Memory Efficient State Space Storage in Explicit Software Model Checking

Evangelista Sami and Pradat-Peyre Jean-François

CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris
email: {evangeli,peyre}@cnam.fr

Abstract. The limited amount of memory is the major bottleneck in model checking tools based on an explicit states enumeration. In this context, techniques allowing an efficient representation of the states are precious. We present in this paper a novel approach which enables to store the state space in a compact way. Though it belongs to the family of explicit storage methods, we qualify it as semi-explicit since all states are not explicitly represented in the state space. Our experiments report a memory reduction ratio up to 95% with only a tripling of the computing time in the worst case.

1 Introduction

Model checking is a powerful and automatic technique for the verification of finite state systems. It consists of enumerating all the possible configurations (states) or actions of the system to track the ones which do not match the specification, usually expressed in a temporal logic, e.g. LTL. To preserve termination and to improve efficiency, model checking algorithms have to keep track of the visited states in a state space (or reachability set).
Algorithms based on an explicit state enumeration suffer from the well-known state explosion problem. Due to the concurrent execution of several components, the number of possible configurations can be far too large to fit in memory or even on a disk. In a first family of techniques designed to alleviate this problem, we can put all those which aim at reducing the number of visited states while still preserving the property to verify. Examples of such techniques include partial order reductions and symmetry-based reductions. A second family of techniques aim at representing the state space in an efficient way in order to limit the memory allocated to store the state space. State compression techniques and state caching are examples of such techniques.
Methods which belong to the first family usually achieve better reductions of the amount of memory required since, in most systems, the number of visited states is an exponential function of the number of concurrent process. However, a wise choice for the representation of the state space can still perform a significant reduction of the memory requirement. Moreover, the use of such techniques is especially needed when model checking is applied to software specification or

models extracted from source code, since the state descriptor of these models can grow very large as it may represent complex data such as heap allocated objects.
This paper presents a compression method which falls into the second family. We qualify our method as semi-explicit since states may not be explicitly represented in the state space. The underlying idea of the method is conceptually simple but reveals to be very useful in practice. Moreover, it is not specific to the formalism used in this paper (colored Petri nets) and it could easily be adapted to other formalisms. Thus we formulate our ideas in general terms. Last but not least, our technique can be further combined with the state collapsing method [1, 2] to lead to better state representations.

The remainder of this work is organized as follows. Section 2 is a brief overview of the methods proposed so far to represent the state space in explicit state model checking. Section 3 recalls some basic concepts of colored Petri nets. Our storage method is presented in Section 4. The results of some experiments made with our model checker are presented in Section 5. Section 6 discusses the compatibility of our method with other techniques used in explicit state model checking and presents some directions for future research. Lastly, Section 7 presents our conclusions.

2 State space representation in explicit model checking

This section draws up an overview of the methods used to represent the state space in explicit model checking. These methods can be classified in three categories: exhaustive storage, partial storage, and lossy storage. The method we propose in this paper belongs to the first family.

Exhaustive storage methods build the complete state space of the system. Each state met during the search is encoded by a reversible mapping into a state vector and then stored in an adequate data structure, e.g., a hash table. In this way, each state can be retrieved from its encoded form and checking whether a state has already been visited becomes trivial. Compression techniques are usually employed to optimize the encoding. Examples of compression techniques are state collapsing [1], recursive indexing [2] (possibly improved by training runs [2]), very tight hashing [3], sharing trees [4], difference compression [5], runtime state compaction [6], or invariant-based compression methods for Petri nets as it is done in [7] and [8]. Each method tries to provide efficient compression ratios without introducing an unbearable increase in the running time.

