State Space Caching Revisited

Patrice Godefroid[*]
Université de Liège
Institut Montefiore B28
4000 Liège Sart-Tilman
Belgium

Gerard J. Holzmann
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
U.S.A.

Didier Pirottin[*]
Université de Liège
Institut Montefiore B28
4000 Liège Sart-Tilman
Belgium

Abstract

State space caching is a state space exploration method that stores all states of just one execution sequence plus as many previously visited states as available memory allows. So far, this technique has been of little practical significance. With a conventional reachability analysis, it allows one to reduce memory usage by only two to three times, before an unacceptable exponential increase of the run-time overhead sets in. The explosion of the run-time requirements is caused by redundant multiple explorations of unstored parts of the state space. Indeed, almost all states in the state space of concurrent systems are typically reached several times during the search. There are two causes for this: firstly, several different partial orderings of statement executions can lead to the same state; secondly, all interleavings of a same partial ordering of statement executions lead to the same state.

In this paper, we describe a method to completely avoid the effects of the second cause given above. We show that with this method, most reachable states are visited only once during the state space exploration. This makes for the first time state space caching a very efficient verification method. We were able, for instance, to completely explore a state space of 250,000 states while storing simultaneously no more than 500 states and with only a three-fold increase of the run-time requirements.

1 Introduction

Memory is the main limiting factor of most conventional reachability analysis algorithms. These verification algorithms perform an exhaustive exploration of the state space of the system being checked. This exploration amounts to simulating all possible behaviors the system can have from its initial state and storing all reachable states. To avoid significant run-time penalties for disk-access, reachable states can only be stored in a randomly accessed memory, i.e. in the main memory available in the computer where the algorithm is executed. Therefore the applicability of these verification algorithms is limited by the amount of main memory available. Typically, it only takes a few minutes of run-time to fill up the whole main memory of a classical computer.

[*]The work of these authors is partially supported by the European Community ESPRIT BRA project SPEC (3096) and by the Belgian Incentive Program "Information Technology" – Computer Science of the future, initiated by Belgian State – Prime Minister's Service – Science Policy Office. The scientific responsibility is assumed by its authors.

During the search, once states have been visited they are stored. Storing states avoids redundant explorations of parts of the state space. If a stored state is encountered again later in the search, it is not necessary to revisit all its successors. It is worth noticing that states that are reached only once during the search do not need to be stored. Storing them or not would not change anything about the time requirements of the method. Of course, it would be preferable not to store them in order to decrease the memory requirements, but with a conventional algorithm it is virtually impossible to predict if a given state will be visited once or more than once.

Typically, almost all states in the state space of concurrent systems are reached several times during the search. There are two causes for this:

- From the initial state, the exploration of different partial orderings of statement executions of the system can lead to the same state.
- From the starting state, the exploration of all interleavings of a same partial ordering of statement executions lead to the same state.

In this paper, we give a way to completely get rid of the second cause given above. Then we study the impact of this new technique on real-protocol state spaces. In many cases, when using this method, most of the states are now reached *only once* during the search.

Sadly, it is not possible to determine which states are visited only once before the search is completed. However, the risk of double work when not storing an already visited state becomes very small since the probability that this state will be visited again later during the search becomes very small. This enables us not to store most of the states that have already been visited without incurring too much redundant exploration of parts of the state space. The memory requirements can thus strongly decrease (more than 100 times) without seriously increasing the time requirements (only 3 or 4 times). This makes possible the complete exploration of very large state spaces (several tens of million states) that can not be explored exhaustively by any other known method. With this technique, time becomes the main limiting factor.

In the next Section, we recall the principles of state space caching and present some results obtained with this method for the verification of four real-protocols. Then we show how this verification method can be substantially improved by the use of *"sleep sets"*. Sleep sets were introduced in [God90, GW91b]. In Section 3, we recall the basic idea behind sleep sets. Then, we give a new simple and efficient implementation of the sleep set scheme. We study properties of sleep sets and prove two new theorems. Section 4 presents and compares the results obtained with the state space caching method with and without the use of sleep sets. In Section 5, some suggestions to further improve the effectiveness of the method are investigated.

