Chapter 19. Overview of PAN Options

> "The only reasonable way to get a program right is to assume that it will at first contain errors and take steps to discover these and correct them."
>
> —(Christopher Strachey, 1916–1975)

This chapter summarizes all verification options. The options apply to the verification code that is generated with SPIN's run−time option `-a`. Also included is an explanation of the information that is generated by the program at the end of a verification run (unless disabled with PAN run−time option `-n`).

The three main sections of this chapter cover:

- PAN Compile−Time Options:

  Options that are available at the time of compilation of the verifier source code.
- PAN Run−Time Options:

  Options that are available as command−line arguments to the executable PAN code that is generated by the compiler.
- PAN Output Format:

  An explanation of the information that is generated by the PAN verifiers at the end of a run.

The primary reason for the reliance of compile−time options for the automatically generated verifier code is efficiency: using compile−time directives allows for the generation of more efficient executables than if all options were handled through the use of command line arguments.

If the XSPIN user interface is used, most options are selected automatically by XSPIN (based on user preferences), so in that case there is no strict need to be familiar with the information that is presented in this chapter.

PAN Compile–Time Options

There are quite a few compile–time options for the PAN sources. We will divide them into the following groups, depending on their main purpose:

- Basic options
- Options related to partial order reduction
- Options to increase speed
- Options to reduce memory use
- Options for use only when prompted by PAN
- Options for debugging PAN verifiers
- Experimental options

Usage of all compile–time directives is optional. In its most minimal form, the generation, compilation, and execution of the verification code would simply proceed as follows:

```
$ spin –a spec
$ cc –o pan pan.c
$ ./pan
```

The compile–time directive can modify the default behavior of the verifier to achieve specific effects, as explained in more detail shortly. For instance, to enable breadth–first search and bitstate hashing, the compilation command would change into:

```
$ cc –DBFS –DBITSTATE –o pan pan.c
```

Basic Options

–DBFS

Arranges for the verifier to use a breadth–first search algorithm rather than the standard depth–first search. This uses more memory and restricts the type of properties that can be verified to safety properties only, but within these restrictions it is the easiest way to find a short error path. This option can be combined with the various methods for reducting memory use, such as hash–compact, bitstate hashing, collapse compression, and minimized automaton compression.

–DMEMCNT=N

Sets an upper–bound to the amount of memory that can be allocated by the verifier to 2^N bytes. This limit should be set as closely as possible to the amount of physical (not virtual) memory that is available on the machine. Without this limit, the verifier would pass this limit and start using virtual memory, which in this type of search can lead to a serious degradation of performance, and in the worst case (when the amount of virtual memory used exceeds the amount of physical memory used) to thrashing. For example,

```
$ cc –DMEMCNT=29 –o pan pan.c
```

sets the memory limit at 2^{29} = 512 Megabyte. The next step up would bring this to 1 Gigabyte. Somewhat finer control is available with the directive `MEMLIM`.

–DMEMLIM=N

Sets an upper–bound to the amount of memory that can be allocated by the verifier to `N` Megabytes. For example,

```
$ cc –DMEMLIM=600 –o pan pan.c
```

sets the limit at 600 Megabyte.

–DNOCLAIM

If a PROMELA `never` claim is part of the model, the addition of this directive will exclude it from the verification attempt. It is safe to use this directive even if no `never` claim is present. The code that would ordinarily be used for the handling of the claim is disabled, which can also improve performance slightly.

–DNP

Includes the code in the verifier for non–progress cycle detection, which in turn enables run–time option `-l` and simultaneously disables run–time option `-a` for the detection of standard acceptance cycles.

–DON_EXIT=STRING

The name `ON_EXIT` can be used to define an external procedure name that, if defined, will be called immediately after the verifier has printed its final statistics on a verification run and just before the verifier exits. A possible use can be, for instance:

```
$ spin -a spec
$ cc -DON_EXIT=mycall() -o pan pan.c user_defined.c
$ ./pan
```

where the file `user_defined.c` contains the definition of procedure `mycall()`.