Partial storage methods are based on the idea that it is not necessary to store all the visited states to preserve the termination of the search algorithm, but rather a portion of the state space. Perhaps the most famous method that follows this paradigm is the state space caching method [9]. In state space caching, only the states which belong to the search stack are stored in the state space in order to

avoid entering cycles which would endanger the termination of the algorithm. Some other states may also be stored depending on the amount of memory still available. The main limitation of state caching is the amount of redundant work performed. Indeed, if only stack states are stored, each state s will be explored once for every path leading from the initial state to s, causing an unacceptable blowup in the execution time. It was observed in [9] that the use of partial order methods greatly reduces this problem by limiting the exploration of redundant paths. The main drawback of caching techniques is that the best replacement strategy to adopt and the run time increase heavily depends on the structure of the state space, and is thus hard to predict as observed in [10].
In [11], the authors proposed to improve reachability analysis via the use of pseudo-root states. These states are characterized by the fact that all their predecessors have already been visited during the search. This gives the possibility to forget these states without endangering the termination of the search. Experiments reported result in 2- to 16-fold improvement in space requirements. This technique has the advantage of introducing little runtime overhead, since any state is only visited once. Nevertheless, it relies on the ability to compute predecessors of states which seems to us a strong requirement.
The sweep line method, another technique based on this observation, was recently introduced in [12, 13]. It is inspired from the garbage collection mechanism. It uses the notion of progress present in some systems, e.g., timed protocols, to safely delete from the state space the states which cannot be reached again from the set of unprocessed states. Preliminary results on this method are quite promising. However, it suffers from the fact that a progress measure function has to be supplied by the user. In [14] Schmidt gave a method to automatically derive such a function from the incidence matrix of Petri nets, but it seems hard to construct a "good" function for more complex formalisms.
More recently, Behrmann, Larsen and Pelánek proposed in [15] several strategies to decide whether or not a state has to be kept in the reachability set while still preserving termination and efficiency.
Verisoft [16] is a software model checker that illustrates the partial storage strategy to the extreme. Verisoft does not store any states at all. The termination is guaranteed by limiting the search to a specified depth. It makes use of partial order methods (sleep sets) to avoid multiple revisits of the same state.

Lossy storage In a lossy storage scheme, each state is mapped to a state vector in a non-injective way before its insertion into the state space. Two different states may thus share the same compressed representation. An example of technique that belongs to this family is the bitstate hashing technique of Holzmann (or supertrace) [17]. Lossy storage methods greatly cut down the memory requirement since an arbitrary small number of bits can be used to represent each state, but suffer from an omission probability that a state may be erroneously declared as already visited whereas it is new, leading to unexplored portions of the state space. Wolper and Leroy showed in [18] that this probability can be reduced by storing the whole hash signature in a hash table with conflict resolution (the hashcompact method), or by using multiple hash functions (the multihash method). However, this kind of technique cannot be used for verification purposes, but rather at a debugging stage since the omission probability cannot be reduced to 0.

3 Colored Petri nets

We develop our method in the context of colored Petri nets [19]. Colored Petri nets allow the modeling of complex systems in a compact way and support numerous analysis techniques. In a colored Petri net, a place contains typed (or colored) tokens instead of anonymous tokens of Petri nets, and a transition may be fired in multiple ways, i.e., instantiated. To each place and each transition is attached a type (or a color domain). Each arc of the net between a place and a transition is labeled by a color mapping which specifies the type and the number of tokens produced or consumed by the firing of the transition for a given instantiation.
The definition of colored Petri nets is based on multi-sets. A multi-set over a set S is a mapping from S to the set of positive integers. The set of multi-sets over a set S is noted $Bag(S)$. Addition, subtraction, and comparison of multi-sets are defined as usual. States of colored Petri nets are also called markings.

Definition 1. *A colored Petri net is a tuple $N = \langle P, T, \Sigma, C, W^-, W^+, m_0 \rangle$ where P is a finite set of* **places***; T is a finite set of* **transitions***, with $P \cap T = \emptyset$; Σ, the* **colors set***, is a finite set of finite and non empty sets; C, the* **color domain** *application, is a mapping from $P \cup T$ to Σ; W^- and W^+, the* **backward and forward incidence matrixes** *associate to each $(p,t) \in P \times T$ a color mapping from $C(t)$ to $Bag(C(p))$; and m_0, an* **initial marking** *is an element of $\mathbb{M}_N$, the set of mappings which associate each $p \in P$ to an element of $Bag(C(p))$.*

We now define the transition relation and the state space of colored Petri nets.

