Chapter 9. Search Optimization

"Don't find fault. Find a remedy."

—(Henry Ford, 1863–1947)

The basic algorithms for performing explicit state verification, as implemented in SPIN, are not very complicated. The hard problem in the construction of a verification system is therefore not so much in the implementation of these algorithms, but in finding effective ways to scale them to handle large to very large verification problems. In this chapter we discuss the methods that were implemented in SPIN to address this issue.

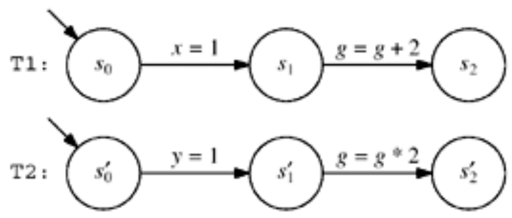
The optimization techniques we will review here have one of two possible aims: to reduce the number of reachable system states that must be searched to verify properties, or to reduce the amount of memory that is needed to store each state.

SPIN's partial order reduction strategy and statement merging technique fall into the first of these two categories. In the second category we find techniques that are based on either lossless or lossy compression methods. The former preserve the capability to perform exhaustive verifications, though often trading reductions in memory use for increases in run time. The lossy compression methods can be more aggressive in saving memory use without incurring run−time penalties, by trading reductions in both memory use and speed for a potential loss of coverage. The bitstate hashing method, for which SPIN is perhaps best known, falls into the latter category. A range of lossless compression methods is also supported in SPIN. We will briefly discuss the principle of operation of the hash−compact method, collapse compression, and the minimized automaton representation. We begin with a discussion of partial order reduction.

Partial Order Reduction

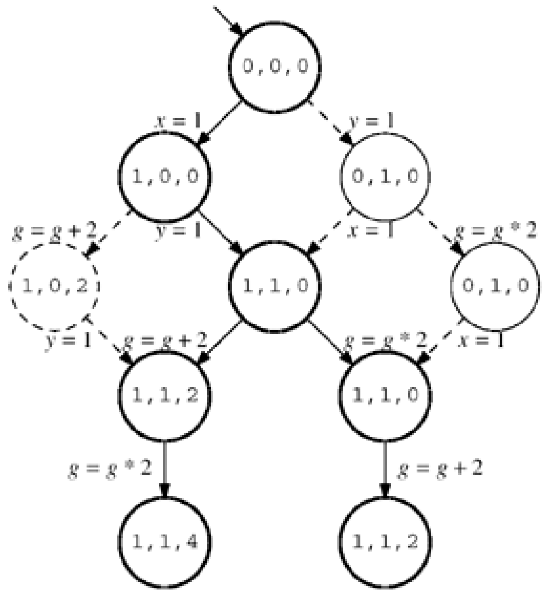
Consider the two finite state automata T1 and T2 shown in Figure 9.1. If we interpret the labels on the transitions, we can see that the execution of each system is meant to have a side effect on three data objects. The automata share access to an integer data object named g, and they each have access to a private data object, named x and y, respectively. Assume that the initial value of all data objects is zero, and the range of possible values is 0...4.

Figure 9.1. The Finite State Automata T1 and T2



The expanded asynchronous product of T1 and T2 (cf. Appendix A) is illustrated in Figure 9.2. We have used the state labels in Figure 9.2 to record the values of the data objects in the order: x, y, g.

Figure 9.2. Expanded Asynchronous Product of T1 and T2



The paths through the graph from Figure 9.2 represent all possible interleavings of the combined execution of the four statements from automata T1 and T2. Clearly, the two possible interleavings of the statements x = 1 and y = 1 both lead to the same result, where both x and y have value 1. The two possible interleavings of the statements g = g + 2 and g = g * 2, on the other hand, lead to two different values for g. The underlying notion of data independence and data dependence can be exploited to define an equivalence relation on runs.

The system is small enough that we can exhaustively write down all finite runs. There are only six:

$\sigma_1 = \{(0, 0, 0), (1, 0, 0), (1, 0, 2), (1, 1, 2), (1, 1, 4)\}$

σ_2 = {(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 2), (1, 1, 4)}

σ_3 = {(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 0), (1, 1, 2)}

σ_4 = {(0, 0, 0), (0, 1, 0), (0, 1, 0), (1, 1, 0), (1, 1, 2)}

σ_5 = {(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 0), (1, 1, 2)}

σ_6 = {(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 2), (1, 1, 4)}

The sequence of statement executions that correspond to these six runs can be written as follows:

1: x = 1; g = g+2; y = 1; g = g*2;

2: x = 1; y = 1; g = g+2; g = g*2;

3: x = 1; y = 1; g = g*2; g = g+2;

4: y = 1; g = g*2; x = 1; g = g+2;

5: y = 1; x = 1; g = g*2; g = g+2;

6: y = 1; x = 1; g = g+2; g = g*2;

The first two runs differ only in the relative order of execution of the two transitions y = 1 and g = g+2, which are independent operations. Similarly, runs σ_4 and σ_5 differ only in the relative order of execution of the independent operations x = 1 and g = g*2, By a process of elimination, we can reduce the number of distinct runs to just two, for instance to:

2: x = 1; y = 1; g = g+2; g = g*2;

3: x = 1; y = 1; g = g*2; g = g+2;

The four other runs can be obtained from these two by the permutation of adjacent independent operations. We have the following mutual dependencies in this set of transitions:

g = g*2 and g = g+2 because they touch the same data object

x = 1 and g = g+2 because they are both part of automaton T1

y = 1 and g = g*2 because they are both part of automaton T2

The following operations are mutually independent:

x = 1 and y = 1

x = 1 and g = g*2

y = 1 and g = g+2

Using this classification of dependent and independent operations, we can partition the runs of the system into two equivalence classes: $\{\sigma_1, \sigma_2, \sigma_6\}$ and $\{\sigma_3, \sigma_4, \sigma_5\}$. Within each class, each run can be obtained from the other runs by one or more permutations of adjacent independent transitions. The eventual outcome of a computation remains unchanged under such permutations. For verification, it therefore would suffice to consider just one run from each equivalence class.