–DPROV=file

If the name `PROV` is defined, the verifier will arrange to execute whatever code segment is defined in the file `file` into the verifier source text, at a point just before the search starts. The code segment should be a sequence of statements that will be inserted, via an include directive, at the start of the `run()` procedure. All variable access must be done with the proper local prefixes, using knowledge about the internal data structures used by the verifier, so the proper use of this option will require some knowledge of the internals of the verifier. The option can be used to set provisioning information, for instance, by assigning values to variables declared to be `hidden` that are used as constants in the model.

Options Related to Partial Order Reduction

–DNOREDUCE

Disables the partial order reduction algorithm and arranges for the verifier to perform an exhaustive full state exploration, without reductions. This clearly increases both the time and the memory requirements for the verification process. The partial order reduction method used in SPIN is explained in Chapter 9 (p. 191).

–DXUSAFE

Disables the validity checks on `xr` and `xs` assertions. This improves the performance of the verifier and can be useful in cases where the default check is too strict.

Options Used to Increase Speed

–DNOBOUNDCHECK

Disables the default check on array indices that is meant to intercept out–of–bound array indexing errors. If these types of errors are known to be absent, disabling the check can improve performance.

–DNOFAIR

Disables the code for the weak fairness algorithm, which means that the corresponding run–time option `-f` will disappear. If it is known that the weak fairness option will not be used, adding this directive can improve the performance of the verifier.

–DSAFETY

Optimizes the code for the case where no cycle detection is needed. This option improves performance by disabling run–time options `-l` and `-a`, and removing the corresponding code from the verifier.

Options Used to Decrease Memory Use

–DBITSTATE

Uses the bitstate storage algorithm instead of default exhaustive storage. The bitstate algorithm is explained in Chapter 9 (p. 206).

–DHC

Enables the hash–compact storage method. The state descriptor is replaced with a 64–bit hash value that is stored in a conventional hash table. Variations of the algorithm can be chosen by adding a number from zero to four to the directive: `HC0`, `HC1`, `HC2`, `HC3`, or `HC4` to use 32, 40, 48, 56, or 64 bits, respectively. The default setting with `HC` is equivalent to `HC4`, which uses 64 bits. The hash–compact algorithm is explained in Chapter 9 (p. 212).

–DCOLLAPSE

Compresses the state descriptors using an indexing method, which increases run time but can significantly reduce the memory requirements. The collapse compression algorithm is explained in Chapter 9 (p. 198).

–DMA=N

Enables the minimized automaton storage method to encode state descriptors. Often combines a very significant reduction in memory requirements with a very significant increase in the run–time requirements. The value `N` sets an upper–bound to the size of the state descriptor as stored. This method can often fruitfully be combined with `–DCOLLAPSE` compression.

–DSC

Enables a stack cycling method, which can be useful for verifications that require an unusually large depth–limit. The memory requirements for the stack increase linearly with its maximum depth. The stack cycling method allows only a small fraction of the stack to reside in memory, with the remainder residing on disk. The algorithm swaps unused portions of the search stack to disk and arrange for just a working set to remain in–core. With this method, the run–time flag `–m` determines only the size of the in–core portion of the stack, but does not restrict the stack's maximum size. This option is meant only for those rare cases where the search stack may be millions of steps long, consuming the majority of the memory requirements of a verification.

Options to Use When Prompted by PAN

If the verifier discovers a problem at run time that can be solved by recompiling the verifier with different directives, the program prints a recommendation for the recompilation before it exits. This applies to two directives in particular: `-DNFAIR` and `-DVECTORSZ`.

–DNFAIR=N

Allocates memory for enforcing weak fairness. By default, that is, in the absence of an explicit setting through the use of this directive, the setting used is `N=2`. If this setting is insufficient, the verifier will prompt for recompilation with a higher value. The default setting can be exceeded if there is an unusually large number of active processes. Higher values for `N` imply increased memory requirements for the verification.

