Appears in: Computer Aided Verification, Proc, Int. Workshop, Elounda, Crete, Lecture Notes in Computer Science, Vol. 697, Springer-Verlag, June, 1993, pp. 59-70.

Reliable Hashing without Collision Detection*

Pierre Wolper and Denis Leroy

Université de Liège, Institut Montefiore, B28, 4000 Liège Sart Tilman, Belgium.
Email : {pw,leroy}@montefiore.ulg.ac.be

Abstract. Thanks to a variety of new techniques, state-space exploration is becoming an increasingly effective method for the verification of concurrent programs. One of these techniques, hashing without collision detection, was proposed by Holzmann as a way to vastly reduce the amount of memory needed to store the explored state space. Unfortunately, this reduction in memory use comes at the price of a high probability of ignoring part of the state space and hence of missing existing errors. In this paper, we carefully analyze this method and show that, by using a modified strategy, it is possible to reduce the risk of error to a negligible amount while maintaining the memory use advantage of Holzmann's technique. Our proposed strategy has been implemented and we describe experiments that confirm the excellent expected results.

1 Introduction

The sceptic often dismisses state-space exploration as a simple-minded, brutal, and hence ineffective concurrent program verification technique. Indeed, this technique is based on the straightforward idea of exploring all possible behaviors of the concurrent program and relies more on raw computing power than on intricate mathematics. Nevertheless, its effectiveness on a large class of problems is becoming more and more apparent as tools are being developed [BCD85, RRSV87, Hol91]. Moreover, combining the mere powerful methodicalness of state-space exploration with even limited subtlety can yield very impressive results.

There are two main ways to use computing power more effectively for state-space exploration. The first is to tackle directly the central problem of this technique: the huge number of states of most systems. This is done by showing that, under some conditions, one can reliably analyze a system with many fewer states. Examples of this approach are abstraction techniques [Wol86, CGL92, BBLS92] and "partial-order" techniques [Val90, GW91a, GW91b, HGP92, McM92]. The former replace the system to be analyzed by a simpler one, the latter attempt to avoid the explosion of the state space due to the modelling of concurrency by interleaving.

* This work was supported by the Esprit BRA action REACT and by the Belgian Incentive Program "Information Technology" - Computer Science of the future, initiated by the Belgian State - Prime Minister's Office - Science Policy Office. The scientific responsability is assumed by its authors.

The second approach is to squeeze as much performance as possible out of the technique. "On the fly" methods are a step in this direction [Hol85, VW86, JJ89, FM91, CVWY92]. They are based on the observation that for many state-exploration based verification problems including deadlock detection, temporal logic model-checking and testing bisimulation equivalence, one just needs to visit the entire state-space without ever having to store it entirely for further use. This allows substantial improvements in the amount of memory required to implement the search through the state-space. Indeed, if as usual one uses a depth-first search, one only needs to store the current search stack and, in order to avoid duplicate work, a table containing the states that have already been visited.

Hashing without collision detection is an idea introduced by Holzmann for minimizing the amount of memory necessary for keeping the table of visited states [Hol88, Hol91]. The table is a table of bits all initially set to 0. To add a state to the table, one hashes the state description into an address in the table and sets the bit at this address to 1. To determine if a state is in the table, one applies the hash function to the state and checks whether the bit appearing at the computed address is 1. Of course, the drawback of the method is that the hash function can compute the same address for distinct states and hence one can wrongly conclude that a state has been visited, whereas one has actually encountered a hash collision. It has been argued that this is not too serious because the probability of collision can be kept small if the table is not too full. Moreover, repeating the search with different hash functions can further reduce the probability of collisions. Alternatively, as advocated by Holzmann, one can use two hash functions and store two bits in the table for each state. One then only concludes that a state is present in the table if the bits at both the computed addresses are set to 1.