For the system from Figure 9.2 it would suffice, for instance, to consider only runs σ_2 and σ_3. In effect this restriction amounts to a reduction of the graph in Figure 9.2 to the portion that is spanned by the solid arrows, including only the states that are indicated in bold. There are three states fewer in this graph and only half the number of transitions, yet it would suffice to accurately prove LTL formulae such as:

$$\square (g \equiv 0 \lor g > x)$$
$$\Diamond (g \geq 2)$$
$$(g \equiv 0)\ \boldsymbol{U}\ (x \equiv 1)$$

Visibility

Would it be possible to formulate LTL properties for which a verification could return different results for the reduced graph and the full graph? To answer this question, consider the LTL formula

$$\Box (x \geq y).$$

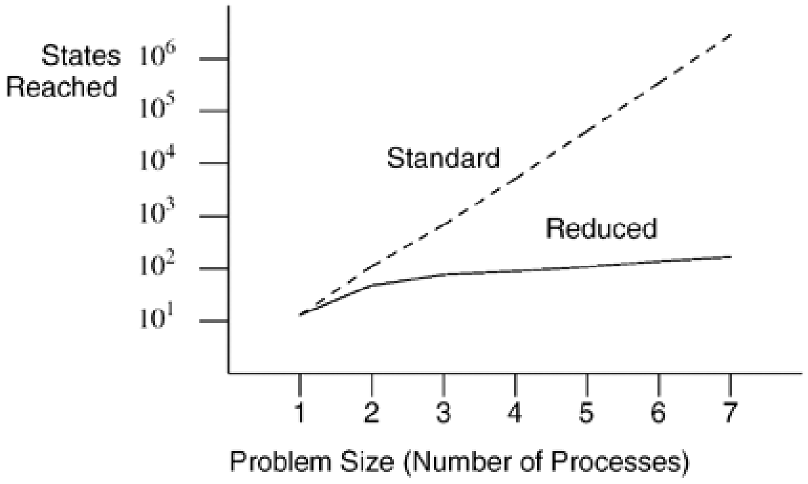
This formula indeed has the unfortunate property that it holds in the reduced graph but can be violated in the full graph.

What happened? The formula secretly introduces a data dependence that was assumed not to exist: it relates the values of the data objects x and y, while we earlier used the assumption that operations on these two data objects were always independent. The dependence of operations, therefore, does not just depend on automata structure and access to data, but also on the logical properties that we are interested in proving about a system. If we remove the pair x = 1 and y = 1 from the set of mutually independent operations, the number of equivalence classes of runs that we can deduce increases to four, and the reduced graph gains one extra state and two extra transitions.

The new graph will now correctly expose the last LTL formula as invalid, in both the full and in the reduced graph.

The potential benefits of partial order reduction are illustrated in Figure 9.3. Shown is the reduction in the number of states in the product graph that needs to be explored to perform model checking when partial order reduction is either enabled (solid line) or disabled (dashed line). In this case, the improvement increases exponentially with the problem size. It is not hard to construct cases where partial order reduction cannot contribute any improvement (e.g., if all operations are dependent). The challenge in implementing this strategy in a model checker is therefore to secure that in the worst case the graph construction will not suffer any noticeable overhead. This was done in the SPIN model checker with a static reduction method. In this case, the dependency relations are computed offline, before a model checking run is initiated, so that no noticeable run−time overhead is incurred.

Figure 9.3. Effect of Partial Order Reduction Increase in Number of States as a Function of Problem Size (Sample of Best Case Performance for Leader Election Protocol)



The partial order reduction strategy is enabled by default for all SPIN verification runs. There are a small number of language constructions that are not compatible with the enforcement of a partial order reduction strategy. They are listed in Chapter 16. In these cases, and for experimental purposes, the partial order reduction strategy can be disabled by compiling the verification code that is generated by SPIN with the compiler directive `-DNOREDUCE`.

Statement Merging

A special case of partial order reduction is a technique that tries to combine sequences of transitions within the same process into a single step, thus avoiding the creation of intermediate system states after each separate transition. The merging operation can be performed, for instance, for sequences of operations that touch only local data. In effect, this technique automatically adds `d_steps` into a specification, wherever this can safely be done.

To see the potential effect of statement merging, consider the following example:

```
#ifdef GLOBAL
        byte c;
#endif

active proctype merging()
{
#ifndef GLOBAL
        byte c;
#endif
        if
        :: c = 0
        :: c = 1
        :: c = 2
        fi;
        do
        :: c < 2 -> c++
        :: c > 0 -> c--
        od
}
```

If we make the declaration for variable `c` global, none of the operations on this variable can be considered safe under the partial order reduction rules, and the statement merging technique cannot be applied.

Note that `proctype merging` has five control states, and variable `c` can take three different values, so there can be no more than fifteen system states.

> There is one control state before, and one after the `if` statement. Then there is also one control state at each of the two arrow symbols. The fifth control control state is the termination state of the process: immediately following the `do` construct.

It turns out that only eight of these fifteen states can be reached, as confirmed by this first run:

```
$ spin -DGLOBAL -a merging.pml
$ cc -o pan pan.c
$ ./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 16 byte, depth reached 6, errors: 0
       8 states, stored
       4 states, matched
      12 transitions (= stored+matched)
       0 atomic steps
...
```

If we now turn `c` from a global into a local variable, all operations on this variable become local to the one process in this system, which means that the SPIN parser can recognize the corresponding transitions as necessarily independent from any other statement execution in the system. The statement merging technique can now combine the two option−sequences inside the `do` loop into a single step each, and thereby removes two of the control states. The result should be a reduction in the number of states that is reached in a verification. If we perform this experiment, we can see this effect confirmed:

```
$ spin -a merging.pml
$ cc -o pan pan.c
$ ./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 16 byte, depth reached 3, errors: 0
       4 states, stored
       4 states, matched
       8 transitions (= stored+matched)
       0 atomic steps
...
```

There is only one downside to the statement merging technique: it can make it harder to understand the automaton structure that is used in the verification process. Statement merging can be disabled with an extra command−line option in SPIN. For instance, if we generate the verifier as follows:

```
$ spin -a -o3 merging.pml
```

statement merging is surpressed, and the system will again create eight system states during a verification.

