State Space Compression in SPIN with GETSs

J.-Ch. Grégoire
INRS-Télécommunications
gregoire@inrs-telecom.uquebec.ca

Abstract

We explore the application of a state space compression algorithm to SPIN. Compression techniques are notably unpredictable: we tend to use them when they work well for us (e.g. ZIP),but we tend to forget that, according to the theory, they can also result in an expansion (e.g., ZIPtwice).

How well a compression algorithm work in a given context is typically discovered experimentally. In this paper, we describe such an experiment, where a compression algorithm based on sharing of state information in a graph structure is used as a state store for a SPIN-generated verification programs.

The positive results are encouraging for further investigation of the use of this technique in SPIN-like tools.

keywords state space compression, reachability analysis, GRAPH ENCODED TUPLE SETS, ARBRES PARTAGÉS, SPIN.

Contents

1 Introduction **3**

2 Definitions **4**

3 Behavior of GETSs **5**

4 Operations on GETSs **8**
4.1 Operators 8
4.2 Normalized insertion 8
4.3 Normalized deletion 9
4.4 Other Operators 9

5 Application to SPIN and Analysis **10**
5.1 State Coding 10
5.2 Stack Presence Coding 11
5.3 snoopy 11

5.4 pftp 11
5.5 gsm 12
5.6 Observations 15
5.7 Discussion 16
5.8 SPIN's Compression 17

6 Conclusion **18**

1 Introduction

The amount of main storage available on a given computer is the usual bottleneck in the verification of finite state systems. In SPIN [Holzmann91], this obstacle is alleviated somewhat in two different ways. The first is through the use of "partial order" techniques [Holzmann et al.94], which reduce the size of the system to explore while preserving the properties we want to verify. The second consists of changing the search from an exhaustive one to a partial, stochastic one, but without guaranteing a complete exploration [Holzmann91]. Those two techniques are orthogonal and can be combined.

We explore here the use of a data structure, called Graph Encoded Tuple Sets (GETS) [1] to support the computations of SPIN. This data structure acts as a compression mechanism for the state space. By encoding the state representation, it has the potential of reducing the amount of storage required to hold the state space. However, since it acts as a compression mechanism, it also has the potential of actually using *more* storage than would be required without compression.

In earlier work [Gagnon et al.95], we have already studied GETS in the context of a custom reachability analysis tool. Here our focus is on SPIN, and on whether or not this algorithm will behave favorably (i.e. achieve major compression in reasonable time) in the context of SPIN, and with "partial order" reduced explorations. Our results in this experiment have been largely positive, but we should stress they are only the first step towards a GETS-based SPIN.

Next, the data structure is defined, and its properties illustrated. We then define a few of its operators. We go on to explore how they can be used in SPIN, and show the results we have obtained in a few experiments. Some observations are made and discussed.

[1] the original french name of the data structure was Arbres Partagés, or "shared trees", and dubbed in english "sharing trees" by original authors. In practice, we have found the name to be misleading, and prefer the GETS terminology; GETS however are indeed Arbres Partagés.

2 Definitions

Let us begin with a formal definition of the data structure. This definition is important to highlight some critical properties.

Definition A GETS is a rooted acyclic graph $(N, V, val, succ)$ such that $N = N_0 \cup \ldots \cup N_k, k \geq 0$, is a finite set called the set of nodes (the nodes are organized in layers, N_i is the set of nodes of layer i), V is a set of values, $val : N \longrightarrow V + \{\top, \bot\}$ is the valuation function, $succ : N \longrightarrow \mathcal{P}(N)$ [2] gives the successors (i.e. sons, on the next layer) of a node; and the following properties hold:

1. $\forall i \; s.t. \; 0 \leq i < k, \forall n \in N_i, succ(n) \subseteq N_{i+1}$: each node has all its successors on the next layer;

2. $\forall i \; s.t. \; 0 \leq i \leq k, \forall n_1 \& n_2 \in N_i, n_1 \neq n_2, val(n_1) = val(n_2) \Rightarrow succ(n_1) \neq succ(n_2)$: two nodes holding equal values in the same layer do not have the same set of sons;

3. $\forall n \in N, \forall s_1 \& s_2 \in succ(n), s_1 \neq s_2 \Rightarrow val(s_1) \neq val(s_2)$: a node does not have two sons with equal values;

4. $|N_0| = 1$ and $\forall n \in N, val(n) = \top \Leftrightarrow n \in N_0$: the first layer N_0 contains only one element (called the root), the only one with value $\top$;

5. $val(n) = \bot \Rightarrow succ(n) = \emptyset$ and
   $succ(n) = \emptyset \Rightarrow (val(n) = \bot \vee val(n) = \top)$.