In this paper, we first look at the simple analysis of the probability of collisions in the scheme proposed by Holzmann. Our conclusion which is confirmed by experiments with Holzmann's SPIN system is that, even if the ratio of number of states to table entries is less than 1%, as recommended for good results in [Hol91], the probability of collision is unacceptably high and states are missed in such searches. We then analyze two variants of the method. The first is simply to increase further the number of hash functions. The second is to increase the number of bits stored in the table and to use a collision resolution scheme for the values stored. Then, assuming a fixed size memory, we compute optimum values for the number of hash functions in the first scheme and for the number of bits stored in the table in the second scheme. For these optimal values, the analysis shows that both techniques yield essentially the same result which is dramatically better than what can be achieved with Holzmann's scheme. Collision probabilities of 10^{-3} or even 10^{-6} can easily be obtained while memory use is of the order of 40–100 bits per stored state. Compared with Holzmann's scheme, collision probability is thus drastically reduced whereas memory use is essentially unchanged (assuming a ratio of number of states to table size of no more than 1%).

We then discuss the pragmatic consequences of our analysis and make a practical recommendation. We have implemented our recommendation in the context of the SPIN system and our experiments perfectly confirm the theoretical analysis.

2 State-Space Exploration and its Memory Requirements

The exact problem we consider is the following. We are given a program P represented by an initial state s_0 and a function succs(s) which yields the set of immediate successors of any sate s. The problem is to explore all reachable states of the program in order to check for some property. Since it is irrelevant to our present purpose, we ignore the property to be checked and focus on the search process. The algorithm used for the search is described in Fig. 1 where the variable *Stack* denotes a stack structure and the variable T denotes a lookup table.

```
1. Initialize: Stack := [s0]; T := {s0};
2. Loop: while Stack ≠ ∅ do
           begin
              s := pop(Stack);
              for all s' ∈ succs(s) do
                 begin
                 if s' ∉ T then
                    begin
                    insert s' in T;
                    push s' onto Stack;
                    end
                 end
           end
```

Fig. 1. Search algorithm

The memory requirements of this algorithm are thus a stack and a table. The stack is sequentially accessed and has its length bounded by the depth of the state-space graph. It is usually not the crucial element from a memory usage point of view. On the other hand, the table has to allow direct access to its elements and will eventually contain the whole state-space. It is thus essential to carefully choose the corresponding data structure.

The natural choice is a hash table. However, the state descriptors that have to be stored in this table are often rather large (of the order of a 100 bytes). Thus, on a typical computer with 64 Mbytes of memory, one is limited to only several hundreds of thousand states.

To improve on this rather drastic limit, Holzmann has suggested the following strategy. One uses a table of bits T such that that initially $T[i] = 0$ for all i. To access this table, one uses a hash function h, and storing a state s in the table amounts to setting $T[h(s)]$ to 1. To determine if a state s is not in the table, one then simply checks that $T[h(s)]$ is still 0. This scheme is great if the hash function is perfect (produces no collisions). Since this is not a reasonable assumption, Holzmann advocates the following use of the method.

First, rather than using one hash function, one uses two hash functions h_1 and h_2 (this is actually what is implemented in the SPIN system [Hol91]). To insert a state s, one sets both $T[h_1(s)]$ and $T[h_2(s)]$ to 1. Furthermore, one only concludes that a state is present if both $T[h_1(s)]$ and $T[h_2(s)]$ are 1. Second, one only relies on the result if the table remains less than 1% full. Precisely, Holzmann argues that in this case, the fraction of the state space that is explored is large (say 99%). On our prototypical 64 Mbyte computer, this now allows storing a few million states. Moreover, the method degrades gracefully when the number of states increases beyond what is allowable. Indeed, as the table fills up, the probability of missing part of the state space increases, but the search can continue and still provide useful information.

Unfortunately, even exploring 99% of the state space does not guaranty that all errors in the protocol will be discovered. Our goal is to determine if Holzmann's method cannot be modified in such a way that the probability of missing even a single state is negligible (say 10^{-3} or even 10^{-6}). For doing this, the next section turns to a probabilistic analysis.

3 An Analysis of Hashing without Collision Detection

Doing a probabilistic analysis of Holzmann's scheme is very simple. We first deal with the case of a single hash function and use the following notation:

- the size (number of possible entries) in the table is t,
- the number of states to be inserted in the table is n.

