Runtime Efficient State Compaction in Spin

J Geldenhuys and PJA de Villiers

Department of Computer Science, University of Stellenbosch,
7600 Stellenbosch, SOUTH AFRICA
{jaco,pja}@cs.sun.ac.za

Abstract. Spin is a verification system that can detect errors automatically by exploring the reachable state space of a system. The efficiency of verifiers like Spin depends crucially on the technique used for the representation of states. A number of recent proposals for more compact representations reduce the memory requirements, but cause a considerable increase in execution time. These methods could be used as alternatives when the standard state representation exhausts the memory, but this is exactly when the additional overhead is least affordable.

We describe a simple but effective state representation scheme that can be used in conjunction with Spin's normal modes of operation. We compare the idea to Spin's standard state representation and describe how Spin was modified to support it. Experimental results show that the technique provides a valuable reduction in memory requirements and simultaneously *reduce* the execution time. For the cases considered an average reduction in memory requirements of 40% was measured and execution time was reduced on average by 19%. The proposed technique could therefore be considered to replace the default technique in Spin.

1 Introduction

Model checking has been applied successfully to detect errors in systems of realistic size and complexity [1,6]. Spin is a well-known model checker that does on-the-fly state exploration to verify correctness claims stated as temporal logic formulae. With model checkers such as Spin there are two main concerns: (1) the efficiency of state generation and (2) the amount of memory needed to store unique states. The state generation algorithm used in Spin is highly efficient, but it seems possible to improve state storage.

Currently, Spin must be used in *supertrace mode* to cope with large state spaces. This involves using a controlled partial search. The technique is based on hashing without collision detection and when collisions do occur, search paths are sometimes terminated prematurely. Unfortunately this means that errors could be missed.

A full state space exploration is always preferable, but this is only possible for much smaller models because all unique states must be stored in memory. One remedy is to reduce the state size by storing states in compacted form. Various state compaction techniques have been investigated, but all increase the execution time significantly [3, 5, 7]. In this paper we describe a state representation technique that is simple to implement. It does not only reduce the memory requirements, but often reduces the execution time as well. Section 2 describes the principles underlying our technique and Section 3 explains how the SPIN source code was modified to incorporate the proposed technique; results are presented in Section 4.

2 Proposed state representation technique

The global state of a PROMELA model consists of the values of the variables in all processes. Usually each state includes several data variables and a single location variable per process. These variables are aligned on byte boundaries because PROMELA supports various 8-bit, 16-bit and 32-bit types. Because one or more bytes are allocated to each variable, state sizes of more than a hundred bytes are not exceptional. Therefore, if all unique states are stored explicitly in uncompacted form, a model that generates a few million states quickly exhausts the memory available on typical workstations.

Most variables in validation models only assume a small subset of their potential values. An efficient technique to map these values onto consecutive integers would save a substantial amount of memory. One possible strategy is to enter unique states as they are encountered in lookup tables. This is the principle behind the approaches in [5, 7]. The position (index) in the table where a given state is entered then serves to identify the much larger state value. However, the overhead of manipulating these tables increases the execution time.

A significant reduction in state size is possible by simply placing tighter bounds on the ranges of variables and packing them into the minimum space required. This idea is the essence of the technique proposed here. Users can easily supply the information about the ranges of variables if the validation language supports user-definable types. For example, type definitions such as "ProcNumber = 0...4" are easy to use and provide enough information to store variables in compacted form.

A small example will illustrate the basic idea. Assume that a model contains three variables v_1, v_2, and v_3 which can respectively assume values from the ranges $0\dots4$, $0\dots2$, and $0\dots6$. Reflecting the number of different values each variable can assume, the compacted form V of each given state is computed as

$$\begin{aligned} V &= v_3 + 7(v_2 + 3v_1) \\ &= v_3 + 7v_2 + 7 \cdot 3v_1 \end{aligned}$$

Two constant factors are associated with each variable v_i. These factors, known as the lower and upper factors of each variable, are denoted by v_i^l and v_i^u respectively. In the example above, $v_3^l = 1$, $v_3^u = 7$, $v_2^l = 7$, $v_2^u = 7 \cdot 3 = 21$, $v_1^l = 21$, and $v_1^u = 7 \cdot 3 \cdot 5 = 105$. These factors are used as masks to extract and update the value of a specific variable in the compacted representation of a state.

