Chapter 11. Using SPIN

> "The difference between theory and practice is a lot bigger in practice than in theory."
>
> —(Peter van der Linden, Expert C Programming, p. 134)

Although SPIN verifications are often performed most conveniently with the help of the graphical interface XSPIN, we will postpone a discussion of that tool for one more chapter and concentrate here on a bare bones use of SPIN itself. It can be very useful to know how to run short verification jobs manually through SPIN's command line interface, especially when trying to troubleshoot unexpected results.

The number of industrial applications of SPIN is steadily increasing. The core intended use of the tool, though, has always been to support both research and teaching of formal verification. One clear advantage of this primary focus is that the sources to SPIN remain freely available, also for commercial use.[1] The large number of users of the tool mean that any flaws that occasionally slip into the software are typically reported and fixed much more rapidly than would be possible for a commercial tool. A possible disadvantage is that the tool continues to evolve, which means that whatever version of the software you happen to be using, there is likely to be a more recent and better version available by the time you figure out how to use it. The basic use of the tool, though, does not change much from version to version, and it is this basic use that we will discuss in this chapter.

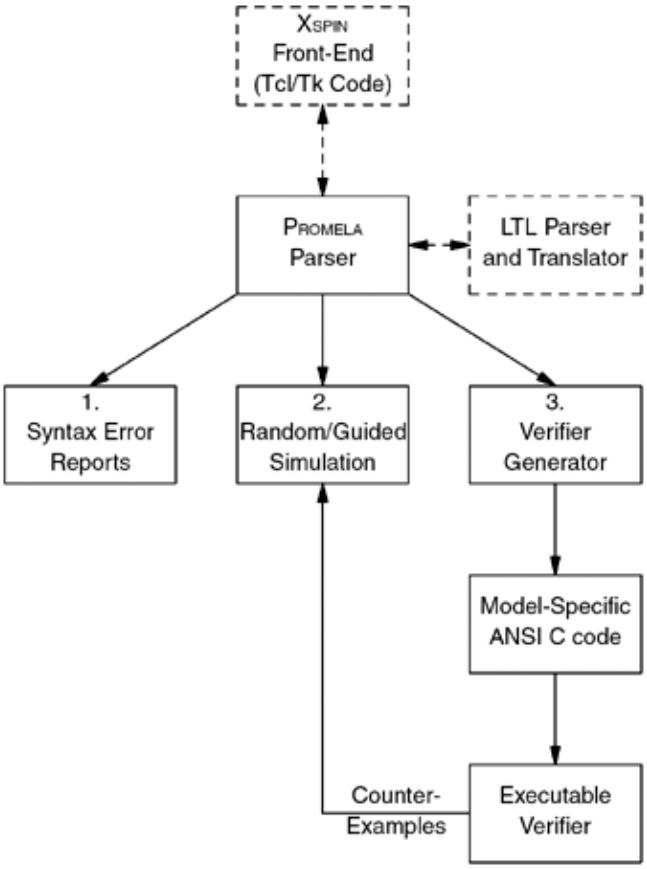
[1] Downloading instructions for SPIN can be found in Appendix D (p. 579).

SPIN Structure

The basic structure of SPIN is illustrated in Figure 11.1. The workflow starts with the specification of a high−level verification model of a concurrent system, or distributed algorithm, optionally using SPIN's graphical front−end XSPIN. After fixing syntax errors, interactive simulation is performed until the user gains the basic confidence that the model has the intended properties. Optionally, a PROMELA correctness claim can be generated from an logic formula specified in linear temporal logic (LTL). Then, in the third main step, SPIN is used to generate an optimized on−the−fly verification program from the high−level specification. This verification program is compiled, with possible compile−time choices for the types of reduction algorithms that are to be used, and it is then executed to perform the verification itself. If any counterexamples to the correctness claims are detected, these can be fed back into the SPIN simulator. The simulation trail can then be inspected in detail to determine the cause of the correctness violation.[2]

[2] When embedded C code is used, the trail can only be reproduced by the verifier itself. See Chapter 17 for details.

Figure 11.1. The Structure of SPIN



In the remainder of this chapter we give some more detailed guideliness of how each of the main steps of specification, simulation, and verification can be performed.

Roadmap

A verification is often performed in an iterative process with increasingly detailed models. Each new model can be verified under different types of assumptions about the environment and for different types of correctness properties. If a property is not valid under a given set of assumptions, SPIN can produce a counterexample that shows explicitly how the property may be violated. The model can then be modified to prevent the property violation.

Once a property has been shown to hold, it is often possible to then reduce the complexity of that model by using the now trusted property as a simplifying assumption. The simpler model may then be used to prove other properties.

A more detailed sequence of steps that the (human) verifier can take in tackling a systems verification problem is as follows. We will assume that an initial verification model, saved in a file called `model`, has been built, including an initial formalization of all relevant correctness properties.

