant 2],$$

or, equivalently,

$$\mathsf{G}\,[\#leaders \leqslant 1].$$

This property is checked using SPIN in the following way. One defines the formula as `[] p` where `[]` corresponds to G and `p` is defined as

```
#define p  (nr_leaders <= 1)
```

where `nr_leaders` is an auxiliary global variable in the PROMELA specification which keeps track of the number of leaders in the system. Therefore the PROMELA specification is extended in the following way.

```
byte nr_leaders = 0;

proctype process (chan in, out; byte ident)
{ .....as before.....

activ:
  d = ident;
  do :: true -> out!d;
                in?e;
                if :: (e == d) ->
                           printf("MSC: %d is LEADER\n", d);
                           nr_leaders = nr_leaders + 1;
                           goto stop  /* process d is leader */
                   :: else -> skip
                fi;

   .....as before.....
}
```

The property `[] p` is automatically converted by SPIN into a labelled Büchi automaton. In essence, this has been achieved by implementing the basic part of this chapter (Holzmann, 1997). One of the major differences is that SPIN labels transitions with atomic propositions, whereas we have labeled the states with (sets of) atomic propositions. The automaton generated by SPIN is as follows.

```
/*
* Formula As Typed: []  p
* The Never Claim Below Corresponds
* To The Negated Formula !([]  p)
* (formalizing violations of the original)
*/

never {    /* !([]  p) */
T0_init:
        if
        :: (1) -> goto T0_init
        :: (! ((p))) -> goto accept_all
        fi;
accept_all:
        skip
}
```

Notice that this is the Büchi automaton that corresponds to the negation of the formula to be checked; in our notation this would be $A_{\neg\,\phi}$, where ϕ corresponds to $\mathsf{G}\,[\#leaders \leqslant 1]$. If this property is saved in the file `property1`, then SPIN can automatically check it if we add at the end (or beginning) of our PROMELA specification the statement:

```
#include  "property1"
```

The result of the verification of this property is positive:

```
Full state space search for:
never-claim          +
assertion violations + (if within scope of claim)
cycle checks           - (disabled by -DSAFETY)
invalid endstates      - (disabled by never-claim)

State-vector 140 byte, depth reached 155, errors: 0
   16585 states, stored
   44589 states, matched
   61174 transitions (= stored+matched)
      18 atomic steps
hash conflicts: 1766 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
2.388  equivalent memory usage for states
                         (stored*(State-vector + overhead))
2.046  actual memory usage for states (compression: 85.65%)
State-vector as stored = 119 byte + 4 byte overhead
2.097  memory used for hash-table (-w19)
```

```
0.200  memory used for DFS stack (-m10000)
4.448  total actual memory usage
```

as indicated by the fact that `errors: 0`. Even for this small protocol with 5 processes (plus their interconnecting channels) the state space is already rather large. SPIN therefore provides various mechanisms in order to make the state space smaller, either through storing it in a compact way (using partial-order reduction) or by avoiding searching the entire state space (using hashing), or even a combination of both. These techniques are further discussed in Chapter 5 of these lecture notes. The effect of applying partial-order reduction to the above example is that only 3189 states and 5014 transitions are stored rather than 16585 states and 61172 transitions, the values if partial-order reduction is not applied. In the remainder of this example we will use partial-order reduction as a standard option.

In a similar way as above the following properties could be verified: always a leader will be elected eventually, the elected leader will be the process with the highest number, if a process is elected as a leader it remains a leader forever. The interested reader is encouraged to formulate these properties in PLTL and perform these verifications.

In order to see what happens when we check a property that is violated we check $\mathsf{G}\,[\#leaders = 1]$. Obviously, this property is violated in the state where all process instantiations are created, since then there is no leader. If we run SPIN we indeed obtain an error, and a counter-example is generated and stored in a file. In order to analyze the error, a guided simulation run can be carried out, based on the generated counter-example. The shortest trace that leads to the error found can be generated on request. This option can be indicated when the verification is started. Basically, SPIN then performs a breadth-first search rather than a depth-first search.

3.5 Verification using assertions

An alternative and often more efficient way of verification is the use of so-called *assertions*. An assertion has a boolean expression as argument. Its execution has no side-effects. If a PROMELA specification contains an assertion, SPIN checks whether there exists a run in which the assertion is violated. In such a case, an error is indicated. Since, in general, the state space is not increased when checking PLTL-properties (where the product automaton of the property and the specification is constructed) checking for assertions is usually more efficient. As an example of the use of assertions, suppose we want to check whether it is always the case that the number of leaders is at most one. This can be checked by adding the in-line assertion

```
assert(nr_leaders <= 1)
```

to our PROMELA specification at the point where a process determines that it has won the election and increments the variable `nr_leaders`. So we obtain as the body of a process:

```
byte nr_leaders = 0;

proctype process (chan in, out; byte ident)
```

```
{ .....as before.....

activ:
  d = ident;
  do :: true -> out!d;
                in?e;
                if :: (e == d) ->
                         printf("MSC: %d is LEADER\n", d);
                         nr_leaders = nr_leaders + 1;
                         assert(nr_leaders <= 1);
                         goto stop  /* process d is leader */
                   :: else -> skip
                fi;

   .....as before.....
}
```

Verification of this PROMELA specification yields no errors, and uses the same number of states and transitions as the verification of the corresponding PLTL-formula $\mathsf{G}\,[\#leaders \leqslant 1]$.

3.6 Selected references

SPIN:

- G.J. HOLZMANN. Designing bug-free protocols with SPIN. *Computer Communications* **20**: 97–105, 1997.
- G.J. HOLZMANN. The model checker SPIN. *IEEE Transactions on Software Engineering* **23**(5): 279–295, 1997.
- G.J. HOLZMANN. Design and validation of protocols: a tutorial. *Computer Networks & ISDN Systems* **25**: 981–1017, 1993.
- G.J. HOLZMANN. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- R. GERTH. *Concise Promela reference*, 1997. (available on the web-site for SPIN.)

Leader election protocol:

- D. DOLEV, M. KLAWE, AND M. RODEH. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms* **3**: 245–260, 1982.