State Compression

The aim of the partial order reduction strategy is to reduce the number of system states that needs to be visited and stored in the state space to solve the model checking problem. An orthogonal strategy is to reduce the amount of memory that is required to store each system state. This is the domain of memory compression techniques.

SPIN supports options for both lossless and lossy compression: the first type of compression reduces the memory requirements of an exhaustive search by increasing the run−time requirements. The second offers a range of proof approximation techniques that can work with very little memory, but without guarantees of exhaustive coverage.

We first consider lossless state compression. SPIN has two different algorithms of this type. The `COLLAPSE` compression mode exploits a hierarchical indexing method to achieve compression. The `MA`, or minimized automaton, compression mode reduces memory by building and updating a minimized finite state recognizer for state descriptors.

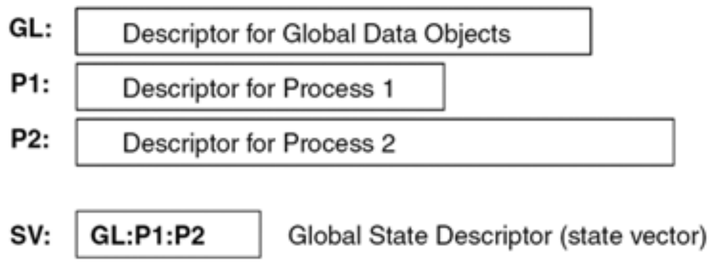
Collapse Compression

At first sight, it may strike us as somewhat curious that the number of distinct system states that the verifier can encounter during a search can become so large so quickly, despite the fact that each process and each data object can typically reach only a small number of distinct states (i.e., values). The explosion in the number of reachable system states is only caused by the relatively large number of ways in which the local states of individual system components, such as processes and data objects, can be combined. Replicating a complete description of all local components of the system state in each global state that is stored is therefore an inherently wasteful technique, although it can be implemented very efficiently.

SPIN's collapse compression mode tries to exploit this observation by storing smaller state components separately while assigning small unique index numbers to each one. The unique index numbers for the smaller components are now combined to form the global state descriptor.

There are now several choices that can be made about how precisely to break down a global system state into separate components, ideally with as little correlation as possible between components. SPIN assigns components as illustrated in Figure 9.4.

Figure 9.4. State Components for COLLAPSE Compression


GL:
Descriptor for Global Data Objects
P1:
Descriptor for Process 1
P2:
Descriptor for Process 2
SV:
GL:P1:P2
Global State Descriptor (state vector)

- A first component is formed by the set of all global data objects in the model, including the contents of all message channels, irrespective of whether they were declared locally or globally. This component also includes a length field that records the original length of the state vector (i.e., the state descriptor) before compression.
- Next, there is one component for each active process, recording its control state together with the state of all its local variables, but excluding the contents of locally declared channels.

Because the number of component states cannot be known in advance, the method should be able to adjust the number of bits it uses to record index values. The SPIN implementation does this by adding an extra two bits to each component index to record how many bytes are used for the corresponding index field. In this way, the compressor can use up to four bytes per index, which suffices for up to 2^{32} possible separate component states. The table of index widths is added to the global variables component. The width of the index for the global variables component itself is stored in a fixed separate byte of the compressed state descriptor.

To make sure no false partial matches can occur, the length of each separately stored component is also always stored with the component data.

The collapse compression method is invoked by compiling the verifier source text that is generated by SPIN with the extra compile–time directive `-DCOLLAPSE`. For instance:

```
$ spin -a model
$ cc -DCOLLAPSE -o pan pan.c
$ ./pan
...
```

There should be no change in the results delivered by the verifier, other than that run time may increase while the memory requirements decrease.

To see the effect of the COLLAPSE algorithm, consider the example PROMELA model of the leader election protocol that is part of the standard SPIN distribution. We will set the number of processes to seven (changing it from the default value in the distributed version of five) and will disable the partial order reduction method to increase the state space size to a more interesting value. We proceed as follows:

```
$ spin -a leader.pml
$ cc -DNOREDUCE -DMEMLIM=200 pan.c
$ time ./pan
(Spin Version 4.0.7 -- 1 August 2003)

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

State-vector 276 byte, depth reached 148, errors: 0
  723053 states, stored
3.00211e+006 states, matched
3.72517e+006 transitions (= stored+matched)
      16 atomic steps
hash conflicts: 2.70635e+006 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
205.347 equivalent memory usage for states (...)
174.346 actual memory usage for states (compression: 84.90%)
        State-vector as stored = 233 byte + 8 byte overhead
1.049   memory used for hash table (-w18)
0.240   memory used for DFS stack (-m10000)
175.266 total actual memory usage

unreached in proctype node
        line 53, state 28, "out!two,nr"
        (1 of 49 states)
unreached in proctype :init:
        (0 of 11 states)

real 0m16.657s
user 0m0.015s
sys  0m0.015s
```

Running on a 2.5 GHz PC, the search took 16.7 seconds, and it consumed 175.2 Mbytes of memory. A statistic is also printed for the "equivalent memory usage," which is obtained by multiplying the number of stored states with the size of each state descriptor, plus the overhead of the lookup table. The default search does a little better than this by always using a simple byte masking technique that omits some redundant information from the state descriptors before they are stored (e.g, padded bytes that are inserted to secure proper alignment of components inside the state descriptor).