Let us now consider that each component of a tuple is mapped to a specific layer, according to some (total) order. The second condition guarantees that there will be *suffix merging* of tuples that share equal ends, while the third condition guarantees the *prefix merging* of tuples that share equal beginnings. This condition furthermore imposes that the creation of the longest sharable prefixes, or in other words, *every prefix is unique.*

Any path starting from the root node $r \in N_0$ and terminated by $\bot$ represents one tuple of values encoded in the GETS.

The following GETS (fig. 1) represents the set $\{(a, b, d, e),\ (a, b, d, f),\ (a, b),\ (a, c),\ (a, c, d, f)\}$. Note that tuples of different lengths can be represented in the same GETS. Notice also the repetition of a node with value d in the 3rd layer, which is legitimate since the set of suffixes is different for the two nodes.

A fundamental property of GETSs is the *canonicity* of the coding: there is a unique representation for any set of tuples. This, however, does not imply *minimality*; it may often be possible to find graph codings of tuples which would have a smaller number of nodes and/or vertices.

The canonicity property derives from the uniqueness of the prefixes (like tries), and the sharing of identical suffixes on the same level. Imagine [3] two GETSs holding the same set of tuples. Let us also imagine that, following a path simultaneously through both GETSs,

[2] $\mathcal{P}(N)$ is the *powerset* of N.

[3] A more formal proof is given in [Zampunieris et al.95].

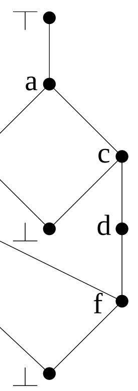

Figure 1: Example GETS

we encounter a node which has a different number of sons in each GETS. This would mean that, in one of the two GETSs, there would have to be another path leading to (one of) the son(s) missing in the other, since both GETSs hold the same information. This however would break the rule of the sharing of the longuest prefix. The same reasoning applies to suffixes.

GETSs have other interesting properties:

explicitness each contribution to the tuple is present in the graph;

constant time projection all projections of the set of tuples on any component of the tuple can be done in constant time;

heterogeneity the domains of the components of the tuples can be different and the tuples can have varying lengths;

insertion order independence since the representation is canonical, it does not depend on the order in which the tuples are inserted in the GETS;

isomorphism again, because of the canonicity property, two GETSs containing the same set of tuples are isomorphic.

3 Behavior of GETSs

The following example will further illustrate the behavior of GETSs and the problem we have to consider for their implementation. Consider the following GETS holding the set $\{(a, b, b, a)$, (a, b, b, b), (b, b, b, a) and $(b, b, b, b)\}$. This structure is represented on figure 2. We add to it the tuple (a, b, b, c). The structure of the graph changes significantly, as it is split to isolate the two longest prefixes (a, b, b) and (b, b, b) as shown on figure 3.

We see that, to preserve canonicity, it was necessary to perform a *splitting* operation on the tree: a shared prefix was split to create the longest shared unique prefix (a, b, b) for the three suffixes (a), (b) or (c).

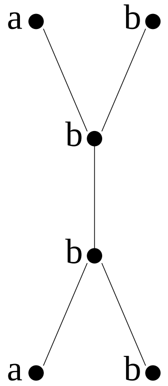

Figure 2: Simple GETS

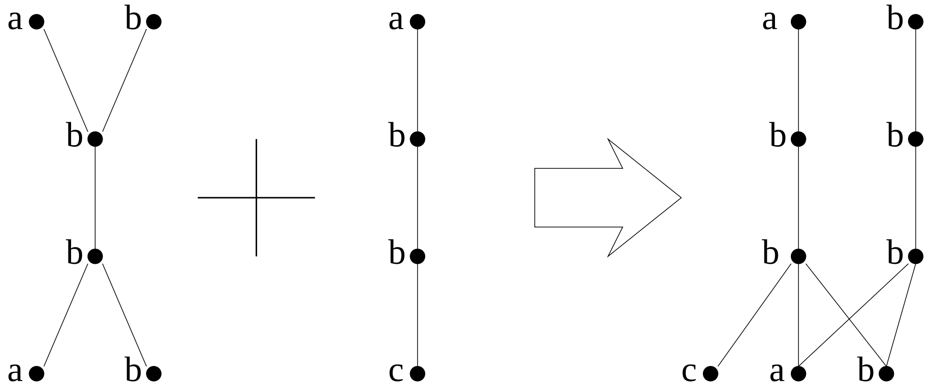

Figure 3: First addition

Observe also that this structure is not minimal in the number of node and links. Figure 4 shows an alternate graph holding the same information, with the same number of nodes, but less links. It is however not a GETS since the prefix (a, b, b) is duplicated.

Let us now add the tuple (b, b, b, c) to the previous GETS. Observe on figure 5 that the result is more compact. The addition of the tuple has allowed *suffix merging*: the creation of a common sharable suffix, its merging and, in this case, the recursive merging of the paths leading to it in a common one.

