Informatics and Mathematical Modelling
Technical University of Denmark
Building 321
DK–2800 Lyngby
Denmark

HHL 11–09–2007

Hans Henrik Løvengreen:

Introduction to SPIN

Version 1.2

In this note we introduce the SPIN model checker for verification of concurrent programs.

The reader is assumed to be familiar with concurrent programming notions as well as with basic concurrency theory, especially the modelling of concurrency by interleaving of atomic actions (see eg [Løv05]).

Contents

1 Introduction

Concurrent programs are notoriously difficult to write, test and debug due to their inherent non-determinism. Concurrent processes may interact in unforeseen ways leading to *race conditions* that may result in inconsistent states or erroneous results.

An approach to address these issues is to formally prove a concurrent program to satisfy certain safety and liveness properties. However, in practice such proofs may be cumbersome to carry out and are subject to errors like the programs they are supposed to verify.

An alternative program verification approach is to model the program in an abstract way leading to the reachable state space being finite. In that case the reachable state space may be exhaustively explored and desired properties verified to hold for all states. This approach is commonly known as *model checking*.

One particular tool for model checking of concurrent programs is the SPIN model checker developed by Gerald Holzmann. In this note, we give a brief introduction to the most basic notions of the SPIN verifier. The reader is referred to the comprehensive reference material for details.

2 Background

Based on the earlier work of PAN (Protocol ANalazer), SPIN was developed by Gerald J. Holzmann at Bell Labs in the 1980ies as an in-house tool for verification of various telecommunication protocols. The first version of the SPIN-tool was released in 1989 and has since been further developed and fine-tuned. Today SPIN is a standard reference within verification and is used as a verification back-end for several language specific verification tools.

SPIN has been applied for verification of many examples of critical systems — including a number of NASA missions.

The tool, its usage and its theoretical background are thoroughly described by its author in the recent book [Hol04]. Further documentation and tool reference material can be found at the tool home page:

`spinroot.com`

The tool is an open source project and is free to use for educational purposes. The tool can be freely downloaded from the home page.

SPIN can work both as a *simulator* and a *verification engine*. In this note, we focus on the verification aspect of SPIN.

3 Promela

The starting point for any use of SPIN is a model of a concurrent program (or system) expressed in the language *Promela* (**Pro**cess **me**ta **la**nguage).

Promela is a textual modelling language based on Dijsktra's Guarded Command extended with CSP-like communication primitives. The concrete syntax is C-like.

A Promela program consists of global declarations of shared variables and communication channels plus a number of process type declarations. A finite number of processes can be instantiated for the initial system state. Further processes may be dynamically created during the execution of the model.

Processes may *interact* by sharing global variables and/or communicating over synchronous or asynchronous channels. Each process type may declare local variables to be used only within the process.

To simplify the notion of system state, procedures are not provided and hence no notion of stacks is needed. A simple macro expansion mechanism can be used if needed.

3.1 Example

The following example shows a Promela model of an archetypal concurrent program using shared variables

```
/* Peterson's solution to the mutual exclusion problem - 1981 */

bool in1, in2 = 0;
byte turn = 1;

active proctype P1() {
  do :: in1 = 1;
        turn = 2;

        /*await*/ (in2 == 0 || turn == 1) ->

        /*  critical section */

        in1 = 0;

        /*  non-critical section */
        if :: skip :: break fi
   od
}

active proctype P2() {
  do :: in2 = 1;
        turn = 1;

        /*await*/ (in1 == 0 || turn == 2) ->

        /*  critical section */

        in2 = 0;

        /*  non-critical section */
        if :: skip :: break fi
  od
}
```

The example shows two processes that implement a critical region using shared variables. Details will be elaborated below.

