Chapter 18. Overview of SPIN Options

"An error does not become a mistake unless you refuse to correct it."

—(Manfred Eigen, 1927–)

In this chapter we discuss all available SPIN options. The options are grouped into seven categories according to their primary use, as follows:

- Compile−time options, which can be used to compile the SPIN source itself for different platforms and for different types of use
- Simulation options, which can be used for customizing simulation runs of PROMELA models
- Syntax checking options, for performing a basic check of the syntactic correctness of PROMELA models
- LTL conversion options, describing various ways in which the conversion from LTL formulae to PROMELA `never` claims can be done
- Model checker generation options
- Postscript generation options
- Miscellaneous options

Except for the first category above, all these options are defined as run−time parameters to the SPIN tool. Since SPIN is an evolving tool, new options will continue to be added to the tool from time to time. An up−to−date list of these options in the specific version of SPIN that is installed on your system can always be printed with the command:

```
$ spin --
```

The discussion in this chapter covers all compile−time and run−time options that are available in SPIN version 4.

Compile–Time Options

There are eight compile–time directives that can be used to change the default settings for the compilation of the SPIN sources. They are: `__FreeBSD__`, `CPP`, `MAXQ`, `NO_OPT`, `NXT`, `PC`, `PRINTF`, and `SOLARIS`.

These directives are typically set in the SPIN `makefile` at the time the tool is first installed on a system, and should rarely need modification later. The settings are typically included by changing the definition of the `CFLAGS` parameter in SPIN's `makefile`. As an example, to change the default location of the C preprocessor in the compiled version of SPIN, we could change the line

```
CFLAGS=-ansi -D_POSIX_SOURCE
```

into this new setting

```
CFLAGS=-ansi -D_POSIX_SOURCE -DCPP=/opt/prep/cpp
```

We first discuss the use of the `CPP` directive together with the two special cases `__FreeBSD__` and `SOLARIS`.

–DCPP=..., –DSOLARIS, –D__FreeBSD__

SPIN always preprocesses the source text of PROMELA models with the standard C preprocessor before it attempts to parse it. The preprocessor takes care of the interpretation of all `#include` and `#define` directives. On most UNIX systems, the location of the C preprocessor is `/lib/cpp`, so the default compilation of SPIN implies `-DCPP=/lib/cpp`.

On PCs the preprocessor is usually installed as a separate program, and therefore when SPIN is compiled with the directive `-DPC`, the default setting of the `CPP` parameter changes to `-DCPP=cpp`.

To override the default settings, an explicit value for the `CPP` parameter can be provided.

Two standard cases are predefined. By compiling SPIN with the directive `-D__FreeBSD__`, the default setting for the preprocessor changes to `cpp`, which matches the requirements on these systems. By compiling SPIN with the directive `-DSOLARIS`, the default setting for the preprocessor changes to `/usr/ccs/lib/cpp`, matching the location of the C preprocessor on Solaris systems.

These settings only affect the compilation of the `main.c` file from the SPIN sources.

There is also another way to define a switch to another preprocessor—by using SPIN's command line arguments `P` and `E`. For instance, on an OS2 system one might say:

```
$ spin −Picc −E/Pd+ −E/Q+ model
```

to notify SPIN that preprocessing is done by calling `icc` with parameters `/Pd+ /Q+`. Similarly, on a Windows system with the Visual C++ compiler installed, one can say:

```
$ spin −PCL −E/E model
```

independent of the settings for `CPP` with which SPIN itself was compiled. For more information on these run−time options, see the miscellaneous options section below.

−DMAXQ=N

The maximum number of message channels that can be created during a verification run is fixed to the value 255. This limit prevents both runaway models that may be attempting to create infinite numbers of channels, and it secures nicely that the identifying number, used internally by the verifier to keep track of channels, always fits within a single byte. In simulation, the requirements are less strict. In early versions of SPIN, the maximum number of channels that could be created in a simulation run was set at 2,500. In SPIN version 4.0 and later, the limit is set the same for both simulation and verification, which means that it is 255 in both modes. The default setting for simulations can be changed by compiling the SPIN sources with a different upper−bound, for instance, `−DMAXQ=2500`. This override only affects the compilation of the file `mesg.c` and it only affects the behavior of SPIN in simulation mode.