Definition 2. *Let $N = \langle P, T, \Sigma, C, W^-, W^+, m_0 \rangle$ be a colored Petri net, $t \in T, c_t \in C(t)$ and $m \in \mathbb{M}_N$. The transition instance c_t of t, noted (t, c_t) is* **firable***, at m (noted $m[(t,c_t)\rangle$) if and only if $\forall p \in P, m(p) \geq W^-(p,t)(c_t)$. The* **firing** *of (t,c_t) at m leads to a marking m', (noted $m[(t,c_t)\rangle m'$) defined by $\forall p \in P, m'(p) = m(p) - W^-(p,t)(c_t) + W^+(p,t)(c_t)$. The* **state space** *(or reachability set) of N, denoted by $\mathbb{R}_N$, is defined recursively as the set $\{m_0\} \cup \{m \in \mathbb{M}_N | \exists m' \in \mathbb{R}_N, t \in T, c_t \in C(t) | m'[(t,c_t)\rangle m\}$.*

4 The Δ-markings storage method

This section describes the Δ-markings method and is organized as follows. Firstly, we present the general idea of the method. In a second step we give a depth first search (DFS for short) algorithm based on it. Two direct optimizations of the algorithm are then described. It is then shown how our method can be combined with the state collapsing compression method to obtain very compact representations. Lastly, we close the section with a short analysis of the method.

General idea In colored Petri nets, as in many other formalisms, the transition relation is a deterministic mechanism: the firing of a transition instance at a marking leads to a single marking. On the basis of this determinism, we propose to store some markings of the reachability set in a non explicit way: instead of storing the actual value of a marking m, we only store a reference to one of its predecessors m' and a transition instance (t, c_t) whose execution leads from m' to m. Because of the determinism of the transition relation, this representation of m' is unambiguous although it is not canonical since a marking may have several predecessors. Markings stored in this manner are called Δ-markings and are said to be stored symbolically while markings stored in the usual way are said to be stored explicitly.
Storing a reference to a marking and a transition instance should obviously lead to better state representations, especially when the modeled system exhibits large state vectors. However, this representation presents a drawback: the test for checking whether or not a marking m is new or not can be significantly slowed. This test usually entails comparing m to some marking(s) m' stored in the state space. In the classical scheme, m' is stored as a vector of bits and so is encoded m before its insertion into the state space. The comparison can then be efficiently implemented by a bits vectors comparison. When the reachability set contains Δ-markings, the operation is more complicated. Let us assume that we have a sequence of markings $m_1, m_2, \ldots, m_n = m'$ such that m_1 is stored explicitly and each $m_i \neq m_1$ is stored as a Δ-marking which points to m_{i-1} with the binding (t_{i-1}, c_{i-1}) such that $m_{i-1}[(t_{i-1}, c_{i-1})\rangle m_i$. The idea is then to backtrack to m_1, and to apply to it the firing of bindings sequence $(t_1, c_1).(t_2, c_2)\ldots(t_{n-1}, c_{n-1})$ to have an "explicit view" of m'. Once this operation realized, the comparison of m and m' becomes straightforward.
We will call a *reconstitution* the operation which consists in finding the actual value from a Δ encoding, and the sequence of transition bindings which enables to reconstitute a marking will be called a *reconstituting sequence*. The principle of the reconstitution mechanism can be illustrated with the help of Figure 1. Let us suppose, for instance, that we have to reconstitute marking m. To do so, we will first have to backtrack to m'. Since it is not stored explicitly, we will then have to backtrack to m_0 and finally apply to it the reconstituting sequence $(t', c').(t, c)$. This operation allows us to retrieve the actual value of marking m.