2 State Space Caching

State space exploration can be performed by a classical depth-first search algorithm, as shown in Figure 1, starting from the initial state s_0 of the system. The main data structures used are a *Stack* to hold the states of the current explored path, and a hash table

```
Initialize: Stack is empty; H is empty;
Search() {
   enter $s_0$ in H;
   push ($s_0$) onto Stack;
   DFS();
   }
DFS() {
   s = top(Stack);
   for all t enabled in s do {
       s' = succ(s) after t; /* execution of t */
       if s' is NOT already in H then {
          enter s' in H;
          push (s') onto Stack;
          DFS();
          }
       /* backtracking of t */
       }
   pop s from Stack
   }
```

Figure 1: Algorithm 1 — classical depth-first search

H to store all the states that have already been visited during the search. Algorithm 1 simulates all possible transitions sequences the system is able to perform. The exploration can be performed "on-the-fly", i.e. without storing the transitions that are taken during the search. This reduces substantially the memory requirements. Unfortunately, the number of reachable states can be very large and it is then impossible to store all these states in *H*.

However, it is well-known that a completely exhaustive state space exploration can be performed without the storage of any other part of the full state space than a single sequence of states leading from the initial state to the currently explored state, i.e. the *Stack* used in Algorithm 1. Such a search, termed "Type-3" or stack-search algorithm in [Hol90], reduces the memory requirements while still guaranteeing a complete exploration of any finite state space. This strategy was used in, for instance, the first Pan system [Hol81], and in the Pandora system [Hol84]. The problem is that, if an execution path joins a previously analyzed sequence in a state that is no more onto the stack, this search strategy will do redundant work. Hence the run-time requirements of this type of search go up dramatically. The result is that even state spaces that could otherwise comfortably be stored exhaustively become unsearchable with even the fastest implementations of a stack-search discipline.

A trade-off between these two strategies consists of storing all the states of the current path and storing as many other states as possible given the remaining amount of available memory. This strategy is called *state space caching* [Hol85]. It creates a restricted *cache* of selected system states that have already been visited. Initially, all system states encountered are stored into the cache. When the cache fills up, old states are deleted to accommodate new ones. This method never tries to store more states than possible in the cache. Thus, if the size of the cache is greater than the maximal size of the stack during the exploration, the whole state space can be explored without any problems.

We have implemented such a caching discipline in an efficient automated protocol validation system called SPIN[1] [Hol91], which includes an implementation of a classical search as described in Figure 1. The details of PROMELA, the validation language that SPIN accepts, can be found in [Hol91]. PROMELA defines systems of asynchronously executing concurrent processes that can interact via shared global variables or message channels. Interaction via message channel can be either synchronous (i.e. by rendez-vous) or asynchronous (buffered), depending on what type of channel is declared.

Experiments with our implementation were made on four sample real protocols:

1. PFTP is a file transfer protocol presented in Chapter 14 of [Hol91], modeled in 206 lines of PROMELA.
2. URP is the AT&T's Universal Receiver Protocol, modeled in 405 lines of PROMELA.
3. MULOG3 is a protocol implementing a mutual exclusion algorithm presented in [TN87], for 3 participants, modeled in 97 lines of PROMELA.
4. DTP is a data transfer protocol, modeled in 406 lines of PROMELA.

The results of our experiments with Algorithm 1 and different cache sizes are presented in Figure 4. All measurements were run on a SPARC2 workstation (64 Megabytes of RAM). Time is user time plus system time as reported by the UNIX system time command. The experiments were performed using a random replacement strategy (see Section 5).

The results show clearly that the number of stored states can be reduced by approximately two to three times without seriously affecting the run time. If the cache is further reduced, the run time increases dramatically.

These results confirm the ones presented in [Hol85, Hol87]. As first pointed out in [Hol87], whether a large reduction of the memory requirements without a significant blow-up of the time complexity can be achieved depends largely on the structure of the state space, which is protocol dependent and highly unpredictable. The conclusion from these early studies was that the effect of the state space caching discipline are too unpredictable to be useful in a general verification tool. Indeed, it is necessary to know how many states the full state space contains to find the optimal caching setup since the blow-up of execution time starts too soon, and is too steep. The results of these experiments were more recently confirmed in a series of independent experiments [JJ89, JJ91].

The critical point for a caching algorithm is the risk of double work incurred by joining a previously visited state that has been deleted from memory. This risk depends on the state space: if the states are reached several times during the search, the risk is greater than if they are reached only once. For the state spaces of the examples above, one can see in Table 1 that the number of transitions is about 3 times the number of states. This means that each state is, on average, reached 3 times during the search. The risk is too high. This is why this technique is not very efficient.