–DVECTORSZ=N

The default maximum size for the state vector (i.e., state descriptor) is 1,024 bytes. If this is insufficient, for unusually large models, the verifier will prompt for recompilation with a higher value. For example:

```
$ cc -DVECTORSZ=2048 -o pan pan.c
```

There is no predefined limit for the size of the state vector that can be set in this way. Often, a large state vector can successfully be compressed losslessly by also using the `-DCOLLAPSE` directive.

Options for Debugging PAN Verifiers

–DVERBOSE

Adds elaborate debugging printouts to the run. This is useful mostly for small models, where a detailed dump of the precise actions of the verifier is needed to trace down suspect or erroneous behavior.

–DCHECK

Provides a slightly more frugal version of the `-DVERBOSE` directive.

–DSVDUMP

Enables an additional run–time option `-pN` to the verifier which, if selected, writes a binary dump of all unique state descriptors encountered during a verification run into a file named `sv_dump`. The file is only generated at the end of the verification run, and uses a fixed integer size of `N` bytes per recorded state. State descriptors shorter than `N` bytes are padded with zeros. See also `-DSDUMP`.

–DSDUMP

If used in combination with the directive `-DCHECK` this adds an ASCII dump of all state descriptors encountered in the search to the verbose debugging output that is generated.

Experimental Options

–DBCOMP

If used in combination with the directive `-DBITSTATE`, modifies the code to compute hash functions over not the original but the compressed version of the state descriptor (using the standard masking technique). In some cases this has been observed to improve the coverage of a bitstate run.

–DCOVEST

If used in combination with the directive `-DBITSTATE`, this option compiles in extra code for computing an alternative coverage estimate at the end a run. On some systems, the use of this code also requires linkage of the object code with the math library, for instance, with the compiler flag `-lm`.

The experimental formula that is used to compute the coverage in this mode was derived by Ulrich Stern in 1997. Stern estimated that when a run has stored R states in a hash array of B bits, then the true number of reachable states R' is approximately

$$R' = \frac{\ln(1 - R/B)}{\ln(1 - 1/B)}.$$

When the verifier is compiled with directive `-DCOVEST` it reports the estimated state space coverage as the percentage of states that was reached compared to the estimated total number of reachable states, that is:

$$\frac{R}{R'} \times 100\%$$

–DCTL

Allows only those partial order reductions that are consistent with branching time logics, such as CTL. The rule used here is that each persistent set that is computed contains either all outgoing transitions or precisely one.

–DGLOB_ALPHA

Considers process death to be a globally visible action, which means that the partial order reduction strategy cannot give it priority over other actions. The resulting verification mode restores compatibility with SPIN version numbers from 2.8.5 to 2.9.7.

–DHYBRID_HASH

Using this option can reduce the size of every state descriptor by precisely one word (4 bytes), but this benefit will only be seen in 25% of all cases. In the standard storage method, when the state descriptor is one, two, or three bytes longer than a multiple of four, the memory allocator pads the amount of memory that is effectively allocated with one, two, or three bytes, respectively. This padding is done to secure memory alignment. To avoid this in at least some of the cases, the `HYBRID_HASH` will consider state descriptors that exceed a multiple of four by precisely one byte, and truncate the state vector by that amount. The one byte that is removed is now added to the hash value that is computed. This can cause more hash collisions to occur, but it does preserve a correct search discipline, and it can save memory.

–DLC

If used in combination with the directive `–DBITSTATE`, this option replaces exhaustive storage of states in the depth–first search stack with a four–byte hash–compact representation. This option slows down the verification process, but it can reduce the memory requirements. There is a very small additional risk of hash collisions on stack states which, if it occurs, can affect the effective coverage achieved. This option is automatically enabled when `–DSC` is used in combination with `–DBITSTATE`.

–DNIBIS

Applies a small optimization of partial order reduction technique. The attempt is to avoid repeating the exploration of a successor state in cases where the exploration of the reduced set of transitions fails (e.g., because it closed a cycle). This requires extra testing to be done during the search to see if the optimization applies, which in many cases can more than cancel the benefit of the optimization.