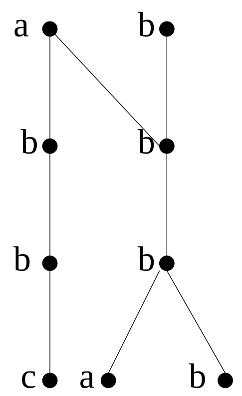

Figure 4: Smaller graph

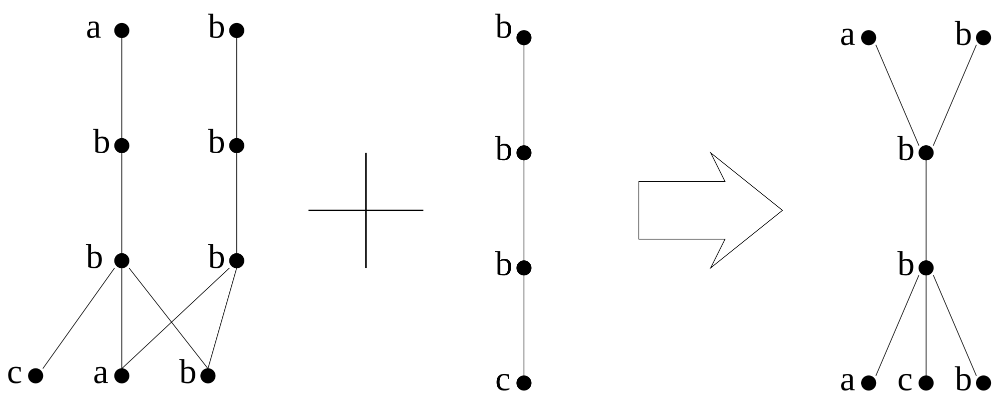

Figure 5: Second addition

4 Operations on GETSs

We present here a few operators to manipulate GETSs. In this presentation, we will use a mix of terminology derived from graph and language theory. A GETS is an undirected graph structure organized along layers, where vertices connect only nodes on adjacent layers. We have already indicated that any path between the top layer of a GETS and terminated by the special $\perp$ value corresponds to a tuple of information stored in. Any path is composed of an *prefix,* a *body* and a *suffix,* any of which might be empty. When we'll talk about the insertion of a tuple of values into a GETS, we'll have to take into account existing prefixes and suffixes which can be shared to some extend. A prefix here is the part of a path starting at $\top$ leading to a node with more than one parent. A suffix is the part of the path ending at $\perp$ which has nodes with single sons only.

4.1 Operators

GETSs can support all the operators of set manipulation [Zampunieris et al.95]. For the purpose of this presentation, however, there are only three basic operators needed, namely *membership, insertion* and *deletion*. An important issue, however, is whether an insertion or removal operation should preserve the GETS, or some of the constraints could be relaxed for a while, until some *normalization* operation is performed. In fact, one original implementation of GETSs was designed exactly that way. However, our analysis of several tests cases have convinced us that, on any number of repetitions, using an operation preserving the GETS resulted in general better performance than a non-GETS preserving operation, followed by a normalization [Gagnon et al.95].

4.2 Normalized insertion

1. identify in the GETS the longest prefix, the body and the longest suffix matching the tuple of values to insert; the prefix and the body are contiguous, but the suffix may be detached; if either the prefix or the body end with $\perp$, then the tuple already exists in the GETS and we can terminate, otherwise:

2. if the prefix isn't empty, then duplicate and split the body on its whole length; the body is adequate otherwise;

3. expand the path from the end of the body (or the prefix if the body is empty, or from $\top$ if both are empty) to the start of the longest suffix or to $\perp$ if this latter is empty, and

4. merge suffixes upward.

In the previous example (fig. 3), the prefix was (a, b), the body (b, b), and the suffix was empty. The body was duplicated to create a (longest) prefix (a, b, b) in the GETS. A new suffix was then added. In the second example (fig. 5), the prefix was (b, b, b), the body was empty, and the suffix was (c). A link was added from the end of the prefix to (c), then the nodes were merged upwards.

Adding new suffixes creates an opportunity for sharing similar suffixes. We must identify, from the point where the path merges with the suffix if another node with the same set of sons already exists. If it is the case, the nodes can be merged, and we renew the operation upward.

It is quite clear that this algorithm terminates. Once we have completed the upward merge, nothing is left to do since the longest unique prefix constraint was established from the start, and is not invalidated by a merge. We see also that the procedure is linear in the number of layers.

4.3 Normalized deletion

Normalized deletion operates similarly to normalized insertion.