[1]The original version of SPIN can be obtained free of charge via email, for educational purposes. To get instructions, send an arbitrary one-line message to "netlib@research.att.com". The response is automated.

In the next section, we show how it is possible to strongly reduce the number of transitions that have to be explored during the search, which reduces the risk and makes state space caching manageable.

3 Sleep Sets

The classical depth-first search presented in Figure 1 explores all enabled transitions from each state encountered during the search. However, in case of concurrent systems, it is possible to explore all the reachable states of the state space *without* exploring systematically all enabled transitions in each state. This can be done by using *sleep sets*.

Sleep sets were introduced in [God90, GW91b] where it was shown that most of the state explosion due to the modeling of concurrency by interleaving can be avoided. The basic idea of this verification method was to describe the behavior of the system by means of partial orders rather than by sequences. More precisely, Mazurkiewicz's traces [Maz86] were used as a semantic model.

Traces are defined as equivalence classes of sequences. Given an alphabet Σ and a dependency relation $D \subseteq \Sigma \times \Sigma$, two sequences over Σ belong to the same trace with respect to D (are in the same equivalence class) if they can be obtained from each other by successively exchanging adjacent symbols which are independent according to D. For instance, if a and b are two symbols of Σ which are independent according to D, the sequences ab and ba belong to the same trace. A trace is usually represented by one of its elements enclosed within brackets and, when necessary, subscripted by the alphabet and the dependency relation. Thus the trace containing both ab and ba could be represented by $[ab]_{(\Sigma,D)}$. A trace corresponds to a partial ordering of symbol occurrences and represents all linearizations of this partial order. If two independent symbols occur next to each other in a sequence of a trace, the order of their occurrence is irrelevant since they occur concurrently in the partial order corresponding to that trace.

In a PROMELA program, dependency can arise between statements that refer to the same global objects, i.e. same global variables or same message channels. For instance, two write operations on a same shared global variable in two concurrent processes are dependent, while two concurrent read operations on the same object are independent since they can be shuffled in any order without changing the possible outcome of the read. Tracking dependencies between PROMELA statements is by no means a trivial point. We refer the reader to [HGP92] for a detailed presentation of that topic.

In the context of [God90, GW91b], sleep sets were one of the means used by an algorithm devoted to the exploration of at least one (sequence) interleaving for each possible trace (partial ordering of transitions) the concurrent system was able to perform. More precisely, the specific aim of sleep sets was to avoid the wasteful exploration of all possible shufflings of independent transitions.

Let us consider an example to illustrate the basic idea behind sleep sets. Consider a classical depth-first search and assume there are two independent transitions t_1 and t_1' from the current state s (see the top of the right part of Figure 2). Assume that transition t_1 is explored before transition t_1' and that t_1 leads to a successor state s'. When all immediate successor states of s' have been explored, the transition t_1 is backtracked and the depth-first

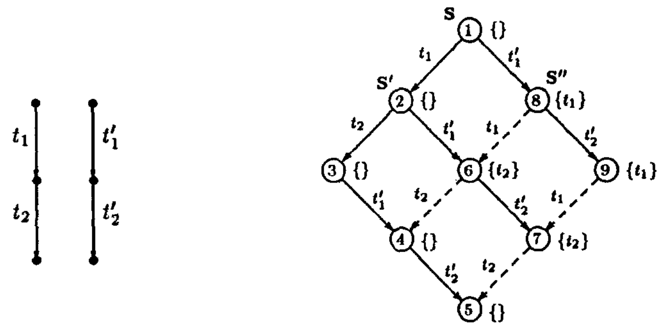

Figure 2: A concurrent system (left) and the exploration performed by Algorithm 2 (right)

Same as Algorithm 1 except:

- **A variable *Sleep* is added. Its initial value is {}.**
- **Instead of executing systematically all enabled transitions from each state, execute only all enabled transitions that *are not in Sleep*.**
- **Each time a transition *t* is executed from a state *s*, delete all transitions from *Sleep* that are dependent with *t* (the result gives the value of *Sleep* that has to be associated to the successor state *s'* of *s*).**
- **Each time a transition *t* that has led to a state *s'* is backtracked, restore the value of *Sleep* before the execution of *t*. If *s'* is not in *Stack*, then add *t* to *Sleep*.**

Figure 3: Algorithm 2 — depth-first search with sleep sets

search backs up to state s. Then t'_1 is executed from s, leads to a successor node s'' and the search goes on from s''. Since t_1 and t'_1 are *independent*, t_1 is still enabled in s''. But it is not necessary to explore transition t_1 from state s'' since the result of another shuffling of these independent transitions, namely the sequence $t_1t'_1$, has already been explored from s. In order to prevent the execution of t_1 in s'', we use sleep sets: we put t_1 in the sleep set associated to s''.