1. The first step is to perform a sanity check of the PROMELA code by executing the command:[3]

   [3] In the command line examples that follow, the # symbol indicates the start of a comment.

   ```
   $ spin -A model       # perform thorough syntax check
   ```

   The output from SPIN will include warnings about syntax errors, possibly dubious constructs that were used, as well as suggestions on possible improvements of the model. A second, more basic check of the model can be performed by attempting to generate the source code for a verifier from the model with the command:

   ```
   $ spin -a model       # generate verifier
   ```

2. Once the first sanity checks are completed, and any flaws that were found have been repaired, a good insight into the behavior that is captured can be obtained with a series of either random or interactive simulation runs. If enough information can be gleaned from just the execution of print statements that are contained within the model, it may suffice to say simply:

   ```
   $ spin model          # non-verbose simulation
   ```

   However, especially for the first few runs, or when it is known that the simulation could continue ad infinitum, it is wise to add a more verbose output option, and to limit the maximum number of steps that may be executed, for instance, by executing the command:

```
$ spin -p -u200 model # more verbose simulation
```

With these parameters every single step that is executed will produce at least some visible output. Also, the simulation will reliably stop after at most 200 steps were made, protecting against a runaway execution.

If the model contains message passing operations, the best information is often obtained from the following type of run:

```
$ spin -c -u200 model # bounded simulation
```

To obtain better insight into seemingly obscure behavior, it is often very useful to add some extra print statements or assertions to the model. User−defined print statements can provide additional information about the evolving state of the model that may not be available from predefined simulation options.

Unfortunately, if the model contains embedded C code, a direct simulation of the model with SPIN as suggested above cannot reveal too much information about an execution. In that case we will have to gain insight into the model's execution from the verifier, as outlined in the next few steps.

3. Once all small modeling errors have been fixed, a more thorough verification process can begin by generating, compiling, and executing the model−specific verifier. In its most basic mode, this is done as follows:

```
$ spin -a model       # generate verifier
$ cc -o pan pan.c     # compile verifier
$ ./pan               # perform verification
```

The verifier can be generated and compiled in several different ways, depending on the type of verification that is desired, as more fully explained later in this chapter, and also in Chapter 18 (p. 513).

4. If a counterexample to a correctness property is generated by the verifier, it can be explored in detail with SPIN's guided simulation options. For models without embedded C code, this is done by executing, for instance:

```
$ spin -t -p model    # replay error trail
```

All options that are available for random and interactive simulations are again available to determine the precise cause of the property violation that was discovered. For instance, we can skip the first 200 steps in a long simulation trail by executing:

```
$ spin -t -p -j200 model # skip first 200 steps
```

or we can revert to the mode where message passing details are shown:

```
$ spin -t -c model # simulation with io details
```

If the model contains embedded C code, a guided simulation run with SPIN will only be able to print but not execute the embedded code fragments. In this case we can execute the error trail with an additional option in the verifier itself, for instance, as follows:

```
$ ./pan -C    # replay a trail with embedded C code
```

These options are more fully explained in Chapter 17 (p. 495). By analyzing the counterexample, we can conclude that either the model or the correctness properties were at fault, and the appropriate remedy can be taken.

5. There is a range of options available in the SPIN verifiers to control the complexity of a large verification run, or to perform various types of approximate verification runs. The details can be found later in this chapter and also in Chapters 9, 18, and 19 (pgs. 191, 513, and 527, respectively). If, nonetheless, a verification cannot be completed exhaustively, or if an approximate verification run cannot be completed with sufficient coverage, it is time to reconsider the PROMELA model itself and remove any potential causes of computational complexity that can be identified. This approach is discussed in more detail in Chapter 5 (p. 101).

The main steps in the verification process sketched above are reviewed in a little more detail in the remainder of this chapter. As a general hint, if you are ever unsure which options SPIN supports, typing

```
$ spin --    # list available command-line options
```

will produce a list, with a brief explanation of each option. The command

```
$ spin -V    # print version number and exit
```

prints the number and date of the specific version of SPIN that you are using. The same principle holds for the verifiers generated by SPIN. Typing

```
$ ./pan --   # list available run-time options
```

lists all available run−time options, and

```
$ ./pan −V   # print version number and exit
```

prints the number of the version of SPIN that was used to generate the verifier, and lists the precise set of compile−time directives that was used to compile the `pan` executable.

Simulation

We will now take a closer look at the main options for performing random, interactive, and guided simulations with SPIN.

