State-Space Caching Revisited

Patrice Godefroid, Gerard Holzmann and Didier Pirottin

Formal Methods and System Design, Kluwer Academic Publishers, volume 7, number 3, pages 1-15, November 1995.

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 verification technique for finite-state concurrent systems. It performs an exhaustive exploration of the state space of the system being checked while storing only all states of just one execution sequence plus as many other previously visited states as available memory allows. So far, this technique has been of little practical significance: it allows one to reduce memory usage by only two to three times, before an unacceptable blow-up of the run-time overhead sets in. The explosion of the run-time requirements is due to 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.

In this paper, we present a method to tackle the main cause of this prohibitive state matching: the exploration of all possible interleavings of concurrent executions of the system, which all lead to the same state. Then, we show that, in many cases, with this method, most reachable states are visited only once during state-space exploration. This enables one not to store most of the states that have already been visited without incurring too much redundant explorations of parts of the state space, and makes therefore state-space caching a much more attractive verification method. As an example, we were able 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

The effectiveness of state-space exploration techniques for debugging and proving correct concurrent reactive systems is increasingly becoming established as tools are being developed. The number of "success stories" about applying these techniques to industrial-size systems keeps growing (e.g., see [Rud92]). The reason why these techniques are so successful is mainly due to

*The work of these authors was partially supported by the European Community ESPRIT BRA projects SPEC (3096) and REACT (6021), 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 the authors.

†New address: AT&T Bell Laboratories, Software Production Research Department, 1000 E. Warrenville Road, P.O. Box 3013, Naperville, IL 60566-7013, U.S.A.

their simplicity: they are easy to understand, easy to implement and, last but not least, easy to use: they are fully automatic. Moreover, the range of properties that they can verify has been substantially broadened in the last decade thanks to the development of model-checking methods for various temporal logics. The only real limit of state-space exploration verification techniques is the often excessive size of the state space.

More precisely, memory is the main limiting factor of most conventional reachability analysis algorithms. Given a finite-state system, these verification algorithms perform an exhaustive exploration of the state space of the system being checked in order to prove or disprove properties of the system. 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.

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:

1. From the initial state, the explorations of all interleavings of a single finite concurrent execution of the system always lead to the same state. This state will thus be visited several times because of all these interleavings.
2. From the initial state, explorations of different finite concurrent executions may lead to the same state.