Assuming uniformity, the probability of no collision p_{nc} is

$$p_{\mathrm{nc}} = \frac{t!}{t^n (t-n)!} \tag{1}$$

For n close to t, this is clearly very close to 0. Furthermore, even for $n \ll t$ the situation is not favorable since one obtains from (1)

$$p_{\mathrm{nc}} \approx e^{-\frac{n^2}{t}}. \tag{2}$$

Thus, for the probability of no collision to be close to 1 (and hence for the method to be reliable), one needs $\frac{n^2}{t}$ to be close to 0 (say 10^{-3}) and hence t must be larger than $10^3 n^2$. For instance, if $n = 10^6$, t must be of the order of 10^{15} which is quite unrealistic.

Let us now see if using two hash functions (as described in the previous section) gives better results. With an eye towards generalizing this idea, we immediately analyze the case of k hash functions. There are t^{kn} ways in which the k hash functions can map the n states into a table of size t. There are

$$t^k$$

ways in which the first element can be inserted into the table. For the second element, there are approximately

$$t^k - k^k$$

possibilities that do not lead to a collision. The approximation made is to assume that, for the first element, the k hash functions have yielded distinct values. This is reasonable since, on average, the number of table entries set to 1 after the insertion of the first state is very close to k. Carrying on with the same approximation, we take the number of 1 entries after the insertion of i elements to be ik. The number of ways of inserting the element $i+1$ without collision is thus

$$t^k - (ik)^k$$

and the probability of no collision is

$$p_{\mathrm{nc}} = \frac{\prod_{i=0}^{n-1}(t^k - i^k k^k)}{t^{kn}} \tag{3}$$

for $nk \ll t$, one can obtain

$$p_{\mathrm{nc}} \approx e^{-\frac{k^k n^{k+1}}{t^k}} \tag{4}$$

If we take $k = 2$, then for p_{nc} to be close to 1, $\frac{4n^3}{t^2}$ should be close to 0 (e.g. 10^{-3}). Thus for $n = 10^6$, t must be of the order of $6 \cdot 10^{10}$, much smaller than what was required with a single hash function, but still impractically large.

The natural question to ask at this point is why stop at $k = 2$. One expects that larger values of k will push the probability of no collision closer to 1, though at some point the benefit will disappear because the table will fill up too fast. Let us thus try to find the optimal value of k. For doing this, we fix p_{nc} to a value acceptably close to 1, fix the size of the table t to be equal to the available memory M (in bits) and determine for which value of k, the number of states that can be stored is maximal. Fig. 2 shows the number of states that can be stored with a probability of collision of 10^{-3} ($p_{\mathrm{nc}} = 1 - 10^{-3}$) as a function of k and for table sizes ranging from 1 Mbit (128KBytes) to 1Gbit (128MBytes). The next figure (Fig. 3) also shows the number of states that can be stored as a function of k, but this time for a fixed memory size (100Mbit) and for a probability of collision ($1 - p_{\mathrm{nc}}$) ranging from 10^{-1} to 10^{-6}.

From these figures, we see that there is an optimal value of k that, for a large range of memory sizes and collision probabilities, can be taken to be $k = 20$. For this value, memory use is of the order of 60-100 bits per state. Note that for the optimal value, the table is approximately 35% full. This might seem