Variables are manipulated by two basic operations which must be implemented as efficiently as possible. The operation *GetValue* used to extract the value of a given variable v_i from a compacted state V is simple: $v_i = (V\ mod\ v_i^u)$ $div\ v_i^l$. A simple adjustment is necessary to accommodate variables with non-zero offsets. The operation *SetValue*, which is used to change the value of a variable v_i to v_i', is a little more complex. The new compacted state V' is given by $V' = V + v_i^l \cdot (v_i' - v_i)$. The operations *GetValue* and *SetValue* represent the only overhead at execution time.

Assume that the number of values allowed for variable v_i is denoted by $|v_i|$ and that the lower and upper factors associated with v_i are denoted by v_i^l and v_i^u, respectively. The lower factor of variable v_1 is 1 and its upper factor is $|v_1|$. For $i > 1$ the lower and upper factors of variable v_i are given by

$$\begin{aligned} v_i^l &= v_{i-1}^u \\ v_i^u &= |v_i| \cdot v_i^l \end{aligned}$$

These constants are computed only once for each variable and therefore contribute only a small constant overhead to the validation run. The number of bits required to store a compacted state with n variables is

$$\lceil \log_2 \prod_{i=1}^{n} |v_i| \rceil$$

For instance, the number of bits required to store v_1, v_2, and v_3 in the example above is $\lceil \log_2 5 \cdot 3 \cdot 7 \rceil = 7$.

3 Experimental implementation using SPIN

A detailed description of SPIN can be found in [4]. Most relevant here, is the format of the state vector as shown in Figure 1. The state vector stores the current state. It contains the number of processes (nr_pr), the number of queues (nr_qs), flags for cycle detection (a_t) and the contents of processes and queues. Each process (such as p_0) contains a unique identifier, its local state, the process type, and the values of its local variables. Each queue (such as q_1) contains the queue length, queue type, and the values of its elements. Unused elements are stored as zeros. The state vector is continually referenced as the values of variables are read and modified. It is kept in uncompacted form to speed up references to variables.

The SPIN verifier explores the state space in a depth-first manner, storing only the current execution path in a stack. The amount of memory required by the stack is not a serious concern; although the stack may sometimes contain several thousands of states, it is usually small relative to the size of the state space. A more serious problem is that in full search mode all unique states must be stored in a state table. In practice, the number of states that can be stored in the table determines the maximum size of models that can be handled. This is where compaction pays off: it makes it possible to store more states in the same amount of memory.