In this paper, we introduce a technique to avoid the effects of the first cause given above. Then we study the impact of this technique on real-protocol state spaces. In many cases, when using this method, most of the states are 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 one not to store most of the states that have already been visited without incurring too much redundant explorations of parts of the state space. The memory requirements can thus strongly decrease without seriously increasing the time requirements. This makes possible the complete exploration of very large state spaces (several tens of million states) in a reasonable time (a few hours). In most cases, time, not memory, is the main limiting factor of this verification technique.

```
Initialize: Stack is empty; H is empty;
Search() {
   enter s0 in H;
   push (s0) 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

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, GW93]. In Section 3, we recall the basic idea behind sleep sets. Then, we present a new simple version of the sleep set scheme. We study properties of sleep sets and prove two 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 H to store all the states that have already been visited during the search. Algorithm 1 executes all enabled transitions at each state the system can reach from its initial state. The exploration can be performed "on-the-fly", i.e., without storing the transitions that are taken during the search. This substantially reduces 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, for instance, a single sequence of states leading from the initial state to the current 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 longer in 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 explored path plus 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.

We have implemented such a caching discipline in an automated protocol validation system called SPIN [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 AT&T's Universal Receiver Protocol [FM89], 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.

Information about the state spaces of these protocols are obtained by running Algorithm 1 and are given in Table 1. States and transitions are respectively the number of states and transitions visited by the corresponding algorithm. "Matched" is the number of state matchings that occurred during the search. "Depth" is the maximum size of the stack during the search. All measurements reported in this paper 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. Parameters in these protocols are set in such a way that full state spaces can be stored in 64 Megabytes of RAM, for experimental purposes.

Protocol	states	matched	transitions	depth	time (sec)
PFTP	409,257	771,265	1,180,522	5,044	219.4
URP	15,378	27,709	43,087	202	6.9
MULOG3	100,195	254,183	354,378	119	35.2
DTP	251,409	397,058	648,467	545	97.8

Table 1: State-space exploration with Algorithm 1

The results of our experiments with Algorithm 1 and different cache sizes are presented in Figure 4 in Section 4. These results clearly show that, for these examples, 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 state-space caching discipline is useful for exploring state spaces that are only a few times larger than the size of the cache, 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.

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, GW93] as part of a verification method that can avoid most of the state explosion due to the modeling of concurrency by interleaving. 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 a set T and a symmetrical binary relation $D \subseteq T \times T$ called "dependency" relation, two sequences over T 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 t_1 and t_2 are two symbols of T which are independent according to D, the sequences t_1t_2 and t_2t_1 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 T and the dependency relation. Thus the trace containing both t_1t_2 and t_2t_1 could be represented by $[t_1t_2]_{(T,D)}$. A trace corresponds to a partial ordering of symbol occurrences and contains 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 the context considered here, T will be the set of *transitions* that appear in the code of the PROMELA program being verified. When should we consider transitions to be independent? Our intuitive idea is that transitions are independent if their order does not matter. More precisely, two transitions[1] are *independent* if the following two conditions are satisfied in all reachable states (otherwise they are said to be *dependent*):

1. if t_1 is enabled in s and[2] $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s' (independent transitions can neither disable nor enable each other); and
2. if t_1 and t_2 are enabled in s, then there is a unique state s' such that both $s \stackrel{t_1t_2}{\Rightarrow} s'$ and $s \stackrel{t_2t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

One can wonder if this definition is of more than semantic use. Indeed, it is not practical to check the two properties listed above for all pairs of transitions in all reachable states to determine which transitions are independent and which are not. Fortunately, it is possible to give easily checkable syntactic conditions that are sufficient for transitions to be independent. In a PROMELA program, dependency can arise between program transitions 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. Carefully tracking dependencies in a PROMELA program is by no means a trivial task. We refer the reader to [GP93] for a detailed presentation of that topic.

Given a dependency relation, sequences of transitions can be grouped into equivalence classes, i.e., traces. Moreover, if a state s of the system is reachable from the initial state

[1] We assume that transitions are deterministic, i.e., that the execution of a transition leads to a unique successor state. This is not a restriction since "nondeterministic transitions" can always be modeled by a set of deterministic ones.

[2] We write $s \xrightarrow{t} s'$ to mean that the transition t leads from the state s to the state s' and $s \stackrel{w}{\Rightarrow} s'$ to mean that the sequence of transitions w leads from s to s'.

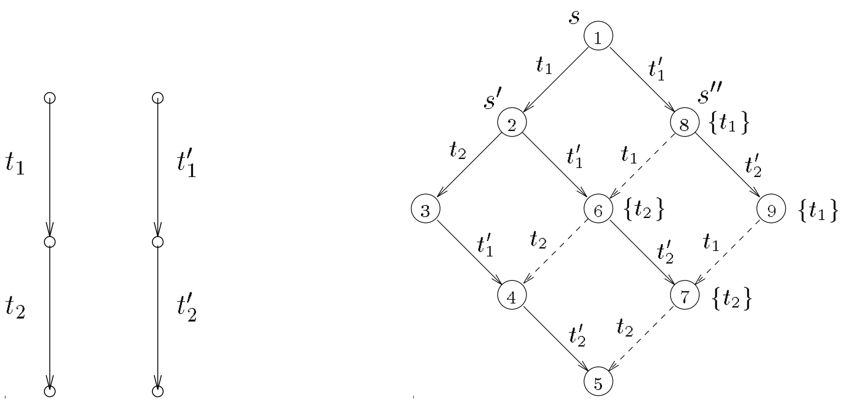

Figure 2: A concurrent system (left) and the exploration performed by Algorithm 2 (right)

s_0 after executing a transition sequence w, all transition sequences belonging to $[w]$ also lead to state s from s_0 [GP93]. In [God90, GW93], 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 search backs up to state s. Then t_1' is explored 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 exploration of t_1 in s'', we use sleep sets: we put t_1 in the sleep set associated with s''.

A sleep set is defined as a set of transitions. A 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 explored* 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 of Figure 1. The sleep set associated with a state $\mathbf{s}$ is denoted by $\mathbf{s}.Sleep$. Initially, one has $\mathbf{s}_0.Sleep = \emptyset$ (line 3). Each time a new state $\mathbf{s}$ is encountered during the search, all enabled transitions that are not in $\mathbf{s}.Sleep$, i.e., the sleep set that has been computed to be associated with $\mathbf{s}$ and has been stored with it in the *Stack*, are selected to be explored from