1. identify a complete (i.e. to $\perp$) path matching the tuple of values we want to remove, terminate if it is not found; otherwise:

2. decompose the path into longest prefix, body and suffix;

3. if the prefix isn't empty, split the path from the beginning of the prefix to, and including the node where the suffix starts;

4. remove the suffix from that node and

5. merge the suffixes upward.

The intuition behind this algorithm is quite simple. We must isolate the tree holding the path to be removed from the effects of sharing, then we remove its distinctive part, preserving the longest prefix distinguishing the tuple to be removed, and normalize the new resulting suffix.

4.4 Other Operators

Membership is done trivially by looking for the tuple in the GETS. We have also implemented an *iterator* on GETS. An iterator call successively returns all the tuple instances held in a GETS. Our iterators are robust in the presence of modifications to the GETS, but do not guarantee that all the tuple stored will be found in that case.

For our purposes, we do not require operators to merge GETSs together or to extract subsets of them.

5 Application to SPIN and Analysis

To analyze the performance of GETS in a "real" context, we have modified analyzers generated by SPIN (i.e. "pan" programs) so that they would use this data structure rather than the hash table storage. Since this was only a feasibility study, we haven't made any attempt at guaranteing that the substitution was compatible with all the algorithms supported by SPIN in full state search mode. We have focused only on safety properties, with or without partial orders reduction.

We have analysed three examples: two tests from the SPIN distribution, namely *snoopy* and *pftp,* and *gsm,* a development of our group, inspired by one of the GSM protocols.

We first explore the issues we were confronted with, namely *state coding* and *stack presence coding.* We then cover the examples, report the performance measured, make some observations and discuss them.

5.1 State Coding

The most significant difficulty of using GETS is to define an encoding for state information. The most straightforward is, of course, to allocate a layer to every variable in the PROMELA model, as well as the bookkeeping variables of SPIN (e.g. the " process counters" _p).

In our experiments, we have done some clustering of the variables. The simple rule has been to cluster variables in each of the global-queue-process groups to keep the size of the domain of the cluster to less then 2^{16} (a *short word*).

The clusters have been assigned to layers following simple rules: put all the clusters of a group item together and try to put in close proximity the group items interacting together.

Our implementation of GETS uses a static allocation of pointers for each element of the domain of clusters. Having a sparse cluster thus leads to a waste of space. When variables with a sparse domain are combined together in a cluster, the wastefullness is compounded. Independently of the (PROMELA) model itself, transitions identifiers typically have a sparse domain in SPIN. This phenomenon is thus real. However, its importance is relative: the waste in pointer space may or may not not be very significant, depending on the size of the domain, and the degree of sparseness.

How clustering is done is another possible contributor to space or time overhead. The most straightforward form of clustering is to compose the *declared* (i.e. in the model) domains of the variables. PROMELA is however notably deficient in this respect, since declared domains can be much larger than the reachable domains. We have tried instead to use the number of bits required to store the reachable domains. Support for such an idenfication exists in SPIN (`VAR_RANGES`) . It could also largely be done by a static analysis of the PROMELA model. In practice, on our examples, this identification has been straightforward [4]. The domains are rounded up to the next power of two and the clusters are assembled through shifting and "or"-ing.

More complex solutions exist which would involve a look-up to compress sparse domains, and clustering by multiplication rather than by shifting, for a most compact clustering. We

[4] Encoding and implementation were done in the course of a couple of hours in all cases.

haven't used these techniques in our experiments.

5.2 Stack Presence Coding

Supporting partial orders has been challenging since the algorithm used in SPIN requires the recognition of whether or not a state already encountered is on the stack. In full state search mode in SPIN, this is done by tagging the state information in the store. The value is the tag is set when the state is met the first time, and reset when it is popped from the stack.

We have found two alternatives to support such a mechanism with GETS. The first, obvious one is to have another GETS to hold a copy of the states currently on the stack. For any state, we would have three GETS operations: addition to the store and to the stack, deletion from the stack. We have however tried another method, borrowing from the idea of tagging the state. We store the state *twice* in the state store, with different tag values appended: one, with a tag value of 0, when the state is first encountered and thus is on the stack, and another, with a tag value of 1, when it is popped from the stack. Except in the REACHvariation, where the state space is always going to be explored at some specific depth, and we need to memorize the depth information with the state, this is sufficient to quickly decide whether or not a state is on the stack.

The cost of this algorithm is two insertions, rather than one, and a doubling of the amount of states stored in the GETS. However, since we know that both instances of the state will end up being in the store, this doubling practically involves a lot of sharing and shouldn't lead to significant space overhead.

5.3 snoopy

We have tested snoopy only in NOREDUCE'd mode. The amount of memory required otherwise is too small to give significant results. The encoding uses 21 layers, the largest domain used, for the `cache` processes is 2560.

	states	memory (M)	time (s)
FS, spin	91920	9.934	13.45
FS, GETSs	91920	0.55(GETS) + 0.375(spin)	45.52

The memory contribution for GETS is broken down in two parts: GETS *per se* and SPIN which is mostly the memory required for the stack. We have a compression factor specifically for the state space of about 18 in this case.