Random Simulation

Given a model in PROMELA, say, stored in a file called `model`, the easiest mode of operation is to perform a random simulation. For instance,

```
$ spin -p model
```

tells SPIN to perform a random simulation while printing the process moves selected for execution at each step. If invoked without any options, SPIN prints no output other than what is explicitly produced by the model itself with print statements. Sometimes, simulation runs can go on indefinitely, and the output can quickly become overwhelming. One simple way of controlling the flow of output would be to pipe the output into a UNIX paging tool, for instance

```
$ spin -p model | more
```

This does not work on all PCs though.[4] We can restrict the output to the first `N` steps by using the `-uN` option. Similarly, it is also possible to skip over an initial sequence of `N` steps with the `-jN` option. The two options can be used in combination, for instance, to see only the detailed output for one hundred execution steps starting at the 100th statement execution; one would say:

[4] A recommended way to make this work on a PC is to install the (free) `cygwin` toolset from www.cygwin.com, which approximates a very usable UNIX environment on Windows PCs.

```
$ spin -p -j100 -u200 model     # print steps 100 to 200
```

This type of simulation is random, which means that every new simulation run may produce a different type of execution. By default, the current time is used to seed the random number generator that SPIN uses in random simulation mode. To fix a user-defined seed value instead, and to make the simulation run completely reproducible, we can say, for instance

```
$ spin -p -j100 -u200 -n123 model     # fix seed
```

which initializes the random number generator with a user-defined seed of 123 in this case.

A range of options exists to make the results of a simulation more verbose, for example, by adding printouts of local variables (add option `-l`), global variables (option `-g`), send statements (option `-s`), or receive statements (option `-r`).

Options can be combined in arbitrary order, as in:

```
$ spin -p -l -g -r -s -n1 -j10 -u20 model
```

which can look quite baffling at first, but quickly starts to make sense.

A quick inspection of the available options with the command

```
$ spin --
```

usually suffices to select the right parameters for a simulation run of this type.

Interactive Simulation

It is not always desirable to have SPIN automatically resolve all non−deterministic choices in the model with calls on a random number generator. For these cases, there is an interactive simulation mode of SPIN that is selected through command−line option `−i`. For instance, when we type:

```
$ spin −i −p model
```

a menu with choices is offered each time that the execution can proceed in more than one way. For instance, for the leader election model from the standard SPIN distribution, we might see:[5]

> [5] We have deleted line numbers and source file references from the output for layout purposes.

```
$ spin −i −p leader
0: proc − (:root:) creates proc 0 (:init:)
1: proc 0 (:init:) ... (state 10) [proc = 1]
2: proc 0 (:init:) ... (state 8) [.(goto)]
Select stmnt (proc 0 (:init:) )
    choice 1: ((proc<=5))
Select [0−2]: 1
3: proc 0  (:init:) ... (state 7) [((proc<=5))]
4: proc 0  (:init:) creates proc 1 (node)
4: proc 0  (:init:) ... (state 3) [(run node(...)]
5: proc 0  (:init:) ... (state 4) [proc = (proc+1)]
6: proc 0  (:init:) ... (state 8) [.(goto)]
Select stmnt (proc 0 (:init:) )
    choice 0: other process
    choice 1: ((proc<=5))
Select [0−2]: 0
Selet a statement
    choice 1: proc 1 (node) ... (state 1) [printf(...)]
    choice 2: proc 0 (:init:) ... (state 7) [((proc<=5))]
Select [1−3]: q
$
```

Everything typed by the user after SPIN starts executing in response to a `Select` request from the tool is indicated in bold. The user is asked to make a choice from one or more non−deterministic alternatives for execution by typing a number within the range that is indicated by the `Select` request. In most cases, if there is only one choice, SPIN will immediately select that option without asking the user for guidance, but this is not always the case, as illustrated by the first query that the tool issues in the preceding example. The simulation can be stopped at any point by typing the letter `q` (for 'quit') at a selection menu.

If initial steps in the execution are skipped with a `−jN` option, then SPIN will resolve the non−determinism for those steps internally with the random number generator, and yield control to the user at the desired point. Again, it is wise to fix a seed to SPIN's random number generator in this case to make sure that the intial part of the simulation run proceeds in a reproducible way. An upper limit, specified with option `−uN`, will stop the

simulation after `N` steps have been executed.

Simulations, of course, are intended primarily for the debugging of a model. Only basic assertions are checked in this mode, but even if none of these assertions are violated in a large battery of random simulation runs, we cannot conclude that such violations are impossible. To do so requires verification.

Guided Simulation

SPIN can also be run in guided simulation mode. To do so, though, requires the existence of a specially encoded trail file to guide the search. These trail files are generated only in verification mode, when the verifier discovers a correctness violation. The execution sequence leading up to the error is stored in the trail file, allowing SPIN to replay the scenario in a guided simulation, with access to all user−defined options that were discussed earlier for random simulation. We will return to this option on page 258.

Verification

Perhaps the most important feature of SPIN is that it can generate optimized verifiers from a user−defined PROMELA model. SPIN does not attempt to verify properties of a model directly, with any generic built−in code. By generating a verifier that can be compiled and run separately a significant gain in performance can be realized.