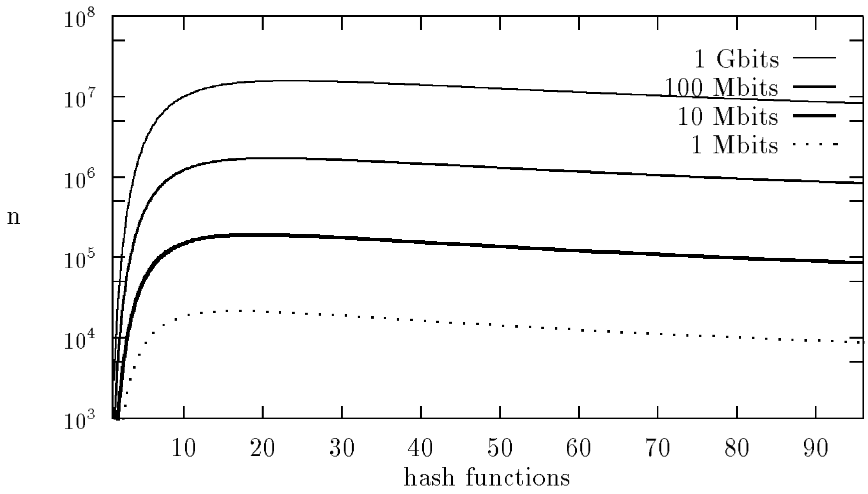

Fig. 2. States stored with $p_{\rm nc} = 1 - 10^{-3}$

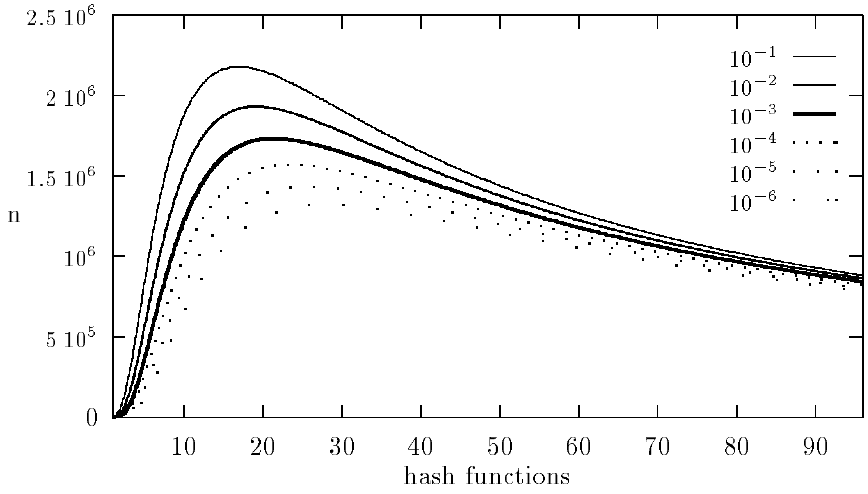

Fig. 3. States stored with M = 100 Mbits

incompatible with the assumption that $nk \ll t$, but a more careful look shows that $nk = 0.35t$ is actually sufficient for our analysis to be valid. Moreover, most of our approximations are pessimistic (i.e., overestimate the probability of collision) and hence are safe.

So, the conclusion is that one should use, not 2 hash functions, but 20. This is indeed much preferable to Holzmann's approach when the size of the state space is less than 1/100th of the number of available memory bits. One then obtains full coverage with a high probability which was not at all the case with Holzmann's method. However, for state spaces substantially larger than the safe maximal values we have computed, the table will fill up more quickly for a larger value of k, and thus coverage (the fraction of the state-space actually visited) might be better for a small value of k, though it will be very far from 100%. Finally, computing 20 hash functions is quite expensive and will substantially slow down the search. In the next section we thus present a method that provides similar benefits to multiple hash functions, which we will from now on call *multihashing*, but without its computational overhead.

However, before doing this, we discuss an alternative to the scheme analyzed in this section that might have occurred to the reader and appeared to be preferable. The idea is, rather than using k hash functions which compute an address in a single table, to use k hash functions that compute addresses in k distinct tables of size t/k. This amounts to partitioning the hash table in k equal parts and ensuring that the range of each hash function is limited to one of these parts.

If we take a second look at the analysis appearing above, we notice that partitioning the hash table does not require it to be modified much. Indeed, the number of ways of inserting element $i+1$ without a collision is

$$(t/k)^k - (i)^k$$

Again, this is an approximate number since we have assumed the number of elements in each part of the table to be exactly i after the insertion of the ith element. From this, we obtain that

$$p_{\mathrm{nc}} = \frac{\prod_{i=0}^{n-1}((t/k)^k - i^k)}{(t/k)^{kn}} \tag{5}$$

which turns out to be identical to (3). Actually, the only difference between the two cases is that our approximation underestimates the probability of no collision slightly more in the case of a nonpartitioned table than in the case of a partitioned table. The nonpartitioned table is thus preferable.