5.4 pftp

This example being larger, we have been able to test it both in REDUCE'd and in NOREDUCE'd form. We have 19 layers, and the largest domain is 4096.

	states	memory (M)	time (s)
FS, spin	439895	54.49	86.71
FS, GETSs	439895	2.25(GETS) + 0.334(SPIN)	179.59
RS, spin	95241	13.33	10.41
RS, GETSs	95241	1.864(GETS) + 0.334(SPIN)	42

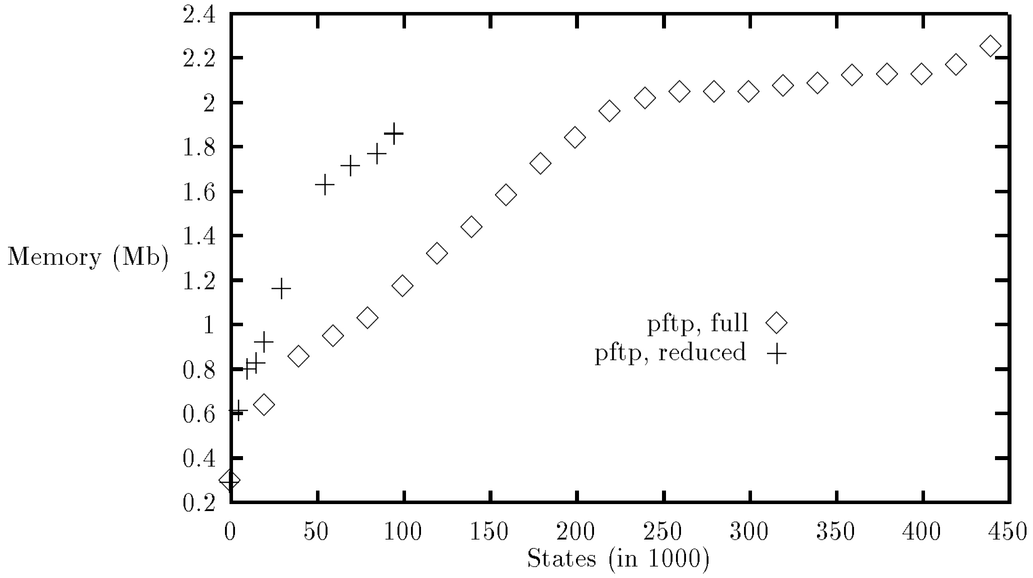

Figure 6: pftp, spin & GETS, memory behaviour

We achieve here a compression rate of 24 for the NOREDUCE 'd mode, and about 7 for the REDUCE 'd mode. Note also that, in REDUCE 'd mode, the difference in time performance is a factor of about 4, but we store twice at many states in the store!

The static part of the GETS is only about 135k.

The following figures represent the evolution in the memory and time requirements of the computation as a function of the number of states for the GETS-based examples. The time results are also represented for SPIN: in this case, the use of memory is always linear as a function of the states [5].

The memory consumption behavior represented in figure 6 is quite interesting. The use of memory isn't linear, but presents a bit of a quirk. This means that the use of memory improves at higher capacities in this case. Both the REDUCE 'd and non-REDUCE 'd explorations present the same phenomenon, although it is more distinct in the non-REDUCE 'd case. Predictably, reducing the number of states has removed some opportunities for sharing.

The time behaviors are interestingly enough almost linear in both cases (fig. 7) and also for the SPIN versions (fig. 7). The linearity is somewhat surprising, since we could have expected the quirk to show up in the time measurements. SPIN 's behavior is also linear in terms of the number of states, which means that, in general, the rate of discovery of new states tends to remain constant. This rule also applies to the GETS version.

5.5 gsm

This last example is not a "classic" of the SPIN distribution. It is a small, simplified model of a call set-up for a mobile in a GSM system. The state vector is 144 bytes wide, the maximum

[5] variations can be due to the memory used by the stack, but this one tends to be allocated early in the execution.