Next, we recompile the model checker with `COLLAPSE` compression enabled and repeat the search.

```
$ cc -DMEMLIM=200 -DNOREDUCE -DCOLLAPSE pan.c
$ time ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Compression

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

State-vector 276 byte, depth reached 148, errors: 0
  723053 states, stored
3.00211e+006 states, matched
3.72517e+006 transitions (= stored+matched)
      16 atomic steps
hash conflicts:
3.23779e+006 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
208.239 equivalent memory usage for states (...)
23.547  actual memory usage for states (compression: 11.31%)
        State-vector as stored = 21 byte + 12 byte overhead
1.049   memory used for hash table (-w18)
0.240   memory used for DFS stack (-m10000)
24.738  total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 2765 129 2 ]
unreached in proctype node
        line 53, state 28, "out!two,nr"
        (1 of 49 states)
unreached in proctype :init:
        (0 of 11 states)

real 0m20.104s
user 0m0.015s
sys  0m0.015s
```

As expected, the same number of states was reached, but this time the search took 20.1 seconds (about 20% more than the first run) and the memory usage dropped to 24.7 Mbytes (a decrease of about 85%). The use of `COLLAPSE` is well rewarded in this case.

At the end of the run, a few additional statistics are printed to give some impression of how many components of each basic type were seen. The components are called templates in this list, and the maximum number of entries made in the lookup table for each basic type (global variables, message channels, and processes) is given here as 2,765, 129, and 2.

Although the effect of the reduction looks impressive, if we repeat this search with the standard partial order reduction strategy enabled, the state space size reduces to just 133 states in this case. As a result, the run time drops to a fraction of a second, and the memory requirements to about 1.5 Mbytes. Clearly, state compression is not an alternative to partial order reduction, but can be combined with it very fruitfully.

Minimized Automaton Representation

A second lossless compression method that is supported in SPIN optionally stores the set of reachable system states not in a conventional lookup table, but instead performs state matching by building and maintaining a minimal deterministic finite state automaton that acts as a recognizer for sets of state descriptors. The automaton, represented as a finite graph, is interrogated for every system state encountered during the search, and updated immediately if a new state descriptor is seen. The savings in memory use with this method can be very large, sometimes allowing the verifier to use exponentially smaller amounts of memory than required for the standard search methods. The run–time penalty, though, can be very significant.

Figure 9.5 shows the minimized automaton structure for a state descriptor of three bits, after the first three state descriptors have been stored. All paths in the automaton that lead from node s_0 to the accepting state s_6 are part of the state space, all paths from node s_0 to the non–accepting terminal state s_3 are not part of the state space. The dashed lines separate the subsequent layers in the automaton. The edges between the node in the first (left–most) layer and the second layer represent the possible values of the first bit in the state descriptor, those between the second and the third layer represent possible values of the second bit, and so on.

Figure 9.5. Minimized Automaton Structure After Storing {000, 001, 101}

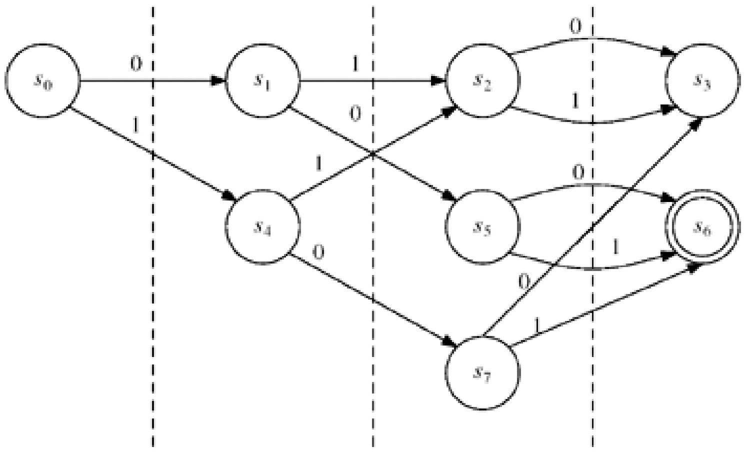
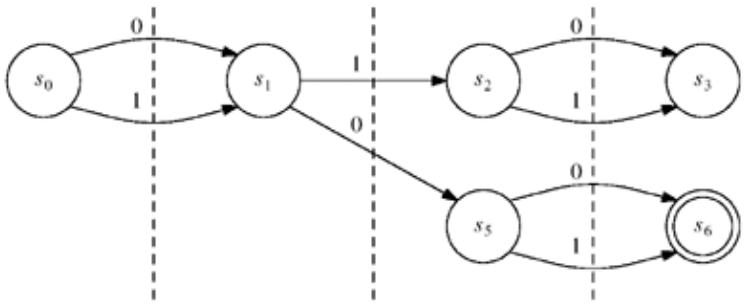


Figure 9.6 shows how this structure is updated when one more state descriptor is added, again restoring the minimized form. There is a close resemblance between this minimized automaton structure and OBDDs (ordered binary decision diagrams). In our implementation of this storage method, each node in the graph does not represent a single bit but a byte of information, and each node therefore can have up to 255 outgoing edges. Edges are merged into ranges wherever possible to speed up the lookup and update procedures.

Figure 9.6. Automaton Structure After Storing {000, 001, 101, 100}



In principle, the minimized automaton has the same expected complexity as the standard search based on the storage of system states in a hashed lookup table. In both cases, an update of the state space has expected complexity O(S), with S the maximum length of the state descriptor for a system state. The constant factors in both procedures are very different, though, which means that the minimized automaton procedure can consume considerably more time than the other optimization algorithms that are implemented in SPIN. Nonetheless, when memory is at a premium, it can well be worth the extra wait to use the more aggressive reduction technique.