Generating a Verifier

When done debugging, we can use SPIN option -a to produce the source code for a model specific verifier, for instance, as follows:

```
$ spin -a model
```

There are actually two different semantic models that may be used to generate the verifier at this point. The alternative semantic model is obtained with the command:

```
$ spin -a -m model
```

By default, send operations are considered to be unexecutable when the channel to which the message is sent is full. With option -m, this semantic changes into one where send operations are always executable, but messages sent to full channels are lost. The standard semantics of PROMELA correspond to the default model, where option -m is not used.

The verifier is generated as a C program that is stored in a number of files with a fixed set of names, all starting with the three-letter prefix[6] pan. For instance, we may see this result:

> [6] The prefix pan is short for protocol analyzer, and a reference to SPIN's earliest predecessor from 1980.

```
$ spin -a leader
$ ls -l pan.?
-rw-r--r--   1 gerard user 2633 Aug 18 12:33 pan.b
-rw-r--r--   1 gerard user 147964 Aug 18 12:33 pan.c
-rw-r--r--   1 gerard user 9865 Aug 18 12:33 pan.h
-rw-r--r--   1 gerard user 13280 Aug 18 12:33 pan.m
-rw-r--r--   1 gerard user 18851 Aug 18 12:33 pan.t
$
```

The file named pan.h is a generic header file for the verifier that contains, for instance, the translated declarations of all global variables, all channels, and all process types. File pan.m defines the executability rules for all PROMELA statements used in the model, and the effect they have on the system state when successfully executed. File pan.b defines how the effect of each statement from pan.m can be undone when the direction of the depth-first search is reversed. File pan.t contains the transition matrix that encodes the labeled transition system for each process type. Finally, file pan.c contains the algorithms for the computation of the asynchronous and synchronous products of the labeled transition systems, and the state space maintenance and cycle detection algorithms, encoding optimized versions of either a depth-first or a breadth-first search.

Compiling the Verifier

The best performance of the SPIN−generated verifiers can be obtained if the physical limitations of the computer system that will be used to run the verifications are known. If it is known, for instance, how much physical (not virtual) memory the system has available, the verifier can take advantage of that. Initially, the verifier can simply be compiled for a straight exhaustive verification, which can also deliver the strongest possible verification result, provided that there is sufficient memory to complete the run. Compile as follows:

```
$ cc −o pan pan.c # compile for exhaustive search
```

The `pan.c` file includes all other files that are generated by SPIN, so the name of only this file needs to be provided to the compiler. If this compilation attempt fails, make sure that you have an ANSI compatible C compiler. Almost all C compilers today conform to this standard. In case of doubt, though, the generally available Gnu C compilers have the right properties. They are often available as `gcc`, rather than `cc`. The result of the compilation on UNIX systems will be an executable file called `pan`.

On a Windows PC system with the Microsoft Visual C++ compiler installed, the compilation would be done as follows:

```
$ cl pan.c
```

which if all is well also produces an executable verifier named `pan.exe`.

If a memory bound is known at the time of compilation, it should be compiled into the verifier so that any paging behavior can be avoided. If, for instance, the system is known to have no more than 512 Megabytes of physical RAM memory, the compiler−directive to add would be:

```
$ cc −DMEMLIM=512 −o pan pan.c
```

If the verifier runs out of memory before completing its task, the bound could be increased to see if this brings relief, but a better strategy is to try some of SPIN's memory compression options. For instance, a good first attempt could be to compile with the memory collapse option, which retains all the benefits of an exhaustive verification, but uses less memory:

```
$ cc −DCOLLAPSE −o pan pan.c # collapse compression
```

If the verifier still runs out of memory before it can complete the search, a good strategy is to attempt the hash−compact option, which uses less memory, at a small risk of incompleteness of the search:

```
$ cc -DHC4 -o pan pan.c # hash-compact strategy
```

If that also fails, the recommended strategy is to use a series of bitstate verifi−cation runs to get a better impression of the complexity of the problem that is being tackled. Although the bitstate verification mode cannot guarantee exhaustive coverage, it is often very successful in identifying correctness violations.

```
$ cc -DBITSTATE -o pan pan.c # bitstate compression
```

Whichever type of compilation was selected, an executable version of the verifier should be created in a file called either `pan` (on UNIX systems) or `pan.exe` (on PCs), and we can proceed to the actual verification step.

Tuning a Verification Run

A few decisions can be made at this point that can improve the performance of the verifier. It is, for instance, useful, though not strictly required, if we can provide the verifier with an estimate of the likely number of reachable states, and the maximum number of unique steps that could be performed along any single non−cyclic execution path (defining the maximum depth of the execution tree). We will explain in the next few sections how those estimates can be provided. If no estimates are available, the verifier will use default settings that will be adequate in most cases. The feedback from the verifier after a first trial run usually provides enough clues to pick better values for these two parameters, if the defaults do not work well.

Next, we must choose whether we want the verifier to search for violations of safety properties (assertion violations, deadlocks, etc.) or for liveness properties (e.g., to show the absence of non−progress cycles or acceptance cycles). The two types of searches cannot be combined.