4 An Alternative Scheme

As we have seen in the previous section, to obtain a small probability of collision with a single hash function, one needs an impractically large table, for example one with 10^{15} entries. However, one can easily simulate such a table when it only contains a limited number of entries. The idea is to compute the address in the large table and actually store it in a much smaller hash table, but this time with a collision resolution scheme. The small hash table thus only needs to have a size of the order of the number of entries that will actually be stored. For example, one could use a hash function to compute 64 bit strings from the state descriptions, and then use a standard hash table to store these strings. This would thus only require approximately $64n$ bits to store n states.

Let us analyze the probability of no collision in this scheme. We assume that we have a table of size t in which entries k bit long (obtained by a hash function from the state descriptor) are stored. Assuming that the overhead required to resolve collisions is negligible, this means that the memory used by our table is of size $tk = M$ bits. This approach simulates hashing without collision detection with a table of size 2^k, and thus using (2), we have that

$$p_{\mathrm{nc}} \approx e^{-\frac{n^2}{2^k}}. \tag{6}$$

If we fix the size M of the available memory and the acceptable probability of no collision, the maximal number of states n that can be stored is obtained when $n = t$ (the hash table is full) and satisfies

$$p_{\mathrm{nc}} \approx e^{-\frac{n^2}{2^{M/n}}}.$$

The following two figures respectively give the values of the optimal n and k as a function of the probability of collision $(1 - p_{\mathrm{nc}})$ and the size of the available memory. From these figures, one easily concludes that taking, for instance,