The run time overhead caused by the method directly depends on the lengths of the reconstituting sequences that are fired when the algorithm has to determine whether a marking is new or not. In order to place an upper bound on the length of these sequences we use the underlying idea of the stratified caching strategy [10]. We use a parameter k_δ defined by the user which belongs to set of positive integers. During the exploration of the state space, each marking met at a depth d such that $d \mod k_\delta = 0$ is stored explicitly. All other markings are stored symbolically and point to one of their predecessors (by a doted arc on the figure). By this way, we can guarantee that the length of each reconstituting sequence is bounded by $k_\delta - 1$. This idea is illustrated by Figure 1.

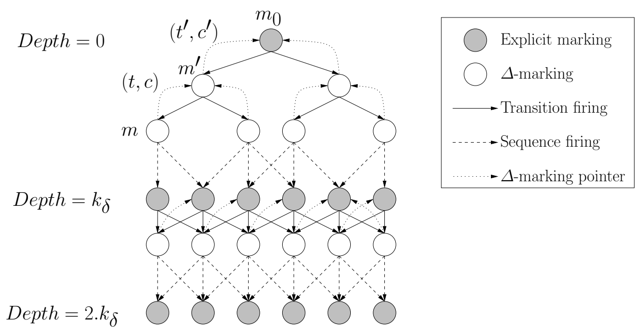

Fig. 1. A state space with Δ-markings

The algorithm We propose a search algorithm based on our method. Let us point out that we arbitrarily choose to present here a depth first search algorithm but that our method is not specific to this kind of search. The algorithm is given in a pseudo code form in Figure 2. For the sake of simplicity, no distinction is done between a transition and a transition instance.
Markings are stored in a hash table S using a hash function noted HASH. To manage collisions, each slot of the hash table contains a list of markings. When a marking m is encountered we compare it to each marking m' of the list contained in the slot HASH(m) to check whether it is new or not. To achieve this, the function RECONSTITUTE is used to reconstitute a marking stored in the hash table. It recursively backtracks to a marking stored explicitly and apply to it the correct reconstituting sequence. Finally, if m is not in the hash table, it is added to the list of slot HASH(m). Depending on its depth, it is stored explicitly or symbolically. If it is stored symbolically, it points to its predecessor in the search stack. To achieve this, three additional parameters are passed to the DFS procedure: the depth of the marking to explore, the predecessor of the marking in the search stack, and the transition which leads from the predecessor to the marking.
We can easily prove that the reconstitution function RECONSTITUTE terminates, i.e., it cannot enter in a cycle of Δ-markings, since each Δ-marking points to its predecessor in the search stack and each k_δ^{th} marking in the search stack is stored explicitly.

Speeding up the reconstitution process An efficient implementation of the reconstitution process is crucial for the performances of our method. Moreover, this makes usable the Δ-markings method with higher values of k_δ, leading to very condensed state spaces. We propose now two optimizations that aim at reducing the reconstitution times. Several experiments, that are not reported in this paper for space constraints, showed that the combination of both optimizations leads to an average reduction of the run time of 60%.

```
PROCEDURE DFS (marking $m$, int depth, stored_marking pred, transition $t$)
  $h \leftarrow$ HASH$(m)$
  for $i \in [1..$LENGTH$(S[h])]$ do
    if $m =$ RECONSTITUTE(ITH$(S[h], i)$) then return end if
  end for
  if $depth = 0$ then
    $new.type \leftarrow explicit$
    $new.m \leftarrow m$
  else
    $new.type \leftarrow delta$
    $new.pred \leftarrow pred$
    $new.t \leftarrow t$
  end if
  ADD$(S[h], new)$
  for $t \in$ ENABLED$(m)$ do
    let $m'$ be such that $m[t\rangle m'$
    DFS$(m', (depth + 1) \mod k_\delta, new, t)$
  end for
FUNCTION RECONSTITUTE (stored_marking $sm$)
  if $sm.type = explicit$ then $r \leftarrow sm.m$
  else
    $m \leftarrow$ RECONSTITUTE$(sm.pred)$
    let $r$ be such that $m[sm.t\rangle r$
  end if
  return $r$
PROCEDURE EXPLORE_STATE_SPACE ()
  INIT$(S)$
  DFS$(m_0, 0, \textbf{nil}, \textbf{nil})$
```