A search for safety properties is the default. This default is changed into a search for acceptance cycles if run−time option `-a` is used. To perform a search for non−progress cycles, we have to compile the `pan.c` source with the compile−time directive `-DNP`, and use run−time option `-l`, instead of `-a`. We will return to some of these choices on page 257.

The Number of Reachable States

The verifier stores all reachable states in a lookup table. In exhaustive search mode, that table is a conventional hash table, with a default size of 2^{18} slots. This state storage method works optimally if the table has at least as many slots as there are reachable states that will have to be stored in it, although nothing disastrous will happen if there are less or more states than slots in the lookup table. Strictly speaking, if the table has too many slots, the verifier wastes memory. If the table has too few slots, the verifier wastes CPU cycles. In neither case is the correctness of the verification process itself in peril.

The built–in default for the size of the hash table can be changed with run–time option `–wN`. For instance,

```
$ ./pan –w23
```

changes the size of the lookup table in exhaustive search mode from 2^{18} to 2^{23} slots.

The hash table lookup idea works basically the same when the verifier is compiled for bitstate verification, instead of for the default exhaustive search. For a bitstate run, the size of the hash table in effect equals the number of bits in the entire memory arena that is available to the verifier. If the verifier is compiled for bitstate verification, the default size of the hash array is 2^{22} bits, that is, 2^{19} bytes. We can override the built–in default by specifying, for instance,

```
$ ./pan –w28
```

to use a hash array of 2^{28} bits or 2^{25} bytes. The optimal value to be used depends primarily on the amount of physical memory that is available to run the verification. For instance, use `–w23` if you expect 8 million reachable states and have access to at least 1 Megabyte of memory (2^{20} bytes). A bitstate run with too small of a setting for the hash array will get less coverage than possible, but it will also run faster. Sometimes increased speed is desired, and sometimes greater coverage.

One way to exploit the greater speed obtained with the small hash arrays is, for instance, to apply an iterative refinement method. If at least 64 Megabytes of physical memory are available, such an iterative search method could be performed as follows, assuming a UNIX system running the standard Bourne shell:

```
$ spin –a model
$ cc –DBITSTATE –DMEMLIM=80 –o pan pan.c
$ for i in 20 21 22 23 24 25 26 27 28 29
  do
        ./pan –w$i
        if [ –f model.trail ]
        then
                exit
        fi
  done
```

```
$
```

The search starts with a hash array of just 2^{20} bits (128 Kbytes), which should not take more than a fraction of a second on most systems. If an error is found, the search stops at this point. If no error is found, the hash array doubles in size and the search is repeated. This continues until either an error is found or the maximal amount of memory has been used for the hash array. In this case, that would be with a hash array of 2^{29} bits (64 Megabytes). The verifier source is compiled with a limit of 80 Megabytes in this case, to allow some room for other data structures in the verifier, so that also the last step can be run to completion.

Search Depth

By default, the verifiers generated by SPIN have a search depth restriction of 10,000 steps. If this isn't enough, the search will truncate at 9,999 steps (watch for this telltale number in the printout at the end of a run). A different search depth of `N` steps can be defined by using run−time option `−mN`, for instance, by typing

```
$ ./pan −m1000000
```

to increase the maximum search depth to 1,000,000 steps. A deeper search depth requires more memory for the search; memory that cannot be used to store reachable states, so it is best not to overestimate here. If this limit is also exceeded, it is probably good to take some time to consider if the model defines finite behavior. Check, for instance, if attempts are made to create an unbounded number of processes, or to increment integer variables without bound. If the model is finite, increase the search depth at least as far as is required to avoid truncation of the search.

In the rare case that there is not enough memory to allocate a search stack for very deep searches, an alternative is to use SPIN's stack−cycling algorithm that arranges for the verifier to swap parts of the search stack to disk during a verification run, retaining only a small portion in memory. Such a search can be set up and executed, for instance, as follows:

```
$ spin −a model
$ cc −DSC −o pan pan.c # use stack-cycling
$ ./pan −m100000
```

In this case, the value specified with the `−m` option defines the size of the search stack that will reside in memory. There is no preset maximum search depth in this mode: the search can go arbitrarily deep, or at least it will proceed until also the diskspace that is available to store the temporary stack files is exhausted.

If a particularly nasty error is found that takes a relatively large number of steps to hit, you can try to find a shorter error trail by forcing a shorter depth−limit with the `−m` parameter. If the error disappears with a lower depth−limit, increase it in steps until it reappears.

Another, and often more reliable, way to find the shortest possible error sequence is to compile and run the verifier for iterative depth adjustment. For instance, if we already know that there exists an error sequence of 1,000 steps, we can try to find a shorter equivalent, as follows:

```
$ spin −a model
$ cc −DREACH −o pan pan.c
$ ./pan −i −m1000    # iteratively find shortest error
```

Be warned, though, that the use of `-DREACH` can cause an increase in run time and does not work for bitstate searches, that is, it cannot be combined with `-DBITSTATE`.