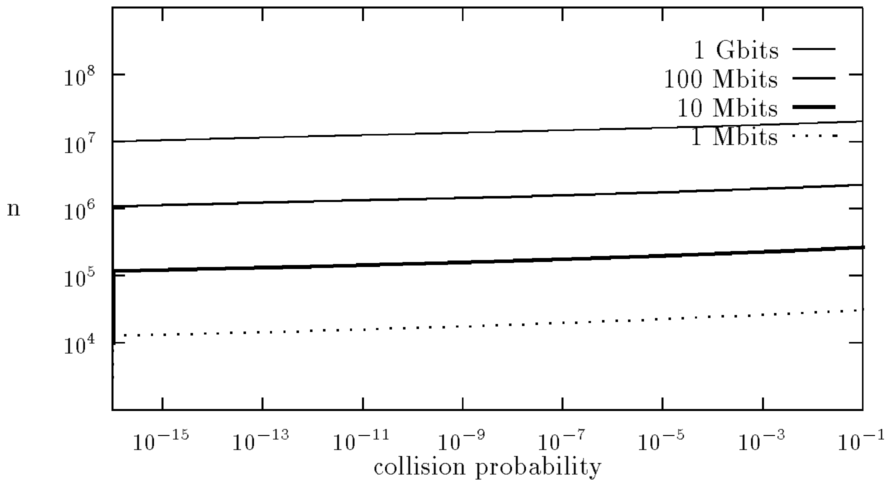

Fig. 4. Optimal n

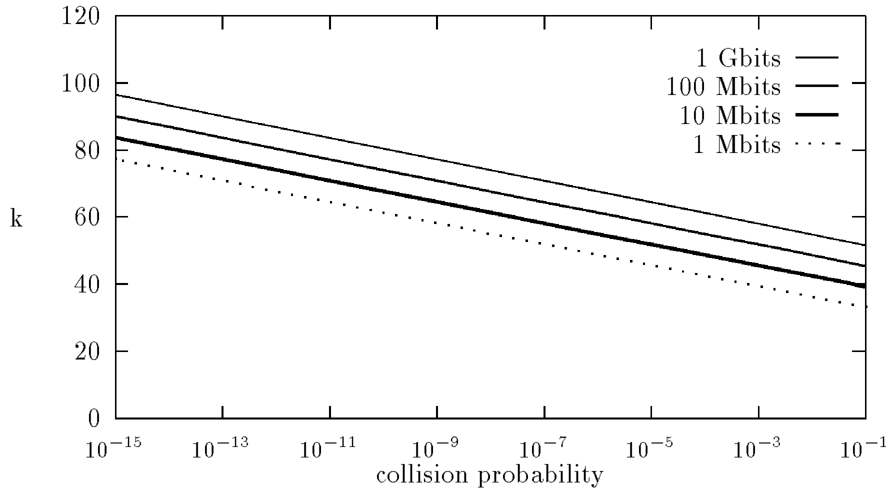

Fig. 5. Optimal k

$k = 64$ is quite sufficient to guarantee a very low probability of collision over the range of memory sizes we have considered. With this value, we obtain a memory use per stored state which is at least as low as with the scheme described in the previous section. Moreover, the present scheme only requires the computation of less than 100 hash bits (the 64 bits to be stored and the address in the hash table) as compared to the approximately 500 needed for the same reliability with the multihashing scheme.

One way to understand the method we have just proposed is to view the hash function applied to the state-description as a compaction function, albeit an unreliable one. We will thus call this scheme *hashcompact*. Note that compared to other compaction proposals (see for instance [HGP92]) it yields a much greater reduction in size, but at the cost of a small probability of error.

5 Recommendation and Discussion

Which of the two schemes *multihash* and *hashcompact* do we recommend ? Fig. 6 compares the number of states that can be stored with a 10^{-3} probability of error when using both schemes optimally. The comparison is to the advantage of hashcompact. This is the first reason for which we recommend this scheme. The second is that it requires less computation than multihash and, the last is that it stays reliable (very small probability of collision) until the memory fills up. This to be contrasted to the behavior of multihash for which the probability of collision increases gradually as the memory fills up. Thus hashcompact warns the user when too little memory is available for an exhaustive search. When

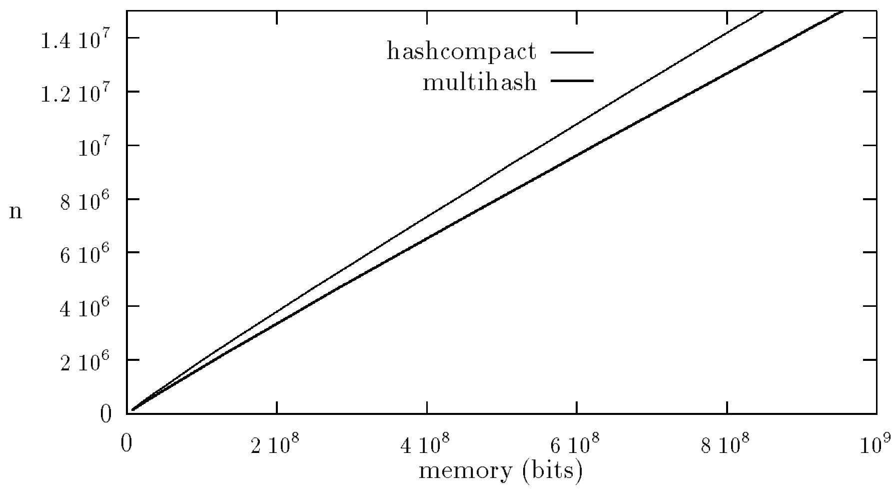

Fig. 6. Comparison of multihash and hashcompact

this actually occurs and one wants to optimize coverage, using a scheme close to Holzmann's original proposal is probably best. However, since this amounts to randomly limiting the search one might also consider alternative ways of doing this.

In practice, compacting the state descriptors into a 64 bit hash value which is then stored reliably is sufficient to ensure a probability of collision lower than 10^{-3} over the range of memory sizes that one can expect fo find on present day machines. This is what we recommend implementing.