–DNOCOMP

Disables the default masking of bits in the state vector during verifications. This can improve performance, but is not compatible with cycle detection or bitstate storage methods.

–DNOSTUTTER

Disables the rule that allows a `never` claim to perform stuttering steps. This is formally a violation of the semantics for LTL model checking. The stuttering rule is the standard way to extend a finite run into an infinite one, thus allowing for a consistent interpretation of Büchi acceptance conditions.

–DNOVSZ

This option removes four bytes from each state descriptor before it is stored in the state space. The field that is removed records the effective size of the state descriptor. In most cases, this information is indeed redundant, so when memory is tight and the exhaustive state space storage method is used, this option may give relief. The number of states stored that is reported at the end of a run should not change when `–DNOVSZ` is enabled. This option is not compatible with `–DCOLLAPSE`. Generally, the latter option will reduce the memory requirements by a more substantial amount, and in a safer way.

–DOHASH

Replaces the default hash function used in the verifier with an alternative one based on the computation of a cyclic redundancy check. In combination with run–time option `-hN`, a choice of 32 different hash functions can be used. The quality of these alternate function is often less than the built–in default.

–DPEG

Includes and enables code in the verifier for performing complexity profiling. With this option, the number of times that each basic statement is executed will be counted, and the counts are printed at the end of the run as a simple aid in identifying the hot spots in the code with respect to verification.

–DPRINTF

Enables the execution of `printf` statements during verification runs. Useful only for debugging purposes.

–DRANDSTOR=N

If used in combination with `-DBITSTATE`, this will randomly prune the number of states that are actually recorded in the hash array. The probability of storage is determined by the parameter `N`. For example,

```
$ cc -DRANDSTOR=33 -DBITSTATE -o pan pan.c
```

would reduce the probability of storage for each state from 100% to 33%. (Only approximately one out of every three unique states encountered is stored.) The value for `N` must be between 0 and 99. Low values will increase the amount of (duplicate) work that has to be done by the verifier, and thus increases the time requirements of a verification. Low values, however, can also increase the effective coverage of a bitstate verification for very large state spaces. This option can be useful also in sequential bitstate hashing runs to improve the cumulative coverage of all runs combined.

–DREACH

Use of this option changes the search algorithm in such a way that the absence of safety errors can be guaranteed within the run–time depth limit that is set by `-m`. The algorithm used is discussed in Chapter 8 (p. 171). This option cannot guarantee that the shortest path to a liveness error is found.

–DVAR_RANGES

Includes and enables code in the verifier for computing the effective value range of all basic (i.e., not PROMELA `typedef` structure) variables. To keep things manageable, all values over 255 are grouped under a single entry in the report that is generated at the end of the run.

PAN Run–Time Options

The following options can be given as command–line arguments to the compiled version of the verifiers generated by SPIN. They are listed here in alphabetical order.

–A

Suppresses the reporting of basic assertion violations. This is useful if, for instance, the verification process targets a different class of errors, such as non–progress cycles or Büchi acceptance cycles. See also `–E`.

–a

Arranges for the verifier to report the existence, or absence, of Büchi acceptance cycles. This option is disabled when the verifier is compiled with the directive `–DNP`, which replaces the option with `–l`, for non–progress cycle detection.

–b

Selecting this bounded search option makes it an error, triggering an error trail, if an execution can exceed the depth limit that is specified with the `–m` option. Normally, exceeding the search depth limit only generates a warning.

–cN

Stops the search after the `N`th error has been reported. The search normally stops after the first error is reported. Using the setting `–c0` will cause all errors to be reported. See also run–time option `–e`.

–d

Prints the internal state tables that are used for the verification process and stops. For the leader election protocol example from the SPIN distribution, the output looks as follows.[1] One state table is generated for each `proctype` that appears in the SPIN model, with one line per transition.