To enable the minimized automaton procedure, the user should provide an initial estimate of the maximal depth of the graph that is constructed for the minimized automaton representation. The estimate is not hard to obtain: the size of the state vector that the verifier prints at the end of a normal run can serve as the initial estimate. The number can be used as an initial value for the compile–time directive that enables the minimized automaton procedure. For instance, if we take the last run for the leader election protocol as a starting point. The state vector size reported there was 276 bytes. The size needed for the minimized automaton structure is typically a little smaller, so we make an initial guess of 270 bytes and recompile and run the verifier as follows:

```
$ cc -DMEMLIM=200 -DNOREDUCE -DMA=270 pan.c
$ time ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Graph Encoding (-DMA=270)

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

State-vector 276 byte, depth reached 148, errors: 0
MA stats: -DMA=234 is sufficient
Minimized Automaton: 161769 nodes and 397920 edges
  723053 states, stored
3.00211e+006 states, matched
3.72517e+006 transitions (= stored+matched)
      16 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
202.455 equivalent memory usage for states (...)
7.235   actual memory usage for states (compression: 3.57%)
0.200   memory used for DFS stack (-m10000)
7.338   total actual memory usage
```

```
unreached in proctype node
        line 53, state 28, "out!two,nr"
        (1 of 49 states)
unreached in proctype :init:
        (0 of 11 states)

real 1m11.428s
user 0m0.015s
sys  0m0.015s
```

Again, the same number of states was reached, but note that the memory requirements dropped to just 7.338 Mbytes, compared to 175 Mbytes for the default search without compression, giving an average of just 10 bytes used per state stored. This impressive reduction comes at a price, though. While the memory requirements were reduced, the run−time requirements increased from 16.7 seconds to about 71.4 seconds: a very noticeable penalty.

At the end of the run, the verifier also tells us that instead of our estimate of 270 bytes, a smaller value of 234 bytes would have sufficed. Using that value could reduce the memory and run−time requirements somewhat. In this case, the run time would reduce to 133 seconds and the memory requirements to 7.301 Mbytes, giving a small additional benefit.

It is safe to combine the minimized automaton compression method with the `COLLAPSE` compression method to achieve additional reductions. If we do this for the leader election protocol (while still suppressing the partial order reduction algorithm to create a large state space), we obtain the following result:

```
$ cc -DMEMLIM=200 -DNOREDUCE -DMA=21 -DCOLLAPSE pan.c
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Compression
        + Graph Encoding (-DMA=21)

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

State-vector 276 byte, depth reached 148, errors: 0
Minimized Automaton:      5499 nodes and 25262 edges
  723053 states, stored
3.00211e+006 states, matched
3.72517e+006 transitions (= stored+matched)
      16 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
208.239 equivalent memory usage for states (...)
0.892   actual memory usage for states (compression: 0.43%)
1.049   memory used for hash table (-w18)
0.200   memory used for DFS stack (-m10000)
2.068   total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 2765 129 2 ]
```

```
unreached in proctype node
        line 53, state 28, "out!two,nr"
        (1 of 49 states)
unreached in proctype :init:
        (0 of 11 states)

real 0m44.214s
user 0m0.015s
sys  0m0.015s
```

After one iteration, we could determine that the value for compiler directive `MA` that suffices now reduces to just 21, and with this value the memory requirements drop to a remarkable 2.068 Mbyte, while the search stores the same number of reachable states as before. The run−time requirements are now also slightly less, reducing to 44.2 seconds, thanks to the smaller, collapsed state descriptors that are now handled by the minimized automaton recognizer. This favorable effect is not always observed, but an experiment like this is often worth trying.

Note carefully that even though this last search consumed only 2.7 bytes for each state stored, the search was still completely exhaustive, and the result of the verification is 100% accurate. This is due to the fact that both the `COLLAPSE` and the `MA` compression methods are lossless. If we give up the requirement of guaranteed exhaustive coverage, a number of other interesting search techniques become possible. Some of these techniques can succeed in analyzing very large problem sizes with very minimal run−time requirements. We will consider two of the techniques that are implemented in SPIN: bitstate hashing and hash−compact.

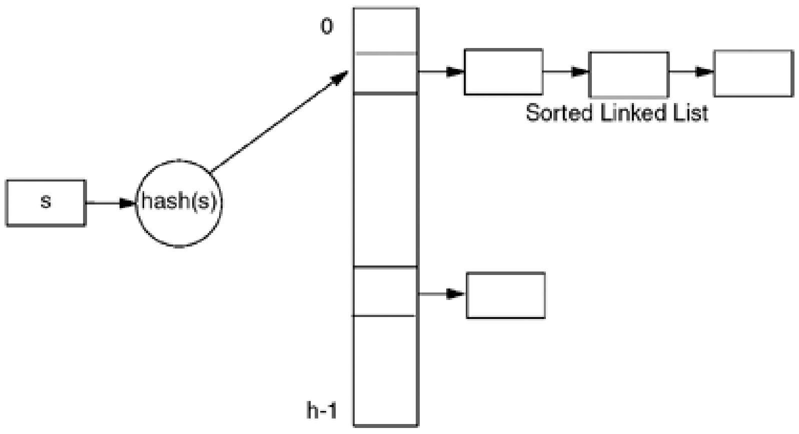
> "An approximate answer to the right question is worth a great deal more than a precise answer to the wrong question."
>
> —(John Tukey, 1915–2000)

Bitstate Hashing

The standard depth–first search algorithm constructs a set of states. Each state that is explored in the verification process is stored in a state space. Since the model checking problem for all practical purposes is reduced to the solution of a reachability problem (cf. Chapter 8), all the model checker does is construct states and check whether they were previously visited or new. The performance of a model checker is determined by how fast it can do this.

The state space structure serves to prevent the re–exploration of previously visited states during the search: it turns what would otherwise be an exponential algorithm into a linear one, that visits every reachable state in the graph at most once. To enable fast lookup of states, the states are normally stored in a hash table, as illustrated in Figure 9.7.

Figure 9.7. Standard Hash Table Lookup



Assume we have a hash table with h slots. Each slot contains a list of zero or more states. To determine in which list we store a new state s, we compute a hash–value hash(s), unique to s and randomly chosen in the range 0..h–1. We check the states stored in the list in hash table slot hash(s) for a possible match with s. If a match is found, the state was previously visited and need not be explored again. If no match is found, state s is added to the list, and the search continues.

> Each state is represented in memory as a sequence of S bits. A simple (but very slow) hashing method would be to consider the array of bits as one large unsigned integer, and to calculate the remainder of its division by h, with h a prime number. A more efficient method, and one of the methods implemented in SPIN, is to use a checksum polynomial to compute the hash values. We now choose h as a power of 2 and use the polynomial to compute a checksum of log(h) bits. This checksum is then used as the hash value.
>
> The default hashing method that is currently implemented in SPIN is based on a method known as Jenkins' hash. It is slightly slower than the checksum polynomial method, but it can be shown to give notably better coverage.