A sleep set is defined as a set of transitions. One sleep set is associated with each state s reached during the search. The sleep set associated with s is a set of transitions that are *enabled* in s but *will not be executed* from s. The sleep set associated with the initial state s_0 is the empty set.

Note that, in the previous example, if t_1 and t'_1 would have been dependent, then it would have been mandatory to explore both shufflings of t_1 and t'_1. (For example, the two shufflings of two write statements on a same global variable performed by two concurrent processes are dependent and leaves the system in two different states.)

Figure 3 shows how to introduce the sleep set scheme in the classical depth-first search algorithm. A single variable *Sleep* is added. The two last rules describe how to set and reset the value of *Sleep* properly during the state-graph traversal. The appropriate rule is applied each time a transition is executed or backtracked during the search.

A simple example of a state-graph traversal performed by Algorithm 2 is given in Figure 2. The system on the left is composed of two completely independent concurrent processes. For each state, the value of *Sleep* when that state has been added to the stack is given between braces beside the state. Dotted transitions are not explored by Algorithm 2.

Note that Algorithm 2 as presented above can be viewed as an efficient version of the procedure that was given in [GW91b] to compute sleep sets. Indeed, with this new version, it is no more necessary to store explicitly sleep sets on the stack as it was suggested in [GW91b]. It is sufficient to store only some information about sleep sets updates in order to restore the value of the sleep set before the execution of a transition when the transition is backtracked. (From our experience in designing several different versions of the sleep set scheme, implementing it as described above can imply a substantial speed-up in the sleep set computations.)

The following theorem ensures that all reachable states of the concurrent system are still visited by Algorithm 2.

Theorem 3.1 *All reachable states are visited by Algorithm 2.*

Proof:

Let s be a state reachable from the initial state s_0. Imagine that we fix the order in which transitions selected in a given state are explored and that we first run Algorithm 1 (depth-first search without sleep sets). Then, we run Algorithm 2 (depth-first search with sleep sets) while still exploring transitions in the same order. The important point is that the order used in both runs is the same, the exact order used is irrelevant. Let then S denote the spanning tree explored by Algorithm 1 and let S_s denote the part of S that contains all states from which the state s is reachable. Since s is reachable, S_s is nonempty and contains s (we do not prove here that a classical depth-first search visits all reachable states). Moreover, the leftmost path of S_s leads to s. We now prove that in the second run, i.e. when using Algorithm 2, the leftmost path of S_s is still explored.

Let $p = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots s_{n-1} \xrightarrow{t_{n-1}} s$ be this path. Since the order used in both runs is the same, p is the very first path of S_s that will be examined during both runs. The only reason for which it might not be fully explored (i.e. until s is reached) by the algorithm using sleep sets is that some transition t_i of p is not taken because it is in the sleep set associated to s_i. There are two possible causes for this. The first cause is that p might contain a state that has already been visited with a sleep set which contained t_i. This is not possible because if such a state existed, the path p would not be the leftmost path in S_s. The second possible cause is that t_i has been added to the sleep set at some point on the path p and then passed along p until s_i. Let us prove that this is also impossible.

Assume that t_i is in the sleep set associated to state s_i and that it has been added to *Sleep* at some previous point on the path p. Precisely, there are states s_j and s_{j+1}, $j < i$, in p such that $t_i \notin$ *Sleep* at s_j and $t_i \in$ *Sleep* at s_{j+1}. This implies that t_i has been explored *before* t_j from s_j since a transition is introduced in *Sleep* once it is backtracked (fourth rule in Figure 3). This also implies that, from s_j, t_i has not led to a state s_l, $l \leq j$ already visited in the path p. Moreover, all transitions that occur between t_j and t_i in p, i.e. all t_k such that $j \leq k < i$, are independent with respect to t_i. Indeed, if this were not the case, t_i would not be in the sleep set of s_i since transitions that are dependent with the transition taken are removed from the sleep set (third rule in Figure 3).