3.2 Data types and expressions

The only data-types available in SPIN are integers, arrays of these and C-like structs (records). Integer variables may be of different sizes allowing for using only as much memory as needed. Some of the integer data types are:

Type	**Range**	**Description**
`int`	$-2^{31}..2^{31}-1$	32-bit signed integer
`short`	$-2^{15}..2^{15}-1$	16-bit signed integer
`byte`	0..255	8-bit unsigned integer
`bool`	0,1	1-bit integer (boolean)

Variables and arrays of these types are declared using C-style declarations:

```
int x = k ;
byte a[n] ;
```

Furthermore, non-negative integer variables of any bit-size k can be declared by:

```
unsigned x : k ;
```

Enumerations may be defined as in

```
mtype = { UP, DOWN, RIGHT, LEFT }
⋮
mtype x = UP;
```

This will assign unique integer codes to the symbols `UP, DOWN, RIGHT, LEFT` , but otherwise the enumeration type `mtype` is equivalent to the `byte` type.

Since a Promela program is passed through the C pre-processor, constants may alternatively be defined by macros like:

```
#define N  8
```

The *expression language* is mostly as in C-like languages.

Usage notes:

- All the integer types are eventually implemented by the C-compiler being used. That means that usually *modulo arithmetic* applies and therefore any attempt of exceeding the range limits of a variable should be avoided.
- All integer types are initialized to 0, unless explicit initialization is applied.
- As in C, any integer expression may be interpreted as a boolean value where 0 represents *false* and all non-zero values represent *true*.
- In general, the smallest integer type possible should be used in order to reduce the program state space.
- No side effects are allowed in expressions.

3.3 Processes

Processes are generally declared as *process types* (`proctype`) of which instances may be created. Process types may be parameterized allowing for actual parameters to be passed at instantiation time.

Processes may be instantiated at program start within the special *initial process* named `init` or dynamically within any other process by statements of the form `run` P`(...)`, where P is a process type. Each process instance is given a unique *process instantiation number* (or *process identifier*, pid) which is returned by he `run` statement creating it. Within the process itself, the identifier can be obtained through the special variable `_pid`.

Process types may also be implicitly instantiated by prepending the type declaration with the keyword `active`. This will create one initial instance of the process type (with any parameters being set to 0). See Peterson's Algorithm for an example. Also arrays of processes may be created by using the form `active [`n`]`.

3.4 Statements

The most fundamental statements of SPIN are *assignments statements* ($x = e$) which change the state and *guards* (b `->`) which block an execution path until a certain conditions is satisfied.

An important dynamic notion of a statement is its readiness to execute. An assignment statement is always *executable* (or enabled) whereas a guard is executable only if it evaluates to true. If a guard evaluates to false, it is *blocked* (or disabled).

Statements may be *sequentially* composed using semicolons: S_1 `;` S_2`;` `...` `;` S_n. Statements may be *grouped* together using braces. The `skip` statement is an empty statement (ie. a statement having no effect) which may be used where a statement is required.

Usage notes:

- The SPIN-syntax analyser seems very sensitive towards improper use of semicolons and may give incomprehensible error messages once it gets confused about the langauge structure. In general, semicolon should be used to *separate statements at the same level*.

  Here are some specific rules:

  - Avoid semicolon before `::`, `fi` and `od`
  - Always semicolon after declarations

The *selection structure*

```
if
   :: S1
   :: S2
       ⋮
   :: Sn
fi
```

provides a branching point from which one of several execution paths may be followed depending on their readiness to execute. When control reaches the statement, a (non-deterministic) choice is made among the executable sub-statements $S_1 \ldots S_n$. As long as all of these are blocked, so is the selection construct.

Since the choice is made among the executable statements, typically the first statement of each S_i is a guard controlling this choice. For instance, a traditional **if** b **then** S_1 **else** S_2 is readily implemented by:

```
if
  ::  b -> S1
  :: !b -> S2
fi
```

Usage notes:

- The non-deterministic choice among executable statements may be used when the precise condition for making a choice has been abstracted from. For instance, in Peterson's Algorithm, we have used a non-deterministic choice to model that the process may or may not want to use the critical section again.