Let r be the number of states stored in the hash table and h the number of slots in that table. When h >> r, each state can be stored in a different slot, provided that the hash function is of sufficiently good quality. The lists stored in each slot of the hash table will either be empty or contain one single state. State storage has only a constant overhead in this case, carrying virtually no time penalty.

When $h < r$, there will be cases for which the hash function computes the same hash value for different states. These hash collisions are resolved by placing all states that hash to the same value in a linked list at the corresponding slot in the hash table. In this case we may have to do multiple state comparisons for each new state that is checked against the hash table: towards the end of the search on average r/h comparisons will be required per state. The overhead incurred increases linearly with growing r/h, once the number of stored states r exceeds h.

Clearly, we would like to be in the situation where $h >> r$. In this case, a hash value uniquely identifies a state, with low probability of collision. The only information that is contained in the hash table is now primarily whether or not the state that corresponds to the hash value has been visited. This is one single bit of information. A rash proposal is now to indeed store only this one bit of information, instead of the S bits of the state itself. This leads to the following trade−offs.

Assume that we have m bits of memory to store the hash table, S bits of data in each state descriptor, r reachable states, and a hash table with h slots. Clearly, fewer than m/S states will fit in memory, since the hash table itself will also take some memory. If $r > m/S$, the search will exhaust the available resources (and stop) after exploring a fraction of $m/(r \cdot S)$ of the state space. Typical values for these parameters are: $m = 10^9$, $S = 10^3$, and $r = 10^7$, which gives a ratio $m/(r \cdot S) = 10^{-2}$, or a coverage of the problem size of only 1%.

If we configure the hash table as an array of 8m bits, using it as a hash table with $h = 8m$ 1−bit slots, we now have $h >> r$, since $8 \cdot 10^9 >> 10^7$, which should give us an expected coverage close to 100%. When, with low probability, a hash collision happens, our model checking algorithm will conclude incorrectly that a state was visited before, and it will skip it. It may now miss other states that can only be reached via a path in the reachability graph that passes through this state. This, therefore, would lead to loss of coverage, but it cannot lead to false error reports. We will see shortly that in almost all cases where this method is used (i.e., when normal state storage is impossible due to limited resources available), coverage increases far more due to the increased capacity to store states than it is reduced due to hash collisions.

This storage discipline was referred to in Morris [1968] as follows:

> "A curious possible use of virtual scatter tables arises when a hash address can be computed with more than about three times as many bits as are actually needed for a calculated address. The possibility that two different keys have the same virtual hash address becomes so remote that the keys might not need to be examined at all. If a new key has the same virtual hash address as an existing entry, then the keys could be assumed to be the same. Then, of course, there is no longer any need to keep the keys in the entry; unless they are needed for some other purpose, they can just be thrown away. Typically, years could go by without encountering two keys in the same program with the same virtual address. Of course, one would have to be quite certain that the hash addresses were uniformly spread over the available addresses.
>
> No one, to the author's knowledge, has ever implemented this idea, and if anyone has, he might well not admit it."

To reduce the probability of collision, we can use multiple independent hash functions, and set more than one bit per state. Using more bits can increase the precision but reduce the number of available slots in the bit hash−table. The trade−offs are delicate and deserve a more careful study.

Bloom Filters

Let m again be the size of the hash table in bits, r is the number of states stored, and k the number of hash functions used. That is, we store k bits for each state stored, with each of the k bit−positions computed with an independent hash function that uses the S bits of the state descriptor as the key.

Initially, all bits in the hash table are zero. When r states have been stored, the probability that any one specific bit is still zero is:

$$\left(1-\frac{1}{m}\right)^{k}\cdot r$$

The probability of a hash collision on the (r + 1)st state entered is then

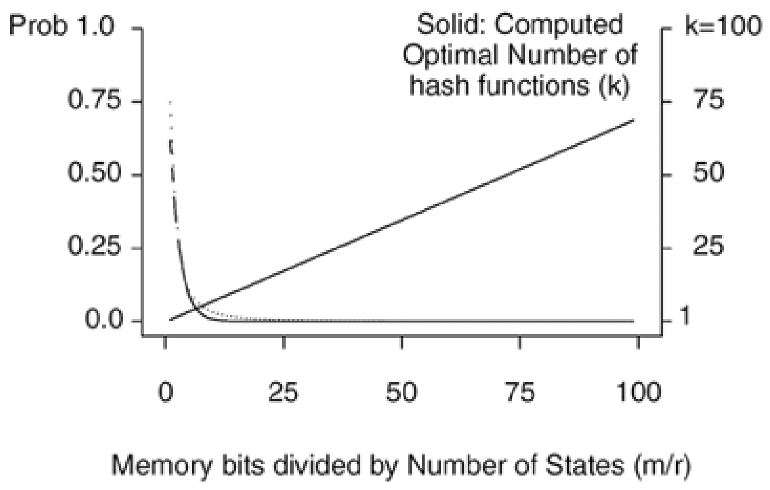
$$\left(1-\left(1-\frac{1}{m}\right)^{k}\cdot r\right)^{k}\approx\left(1-e^{-k\cdot r/m}\right)^{k}$$

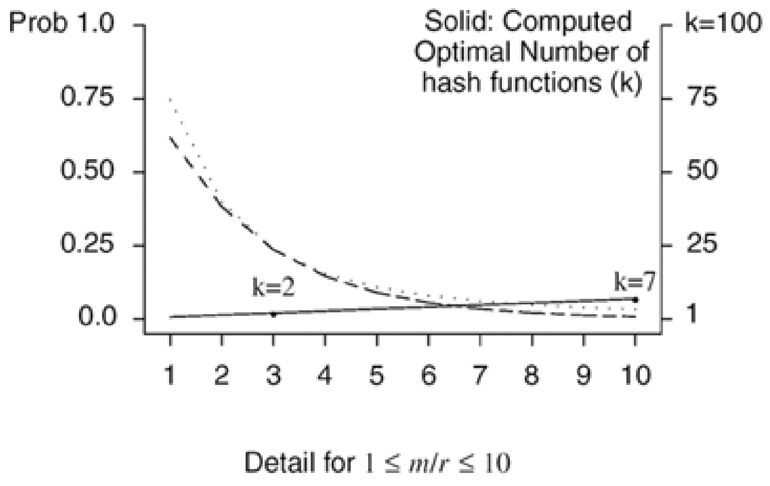
which gives us an upper−bound for the probability of hash collisions on the first r states entered. (The probability of a hash collision is trivially zero for the first state entered.) The probability of hash collisions is minimal when k = log(2) · m/r, which gives

$$\left(\frac{1}{2}\right)^{k}=0.6185^{m/r}$$

For $m = 10^9$ and $r = 10^7$ this gives us an upper−bound on the probability of collision in the order 10^{-21}, for a value of k = 89.315. Figure 9.8 illustrates these dependencies.