−DNO_OPT

In the conversion of LTL formulae into PROMELA `never` claims, SPIN uses a number of simple optimizations that are defined as rewrite rules on formulae. To study the effect of these rewrite rules, SPIN can be compiled without them. The rules are disabled if SPIN is compiled with the compiler directive `−DNO_OPT`. Clearly, it is not recommended to use this setting for normal use of SPIN.

−DNXT

In the syntax that SPIN accepts for the specification of LTL formulae, the standard LTL operator next, written `X`, is not allowed. The restriction secures that all LTL properties that can be expressed are necessarily compatible with SPIN's partial order reduction strategy. Technically, it means that the behavior expressed by these formulae is stutter invariant. Quite apart from this desirable closure property, one can also make a strong argument that the only types of properties that one can sensibly state about distributed system executions preclude the use of the `next` operator. (There is no real notion of a unique next step in a concurrent system.) The conversion algorithm for LTL formulae, though, can easily handle the `next` operator if the above concerns are not applicable. Furthermore, it is also possible to write LTL formulae that do include the `next` operator and that are still stutter invariant, but this is generally hard to determine.

If desired, therefore, SPIN can be compiled with the `next` operator enabled by adding the directive `−DNXT`.

−DPC

When the SPIN sources are compiled under Windows on a PC, a few UNIX−isms are not guaranteed to work. This, for example, includes calls to non−POSIX system library functions such as `sbrk()`. Also, the PC versions of parser generators such as `yacc` are likely to leave their output in files named `y_tab.c` and `y_tab.h`, rather then the multidotted file names used on UNIX systems `y.tab.c` and `y.tab.h`. Adding the compile−time directive `-DPC` will arrange for the right modifications in the compilation of the SPIN sources.

−DPRINTF

During verification runs, PROMELA `printf` statements are normally disabled. This means that execution of a print statement during the search that is performed by the verifier normally produces no output on the user's terminal. Such output would be of very little use. It would be generated in seemingly random order and it would noticeably slow down the verification process if it were not disabled. Nonetheless, for debugging purposes the execution of print statements during verification can be enabled by compiling the SPIN sources with the directive `-DPRINTF`. This change affects only the verification process; print statements are never disabled during simulation runs. The directive affects the compilation of only the source files named `pangen1.c` and `pangen2.c`.

Simulation Options

The command line options to SPIN that we discuss in this section are all meant to define or modify the type of output that is produced during a simulation run, that is, they do not affect any verification settings.

SPIN can be used for three main types of model simulation: random (default), interactive (option `-i`), and guided simulation (option `-t`).

When invoked with only a filename as an argument and no other command−line options, for instance,

```
$ spin model
```

SPIN performs a random simulation of the model that is specified in the file. If no filename is provided, SPIN attempts to read a model from the standard input. This can of course be confusing if it is unexpected, so SPIN gives a warning when it is placed in this mode:

```
$ spin
Spin Version 4.0.7 -- 1 August 2003
reading input from stdin:
```

Typing an end−of−file symbol gets the tool out of this mode again (control−d on UNIX systems, or control−z on Windows systems).

Also possibly confusing at first, even if a filename is provided, may be that a simulation run of the model by itself may not generate any visible output; for instance, when the PROMELA model does not contain any explicit print statements. At the end of the simulation run, though, if it does terminate, SPIN always prints some details about the final state that is reached when the simulation completes. By adding additional options (e.g., `-c`, or `-p`) more detail on a simulation run in progress will be provided, but only the information that the user explicitly requests is generated. Every line of output that is produced by the simulator normally also contains a reference to the source line in the model that generated it.