[1] Not all transitions are shown. Long lines are split into two parts here for layout purposes.

```
$ spin –a leader
$ ./pan –d
proctype node
 state  1 – (tr  8) –> state  3 [id  0  tp  2] [––––L] \
        line 16 => Active  =  1
 state  3 – (tr  9) –> state 30 [id  2  tp  5] [––––L] \
        line 18 =>  out!first,id [(3,2)]
 state 30 – (tr 10) –> state 15 [id  3 tp 504] [––e–L] \
```

```
        line 19 => in?first,number [(2,3)]
 state 30 - (tr 17) -> state 28 [id 16 tp 504] [--e-L] \
        line 19 => in?second,number
 ...
proctype init
 state 10 -(tr 3)->    state  7 [id 33 tp   2] [A---L] \
        line 49 => proc = 1
 state  7 -(tr 4)->    state  3 [id 34 tp   2] [A---L] \
        line 51 => ((proc<=5))
 state  7 -(tr 6)->    state  9 [id 37 tp   2] [A---L] \
        line 51 => ((proc>5))
 state  3 -(tr 5)->    state  7 [id 35 tp   2] [A---L] \
        line 53 => (run node(...))
 state  9 -(tr 1)->    state 11 [id 41 tp   2] [----L] \
        line 51 => break
 state 11 -(tr 7)->    state 0 [id 43 tp 3500] [--e-L] \
        line 58 => -end- [(257,9)]
...

Transition Type: A=atomic; D=d_step; L=local; G=global
Source-State Labels: p=progress; e=end; a=accept;
Note: statement merging was used. Only the first
    stmnt executed in each merge sequence is shown
    (use spin -a -o3 to disable statement merging)
```

The description of each transition specifies a series of numbers and strings. We refer to these with greek symbols, as follows:

```
state α -(tr γ)-> state β [id δ tp ε] [..λ..] \
       line φ => ..ν.. [..η..]
```

The state tables are optimized in three separate steps by the verifier before the verification process begins. The original and intermediate versions of the tables can be generated by using the `-d` argument two, three, or four times in a row (e.g., by typing the command `pan -d -d -d`).

–E

Suppresses the reporting of invalid end–state violations. See also `-A`.

–e

Creates a numbered error trail for all errors that are encountered during the search up to the bound set by the `-cN` argument. By default, only one single error trail is produced per run. The maximum possible number of error trails is therefore generated by the combination:

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

–f

Uses the weak fairness restriction in the search for either acceptance cycles (in combination with option `-a`) or non–progress cycles (option `-l`). The weak fairness algorithm is discussed in Chapter 8 (p. 182).

–hN

If the verifier is compiled with the directive `-DOHASH`, this option replaces the default hash function with an alternative function. The value `N` must be greater than zero and less than 33. The default hash function corresponds to the setting `-h1`.

–i

Enables a fine–grained iterative search method to look for the shortest path to an error. After each error that is found, the verifier will set a new depth limit that is one step smaller than the length of the last error trail. The use of this method can increase complexity. For the method to reliably identify the shortest possible error path, the verifier must be compiled with `-DREACH`. The option is only guaranteed to work for safety properties. Given a safety property, there is also an alternative way to home in on the shortest possible error path, also at increased resource requirements (memory and time). That alternative option is to compile the verifier with option `-DBFS`. In this case, the `-DREACH` option is not needed.

–I

An alternative to run–time option `-i`. Instead of reducing the search depth to one step below the length of the last error trail that was generated, this option reduces it to half that size, in an effort to reduce the number of iterations that has to be made. This will not necessarily find the shortest possible error sequence, but it often gets reasonably close. This option also requires compilation with `-DREACH` for best performance.

–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. This option matches command–line option `-J` to SPIN itself, where it applies to simulation instead of verification runs. To play back an error trail that was generated with the `-J` verification option requires the use of the `-J` option during guided simulation.