Finally, if the property of interest is a safety property (i.e., it does not require a search for cyclic executions), we can consider compiling the verifier for a breadth–first, instead of the standard depth–first, search:

```
$ cc -DBFS -o pan pan.c
$ ./pan
```

This type of search tends to be a little slower than the default search mode, and it can consume more memory, but if these limitations do not prevent it, it is guaranteed to find the shortest path to an error. Combinations with state compression methods are again possible here. Reasonable attempts to control excessive memory use can, for instance, be to compile the verifier with the hash–compact option, using the additional compiler directive `-DHC4`.

Cycle Detection

The most important decision to be made in setting up a verification run is to decide if we want to perform a check for safety or for liveness properties. There are optimized algorithms in the verifier for both types of verification, but only one type of search can be performed at a time. The three main types of search, with the corresponding compilation modes, are as follows:

```
$ spin -a model
$ cc -DSAFETY -o pan pan.c # compilation for safety
$ ./pan                    # find safety violations
$ cc -o pan pan.c          # default compilation
$ ./pan -a                 # find acceptance cycles
$ cc -DNP -o pan pan.c     # non-progress cycle detection
$ ./pan -l                 # find non-progress cycles
```

By default, that is in the absence of option `-l` and `-a`, only safety properties are checked: assertion violations, absence of unreachable code, absence of race conditions, etc. The use of the directive `-DSAFETY` is optional when a search for safety properties is performed. But, when the directive is used, the search for safety violations can be performed somewhat more efficiently.

If `accept` labels are present in the model, for instance, as part of a `never` claim, then a complete verification will require the use of the `-a` option. Typically, when a `never` claim is generated from an LTL formula, it will contain `accept` labels.

Adding run–time option `-f` restricts a search for liveness properties further by enforcing a weak fairness constraint:

```
pan -f -l   # search for fair non-progress cycles
pan -f -a   # search for fair acceptance cycles
```

With this constraint, a non–progress cycle or an acceptance cycle is only reported if every running process either executes an infinite number of steps or is blocked at an unexecutable statement at least once in the execution cycle. Adding the fairness constraint multiplies the time requirements of a verification by a factor that is linear in the number of running processes.

By default, the verifier will always report every statement that is found to be unreachable in the verification model. This reachability report can be suppressed with run–time option `-n`, as, for instance, in:

```
$ ./pan -n -f -a
```

The order in which the options such as these are listed is always irrelevant.

Inspecting Error Traces

If the verification run reports an error, SPIN dumps an error trail into a file named `model.trail`, where `model` is the name of the PROMELA specification. To inspect the trail, and to determine the cause of the error, SPIN's guided simulation option can be used (assuming that the model does not contain embedded C code fragments, cf. p. 495). The basic use is the command

```
$ spin -t -p model
```

with as many extra or different options as are needed to pin down the error. For instance,

```
$ spin -t -r -s -l -g model
```

The verifier normally stops when a first violation has been found. If the first violation is not particularly interesting, run–time option `-cN` can be used to identify others. For instance,

```
$ ./pan -c3
```

ignores the first two violations and reports only the third one, assuming of course that at least three errors can be found.

To eliminate entire classes of errors, two special purpose options may be useful. A search with

```
$ ./pan -A
```

will ignore all violations of basic assertion statements in the model, and a search with

```
$ ./pan -E
```

will ignore all invalid end–state errors. For example, to search only acceptance cycles, the search could be initiated as:

```
$ ./pan -a -A -E
```

To merely count the number of all violations, without generating error trails, use

```
$ ./pan -c0
```

To do the same while also generating an error trail for each violation found, use

```
$ ./pan -c0 -e
```

The error trails now carry a sequence number as part of the file names. To replay a specific numbered trail, say, the `N`th copy, provide the sequence number in the `-t` parameter, for instance,

```
$ spin -t3 model
```

performs a guided simulation for the third error trail found, using the file `model3.trail`.

Internal State Numbers

Internally, the verifiers produced by SPIN deal with a formalization of a PROMELA model in terms of finite automata. SPIN therefore assigns state and transition numbers to all control flow points and statements in the model. The automata state numbers are listed in all the relevant outputs to make it unambiguous (source line references unfortunately do not always have that property). To reveal the internal state assignments, run–time option `-d` can be used. For instance,

```
$ ./pan -d
```

prints a table with all internal state and transition assignments used by the verifier for each distinct `proctype` in the model. The output does not clearly show merged transition sequences. To obtain that output it is best to disable the transition merging algorithm that is used in SPIN. To do so, proceed as follows:

```
$ spin -o3 model
$ cc -o pan pan.c
$ ./pan -d
```

To see the unoptimized versions of the internal state assignments, every repetition of the `-d` argument will arrange for an earlier version of the internal state tables to be printed, up to the original version that is exported by the SPIN parser. Try, for instance, the following command for your favorite model:

```
$ ./pan -d -d -d
```

and compare it with the output that is obtained with a single `-d` argument.