Summarizing: A random simulation is selected by default. If run−time option `-i` is used, the simulation will be performed in interactive mode, which means that the SPIN simulator will prompt the user each time that a non−deterministic choice has to be resolved. The user can then choose from a list of alternatives, and in this way interactively guide the execution towards a point of interest. If run−time option `-t` is used, the simulator is put into guided simulation mode, which presumes the existence of a trail file that must have been produced by the verification engine prior to the simulation.

Alphabetical Listing

–B

Suppresses the printout at the end of a simulation run, giving information on the final state that is reached in each process, the contents of channels, and the value of variables.

–b

Suppresses the output from print statements during a simulation run.

–c

Produces an ASCII approximation of a message sequence chart for a simulation run. For instance:

```
$ spin -c tpc6
proc 0 = :init:
proc 1 = user
proc 2 = local_tpc
proc 3 = manager
proc 4 = Session
proc 5 = Session
q   0   1   2   3   4   5
  4   .   tpc!offhook,0
  4   .   .   tpc?offhook,0
  5   .   .   handler[0]!offhook
  5   .   .   .   handler[0]?offhook
  1   .   .   .   child[0]!start
  1   .   .   .   .   me?start
  1   .   .   .   child[0]!offhook
...
```

For every process that is initiated, SPIN assigns a `pid` and prints it, together with the name of the corresponding `proctype`. In the example, the simulation run starts with the initiation of six processes of the types listed. In the remainder of the listing, the processes and message channels are only referred to by their identifying numbers. The output for each process appears in a different column. Only message send and receive operations are printed in the columnated format. If print statements are executed, the output from the statement also appears within the column of the executing process (but see options `-b` and `-T`). For send and receive operations, the left margin lists the specific channel number that is used for the operation. See also options `-M`, and `-u`.

–g

Shows at each time step the current value of global variables. Normally, this only prints the value of a global variable when a new value was assigned to that variable in the last execution step. To obtain a full listing of all global variable values at each execution step, add options `-v` and `-w`.

−i

Performs an interactive simulation, allowing the user to resolve non−deterministic choices wherever they occur during the simulation run. The simulation proceeds without user intervention whenever it can proceed deterministically.

−J

Reverses the evaluation order of nested `unless` statements, so that the resulting semantics conforms to the evaluation order of nested `catch` statements in Java programs.

−jN

Skips the first `N` steps of a random or guided simulation. `N` most be a positive integer value. See also option `−uN`.

−l

In combination with option `−p`, this shows the current value of local variables of the process. Normally, this only prints the value of a local variable when a value was assigned to that variable in the last execution step. To obtain a full listing of all local variable values at each execution step, add options `−v` and `−w`.

−m

Changes the semantics of send events. Ordinarily, a send action is unexecutable (blocked) if the target message channel is filled to capacity. When the `−m` option is used, send operations are always executable, but messages sent to a full buffer are lost.

−nN

Sets the seed for the random number generator that is used to guide a random simulation to the integer value `N`. There is no space between the `−n` and the integer `N`. Without this option, SPIN uses the current clock time as a seed. This means that to get a reproducible random simulation run, the use of the `−n` option is required.

−p

Shows at each execution step in the simulation run which process changed state, and what source statement was executed. For instance (using also a few other options discussed here):

```
$ spin -b -B -p -u5 tpc6
0: proc - (:root:) creates proc 0 (:init:)
1: proc 0 (:init:) line 213 "tpc6" (state 11) [i = 0]
2: proc 0 (:init:) line 219 "tpc6" (state 8)  [.(goto)]
3: proc 0 (:init:) line 215 "tpc6" (state 7)  [((i<1))]
4: proc 0 (:init:) creates proc 1 (user)
```

```
4: proc 0 (:init:) line 216 "tpc6" (state 3) [run user(i)]
5: proc 0 (:init:) line 216 "tpc6" (state 4) [i = (i+1)]
```

–qN

In columnated output (i.e., using option `-c`) and with options `-s` and `-r`this suppresses the printing of output for send and receive operations on the channel numbered `N`. To discover which channel numbers correspond to which channels, it can be useful to first perform a straight simulation with just the `-c` option.