```
1   Initialize: Stack is empty; H is empty;
2   Search() {
3      s0.Sleep = ∅;
4      enter s0 in H;
5      push (s0) onto Stack;
6      DFS();
7      }
8   DFS() {
9      s = top(Stack);
10     T = enabled(s)\s.Sleep;
11     s.Sleep = {t ∈ s.Sleep | s' = succ(s) after t ⋀ s' is NOT in Stack};
12     for all t ∈ T do {
13        s' = succ(s) after t; /* execution of t */
14        s'.Sleep = {t' ∈ s.Sleep | t' and t are independent};
15        if s' is NOT already in H then {
16           enter s' in H;
17           push (s') onto Stack;
18           DFS();
19           }
20        /* backtracking of t */
21        if s' is not in Stack then {
22           s.Sleep = s.Sleep ∪ {t}
23           }
24        }
25     pop s from Stack
26     }
```

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

state **s** (line 10). Lines 11, 14 and 21–23 describe how to compute the sleep sets associated with the successor states of the current state **s** from the value of its sleep set **s**.*Sleep*. First, all transitions of **s**.*Sleep* that point to a state in *Stack* are removed from **s**.*Sleep* (line 11). (Testing if a transition points to a state in *Stack* can be done by simulating it and then checking if the reached state is in *Stack*.) Next, each time a transition t is explored from state **s**, the set of all transitions of **s**.*Sleep* that are independent with t is the sleep set **s**$'$.*Sleep* of the successor state **s**$'$ of **s** after t (line 14). Finally, each time a transition t from state **s** to a state **s**$'$ is backtracked, t is added to **s**.*Sleep* if **s**$'$ is not in *Stack* (lines 21–23), i.e., if t has not led to an already visited state which is still in the stack.

A simple example of exploration performed by Algorithm 2 is given in Figure 2. The system on the left is composed of two completely independent concurrent processes. On the right, the state-graph explored by Algorithm 2 for this system is presented. The value of the sleep set associated with each state when the state is pushed onto the stack is given between braces beside the state. Dotted transitions are not explored by Algorithm 2.[3]

[3] According to line 11 of Algorithm 2, dotted transitions, i.e., transitions in sleep sets, have to be tested once in

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. Since s is reachable, s is visited by Algorithm 1 (we do not prove here that a classical depth-first search visits all reachable states). We now prove that the first path leading to s explored by Algorithm 1, i.e., the path through which Algorithm 1 reaches s for the first time, is still explored in the second run when using Algorithm 2.