Special Cases

We conclude this chapter with a discussion of some special cases that may arise in the verification of PROMELA models with SPIN. In special circumstances, the user may, for instance, want to disable the partial order reduction algorithm. Alternatively, the user may want to spend some extra time to boost the performance of the partial order reduction by adding some additional declarations that SPIN can exploit. Finally, in most serious applications of automated verification tools, the user will sooner or later run into complexity bottlenecks. Although it is not possible to say specifically how complexity can be reduced in each specific case, it is possible to make some general recommendations.

Disabling Partial Order Reduction

Partial order reduction is enabled by default in the SPIN generated verifiers. In special cases, for instance, when the verifier warns the user that constructions are used that are not compatible with the partial order reduction strategy, the reduction method can be disabled by compiling the verifier's source code with an extra directive:

```
$ spin -a model
$ cc -DNOREDUCE -o pan pan.c # disable p.o. reduction
```

Boosting Performance

The performance of the default partial order reduction algorithm can also be boosted substantially if the verifier can be provided with some extra information about possible and impossible access patterns of processes to message channels. For this purpose, there are two special types of assertions in PROMELA that allow one to assert that specific channels are used exclusively by specific processes. For example, the channel assertions

```
xr q1;
xs q2;
```

claim that the process that executes them is the only process that will receive messages from channel `q1`, and the only process that will send messages to channel `q2`.

If an exclusive usage assertion turns out to be invalid, the verifier will always be able to detect this and report it as a violation of an implicit correctness requirement.

Note that every type of access to a message channel can introduce dependencies that may affect the exclusive usage assertions. If, for instance, a process uses the `len(qname)` function to check the number of messages stored in a channel named `qname`, this counts as a read access to `qname`, which can invalidate an exclusive access pattern.

There are two special operators that can be used to poll the size of a channel in a way that is always compatible with the reduction strategy:

```
nfull(qname)
```

returns true if channel `qname` is not full, and

```
nempty(qname)
```

returns true if channel `qname` contains at least one message. The SPIN parser will reject attempts to bypass the protection offered by these primitives with expressions like

```
!full(qname),
!empty(qname),
!nfull(qname),or
!nempty(qname).
```

In special cases, the user may want to claim that the particular type of access to message channels that is specified in `xr` and `xs` assertions need not be checked. The checks can then be suppressed by compiling the verifier with the extra directive `-DXUSAFE`, for instance, as in:

```
$ cc -DXUSAFE -o pan pan.c
```

Separate Compilation

Often, a verification model is checked for a range of logic properties, and not just a single property. If properties are specified in LTL, or with the Timeline Editor, we can build a library of properties, each of which must be checked against the model. The easiest way to do this is to first generate all property automata from the formulae, or from the visual time line specifications, and store each one in a separately named file. Next we can set up a verification script that invokes SPIN on the basic model, but for each run picking up a different property automaton file, for instance with SPIN's run−time option `−N`.

If the main verification model is stored in a file called `model.pml` and the property automata are all stored in file names with the three−letter extension `.prp`, we can build a minimal verification script, using the UNIX Bourne shell, for instance, as follows:

```
#!/bin/sh

for i in *.prp
do
        echo "property: $i"
        if spin -N $i -a model.pml
        then    ;
        else    echo "parsing error"
                exit 1
        fi
        if cc -o pan pan.c
        then    ;
        else    echo "compilation error"
                exit 1
        fi
        ./pan -a
od
exit 0
```

In most cases, the time that is required to parse the model, to generate the verifier source text and to compile the verifier, is small compared to the time that is required to run the actual verification. But, this is not always the case.

As the model text becomes larger, the time that is needed to compile the verifier source text will also increase. If the compilation process starts to take a noticeable amount of time, and there is a substantial library of properties that need to be checked, we may want to optimize this process.

We can assume that if the compilation time starts to become noticeable, this is typically do to the large size of the basic verification model itself, not the size of the property. In SPIN model checking it would be very rare for a property automaton to exceed a size of perhaps ten to twenty control states. A system model, though, can easily produce an automaton description that spans many thousands of process states. Compiling the automata descriptions for the main verification model, then, can sometimes require significantly more time than compiling the source code that is associated with the implementation of a property automaton.

SPIN supports a method to generate the source code for the model and for the property separately, so that these two separate parts of the source code can also be compiled separately. The idea is that we only need to

generate and compile the source code for the main system model once (the slow part), and we can repeat the generation and compilation of the much smaller source code fragments for the property automata separately.