Consequently, $t_i t_j \ldots t_{i-1} \in [t_j \ldots t_{i-1} t_i]$, i.e. $t_i t_j \ldots t_{i-1}$ and $t_j \ldots t_{i-1} t_i$ are two interleavings of a single concurrent execution (i.e. a single trace) and hence $s_j \overset{t_i t_j \ldots t_{i-1}}{\Rightarrow} s_{i+1}$. Given that s is reachable from s_{i+1}, it is reachable by a path that in state s_j takes the transition t_i. Since t_i has been explored before t_j in s_j and has not led to a state s_l, $l \leq j$ already visited in p, the path p is not the leftmost path in S_s. A contradiction.

∎

Protocol	Algorithm	states	matched	transitions	depth	time (sec)
PFTP	1	409,257	771,265	1,180,522	5,044	219.4
	2	409,257	179,304	588,561	4,550	394.6
URP	1	15,378	27,709	43,087	202	6.9
	2	15,378	1,884	17,262	202	10.7
MULOG3	1	100,195	254,183	354,378	119	35.2
	2	100,195	3,736	103,931	119	53.6
DTP	1	251,409	397,058	648,467	545	97.8
	2	251,409	11,152	262,561	545	160.8

Table 1: Comparison of the performances of Algorithm 1 and 2

In practice, the previous theorem enables us to use Algorithm 2 to verify all properties that can be reduced to a state accessibility problem, like, for instance, deadlock detection, unreachable code detection, assertion violations, safety properties. Moreover, other problems like the verification of liveness properties and model checking for linear-time temporal logic formulae are reducible to a set of reachability problems (see for instance [CVWY90, Hol91, VW86]), for which the method developed in this paper is applicable. By construction, the state-graph G' explored by Algorithm 2 is a "sub-graph" of the state-graph G explored by Algorithm 1. Both state-graphs G and G' contain the same number of states, the only difference is that G' contains always less transitions than G. Of course, if no simultaneous enabled independent transitions are encountered during the search, G' is then exactly equivalent to G.

Since only states, not transitions, are stored during an on-the-fly verification and since the number of states is the same in G and G', Algorithm 1 and Algorithm 2 have exactly the same memory requirements. (As a matter of fact, Algorithm 2 requires a few hundred bytes more for the manipulation of *Sleep*; this overhead can be made insignificant with respect to the global memory requirements [HGP92].)

Table 1 compares the performances of and the state-graphs explored by Algorithm 1 and Algorithm 2 for the protocols presented in Section 2. The advantage of Algorithm 2 is that it explores much fewer transitions than Algorithm 1. The number of state matchings strongly decreases. If the reduction in the number of transitions is sufficient to make up the additional run-time overhead due to the manipulation of sleep sets, an improvement in the general run-time requirements can result. This is not the case for the protocols considered here. (In [HGP92], it is shown that the sleep set scheme can produce a significant reduction in the overall run-time requirements when it is combined with a state compression method.) "Depth" is the maximum size of the stack during the search.

One clearly sees in Table 1 that the number of matched states strongly decreases when using Algorithm 2. This phenomenon can be explained with the following theorem.

Theorem 3.2 *For every reachable state s, Algorithm 2 never completely explores more than one interleaving of a single trace (partial ordering of transitions) that leads to s, and thus never visits s twice because of the exploration of two interleavings of a same trace.*

Proof:

By definition, all $w' \in [w]$, i.e. all interleavings w' of a single concurrent execution $[w]$, can be obtained from w by successively permuting pairs of *adjacent independent* transitions. Let w and w' denote two interleavings of the single trace $[w]$. We now prove that Algorithm 2 does not completely explore both of them.

Let $Pref(w)$ denote the common prefix of w and w' that ends when w and w' differ. Assume the next transition of w after $Pref(w)$ is t and that the next transition of w' after $Pref(w)$ is t'. In state s such that $s_0 \overset{Pref(w)}{\Rightarrow} s$, both transitions t and t' are enabled. Moreover, t and t' are independent since w and w' differ only by the order of independent transitions. Assume that t and t' are not in the current *Sleep* and that the search explores t first. Later, when t is backtracked and the search backs up in s, t is introduced in *Sleep*. Then t' is explored and leads to a state s''. Since t and t' are independent, t is not removed from *Sleep* at state s''.