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 leading to s that will be examined during both runs. The only reason why 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 with 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 very first explored path leading to s. The second possible cause is that t_i has been added to the sleep set associated with some previous state of 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 with state s_i when s_i is pushed onto the stack. Hence, t_i has been added to the sleep set associated with some previous state s_j, $j < i$, of the path p and passed in the sleep set associated with the successor states of s_j along the path p until s_i. Formally, $t_i \notin s_j.Sleep$ when s_j is pushed onto the stack and $t_i \in s_k.Sleep$ for all states s_k, $j < k \leq i$. This implies that t_i has been explored *before* t_j from s_j since a transition is introduced in the sleep set once it is backtracked (line 22 in Figure 3). This also implies that, from s_j, t_i has not led to a state s_l, $l \leq j$, of the path p (line 21). 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 was not the case, t_i would not be in $s_i.Sleep$ since transitions that are dependent with the transition taken are removed from the sleep set (line 14).

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 lead to the same state: $s_j \overset{t_i t_j \ldots t_{i-1}}{\Rightarrow} s_{i+1}$. Moreover, this path from s_j does not intersect any of the states s_l, $l \leq j$, of p. Indeed, if one of the intermediate states of this path was a state s_l, $l \leq j$, of p, one would have $s_j \overset{t_i w}{\Rightarrow} s_l$ where w is a prefix of the sequence $t_j \ldots t_{i-1}$, and at state s_k of p such that $s_j \overset{w}{\Rightarrow} s_k$, t_i would lead to s_l; since transitions that lead to a state in the current stack are removed from the current sleep set (line 10), t_i would have been removed from $s_k.Sleep$ at s_k, which contradicts the fact that t_i is in $s_i.Sleep$ and is therefore impossible. Since there is a path $s_j \overset{t_i t_j \ldots t_{i-1}}{\Rightarrow} s_{i+1}$ from s_j that does not intersect any of the states s_l, $l \leq j$, and since t_i is explored before t_j in s_j, the path p is not the very first path through which Algorithm 1 reaches s. A contradiction.

∎

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 formulas are reducible to a set of reachability problems (see for instance [CVWY90, Hol91, GH93]), for which the method developed in this paper is applicable. By construction, the

order to determine if they lead to a state of the current stack or not. However, "testing" a transition is different from "exploring" it, since, whatever the result of the test is, the successor state of a tested transition is never "explored", i.e., pushed onto the stack, after that test.

Protocol	Algorithm	states	matched	transitions	depth	time (sec)
PFTP	1	409,257	771,265	1,180,522	5,044	219.4
	2	409,257	183,638	592,895	4,786	505.4
URP	1	15,378	27,709	43,087	202	6.9
	2	15,378	1,884	17,262	202	13.8
MULOG3	1	100,195	254,183	354,378	119	35.2
	2	100,195	3,736	103,931	119	80.8
DTP	1	251,409	397,058	648,467	545	97.8
	2	251,409	11,152	262,561	545	206.4

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