Figure 9.8. Optimal Number of Hash Functions and Probability of Hash Collision The dashed line plots the probabilitie for optimal k The dotted line plots the probabilities for fixed k=2 The solid line plots the optimal value for k, 1 $\leq$ k < 100





In practice, k must be an integer (e.g., 90). In a well−tuned model checker, the run−time requirements of the search depend linearly on k: computing hash values is the single most expensive operation that the model checker must perform. The larger the value of k, therefore, the longer the search for errors will take. In the model checker SPIN, for instance, a run with k = 90 would take approximately 45 times longer than a run with k = 2. Although time is a more flexible commodity than memory, the difference is significant. The question is then how much quality we sacrifice if we select a smaller than optimal value of k. The trade−off is illustrated in Figure 9.8.

For the suboptimal value k = 2, the value used in SPIN, the upper−bound on the collision probability becomes $4 \cdot 10^{-4}$, which reduces the expected coverage of the search from 100% to near 99%, still two orders of magnitude greater than realized by a hash table lookup method for this case. We can also see in Figure 9.8 that the hashing method starts getting very reliable for m/r ratios over 100. To be compatible with traditional storage methods, this means that for state descriptors of less than 100 bits (about 12 bytes), this method is not competitive. In practice, state descriptors exceed this lower−bound by a significant margin (one or two orders of magnitude).

The bitstate hashing method is invoked by compiling the verifier source with the additional compiler directive `−DBITSTATE`, for instance, as follows:

```
$ spin -a model
$ cc -DBITSTATE -o pan pan.c
$ ./pan
...
```

A straight bitstate run for the leader election protocol example, for instance, produces this result:

```
$ spin -a leader.pml
$ cc -DNOREDUCE -DMEMLIM -DBITSTATE -o pan pan.c
$ time ./pan
(Spin Version 4.0.7 -- 1 August 2003)

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

State-vector 276 byte, depth reached 148, errors: 0
  700457 states, stored
2.9073e+006 states, matched
3.60775e+006 transitions (= stored+matched)
      16 atomic steps
hash factor: 5.98795 (best coverage if >100)
(max size 2^22 states)

Stats on memory usage (in Megabytes):
198.930 equivalent memory usage for states (...)
0.524   memory used for hash array (-w22)
2.097   memory used for bit stack
0.240   memory used for DFS stack (-m10000)
3.066   total actual memory usage

unreached in proctype node
        line 53, state 28, "out!two,nr"
        (1 of 49 states)
unreached in proctype :init:
        (0 of 11 states)

real 0m28.550s
user 0m0.015s
sys  0m0.015s
```

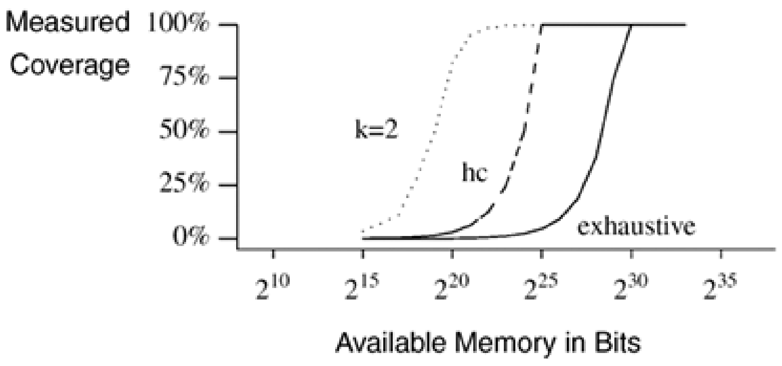
The number of states explored is a little short of the real number of reachable states that we can measure in an exhaustive run. Still, the run reached 96.7% of all reachable states, while the memory requirements dropped from 175 to 3 Mbytes, and the run−time requirements remained relatively low at 28.5 seconds.

Hash−Compact

An interesting variant of this strategy is the hash−compact method, first proposed for use in verification by Pierre Wolper. In this case we try to increase the size of m far beyond what would be available on an average machine, for instance to 2^{64} bits. We now compute a single hash value within the range 0..(2^{64}−1) as a 64−bit number, and store this number, instead of the full state s, in a regular hash table. We have one hash function, so k = 1, and we simulate a memory size of $m = 2^{64} \asymp 10^{19}$ bits. For the value of $r = 10^7$, we then get a probability of collision near 10^{-57}, giving an expected coverage of 100%. To store 107 64−bit numbers takes less than $m = 10^9$ bits. Instead of storing 64 bits at a time, we can also store a smaller or larger number of bits. The maximum number of bits that could be accommodated is trivially m/r. The 64−bit version of hash−compact, then, should be expected to perform best when $r. 64 \leq m \leq r.S$. Unfortunately, although m and S are often known a priori, in most cases r is usually not known before an exhaustive verification is completed, and therefore the optimal ration m/r is also typically unknown.

A measurement of the performance of the hash−compact method and double−bit hashing (i.e., with two independent hash functions) for a fixed problem size r and available memory m varying from 0 to m > r.S is shown in Figure 9.9, which is taken from Holzmann [1998].