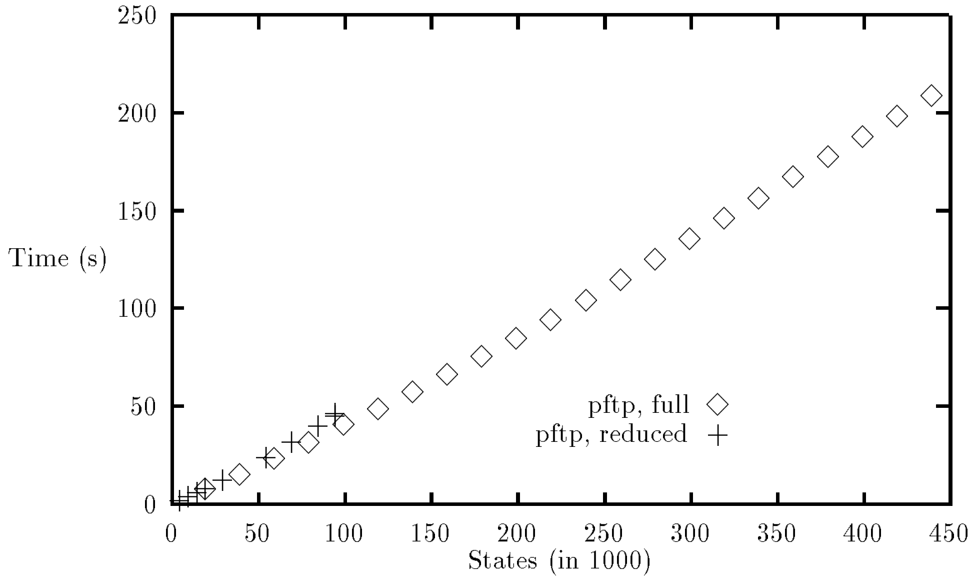

Figure 7: pftp, spin & GETS, time behaviour

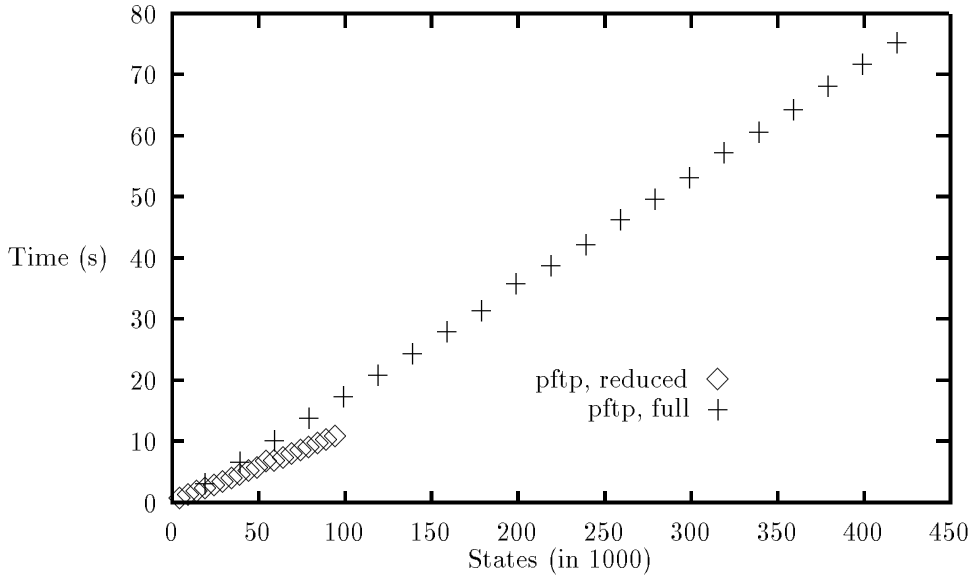

Figure 8: pftp, pure spin, time behaviour

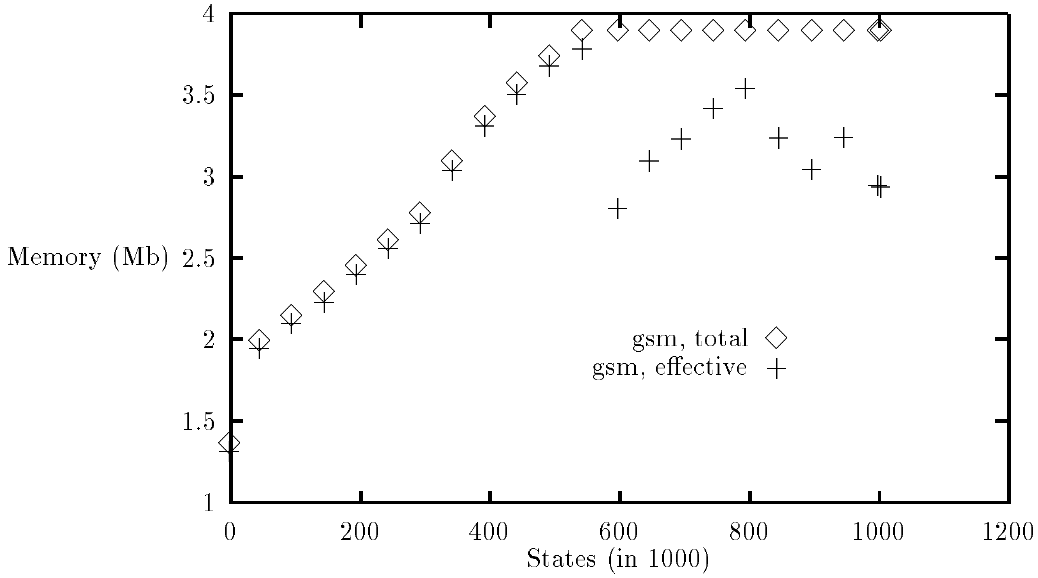

Figure 9: gsm, GETS, memory behavior

depth reached in a REDUCE'd search is 196999. The GETS has 20 layers, and the largest domain size is 16384.