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 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 require exactly the same amount of randomly accessed memory. Indeed, sleep sets are stored in the *Stack*, which can be stored in a sequentially accessed memory.

Table 2 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 when using Algorithm 2. This phenomenon can be explained with the following theorem.

Theorem 3.2 *A sequence w of transitions is said to be* acyclic *if the execution of w from the initial state s_0 never passes twice through a same state. A trace $[w]$ is said to be* acyclic *if $\forall w' \in [w] : w'$ is acyclic. Algorithm 2 never completely explores more than one interleaving per acyclic trace.*

Proof:

By definition, all $w' \in [w]$, i.e., all interleavings w' of a single trace $[w]$, lead to the same state and 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. Let t be the next transition of w after $Pref(w)$ and let t' be the next transition of w' after $Pref(w)$. In state s such that $s_0 \stackrel{Pref(w)}{\Rightarrow} s$, 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. If either t or t' are in $s.Sleep$ when s is pushed onto the stack, the theorem follows. Assume that t and t' are not in $s.Sleep$ and that the search explores t first. Let us show that if w is fully explored, w' cannot be fully explored.

When t is backtracked and the search backs up in s, t is introduced in $s.Sleep$ since it has not led to a state in the stack (otherwise, this would contradict the fact that $[w]$ is acyclic). Then t' is explored and leads to a state s''. Since t and t' are independent, t is in $s''.Sleep$.

During the remainder of the exploration of w' starting from s'', t remains in *Sleep* and is never executed. Indeed, t could be removed from *Sleep* in only two cases. The first case is that a transition t'' dependent with t is executed. 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. The second possible situation is that t leads to a state in the stack. This would contradict the fact that $[w]$ is acyclic, and is therefore impossible. Since t remains in the sleep set associated with the states reached during the remainder of the exploration of w' starting from s'', t is not executed, and since it 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'.

∎

Algorithm 2 never visits a state twice because of the exploration of two interleavings of a same acyclic trace leading to that state. Therefore, if a state is reachable by several interleavings of only one single acyclic trace, Algorithm 2 never completely explores more than one of these interleavings and visits that state only once. Note that all reachable states s are reachable by an acyclic trace: the shortest path w from s_0 to s characterizes an acyclic trace $[w]$ leading to s. 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 is 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 2). 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 2) 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 are 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,*