–l

Arranges for the verifier to report the existence, or absence, of non–progress cycles. This option is not enabled unless the verifier source code is compiled with `-DNP` and disables the search for acceptance cycles (option `-a`).

–mN

Sets the maximum search depth for a depth–first search verification to `N` steps. The default value for `N`, that is, in the absence of an explicit `-m` argument, is 10,000. See also `-b`.

−n

Suppresses the default listing of all unreached states at the end of a verification run.

−q

Adds an extra restriction on end states. Normally a valid end state is one where each process has reached the end of its code or has stopped at a state that was marked with an end−state label. By default, message channels are not required to be empty for a state to be considered a valid end state. The use of this option adds that requirement.

−s

Changes the bitstate search algorithm to use 1−bit hashing instead of the default 2−bit hashing method. The use of this option requires compilation with `-DBITSTATE`.

−V

Prints the SPIN version number that was used to generate the verifier code and stops.

−wN

The default size of the hash table that is used for exhaustive (i.e., non−bitstate) verifications is 2^{18} = 262, 144 slots. For bitstate verifications, the default size of the hash array is 2^{18} bits, which means 2^{15} = 32,768 bytes. This default corresponds to the setting `-w18`. The default size can be changed, with currently an upper−limit of `-w32`.

−X

Option reserved for use by XSPIN. It causes the UNIX standard error output to be printed onto the standard output stream.

−Y

Causes the end of the output to be marked with a special line that can be recognized by postprocessors such as XSPIN.

PAN Output Format

A typical printout of a verification run is shown in Figure 19.1 This is what each line in this listing means:

Figure 19.1 Example Output Generated by Pan

```
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance cycles       - (not selected)
        invalid end states      +

State-vector 32 byte, depth reached 13, errors: 0
      74 states, stored
      30 states, matched
     104 transitions (= stored+matched)
       1 atomic steps
hash conflicts: 2 (resolved)
(max size 2^18 states)

1.533   memory usage (Mbyte)

unreached in proctype ProcA
        line 7, state 8, "Gaap = 4"
        (1 of 13 states)
unreached in proctype :init:
        line 21, state 14, "Gaap = 3"
        line 21, state 14, "Gaap = 4"
        (1 of 19 states)



(Spin Version 4.0.7 -- 1 August 2003)
```

Identifies the version of SPIN that generated the `pan.c` source from which this verifier was compiled.

```
+ Partial Order Reduction
```

The plus sign means that the default partial order reduction algorithm was used. A minus sign would indicate compilation for exhaustive, non–reduced verification with option `-DNOREDUCE`. If the verifier had been compiled with the breadth–first search option, using compiler directive `-DBFS`, then this fact would have been noted here as well.

```
Full statespace search for:
```

Indicates the type of search. The default is a full state space search. If the verifier is instead compiled with one of the various types of state compression enabled (e.g., collapse compression, bitstate search, or hash−compact storage), this would be noted in this line of output.

The next line in the output reads:

```
never claim            - (none specified)
```

The minus sign indicates that no `never` claim or LTL formula was used for this run. If a `never` claim was part of the model, it could have been suppressed with the compiler directive `-DNOCLAIM`. If a trace assertion is used instead of a `never` claim, this would also be reflected in this line of output.

```
assertion violations   +
```

The plus indicates that the search checked for violations of user−specified assertions, which is the default.

```
acceptance cycles     - (not selected)
```

The minus indicates that the search did not check for the presence of acceptance or non−progress cycles. To do so would require a run−time option −a or compilation with `-DNP` combined with the run−time option `-l`.

```
invalid end states    +
```

The plus indicates that a check for invalid end states was done (i.e., for absence of deadlocks).

```
State-vector 32 byte, depth reached 13, errors: 0
```

The complete description of a single global system state required 32 bytes of memory. The longest depth−first search path contained 13 transitions from the root of the tree (that is, 13 statement executions, starting from the initial system state). Of course, this depth is typically smaller when breadth−first search is used than with the default depth−first search. No errors were found in the search, as reflected in the zero error count.