–r

Prints all receive statements that are executed, giving the name and number of the receiving process and the corresponding source line number. For each message parameter this shows the message type and the message channel number and name. For instance:[1]

[1] For layout purposes, line and source file references that are normally part of the listings are omitted here.

```
$ spin -b -B -r -u99 tpc6
29:  proc 3 (manager) ... offhook <- q 5 (handler[0])
43:  proc 4 (Session) ... start   <- q 2 (me)
53:  proc 4 (Session) ... offhook <- q 2 (me)
62:  proc 3 (manager) ... number  <- q 5 (handler[0])
69:  proc 4 (Session) ... number  <- q 2 (me)
timeout
91:  proc 3 (manager) ... onhook  <- q 5 (handler[0])
-------------
depth-limit (-u100 steps) reached
```

–s

Prints all send statements that are executed in a format similar to that produced by the `-r` option. For instance:

```
$ spin -b -B -u99 -s tpc6
25:  proc 1 (user)      ... Sent offhook,0   -> q 1 (tpc)
27:  proc 2 (local_tpc) ... offhook -> q 5 (handler[0])
42:  proc 3 (manager)   ... start    -> q 2 (child[0])
43:  proc 3 (manager)   ... offhook  -> q 2 (child[0])
55:  proc 1 (user)      ... Sent number,0    -> q 1 (tpc)
56:  proc 2 (local_tpc) ... number -> q 5 (handler[0])
68:  proc 3 (manager)   ... number   -> q 2 (child[0])
86:  proc 1 (user)      ... Sent onhook,0    -> q 1 (tpc)
87:  proc 2 (local_tpc) ... onhook -> q 5 (handler[0])
88:  proc 1 (user)      ... Sent offhook,0   -> q 1 (tpc)
89:  proc 2 (local_tpc) ... offhook -> q 5 (handler[0])
96:  proc 3 (manager)   ... onhook   -> q 2 (child[0])
-------------
```

```
depth-limit (-u100 steps) reached
```

See also `-c`.

–T

Suppresses the default indentation of the output from print statements. By default, the output always appears indented by an amount that corresponds to the `pid` number of the executing process. With this option the output appears left–adjusted. See also `-b`.

–t[N]

Performs a guided simulation, following an execution trail that was produced by an earlier verification run. If an optional number `N` is attached (with no space between the number and the `-t`), a numbered execution trail is executed instead of the default unnumbered trail. Numbered execution trails can be generated with verification option `-e`.

–uN

Stops the simulation run after `N` steps have been executed. See also `-jN`.

–v

Performs the simulation in verbose mode, adding more detail to the printouts and generating more hints and warnings about dubious constructs that appear in the model.

–w

Enables more verbose versions of the options `-l` and `-g`.

Syntax Checking Options

−a

Normally, when simulation runs are performed, SPIN tries to be forgiving about minor syntax issues in the model specification. Because the PROMELA model is interpreted on−the−fly during simulations, any part of the model that is not executed may escape checking. It is therefore possible that some semantic issues are missed in simulation runs.

When SPIN option `−a` is used, though, a more thorough check of the complete model is performed, as the source text for a model−specific verifier is also generated. This means that, quite apart from the generation of the verifier source files, the `−a` option can be useful as a basic check on the syntactical correctness of a PROMELA model. More verbose information is generated if the `−v` flag is also added, as in:

```
$ spin −a −v model
```

−A

When given this option, SPIN will apply a property−based slicing algorithm to the model which can generate warnings about statements and data objects that are likely to be redundant, or that could be revised to use less memory. The property−based information used for the slicing algorithm is derived from basic assertion statements and PROMELA `never` claims that appear in the model.

−I

This option will cause SPIN to print the body text of each `proctype` specification after all preprocessing and inlining operations have been completed. It is useful to check what the final effect is of parameter substitutions in `inline` calls, and of ordinary macro substitions.

−Z

This option will run only the preprocessor over the model source text, writing the resulting output into a file named `pan.pre`. Good for a very mild syntax check only. The option is there primarily for the benefit of XSPIN.