	states	memory (M)	time (s)
RS, SPIN	1004630	140.267	251.780
RS, GETS	1004630	4.022(GETS) + 6.4(SPIN)	634.92

We see that the compression factor achieved here was over 33, with an execution time less than the double, and twice the states, since we are in REDUCE'd mode. The runtime behavior is even more interesting. The memory vs. states graph (fig. 9) shows that halfway through the exploration, the program had allocated all the storage it required, and the rest of the time was spent recycling storage. The wide variations in the use of memory do not however show in the time vs. states graph (fig. 10), which is, once again mostly linear.

For a comparison, a comparable [6] bitstate search gave the following results [7], after 192s:

```
./pan -m200000 -w25
(Spin Version 2.8.6 -- 17 May 1996)
        + Partial Order Reduction

Bit statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
```

[6] at least in terms of the amount of storage used.

[7] The depth reached in this case is indeed greater than a full search's; this is not unreasonable, since the greatest depth may be reached through a different path.

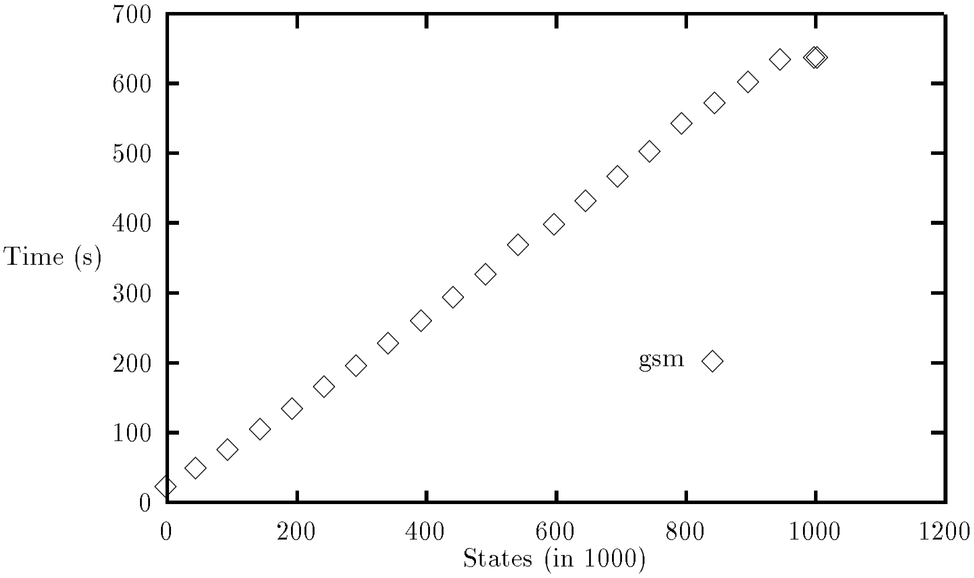

Figure 10: gsm, GETS, time behaviour

```
        invalid endstates       +

State-vector 144 byte, depth reached 197683, errors: 0
1.00088e+06 states, stored
6.15347e+06 states, matched
7.15435e+06 transitions (= stored+matched)
5.56773e+07 atomic steps
hash factor: 33.5248 (expected coverage: >= 98% on avg.)
(max size 2^25 states)

1.49727e+08     equivalent memory usage in bytes (stored*vector + stack)
1.55943e+07     actual memory usage

...
```

We see that the bitstate mode search does indeed give us a fairly good approximation in a good time. It is actually significantly better than the 98% expected coverage (20000 vs. 4000 states) would lead us to know. Of course, in most practical cases, we wouldn't have the full search results.

5.6 Observations

Those few experiments have confirmed some observations we had already made in a different context, and given new insights on the use of GETS.

Compression It is possible to achieve compression of the state space with GETS, for typical SPIN applications. This is a non-trivial statement: GETS are a compression technique which could very well result in a stored space larger than the number of bits required to record the plain data, according to Hamming.

Evolution of gains Gains increase as state space get larger. In earlier work, not directly related to SPIN we had observed compression factor of over 100.

Time There is a performance penalty in the use of GETS. The run time cost appears also to decrease as state spaces become larger.