A *repetition structure* (`do` :: S_1 :: S_2 :: ... :: S_n `od`) works like a repeated selection. The construct can be left by a `break` statement. Hence, a traditional while statement **while** b **do** S may readily be expressed by:

```
do
  :: !b -> break
  ::  b -> S
od
```

For more complex control flows, labels and `goto` statements may be used. Labels may also be used to identify program locations for property specification.

Usage notes:

- The first statement of a branch in a selection/repetition construct cannot be labelled. If needed, a `skip` statement may be inserted first.

3.5 Atomic Statements

By definition every basic statement is considered to be atomic (virtually indivisible) by the SPIN verifier. This is true, even if it contains more than one critical reference and hence cannot be expected to be atomic by the standard assumptions about program execution. Therefore, if non-atomic program statements are to be correctly modelled, they have to be refined into several model statements as usual.

For instance, in the example of Peterson's Algorithm above, the entry condition `(in1 == 0 || turn == 2)` is assumed to be evaluated atomically by SPIN although in a real execution other processes may execute in-between testing the two conditions. It can be argued that for the concrete case, the possible executions are the same though.

When larger atomic actions are required, several statements can be grouped together using the *atomic* construct.

For instance, a conditional action corresponding to a semaphore P-operation $\langle\, s > 0 \rightarrow s := s - 1 \,\rangle$ can be expressed by:

```
atomic { s > 0  ->  s = s - 1 }
```

In general any statement may be "atomized", the semantics being that once chosen for execution, statements within the atomic construct are chosen in favour of statements in other processes as long as some is executable. A blocked statement within the atomic construct will break the atomicity until execution can continue.

Usage notes:

- Although the atomic construct directly expresses the idea of a larger atomic action, it is quite general allowing for non-determinism and intermediate blocking. For the same reasons, its treatment by the verifier may be less efficient. For simple, deterministic actions, the `d_step` (deterministic step) construct should be used instead although its name is less suggestive.

3.6 Message Passing

Promela also allows for interaction through *message passing*. Processes may communicate *messages* over *channels*. The communication is normally assumed to be reliable and ordered. Channels may have a *finite buffer capacity* corresponding to *asynchronous communication* or they may have *zero capacity* corresponding to *synchronous communication* (as in CSP [Hoa78]).

Channels are *typed*, ie. it is assumed that messages sent over a channel have a fixed structure (one or more *fields*). In communication protocol analysis, the first field is often an enumeration value indicating the kind of message being sent.

The primitives for sending and receiving are taken from CSP: $c\,!\,e$ denotes *sending* the value of expression e on channel c and $c\,?\,x$ denotes *receiving* a message from channel c, storing it in variable x. If the channel has more than one message field, the corresponding number of expressions/variables must be provided.

A communication statement is executable when control has reached it and the communication can take place. For synchronous communication, this means that a corresponding communication statement has been reached in another process. For asynchronous communication, a receive statement is executable if the channels holds a message in its buffer. A send statement is executable if there is room in the channel buffer. In Spin, it is optional whether an attempt to send to a full channels should block or give rise to an error.

It is seen that communication statements work both as guards and actions (as in CSP). This way, a server process serving clients on two different channels may be modelled as:

```
do
  :: c?x -> ...
  :: d?y -> ...
od
```

Usage notes:

- There is no direct way to combine boolean guards and communication guards as in CSP. This will have to be simulated by first branching on the boolean conditions and then choosing among a number of different communication selections.

3.7 Macros and Inline Procedures

Promela does not provide proper procedures, but the C pre-processor can be used to define procedure-like macros. For instance, an atomic exchange operation may be defined by:

```
#define SWAP(X,Y) atomic{int t; t = X; X = Y; Y = t}
```

[Note that the temporary variable `t` will become a (multiply declared) local variable used by all calls of this within the same process instance.]