Postscript Generation

–M

Generates a graphical version of a message sequence chart as a Postscript file. A long chart is automatically split across multiple pages in the output that is produced. This representation is meant to closely resemble the version that is produced by XSPIN. The result is written into a file called `model.ps`, where `model` is the name of the file with the PROMELA source text for the model. The option can be used for random simulation runs, or to reproduce a trail generated by the verifier (in combination with option `–t`). See also option `–c` for a non–Postscript alternative.

Model Checker Generation

The following options determine how verification runs of PROMELA models can be performed. The verifications are always done in several steps. First, optimized and model−specific source code for the verification process is generated. Next, that source code can be compiled in various ways to fine−tune the code to a specific type of verification. Finally, the compiled code is run, again subject to various run−time verification options.

−a

Generates source code that can be compiled and run to perform various types of verification of a PROMELA model. The output is written into a set of C files named `pan.[cbhmt]`, that must be compiled to produce an executable verifier. The specific compilation options for the verification code are discussed in Chapter 19. This option can be combined with options `-J` and `-m`.

−N file

Normally, if a model contains a PROMELA `never` claim, it is simply included as part of the model. If many different types of claims have to be verified for the same model, it can be more convenient to store the claim in a separate file. The `-N` option allows the user to specify a claim file, containing the text of a `never` claim, that the SPIN parser will include as part of the model. The claim is appended to the model, that is, it should not contain definitions or declarations that should be seen by the parser before the model itself is read.

The remaining five options control which optimizations that may be used in the verifier generation process are enabled or disabled. Most optimizations, other than more experimental options, are always enabled. Typically, one would want to disable these optimizations only in rare cases, for example, if an error in the optimization code were to be discovered.

−o1

Disables data−flow optimizations in verifier. The data−flow optimization attempts to identify places in the model where variables become dead, that is, where their value cannot be read before it is rewritten. The value of such variables is then reset to zero. In most cases, this optimization will lower verification complexity, but it is possible to create models where the reverse happens.

−o2

Disables the elimination of write−only variables from the state descriptor. It should never be necessary to use this option, other than to confirm its effect on the length of the state vector and the resulting reduction in the memory requirements.

–o3

Disables the statement merging technique. Statement merging can make it hard to read the output of the `pan -d` output (see Chapter 19), which dumps the internal state assignments used in the verifier. Disabling this option restores the old, more explicit format, where only one statement is executed per transition. Disabling this option, however, also loses the reduction in verification complexity that the statement merging technique is designed to accomplish.

–o4

Enables a more experimental rendezvous optimization technique. This optimization attempts to precompute the feasibility of rendezvous operations, rather than letting the model checker determine this at run time. It is hard to find cases where the use of this option brings convincing savings in the verifi–cation process, so it is likely that this option will quietly disappear in future SPIN versions.

–o5

Disables the case caching technique. Leaving this option enabled allows the verifier generator to make smarter use of case statements in the `pan.m` and `pan.b` files, especially for larger models. This allows for a sometimes considerable speedup in the compilation of the generated verifier.

–S1 and –S2

Separate compilation options. If the size of the verification model is much larger than the size of a `never` claims, and there are very many such claims that need to be verified for a single model, it can be more efficient to compile the verification source text for the model separately from the source text for the claim automata. If the file `model.pml` contains both the main model specification and the `never` claim, in the simplest case the verifier is then generated and compiled in two separate steps

```
$ spin –S1 model.pml # source for model without claim
$ spin –S2 model.pml # source for the claim
```

This generates two sets of sources, with file names `pan_s.[chmbt]` and `pan_t.[chmbt]`, respectively. These sources can be compiled separately and then linked to produce an executable verifier:

```
$ cc –c pan_s.c         # source for model without claim
$ cc –c pan_t.c         # source for the claim
$ cc –o pan pan_s.o pan_t.o             # link both parts
```

Alternatively, on a Windows machine using the Gnu C compiler, the command sequence might look as follows:

```
$ gcc -c pan_s.c        # source for model without claim
$ gcc -c pan_t.c        # source for the claim
$ gcc -o pan.exe pan_s.obj pan_t.obj    # link both parts
```

The idea is that the first part, generating and compiling the source for the main model without the claim, needs to be done only once, independent of the number of different `never` claims that must be verified. The second part, generating and compiling the source for the claim automata, can be repeated for each new claim, but is generally much faster, since the claim automata are typically much smaller than the model to be verified. Chapter 11, p. 261, contains a more detailed discussion of these options.