Partial Orders Compression is achieved even in the case of a REDUCE'd search. The highest reported compression factor achieved here were actually realized in such a search. This would tend to show that the type of regularity exploited in GETS is distinct from what "partial order" techniques exploit. This is worthwhile to note because a different behavior had been observed with ROBDDs [Visser95].

Sample Set All our examples are "vintage PROMELA", namely processes communicating through asynchronous communication channels. The success we experience with such models bodes well for the possibilities of significant compression rates with GETSs in general.

5.7 Discussion

Where do GETS fit in the SPIN verification philosophy? Their runtime cost, plus the added cost of generating an encoding do not make them a convenient replacement for either full state or bit state search modes. They would have to be viewed in a complementary way. The other search modes are the most appropriate to start debugging a specification, when the probability of finding an error is high and the duration of the runs low. GETS-based search should come at the end, to try to achieve a "last", complete run, if it cannot be done in full state, and if we do have enough memory to do it, based on estimates of achievable compression and of the number of states (from bit state runs).

The early runs could also give critical information to define the clustering and the domain coding of GETS layers. The automation of such a procedure is possible.

Finally, as we said earlier, we make no claim at this stage that all the algorithms available in SPIN in full state search can be supported as are. SPIN tends to store "marking" information with the state, which can be later modified *in situ*. Such a technique cannot be applied here, but rather would require the manipulation of a another GETS when exploring loops. SPIN's current philosophy is to mark data on the stack to avoid multiple copies of the state space, which was the case originally. We do not have such concern, since multiple copies introduce an opportunity for sharing. GETSs thus give another perspective on the classical verification problems, one for which SPIN's current solutions may not be entirely adequate.

5.8 SPIN's Compression

To make this comparison more complete, we should apply clustering to "vintage" SPIN to see which benefits we could derive from it. However, SPIN already has an experimental compression mode for the state vector, called "collapse"[8]. The principle is quite elegant, but straightforward. For each proctype, each state configuration is memorized and identified. What is stored in the global state vector is no longer the whole information, but the sequence of identifiers for each proctype configuration. Identifiers have to be a (compile-time) fixed multiple of 8 bits, from one to four.

The effect of collapsing on the gsm example are quite interesting. The amount of memory required was reduced from 140Mbytes (in 271s) to 38Mbytes (in 281s). The effective size of state vector was 14 bytes, down from 144! While the GETS-based results remain better, the compression rate achieved has dropped to about 7.8 . The collapsing also has the advantage of being computed "on-the-fly", in true SPIN philosophy.

One could wonder if it might be worthwhile to combine GETS and collapse compression. However, a quick computation shows that only half the storage space is used by the full state vector, while the other half is used to store the unique proctype configurations. We would have to code both to achieve any improvement over our previous results.

[8]the existence of this mode was pointed out to the author by G. Holzmann.

6 Conclusion

We have presented a data structure optimised for a compact encoding of state spaces, and shown how well it performs in the context of SPIN. Our results show that there are major memory gains when a GETS is used to store the state space, at the cost of a significant increase in execution time. The gains are not predictable *a priori*. The execution times tend to be linear in our observations.

The use of GETS is also not entirely compatible with SPIN, in its current instance. They require tighter control over the real domains of variables, and the possibility of clustering variables together. On the algorithmic side, we haven't implemented all the algorithms supported by SPIN in full state search mode. While some could probably be easily adapted, it seems more appropriate to reengineer them to take advantage of the possibilities of GETSs, including the fact that replication is cheap in terms of storage space.

Acknowledgments Our own original GETSs implementation was done by F. Gagnon, with improvements contributed over time by the author and J. Collette. Thanks to Prof. M. Ferguson for pointing out the Hamming argument about compression. Credit is due to D. Zampuniéris who gave us the initial inspiration to apply his Arbres Partagés to reachability analysis.

References

[Gagnon et al.95] Gagnon (F.), Grégoire (J-Ch.) et Zampuniéris (D.). – Sharing trees for "on-the-fly" Verification. *In : Proceedings of FORTE'95.*

[Holzmann et al.94] Holzmann (G.) et Peled (D.). – An improvement in formal verification. *In : Proceedings of FORTE'94.*

[Holzmann91] Holzmann (G.). – *Design and validation of computer protocols.* – Prentice Hall Software Series, 1991.

[Visser95] Visser (W.). – Memory efficient state storage in spin. *In : Proceedings of the 1995 SPIN Workshop.* – URL: http://netlib.att.com/netlib/spin/ws95/spin95_abstracts.html.

[Zampunieris et al.95] Zampuniéris (D.) et Le Charlier (B.). – Efficient handling of large sets of tuples with sharing trees. *In : Proceedings of the Data Compression Conference (DCC'95).*