6 Implementation

We have implemented the hashcompact scheme in the context of the SPIN system [Hol91]. The results confirm our expectations. The table in Fig. 7 illustrates this for two protocols DTP (with a large channel size) and PFTP. It shows that hashcompact does provide full coverage (the same as an exhaustive search storing full state descriptors) with less than 100 bits per state, whereas Holzmann's scheme (bitstate) fails to provide full coverage even when using close to 1000 bits per state.

7 Conclusions

On the fly verification techniques have made it possible to reduce the memory requirements of verification systems to those of a simple state-space search: a

Protocol	Algorithm	Stored States	Memory use	States missed ?
DTP	exhaustive	427,567	73 Mbytes	No
	bitstate	427,446	33.8 Mbytes	Yes
	hashcompact	427,567	3.6 Mbytes	No
PFTP	exhaustive	409,257	34 Mbytes	No
	bitstate	405,969	33.9 Mbytes	Yes
	hashcompact	409,257	3.6 Mbytes	No

Fig. 7. Experimental results

stack and a visited-state table. Since the table needs to be randomly accessed and is the largest of the two structures, it is essential to find the best possible data structures to implement it.

Our starting point was the neat idea of hashing without collision detection used by Holzmann in his SPIN system. Motivated by the desire to optimize this method which is inherently unreliable, we have concluded that a very simple hash compaction scheme could yield comparable storage efficiency and high (though not absolute) reliability.

All the techniques used in this paper are very standard. Our contribution is to show that by using them correctly, one can obtain a very substantial reduction in the space needed to store the visited-state table of state-exploration verifyers. The only price is a small probability of error that can be essentially reduced at will. We have also made a recommendation for implementation our method which is at the same time simple and effective and have shown experimental results.

Finally, the compaction scheme we have proposed is fully compatible with other memory reduction strategies such as state-space caching [GHP92]. To end with a bold statement, let us say that the combination of these techniques has shifted the bottleneck in state-exploration systems from storing the visited-state table to the time needed for completing the search.

Acknowledgements

We thank Costas Courcoubetis and Mihalis Yannakakis for discussions on the subject of this paper.

References

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proc. 4th Workshop on Computer Aided Verification*, Montreal, June 1992.

[BCD85] M. Browne, E.M. Clarke, and D.L. Dill. Automatic circuit verification using temporal logic: Two new examples. In *IEEE Int. Conf. on Computer Design: VLSI and Computers*, Port Chester, October 1985.

[CGL92] E. M. Clarke, O. Grümberg, and D. E. Long. Model checking and abstraction. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 343–354, Albuquerque, New Mexico, January 1992.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[FM91] J.C. Fernandez and L. Mounier. On the fly verification of behavioural equivalences and preorders. In *Proc. 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 181–191, Aalborg, July 1991.

[GHP92] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. 4th Workshop on Computer Aided Verification*, Montreal, June 1992.

[GW91a] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proc. 6th Symp. on Logic in Computer Science*, pages 406–415, Amsterdam, July 1991.

[GW91b] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 332–342, Aalborg, July 1991. Springer-Verlag.

[HGP92] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, June 1992. North-Holland.

[Hol85] G. J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(12):2413–2434, 1985.

[Hol88] G. Holzmann. An improved protocol reachability analysis technique. *Software Practice and Experience*, 18(2):137–161, February 1988.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, 1991.

[JJ89] C. Jard and T. Jeron. On-line model-checking for finite linear temporal logic specifications. In *Automatic Verification Methods for Finite State Systems, Proc. Int. Workshop, Grenoble*, volume 407, pages 189–196, Grenoble, June 1989. Lecture Notes in Computer Science, Springer-Verlag.

[McM92] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. 4th Workshop on Computer Aided Verification*, Montreal, June 1992.

[RRSV87] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in xesar of the sliding window protocol. In *Proc. IFIP WG 6.1 7th Int. Conf. on Protocol Specification, Testing and Verification*, pages 235–250, Zurich, 1987. North Holland.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Rutgers, June 1990. Springer-Verlag.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. Symp. on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming*, pages 184–192, St. Petersburgh, January 1986.

This article was processed using the $\LaTeX$ macro package with LLNCS style