However, Promela also provides its own built-in macro facility called `inline`. For instance, the `SWAP` operation might also be defined by:

```
inline SWAP(X,Y) {
  atomic{int t; t = X; X = Y; Y = t}
}
```

Since the expansion is controlled by Spin, this form enables more informative error messages.

4 How Spin Works

Usually verifiers build an internal representation of the model to be analysed and then uses this to generate a representation of the reachable state space. For efficiency reasons, however, Spin generates a dedicated *verification program* for each verification task. This is an ANSI C program called `pan` (historical from the previous tool) together with a number of auxiliary files. The verification task is then accomplished by compiling and running the `pan` program (with various options).

Both Spin and the **pan** programs are command line programs. Various front-ends may then use these to accomplish verification tasks.

Spin comes with a graphical user interface (GUI) front-end called `xspin` written in the Tcl/tk. It gives access to most of Spin's simulation and verification features, but has a tendency to open confusingly many windows on the user's screen.

An alternative GUI front-end provided by M. Ben-Ari is `jspin` written in Java [BA]. This front-end tries to spare the user from the rough output of Spin/`pan` by presenting it in a processed form within a simple graphical environment. On the other hand, `jspin` does not provide access to all facilities of Spin. The `jspin` tool is the basis for the recent book by Ben-Ari [BA08] which gives a gentle introduction to the use of Spin.

4.1 Verification Algorithms and Options

Basically SPIN does it verification work by constructing the set of states reachable from the initial one through interleaving of actions. In order to recognise states already being visited, a hash table is used. Furthermore, various techniques to compact the state representations may be applied. Usually states are visited using a depth first algorithm, but also breath-first may be selected.

For large systems, an exhaustive search may not be feasible due to space limitations. In that case, the *bitstate hashing* option may be selected. With this technique, states visited are not stored themselves, only their hash codes are marked. Although most reachable states are usually visited using this technique, it cannot be guaranteed since two different states may incidentally have the same hash code with the result that one of them is not further investigated.

Normally *partial order reduction* is turned on. This means that local actions are not arbitrarily interleaved but always taken together with some non-local action (ie a statement containing at least one critical reference).

It is hard to use SPIN without seeing the tool reporting about *never claims*. A never claim is a process representing a finite state automaton (FSA) which "monitors" or "tracks" the behaviour of the system investigated while at the same time constraining the system behaviour. Usually a never claim characterizes undesired behaviour and it is considered an error for the claim to "succeed". Never claims are mostly used for proving liveness properties and are usually generated automatically from temporal logic formulas.

Usage notes:

- Read the reference material if you want to experiment with the verification options to understand the implications.

5 Verification of Safety Properties

Safety properties of concurrent programs informally express that the program does nothing wrong. Typically safety properties are expressed as *invariants*, ie. state predicates that must be satisfied by every reachable state (see [Løv05]).

In SPIN, safety properties are verified by generating the reachable state space incrementally until a safety violation is detected, or the state space has been exhausted.

SPIN should be used in a special *safety mode* when generating the verification program.

5.1 History variables

Many properties of concurrent programs deal with the actual control flow of the individual processes. To express such properties, control assertions like *at a*, *in a* or *after a* are often used.

Although SPIN allows some forms of control assertions, often *history variables* are added to the system to faithfully record control information. In general, history variables may record, but not alter the underlying program state.

A typical example is that of a *critical region* where mutual exclusion should hold between a number of *critical sections*. To express this, a history variable counting the number of processes within the critical region may be added:

```
byte incrit = 0;      /* history variable */

proctype Pi() {
  ⋮
  incrit++
  /* critical section */
  incrit--
  ⋮
}
```

Observe that the increment statements `incrit++` and `incrit--` are implicitly understood to be atomic in Promela and therefore they count correctly.