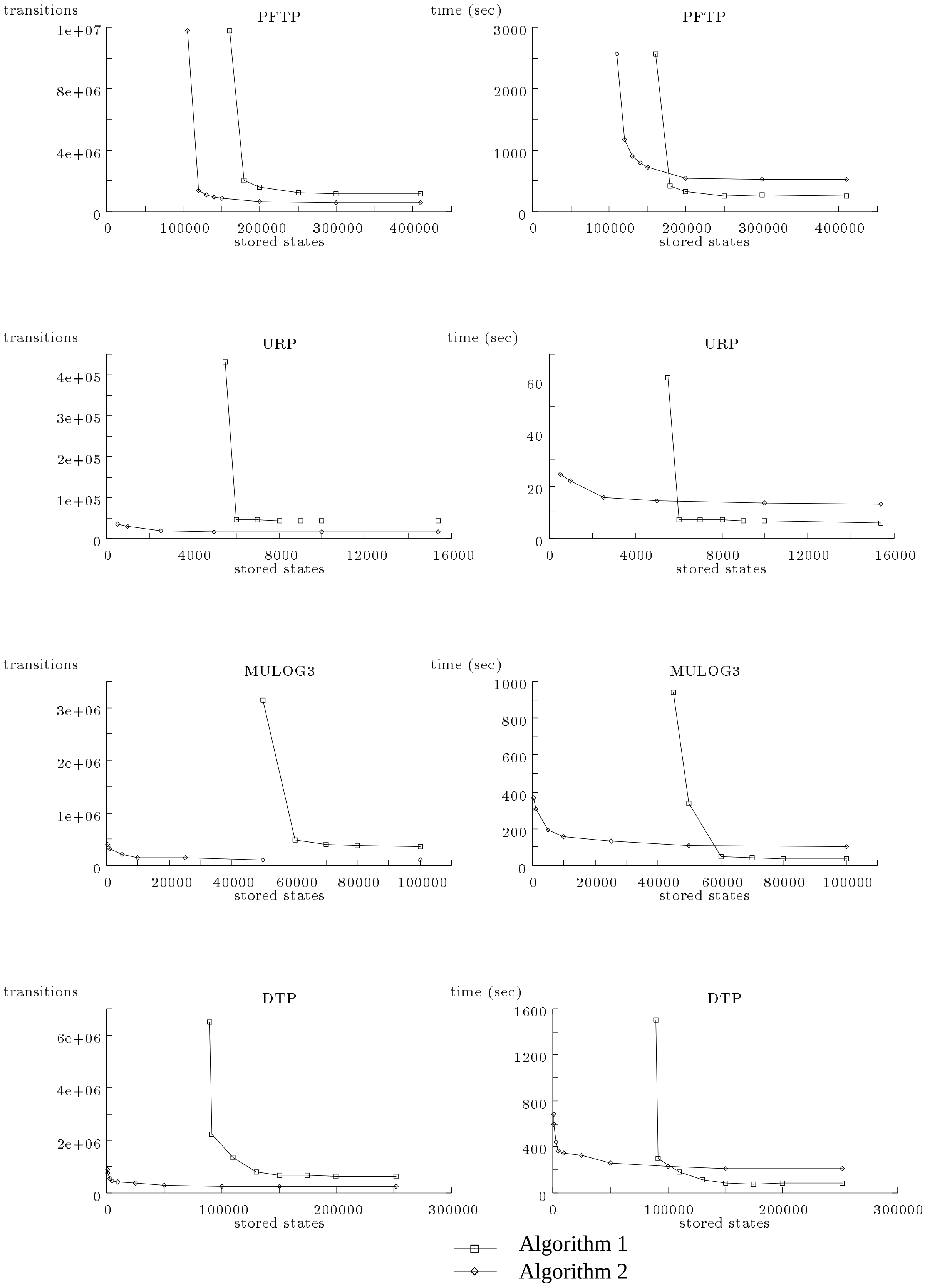

Figure 4: Performances of state-space caching with Algorithm 1 and 2

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. The experiments reported in Figure 4 were performed using a random replacement strategy.

Several replacement strategies were studied in [Hol85]. These strategies were based on the number of times that a state has previously been 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; 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:

> 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

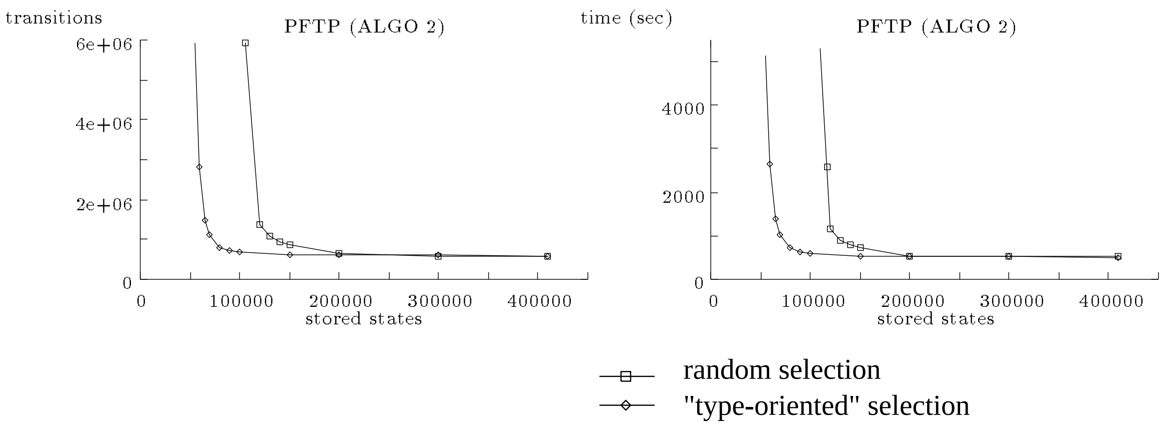

Figure 5: Random vs "type-oriented" replacement strategy

> 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 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. 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 [GW93, God90, 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.

An implementation of the techniques presented in this paper, including a state space caching algorithm and an implementation of the sleep set scheme, is available in an add-on package for the validation tool SPIN [Hol91]. Noncommercial users can obtain the SPIN system via anonymous ftp from research.att.com from the /netlib/spin directory. The add-on package is available free of charge for educational and research purposes by anonymous ftp from montefiore.ulg.ac.be from the /pub/po-package directory.

Acknowledgements

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

A preliminary version of this paper appeared in the Proceedings of the 4th Workshop on Computer-Aided Verification [GHP92].

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.

[FM89] A. G. Fraser and W. T. Marshall. Data transport in a byte stream network. *IEEE Journal on Selected Areas in Communications*, 7(7):1020–1033, September 1989.

[GH93] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proc. 13th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, pages 109–124, Liège, May 1993. North-Holland.

[GHP92] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. 4th Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 178–191, Montreal, June 1992. Springer-Verlag.

[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. Extended version in ACM/AMS DIMACS Series, volume 3, pages 321–340, 1991.

[GP93] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449, Elounda, June 1993. Springer-Verlag.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design, Kluwer Academic Publishers*, 2(2):149–164, April 1993.

[HGP92] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, pages 349–363, 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.

[Rud92] H. Rudin. Protocol development success stories: Part I. In *Proc. 12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, June 1992. North-Holland.

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