LTL Conversion

These two options support the conversion of formulae specified in Linear Temporal Logic (LTL) into PROMELA `never` claims. The formula can either be specified on the command line or read from a file.

–f formula

This option reads a formula in LTL syntax from the second argument and translates it into the equivalent PROMELA syntax for a `never` claim. Note that there must be a space between the `-f` argument and the formula. If the formula contains spaces, or starts with the diamond operator <>, it should be quoted to form a single argument and to avoid misinterpretation by the shell. The quoting rules may differ between systems. On UNIX systems either double or single quotes can be used. On many Windows or Linux systems only single quotes work. In case of problems, use the `-F` alternative below.

–F file

This option behaves like option `-f` except that it will read the formula from the given file and not from the command line. The file should contain the formula on the first line. Any text that follows this first line is ignored, which means that it can be used to store arbitrary additional comments or annotation on the formula.

Miscellaneous Options

–d

Produces symbol table information for the PROMELA model. For each PROMELA object, this information includes the type, name, and number of elements (if declared as an array); the initial value (if a data object) or size (if a message channel), the scope (global or local), and whether the object is declared as a variable or as a parameter. For message channels, the data types of the message fields are also listed. For structure variables, the third field in the output defines the name of the structure declaration that contained the variable.

–C

Prints information on the use of channel names in the model. For instance:

```
$ spin -C tpc6
chan rtpc
    never used under this name
chan manager-child[2]
    exported as run parameter by: manager to Session par 3
    sent to by: manager
chan manager-parent
    exported as run parameter by: manager to Session par 4
    received from by: manager
```

In this example, the names `Session`, `manager`, `parent`, and `child` are the names of `proctype`s in the model. Local channel names are identified as the pair of a `proctype` name and a channel name, separated by a hyphen. A channel name is said to be exported if it appears as an actual parameter in a `run` statement. In effect, the channel is then passed from one process to another. The listing gives the number of the parameter in the call to `run` in which the channel name appears.

When combined with option `-g`, the output will also include information on known access patterns to all globally declared channels:

```
$ spin -C -g tpc6
chan handler
    received from by: manager
    sent to by: local_tpc
chan tpc
    received from by: local_tpc
    sent to by: user
chan rtpc
    never used under this name
chan Session-me
    imported as proctype parameter by: Session par 1
    received from by: Session
chan Session-parent
```

```
    imported as proctype parameter by: Session par 1
    sent to by: Session
chan manager-child[2]
    exported as run parameter by: manager to Session par 3
    sent to by: manager

chan manager-parent
    exported as run parameter by: manager to Session par 4
    received from by: manager
```

–Dxxx

Passes the argument `-Dxxx` in its entirety to the preprocessor that is used. This option allows one to leave the value of some symbolic constants undefined in the model, so that they can be defined on the command line. See also option `-E`.

–Exxx

Passes only the string `xxx` as an argument to the preprocessor.

–Pxxx

Replaces the compiled–in name of the preprocessor with `xxx`. By default, the program used here is the standard C preprocessor `cpp`, but with this option it can be replaced with any other, including user–defined programs.

–V

Prints the SPIN version number and exits.

```
$ spin -V
Spin Version 4.0.7 -- 1 August 2003
```

–X and –Y

These two options are reserved for use by the XSPIN interface. Their effect is limited to small changes in the formatting of the output that can be generated by the other options, such as the addition of blank lines to separate the output from different execution steps.