5.2 Proving safety properties with assertions

Different techniques may be applied to state and check safety properties.

The most immediate method is to use *assert statements* of the form `assert(`b`)`. If control reaches such a statement when the condition b is false, the SPIN verification program will stop with an error message.

Assert statements may be used to express "local" invariants of the form *at* $l \Rightarrow P$, that is, that P holds when control is at l. For example, using the auxiliary variable introduced before, mutual exclusion may be expressed by stating than within the critical section, the number of processes within the region is **exactly** 1.

```
byte incrit = 0;

proctype Pi() {
  ⋮
  incrit++;
  /* critical section */
  assert(incrit == 1);
  incrit--;
  ⋮
}
```

Safety properties may also be expressed by *global invariants*. These may then be checked by a "monitoring" process which continuously checks the invariant. The most efficient way of doing this is to let the process wait for a chance to discover a violation of the invariant and then report this, eg. by asserting false. This way, the monitor process need not take a step in all the situations where the invariant is satisfied.

For example, mutual exclusion can be formulated as the global invariant `incrit` $<= 1$ and checked by adding a monitor process like:

```
⋮
active proctype Check_Mutex() {
  !(incrit <=1) ->
  assert(false)
}
```

Alternatively, global invariants may be stated as LTL formulas (see next section) and automatically translated into the notion of never claims mentioned above.

Finally, more general safety properties may involve history information, like "x cannot become negative before y does". Such properties can be also be encoded as never claims, however details of how to write your own never claims are beyond the scope of this note.

5.3 Checking for deadlocks

A special kind of safety properties are *deadlocks*. In general, a group of processes are in deadlock when they are all blocked, waiting for some condition that must be established by some other process within the group.

SPIN is not able to analyse which process a given blocked process may be waiting for. However, if a situation occurs in which **all** active processes are blocked (or terminated), SPIN will report this a deadlock situation.

To avoid that perfectly terminated processes are reported to be deadlocked, a given control location may be declared to be an acceptable *end point*. This is done by labelling the control point with a special label of the form `end`. . . .

By default, the final control location of each process (at the `}`) is considered an acceptable end point.

For processes which act as servers, the start of the server loop should typically be marked as an end point to prevent a deadlock report in a situation where all clients may have terminated.

Local deadlocks may be detected by characterising the states in which they cannot occur. However, often it is more direct to state desired liveness properties rather than chasing deadlocks.

6 Verification of Liveness Properties

Informally *liveness properties* state that something eventually happens. Combined with safety properties, this implies that something good happens.

Technically, SPIN treats liveness properties through the theory of ω-automata, that is, finite state automata that defines sets of infinite executions. Specifically liveness properties are characterised by never claims where certain states are marked as being *accepting states*. When asked to do so (looking for acceptance cycles) , SPIN will determine whether it is possible to find an infinite system execution that passes through an accepting state infinitely often and report whether such an execution exists.

Usually this is used to prove the existence of a *counter-example* to the property of interest.

6.1 Using LTL to Prove Liveness

Never claims and acceptance conditions can be quite tricky to get right. An alternative is to express intended liveness properties using in some form of *temporal logic*. For most specification purposes, *linear temporal logic* (LTL) is sufficient. An introduction to LTL can be found in [Løv05]. The LTL used in SPIN, however, also allow for binary temporal operators like *weak* and *strong until*. The ASCII-format of the temporal operators in SPIN is as follows:

Operator	**Name**	**SPIN**
$\Box P$	Always P	`[]` P
$\Diamond P$	Eventually P	`<>` P
$P \leadsto Q$	P leads to Q	`[] (`P `-> <>` Q `)`

Any LTL formula can be implemented by a never claim with acceptance conditions. SPIN provides a built-in translation from LTL to never claims which may then be analysed by SPIN. This is more or less automatically handled by the GUI front-ends.