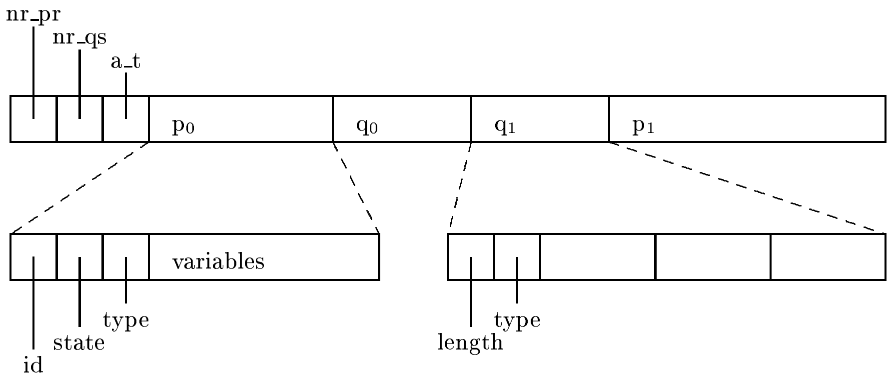

Fig. 1. Standard form of the state vector

It is not practical to store the entire compacted state as a single number and therefore variables are packed into subgroups called *cells*. Each cell is compacted separately and variables may not be split over more than one cell. This causes a small amount of space wastage, but speeds up the manipulation of variables. In the experimental implementation the state vector consists of an array of cells of type `unsigned long`. Each variable in the model is given a unique number, called its *position*. There are three tables: `cell_number` maps each position to the cell where it is stored, `lower_factor` maps each position to its lower factor, and `upper_factor` maps each position to its upper factor. The compacted state vector is stored in a global array variable called `sv`. The implementations of the `SetValue` and `GetValue` operations are simple:

```
void SetValue(int pos, int delta)
{
   sv[cell_number[pos]] += lower_factor[pos] * delta;
}

int GetValue(int pos)
{
   return sv[cell_number[pos]] %
      upper_factor[pos] / lower_factor[pos];
}
```

The `pos` parameter refers to the position of a variable, and the `delta` parameter of `SetValue` is the value $(v_i' - v_i)$, the difference between the old and new values of the variable.

However, this implementation of `GetValue` is too slow. A simple remedy is to keep an uncompacted copy of the current state in variable `usv`. For most models, the size of `usv` is less than 1 Kbyte. It is unnecessary to store `usv` in the state table, but, unfortunately, it must be stored on the stack to allow the current implementation of SPIN to backtrack more than one state at a time. The following more efficient implementations of `SetValue` and `GetValue` were therefore used:

```
void SetValue(int pos, int delta)
{
   sv[cell_number[pos]] += lower_factor[pos] * delta;
   usv[pos] += delta;
}

int GetValue(int pos)
{
   return usv[pos];
}
```

As new variables are created during the execution of a model, they are added to the state vector. This task is performed by the `AddPosition` routine. The parameter `card` indicates the cardinality ($|v_i|$) of the new variable.

```
void AddPosition(int card)
{
   if (MAXCELLVALUE / upper_factor[maxpos - 1] <= card)
   {
      /* use a new cell */
      lower_factor[maxpos] = 1;
      cell_number[maxpos] = cell_number[maxpos - 1] + 1;
   }
   else
   {
      lower_factor[maxpos] = upper_factor[maxpos - 1];
      cell_number[maxpos] = cell_number[maxpos - 1];
   }
   upper_factor[maxpos] = lower_factor[maxpos] * card;
   maxpos += 1;
}
```

The global variable `maxpos` stores the number of the highest position allocated so far. The test determines whether the new variable will fit into the current cell, or whether a new cell must be used. When a new cell is allocated, the last few bits of the previous cell are sometimes left unused. It is possible to minimize the number of bits wasted by reordering the variables, although this was not attempted in the current implementation.

Currently, a slightly restricted version of PROMELA is supported:

– Different orders of process activation along different execution paths are forbidden.

- No global channels are currently allowed. If necessary, models must be adapted so that the `init` process contains all channels, passing them to processes when they are created.
- The ranges of variables must start at 0.

It was necessary to make several modifications to the SPIN source code to implement the compaction technique: the PROMELA grammar was extended with a new data type, the translation of PROMELA statements was modified and a few other routines in the SPIN system were adapted to use the new state representation.

3.1 Modifications to the PROMELA grammar

The range of each variable is needed and the simplest way to obtain this information is to ask the user to supply it. The PROMELA grammar was therefore extended to allow a new data type: `value` n. Instead of declaring a variable as

```
byte x;
```

we write

```
value 12 x;
```

which means that variable `x` can take on 12 different values ranging from 0 to 11. The declaration

```
mtype = { A, B, C, D };
```

is transformed into

```
#define A 4
#define B 3
#define C 2
#define D 1
#define mtype value 5
```

(The values for the messages were chosen to be internally consistent with the current SPIN implementation.) So, instead of

```
chan q = [4] of { mtype, int };
```

we could write

```
chan q = [4] of { mtype, value 6 };
```

In our opinion, it would be better to provide such user-definable types to replace the fixed types of PROMELA. This would allow users to place tighter bounds on the ranges of variables. The notation above was selected for its simplicity; it may be more convenient to also specify the lower bound, to allow types such as "$-2 \ldots 2$" and "$60 \ldots 70$". It may also be useful to allow user-defined subrange types instead of the scheme described above. As an example, a counter that can take on values 0 through 10 could then be declared as follows:

```
subrange Counter = 0...10;
Counter c;
```

3.2 Modification to the manipulation of variables

Normally SPIN translates PROMELA statements to code that performs operations on the normal state vector shown in Figure 1. This system was modified so that the generated code acts on the new compacted state vector. The most important statements are accesses to local variables of processes.