During the remainder of the exploration of w' starting from s'', t remains in *Sleep* and is never executed. Indeed, t could only be removed from *Sleep* after the execution of some transition t'' that is dependent with it. This is impossible because if t'' occurs before t in w' (since t has to occur eventually in w'), w' would differ from w by the order of two dependent transitions t and t'' and thus, w and w' would not be two interleavings of a same trace. Since t is never executed and has to occur in w', w' is not completely explored.

Note that, if an already visited state is reached during the exploration of w' from s'', the exploration of w' stops. It might then be the case that the remainder of w' has already been explored, but it was during the exploration of an interleaving of another trace that has the same suffix than w'.

∎

In other words, if a state is reachable by only several interleavings of a single trace, Algorithm 2 never completely explores more than one of these interleavings and visits that state only once. In the example of Figure 2, all states are visited only once by Algorithm 2. Of course, if one could know it in advance before starting the search, it would not be necessary to store *any* states! Unfortunately, it is impossible to determine which are the states that are encountered only once before the search being completed.

Let us now study the impact of sleep sets on state space caching.

4 State Space Caching and Sleep Sets

Figure 4 compares the performances of Algorithm 1 (classical depth-first search) and Algorithm 2 (depth-first search with sleep sets) for various cache sizes.

As already pointed out in Section 2, the number of transitions that are explored during the search performed by Algorithm 1 blows up when the cache size is approximately the half/third of the total number of states. This causes a run-time explosion, which makes state space caching inefficient under a certain threshold.

With Algorithm 2, for PFTP, this threshold can be reduced to the fourth of the total number of states. The improvement is not very spectacular because the number of matched states, even when using sleep sets, is still too important (see Table 1). The risk of double work when reaching an already visited state that has been deleted from memory is not reduced enough.

For the other three protocols, URP, MULOG3 and DTP, the situation is different: there is no run-time explosion with Algorithm 2. Indeed, the number of matched states is reduced so much (see Table 1) that the risk of double work becomes very small. When the cache size is reduced up to the maximal depth of the search (this maximal depth is the lower bound for the cache size since all states of the stack have to be stored to ensure the termination of the search), the number of explored transitions is still between only two and four times the total number of transitions in the state space. *These protocols, which have between 15,000 and 250,000 reachable states, can be analyzed with no more than 500 stored states. The memory requirements are reduced to 3% up to 0.2%. The only drawback is an increase of the run time by two to four times compared to the search where all states are stored (which may be impossible for larger state spaces).*

The efficiency of the method can be dynamically estimated during the search: if the maximum stack size remains acceptable with respect to the cache size and if the proportion of matched states remains small enough, the run-time explosion will likely be avoided. Else one cannot predict if the cache size is large enough to avoid the run-time explosion.

5 Further Investigations

An important factor when using the state space caching method is the selection criterion for determining which states are deleted when the cache is full.

Holzmann has studied several replacement strategies in [Hol85]. These strategies were based on the number of times that a state has been previously visited. These strategies were: replace the most frequently visited state; replace the least frequently visited state; replace a state from the largest class of states in the current state space (where a class contains states that have been visited equally often); replace randomly a state (blind round-robin replacement); replace the state corresponding to the lowest point in the search tree (smallest subtree). The conclusion of that study was that the best strategy seems to be a random selection. In [Hol87], the probability of recurrence of states (i.e. the probability that once a state has been visited n times it will be visited an $n + 1$st time as well) was investigated and turns out not to be strongly correlated with the number of previous visits.

We have experimented some different replacement strategies. Our motivation was to study the influence of the type of transitions that can lead to a state on the probability that the state is visited again later during the search. For instance, a "labeled" state, e.g. the target of a goto jump, is intuitively more susceptible to be matched than an "unlabeled" state.

First, let us classify transitions into different types:

1. control branches (goto jump, start of do loops, ...);
2. receives on message channels;
3. sends on message channels;
4. assignments to variables;
5. other transitions.