The LTL translator can only deal with LTL formulas over propositional variables. This means that eg. a formula like $\Box(x > 0)$ must be stated as `[] XPOS` and then `XPOS` must be defined by macro expansion:

```
#define XPOS (x > 0)
```

somewhere in the program.

Consider for example the property of critical region fairness for process `P1`. First we introduce labels at the entry protocol and at the critical section of process `P1`:

```
    active proctype P1() {
      do :: in1 = 1;
            turn = 2;

entry_1:    /*await*/ (in2 == 0 || turn == 1);

crit_1:     /*  critical section */

            in1 = 0;

            /*  non-critical section */
            if :: skip :: break fi
       od
    }
```

We may use these labels in *remote references* which are a kind of control assertions. These are used to define two propositional variables:

```
#define at_entry_1  (P1@entry_1)
#define at_crit_1   (P1@crit_1)
```

The usual fairness conditions:

$$\Box(at\ entry_1 \Rightarrow \Diamond at\ crit_1)$$

can now be stated by (using SPINs concrete ASCII syntax):

```
[] ( at_entry_1  ->  <> at_crit_1 )
```

To show this to hold for the program, the GUI front-ends will negate the formula and run an acceptance cycle test on the resulting never claim. If no acceptance cycle exists, the fairness property holds for the program.

Usage notes:

- If you have more than a single instance of a process, say `active [`*n*`] P`, the particular instance must be identified within a remote reference using the form (P[*id*]@*label*). Note, however, that the identifier *id* must be the *process instantiation number* and **not** (as would be natural to think) the array index of the instance.

  Instance numbers are the ones used in output from SPIN. They may be calculated from the number of active processes previously created previously. Alternatively, processes may be started explicitly by an `init` process and their instantiation numbers recorded in global values:

  ```
  pid p1, p2, p3;

  init() {
    atomic {
      p1 = run P();
      p2 = run P();
      p3 = run P()
    }
  }
  ```

  Now you may refer to eg. `P[p2]@entry` in LTL-definitions.

Process Fairness

If some processes may run forever, the interleaving semantics of concurrency allows executions in which other processes starve, ie never are let to execute. This correspond to process implementations in which high priority processes use all computation resources.

For cooperating processes, however, all processes should get at least some chance of making progress in order to contribute to the overall functionality of the system. This may be achieved by implementing processes using a fair scheduling principle like some form of round robin (as found in all desktop operating systems).

To model that all processes are assumed to get a chance to make progress, a *weak fairness* assumption may be enabled in SPIN. This means that any statement whose executability stabilises to true (if not taken) will eventually be executed.

SPIN provides no means of stating *strong fairness* guaranteeing that any periodically executable statement will eventually be executed. This is not really an issue, since general strong fairness cannot be efficiently implemented anyhow.

Usage notes:

- It is recommended always to turn on weak fairness when verifying liveness in SPIN.

7 Ending

In this note, we have tried to give a survey of the basic notions and facilities provided by Spin and Promela with the hope that the reader may start using the tool for practical verification.

The reader is urged to consult the reference material on the web for precise definitions and more advanced verification material.

In the coming years, we are likely to see a new generation of verification tools, eg the Zing tool developed by Microsoft [Zin]. Hopefully experience with SPIN will enable the reader to jump-start utilising these.

References

[BA] M. Ben-Ari. Jspin home.
`http://stwww.weizmann.ac.il/g-cs/benari/jspin`.

[BA08] M. Ben-Ari. *Principles of the SPIN Model Checker*. Springer, 2008.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8), August 1978.

[Hol04] Gerard J. Holtzmann. *The* Spin *Model Checker: Primer and reference manual.* Addison-Wesley, 2004.

[Løv05] Hans Henrik Løvengreen. Basic concurrency theory. Course notes, Informatics and Mathematical Modelling, Technical University of Denmark, September 2005. Version 1.1.

[Zin] Zing home. Microsoft.
`http://research.microsoft.com/zing`.