Normally, the number of errors reported is either zero or one, since by default the search will stop when the first error is found. The verifier can be run in several different modes, though, where it would be allowed to continue the search beyond the first error. The run−time flags `-c` and `-e` can be used for this purpose.

```
74 states, stored
```

This line indicates that a total of 74 unique global system states were stored in the state space (each represented effectively by a state vector of 32 bytes).

```
30 states, matched
```

In 30 cases the search returned to a previously visited state in the search tree.

```
104 transitions (= stored+matched)
```

A total of 104 transitions were explored in the search, which can serve as a statistic for the amount of work that has been performed to complete the verification.

```
1 atomic steps
```

One of the transitions was part of an atomic sequence; all others were outside atomic sequences.

When breadth−first search is used, in addition to these numbers, the verifier also reports a count for the number of nominal states. This number is derived by subtracting all states that would not have to be stored in a depth−first search from the reported number of stored states. This includes all states inside atomic sequences, and all states that record the point of execution in the middle of attempted rendezvous handshakes. Some of those handshake attempts will succeed, and some will fail (if, for instance, the intended recipient of the rendezvous offer is not ready to accept it), so separately, the number of successful rendezvous handshakes will also be reported in this case. These additional numbers are meant to make it easier to compare the performance of breadth−first searches with depth−first searches.

The next line in the output from Figure 19.1 reports:

```
hash conflicts: 2 (resolved)
```

In two cases the default hashing scheme (a weaker version than what is used in bitstate hashing) encountered a collision and had to resolve this collission by placing states into a linked list within the hash table.

```
(max size 2^18 states)
```

The (perhaps default) argument that was specified for the size of the hash table was 2^{18} equivalent to a run−time option `−w18`. If this had been a bitstate search, the size would give the number of bits in the memory arena, rather than the number of slots in the hash table.

```
1.533 memory usage (Mbyte)
```

Total memory usage was 1.533 Megabytes, including the stack, the hashtable, and all related data structures. Choosing smaller values for run−time options `−m` and `−w` than the defaults, would allow memory use to decrease. In this case, with only 74 reachable states of 32 bytes each, this could result in considerable savings.

When the verifier is compiled with directive `−DCOLLAPSE` for memory compression, some extra statistics are printed on how many elements there turned out to be in each of three main groups that are stored and indexed separately. For instance, the report

```
nr of templates: [ globals chans procs ]
collapse counts: [ 3262 18 20 3 ]
```

says that there were 3,262 different versions of the portion of the state descriptor containing global variables values, 18 different versions of the portion containing all channel states, 20 different versions of local process states for processes created from the first `proctype` declaration in the model, and two for processes created from the second `proctype` declaration. If more `proctype` declarations are present in the model, more numbers will follow in this list.

```
unreached in proctype ProcA
        line 7, state 8, "Gaap = 4"
        (1 of 13 states)
unreached in proctype :init:
        line 21, state 14, "Gaap = 3"
        (1 of 19 states)
```

A listing of the state numbers and approximate line numbers for the basic statements in the specification that were not reached. Since this is a full state space search that ran to completion, this means that these transitions are effectively unreachable and constitute dead code in the model.

In bitstate searches the output also contains an estimate for the coverage that was realized in a verification run. This estimate is based on a statistical argument about the likelihood of hash collissions and the number of reachable system states that may have been missed because of this. The coverage estimate is expressed as the so−called hash factor, which measures the relative number of bits in the bitstate space that remained unused. This hash factor is computed as the number of bits that was available for the search in the bitstate space (which can be set with run−time parameter `−w`), divided by the number of states that was reached.

For the default double−bit hashing method, a value for the hash factor over one hundred normally correlates with high confidence with an exhaustive, or nearly exhaustive, coverage. Lower values correlate with lower confidence, with values near one corresponding to near certainty that only a very small fraction of the true

state space was visited in the run. See also the compile−time directive `−DCOVEST`, which can be used to add the computation of an alternative coverage estimate to the search results.