Each state encountered during the search is tagged with the type of the transition that has led to it. We have studied the impact of the following replacement strategy on the run-time requirements of the state space caching method, for each of the four first types of transitions:

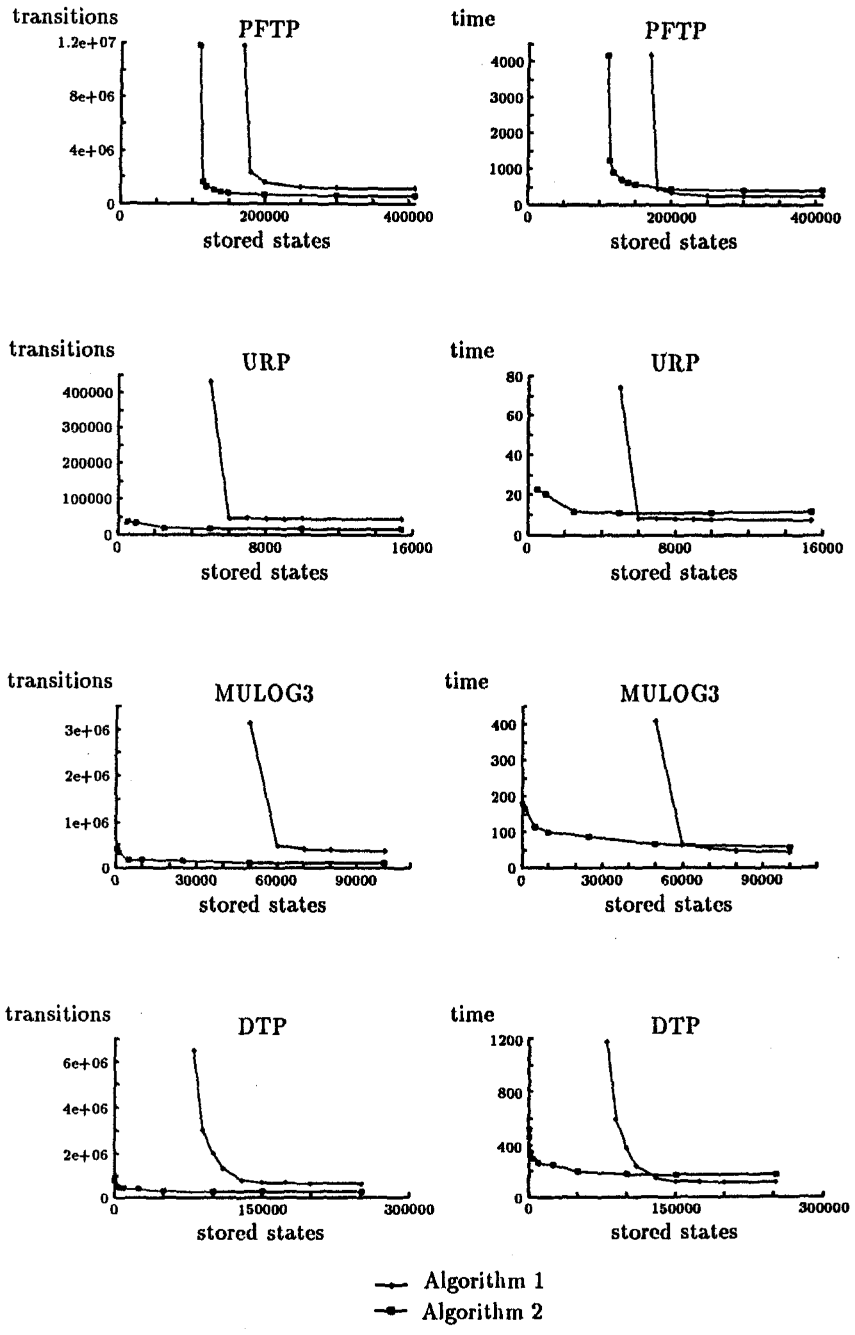

Figure 4: Performances of state space caching with Algorithm 1 and 2

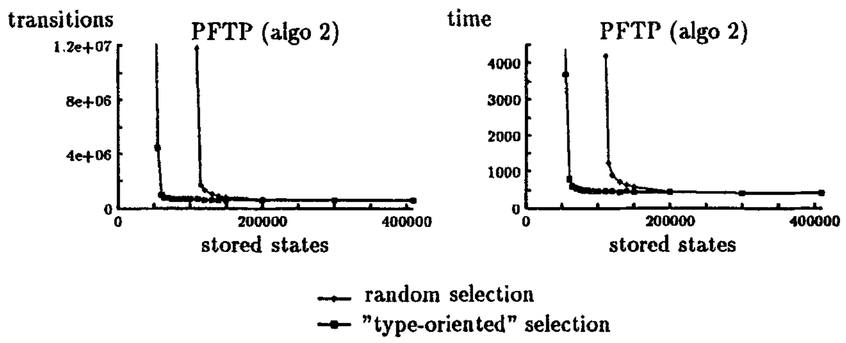

Figure 5: Random vs "type-oriented" replacement strategy

> Each time a state has to be deleted, scan an arbitrarily given number of stored states (scanning too many states incurs an unacceptable overhead; this is why an arbitrary limit is given). If possible, select a state that *is not* tagged with the type considered. Otherwise, select randomly one of them.