A read access of variable `x` is translated as

```
GetValue(pos)
```

where `pos` is the position of `x` in the state vector. A write access, or assignment, `x = e`, is translated as

```
SetValue(pos, e - GetValue(pos))
```

Turning to a more concrete example, consider the PROMELA statement

```
x = y + 2
```

where `x` and `y` are local variables of some process. This is normally translated as

```
case 6: /* STATE 5 - [x = (y+2)] */
   IfNotBlocked
   (trpt+1)->oval = ((int)((P2 *)this)->x);
   ((P2 *)this)->x = (((int)((P2 *)this)->y)+2);
   m = 3; goto P999;
```

In the modified version of SPIN the assignment is translated as

```
case 6: /* STATE 5 - [x = (y+2)] */
   IfNotBlocked
   pos = curproc+14;
   delta = (GetValue(curproc+15) + 2) - GetValue(pos);
   (trpt+1)->oval = delta;
   SetValue(pos, delta);
   m = 3; goto P999;
```

The main difference is that procedures `GetValue` and `SetValue` are used to manipulate variables. The constants `14` and `15` in the example are the positions of variables `x` and `y` relative to the start of the process to which they are local. The starting position of the variables of a process is stored in `curproc`.

4 Results

A number of well-known PROMELA models were used to measure the effectiveness of the compaction technique. Some of the models (leader, pftp, snoopy, and sort) are part of the standard SPIN distribution, but were slightly modified to adhere to the restrictions of our implementation. The other models (cambridge and slide) were collected from the Internet[1].

Table 1 shows the memory required (in Mbytes) followed by the time needed to complete each validation run (in seconds)[2]. The first column (marked SPIN) gives the results for the standard SPIN system. The next column (marked SPIN-C) is for the modified SPIN system with compact representation of states. The last column shows the ratio between the two systems for purposes of comparison.

Table 1. Comparison of standard SPIN against SPIN with the proposed compression technique (SPIN-C)

Model	SPIN		SPIN-C		Ratio	
	memory	time	memory	time	memory	time
cambridge	54.1	6.9	23.8	6.8	0.44	0.99
leader	91.7	12.6	33.2	7.7	0.36	0.61
pftp	66.6	8.9	26.5	7.3	0.40	0.83
snoopy	15.2	1.7	10.6	1.4	0.70	0.84
sort	3.7	0.3	4.4	0.2	1.19	0.77
sort(BIG)	130.7	22.6	70.4	14.7	0.54	0.65
slide	33.3	3.2	20.3	3.2	0.61	0.97
Average					0.60	0.81

Note that the technique works best for the larger models. A useful reduction in memory requirements was measured for most models, the average being 40%. More surprising, however, is that execution time was *decreased* on average by 19%.

As can be expected, the results differ from one model to the next. This is due to differences in the number of variables, the number of read and write accesses, and the structure of the state space. In fact, when models use large amounts of stack memory compared to the amount of state table memory, the technique does not seem to help at all.

For example, a model of six dining philosophers[3] yields a memory ratio of 1.89 and a time ratio of 1.11. This model is unusual because the stack reaches a depth of 152383 states, while the state table contains only 94376 states. For the majority of models, the situation is reversed. The execution time is longer in this case, since a substantial overhead is associated with pushing states onto the stack. This overhead comes from copying the uncompacted state vector to the stack. This overhead is avoided in SPIN's standard mode of operation because the uncompacted state vector is not stored in the stack. A substantial modification of the internal mechanisms of SPIN is required to address this problem, but

[1] `http://cm.bell-labs.com/cm/cs/what/spin/Man/Exercises.html`

[2] Tests were executed on a 400MHz Pentium II with 256 Mbytes of memory.