If we revise the verification script from our first example to exploit this separate compilation option, it would look like this:

```
#!/bin/sh

if spin -S1 model.pml # model, without properties
then    ;
else    echo "parsing error in main model"
        exit 1
fi

if cc -c pan_s.c # compile main part once
then    ;
else    echo "compilation error in main model"
fi

for i in *.prp
do
        echo "property: $i"
        if spin -N $i -S2 model.pml
        then    ;
        else    echo "parsing error in property"
                exit 1
        fi
        # next, compile only the code for the
        # property and link it to the previously
        # compiled module
        if cc -c pan_t.c                # property code
        then    ;
        else    echo "compilation error in property"
                exit 1
        fi
        if cc -o pan pan_s.o pan_t.o     # link
        then    ;
        else    echo "link error"
                exit 1
        fi
        ./pan -a
od
exit 0
```

To get an idea of how much time the separate compilation strategy can save us, assume that we have a library of one hundred properties. If the compilation of the complete model code takes seventy–two seconds, and compilation of just the property related code takes seven seconds, then the first verification script would take

```
100 * 72 = 7,200 seconds = 2 hours
```

The second verification script, using separate compilation, would take:

```
72 + 100 * 7 = 772 seconds = 11 minutes, 12 seconds
```

The time to run each verification would be the same in both scenarios.

In some cases, when the property automata refer to data that is external to the module that contains the property related source code, it can be necessary to add some code into the source file. This can be done via the addition at compile−time of so−called provisioning information, as follows:

```
$ cc -DPROV=\"extra.c\" -c pan_t.c
$ cc -o pan pan_s.o pan_t.o
```

The provisioning information (such as declarations for external variables) is provided in a separate file that is prepared by the user. It can contain declarations for external variables, and also initialization for selected global variables.

Lowering Verification Complexity

If none of SPIN's built−in features for managing the complexity of a verification run seem to be adequate, consider the following suggestions to lower the inherent complexity of the verification model itself:

- Make the model more general; more abstract. Remove everything from the model that is not directly related to the correctness property that you are trying to prove. Remove all redundant computations and redundant data. Use the output from SPIN option `−A` as a starting point.
- Avoid using variables with large value ranges, such as integer counters, clocks, or sequence numbers.
- Try to split channels that receive messages from multiple senders into separate channels, one for each source of messages. Similarly, try to split channels that are read by multiple processes into separate channels, one for each receiver. The interleaving of independent message streams in a single channel can be a huge source of avoidable complexity.
- Reduce the number of slots in asynchronous channels to as small a number as seems reasonable. See Chapter 5, p. 101, on the effect that channel sizes can have on search complexity.
- Group all local computations into `atomic` sequences, and wherever possible into `d_step` sequences.
- Avoid leaving scratch data around in local or global variables. The number of reachable system states can often be reduced by resetting local variables that are used only inside `atomic` or `d_step` sequences to zero at the end of those sequences.

  There is a special keyword in the language that can be used to hide a scratch variable from the verifier completely. It is mentioned only in passing here, since the mechanism is easily misused. Nonetheless, if you declare a global variable, of arbitrary type, as in:

```
hidden byte var;
```

  then the variable, named `var` here, is not considered part of the state descriptor. Clearly, values that are stored in `hidden` variables cannot be assumed to persist. A typical use could be to flush the contents of a channel, for instance, as follows:

```
do
:: nempty(q) -> q?var
:: else -> break
od
```

  If the variable `var` were not hidden, each new value stored in it would cause the creation of a new global state. In this case this could needlessly increase the size of the reachable state space. Use with caution. An alternative method is to use the predefined hidden variable named _ (underscore). This write−only variable need not be declared and is always available to store scratch values. The last example can therefore also be written as:

```
do
:: q?_
:: empty(q) -> break
od
```

- Try to avoid the use of global variables. SPIN's partial order reduction technique can take advantage of the fact that a local variable can only be accessed by a single process.
- Where possible, add the channel assertions `xr` and `xs` (see p. 261 in this chapter).
- Always use the predefined functions `nempty(q)` and `nfull(q)` instead of the equivalent expressions `len(q)>0` and `len(q)<MAX`, respectively; the partial order reduction algorithm can take advantage of the special cases where these expressions are needed.
- Where possible, combine the behavior of multiple processes in a single one. The larger the number of asynchronously executing processes, the greater the potential search complexity will be. The principle to be used here is to generalize the behavior that is captured in a verification model. Focus on properly defining the interfaces between processes, rather than the computation performed inside processes.

In any case: Don't give up. A model checker is a powerful tool that can assist us in proving interesting facts of distributed systems. But, in the end, it is still only a tool. Our own powers of abstraction in formulating problems that the model checker can effectively solve will always outstrip the power of the model checker itself. Fortunately, the ability to predict what types of models can be verified most efficiently grows with experience: you will get better at it each time you use the tool.

> "The road to wisdom is plain and simple to express: Err and err and err again, but less and less and less."
>
> —(Piet Hein, 1905–1996)