Figure 9.9. Measured Coverage of Double Bitstate Hashing (k=2) Compared with Hash−Compact (hc), and Exhaustive Search Problem size: 427567 reachable states, state descriptor 1376 bits



When sufficient memory is available, traditional exhaustive state storage is preferred, since it gives full coverage with certainty. For the problem shown in Figure 9.9 this is the area of the graph with $m > 2^{29}$. Barring this, if sufficient memory is available for the hash−compact method, then this is the preferred method. This is the area of the graph where $2^{23} < m < 2^{29}$. Below that, in Figure 9.9 for all values $m < 2^{23}$, the double−bit hashing method is superior. The latter method, for instance, still achieves a problem coverage here of 50% when only 0.1% of the memory resources required for an traditional exhaustive search are available.

The hash−compact method can be enabled by compiling a SPIN−generated verifier with the compiler directive `HC4`, for instance as follows (see also Chapter 19, p. 530):

```
$ spin −a model
$ cc −DHC4 −o pan pan.c
$ ./pan
...
```

Applying the hash−compact to the leader election protocol from before, using four bytes per state, produces this result:

```
$ cc -DNOREDUCE -DMEMLIM=200 -DHC4 -o pan pan.c
$ time ./pan
(Spin Version 4.0.7 -- 1 August 2003)

Hash-Compact 4 search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance cycles       - (not selected)
        invalid end states      +

State-vector 276 byte, depth reached 148, errors: 0
  723053 states, stored
3.00211e+006 states, matched
3.72517e+006 transitions (= stored+matched)
      16 atomic steps
hash conflicts: 2.41742e+006 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
205.347 equivalent memory usage for states (...)
11.770  actual memory usage for states (compression: 5.73%)
        State-vector as stored = 8 byte + 8 byte overhead
1.049   memory used for hash table (-w18)
0.240   memory used for DFS stack (-m10000)

12.962  total actual memory usage

unreached in proctype node
        line 53, state 28, "out!two,nr"
        (1 of 49 states)
unreached in proctype :init:
        (0 of 11 states)

real 0m15.522s
user 0m0.031s
sys  0m0.000s
```

No states are missed. The memory requirements dropped to 12.9 Megabytes, and the run−time requirements remained largely unchanged from an exhaustive search.

The coverage of both the hash−compact and the double−bit hashing method can be increased by performing multiple searches, each time with an independent set of hash functions. If each search misses a fraction p of the state space, t independent searches could reduce this to p^t. Though potentially expensive in run time, this gives us a capability to increase the quality of a verification under adverse constraints.

Which of all these storage methods is best? There is, alas, no single answer to this question. The behavior of each algorithm can depend to some extent on unpredictable particulars of an application. All compression methods can be expected to bring some improvement, but the maximal improvement is not always achieved with the same technique for all applications. If there is enough memory to complete an exhaustive search, the problem of search optimization need not even be considered. When the problem is too large to be verified exhaustively with the default search method, two experiments that can be performed relatively quickly and without much thought are collapse compression and hash−compact. To go beyond what is achieved with these methods, more thought, and perhaps more search time, may be needed. As a last resort, but only for very large problem sizes, the bitstate hashing method is often hard to defeat, and will likely give the best results.

Bibliographic Notes

A formal treatments of the notions of dependence of actions (transitions) and the equivalence of ω–runs can be found in, for instance, Mazurkiewicz [1986], and Kwiatkowska [1989]. The application of these notions to model checking is described in Peled [1994], and Holzmann and Peled [1994], with a small, but important, adjustment that is explained in Holzmann, Peled, and Yannakakis [1996].

A formal proof of correctness of the partial order reduction algorithm implemented in SPIN is given in Chou and Peled [1999], and is also discussed in Clarke, Grumberg, and Peled [2000].

The statement merging technique that is implemented in SPIN was first proposed in Schoot and Ural [1996]. The SPIN implementation is discussed in Holzmann [1999].

The `COLLAPSE` compression method is described in detail in Holzmann [1997]. The design and implementation of the minimized automaton storage method is detailed in Holzmann and Puri [1999]. There are several interesting similarities, but also significant differences, between the minimized automaton procedure and methods based on the use of BDDs (binary decision diagrams) that are commonly used in model checking tools for hardware circuit verification. A discussion of these and other points can be found in Holzmann and Puri [1999].

The application of the hash–compact method to verification was described in Wolper and Leroy [1993], and also independently in Stern and Dill [1995]. An earlier theoretical treatment of this storage method can also be found in Carter at al. [1978].

Bitstate hashing, sometimes called supertrace, was introduced in Holzmann [1988] and studied in more detail in Holzmann [1998]. The first explicit description of the notion of bitstate hashing, though not the term, appeared in Morris [1968], in a paper on "scatter storage" techniques. In Bob Morris's original paper, the technique was mentioned mostly as a theoretical curiosity, unlikely to have serious applications. Dennis Ritchie and Doug McIlroy found an application of this storage technique in 1979 to speed up the UNIX spelling checking program, as later described in McIlroy [1982].

The original implementation of `spell` was done by Steve Johnson. The new, faster version was written by Dennis Ritchie, and was distributed as part of the 7th Edition version of UNIX. The mathematics McIlroy used in his 1982 paper to explain the working of the method is similar to the elegant exposition from Bloom [1970]. Bloom's 1970 paper, in turn, was written in response to Morris [1968], but was rediscovered only recently. Bob Morris, Dennis Ritchie, Doug McIlroy, and Steve Johnson all worked in the UNIX group at Bell Labs at the time.

The code used in the 1979 version of `spell` for table lookup differs significantly from the version that is used in the SPIN implementation for bitstate hashing, given the differences in target use. The first hash function that was implemented in SPIN for default state storage during verification was based on the computation of 32–bit cyclic redundancy checksum polynomials, and was implemented in close collaboration with (then) Bell Labs researchers Jim Reeds, Ken Thompson, and Rob Pike.

The current hashing code used in SPIN is based on Jenkins [1997]. Jenkins' hash function is slightly slower than the original code, but it incurs fewer collissions. The original hash functions are reinstated when the `pan.c` source code is compiled with directive `-DOHASH`.