The results are the following: for type 1, the procedure described above gives always better results than a simple random selection; for types 2, 3 and 4, the results are unpredictable. In other words, it is preferable not to remove states pointed by type 1 transitions, as far as possible.

Since protocols do not necessarily have transitions of each type, a good heuristic cannot be based only on the selection of states that follow transitions of a single type. Grouping all the four first types together and trying to delete only states that follow transitions of type 5 is not a good solution as well because it degenerates to a random selection since transitions of type 5 are usually not numerous enough. A possible trade-off is to use the following replacement strategy:

> Each time a state has to be deleted, scan an arbitrarily given number of stored states and select one state that is tagged with the highest type (i.e. closest to 5).

The order of the types given above was chosen according to the results of the experiments we made with the different types taken separately. If a state is visited by several transitions, its tag is set to the smallest type of transitions that led to it.

Figure 5 shows the results obtained with this strategy (denoted "type-oriented" strategy) compared to a random replacement discipline for the PFTP protocol. One can see that this strategy does not involve a significant run-time overhead. Moreover, it yields a 50% reduction for the run-time blow-up threshold.

For the other three protocols, there is no significant difference with respect to a random selection strategy. As a matter of fact, in these examples, the random selection strategy is sufficient to reduce the cache size so close to the maximal stack size that no significant further reduction is possible.

6 Conclusions

We have presented a new technique which can substantially improve the state space caching discipline by getting rid of the main cause of its previous inefficiency, namely prohibitive state matching due to the exploration of all possible interleavings of concurrent statement executions all leading to the same state. We have shown with experiments on real protocol models that, thanks to sleep sets, the memory requirements needed to validate large protocol models can be strongly decreased (sometimes more than 100 times) without seriously increasing the time requirements (a factor of 3 or 4). This makes possible the complete exploration of very large state spaces, that could not be explored so far. However, exploring state spaces of several tens of million states takes time, since all these states are visited at least once during the search. Thus time becomes the main limiting factor.

Note that no attempts were made in this paper to reduce the number of states that need to be visited in order to validate properties of a system. However, sleep sets were originally introduced as part of a method intended to master the "state explosion" phenomenon [God90, GW91a, GW91b, HGP92]. Using the full method preserves the beneficial properties of sleep sets that were investigated in Section 3 while enabling a substantial reduction of the number of states that have to be visited for verification purposes.

Acknowledgements

We wish to thank Pierre Wolper for helpful comments on this paper.

References

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242, Rutgers, June 1990.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185, Rutgers, June 1990.

[GW91a] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the 6th IEEE Symposium 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.

[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.

[Hol81] G. J. Holzmann. Pan — a protocol specification analyzer. Technical report, Technical Memorandum 81-11271-5, Bell Laboratories, 1981.

[Hol84] G. J. Holzmann. The pandora system — an interactive system for the design of data communication protocols. *Computer Networks*, 8(2):71–81, 1984.

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

[Hol87] G. J. Holzmann. Automated protocol validation in argos — assertion proving and scatter searching. *IEEE Trans. on Software Engineering*, 13(6):683–696, 1987.

[Hol90] G. J. Holzmann. Algorithms for automated protocol validation. *AT&T Technical Journal*, 69(1):32–44, 1990. Special issue on Protocol Testing and Verification.

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

[JJ89] C. Jard and T. Jeron. On-line model-checking for finite linear temporal logic specifications. In *Workshop on automatic verification methods for finite state systems*, volume 407 of *Lecture Notes in Computer Science*, pages 189–196, Grenoble, June 1989.

[JJ91] C. Jard and Th. Jeron. Bounded-memory algorithms for verification on-the-fly. In *Proc. 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, Aalborg, July 1991.

[Maz86] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324, 1986.

[TN87] M. Trehel and M. Naimi. Un algorithme distribué d'exclusion mutuelle en log(n). *Technique et Science Informatiques*, pages 141–150, 1987.

[VW86] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–21, April 1986.