[3] `http://www.ececs.uc.edu/~imutaban/miniproject/2Phase/2phase.html`

this was not attempted since our goal was simply to test the feasibility of the technique in the context of SPIN.

More information about the models is given in Table 2. The first three columns give the size of the original SPIN state vector in bytes, the number of variables in each model, and the number of cells (of 4 bytes each) occupied by the compacted state vector. The last two columns contain the number of unique states and length of the longest path on the stack.

Table 2. More details about models

Model	SPIN	# variables	cells	states	depth
cambridge	64	52	5	728544	13004
leader	148	223	18	341290	148
pftp	140	109	11	439895	5780
snoopy	196	105	14	91920	12250
sort	140	92	16	15351	115
sort(BIG)	188	147	23	659683	202
slide	52	43	4	510111	51072

5 Discussion and conclusions

Compact representation of states is crucial in verifiers like SPIN and various compaction techniques have been investigated by modifying the SPIN source code. In Table 3 a comparison is given for five such techniques. (Only the pftp and snoopy data are available for all techniques considered.) The measurements in the table were taken from the literature [3, 5, 7]. Although they were measured on different platforms and cannot be compared directly to each other, they provide a rough indication of the cost of each technique.

Because the platforms and validation models differ, it is difficult to draw general conclusions about the effectiveness of specific techniques. However, all compression techniques proposed thus far seem to increase execution times while the technique proposed in this article can *reduce* the runtime required. This is surprising. A possible explanation is that the reduction in state size—which speeds up state comparison and manipulation—must be more than enough to compensate for the compression overhead. On the negative side, only a fair amount of compression can be expected in general.

The impressive performance of some compression techniques rely on extra information that must be supplied by users. For example, it may be necessary to obtain information about the behaviour of variables from training runs [3, 5]. In addition, the most powerful technique considered ([3]) requires that users define an encoding for state information. To be fair, this extra time and effort cannot be ignored. The technique proposed here only needs information that can be derived automatically from the declaration of variables at compile time.

Table 3. Comparison of state compression techniques. The first four rows of measurements were taken directly from the literature, while the last two rows were measured on a 400MHz Pentium II.

Technique	pftp		snoopy	
	memory	time	memory	time
GETSs [3]	0.05	2.07	0.09	3.38
State compression [7]	0.25	1.24	0.17	0.29
Recursive indexing [5]	0.18	1.41	0.33	2.45
Two-phase compression [5]	0.15	2.42	0.22	3.23
Recursive indexing [5]	0.18	2.54	0.27	3.47
State compaction (proposed technique)	0.40	0.83	0.70	0.84

The technique relies on extra information regarding the minimum and maximum values assumed by variables. It is easy to obtain this information if user-definable types are used instead of the predefined fixed types of PROMELA. In fact, based on our experience with ESML [2]—a specification language similar to PROMELA—we know that it requires little extra effort of users to define the ranges of variables more precisely. When it comes to the implementation of such models, it is trivial to find the optimal representation for integer subranges.

We have demonstrated the feasibility of our idea in a well-known context by modifying the SPIN source code to test the technique. The results indicate that substantially larger models can be handled in full search mode, while reducing execution time as a bonus.

References

1. E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. ACM Computing Surveys, 28(4):626–643, December 1996.
2. P. J. A. de Villiers and W. C. Visser. ESML—a validation language for concurrent systems. South African Computer Journal, 7:59–64, July 1992.
3. J.-Ch. Grégoire. State space compression in SPIN with GETSs. In Proceedings of the 2nd SPIN Workshop, 1996.
4. G. J. Holzmann. Design and Validation of Computer Protocols. Prentice Hall Software Series, 1991.
5. G. J. Holzmann. State compression in SPIN: recursive indexing and compression training runs. In Proceedings of the 3rd SPIN Workshop, 1997.
6. J. Rushby. Mechanized formal methods: progress and prospects. In Proceedings of the 16th Conference on the Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science #1180, pages 43–51. Springer-Verlag, December 1996.
7. W. C. Visser. Memory efficient state storage in SPIN. In Proceedings of the 2nd SPIN Workshop, 1996.