Fig. 2. A DFS algorithm based on the Δ-markings method

Updating the predecessors of Δ-markings The idea of this first optimization, illustrated by Figure 3, is to update the predecessor of a Δ-marking when a shorter path to an explicit marking is found. Let us suppose that the search algorithm successively visits a sequence of markings $m_1[t_1\rangle m_2[t_2\rangle \ldots [t_{n-1}\rangle m_n$ and that m_1 is its only marking stored explicitly by the algorithm. Each $m_i \neq m_1$ points to m_{i-1}. When m_n has to be reconstituted, we have to backtrack to m_1 and apply to it the reconstituting sequence $t_1.t_2 \ldots t_{n-1}$. Let us suppose that the algorithm visits later a sequence of markings $m[t\rangle \ldots [t'\rangle m'[t''\rangle m_n$ shorter than the initial one and such that m is its only marking stored explicitly. We can update the predecessor of m_n and set it to m'. As a consequence, each time the algorithm will reconstitute m_n it will backtrack to m and apply a shorter reconstituting sequence. Not only the reconstitutions of m_n will be sped up, but also the reconstitutions of all the Δ-markings which point to m_n (the set S on the figure).
Several experiments pointed out that the speed improvement is proportional to the parameter k_δ which is not surprising. Nevertheless, we expect that the

benefits obtained with this optimization decrease if some partial order technique is used in combination with our storage method. Since the goal of partial order methods is to reduce the exploration of redundant paths, the possibilities of updating the predecessor of a Δ-marking should naturally be reduced.

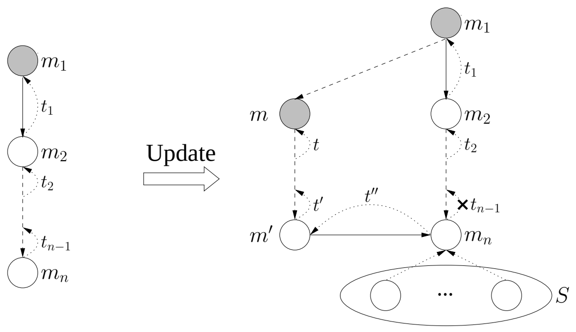

Fig. 3. Updating the predecessor of m_n

Backward firing of the reconstituting sequence This second optimization is more an implementation trick, but it reveals to be very efficient in terms of reduction of the execution time.
The reconstitution of a Δ-marking δ involves two costly operations: the decoding of the explicit marking e which enables to reconstitute δ, and the firing of the reconstituting sequence σ. In comparison, in a "classical" storage scheme, checking that two markings are equal is simply done by comparing two vectors of bits. However, this reconstitution can be avoided by performing a backward firing, i.e. an "unfiring", of sequence σ. Instead of backtracking to e and firing σ, the idea is to start from m and to unfire σ on it, i.e., find the marking m' which is such that $m'[\sigma\rangle m$. Finally, δ and m correspond to the same marking if and only if $m' = e$. If at some step of the unfiring of σ, let us say after the unfiring of σ_2 such that $\sigma = \sigma_1.\sigma_2$ we obtain a marking m'' which is such that $m''(p) < 0$ for some place p of the net, then this means that the unfiring of σ is not possible at m. We can therefore claim that the reconstitution of δ does not produce the marking m. The full backtrack to e is thus avoided, as well as the decoding of e and the unfiring of σ_1.
The possibility to reverse the transition relation is a prerequisite for this optimization. This one is met for colored Petri nets but it would be more problematic to adapt this optimization to formalisms which do not provide such a facility.

The function RECONSTITUTE of Figure 2 can thus be replaced by function UNFIRE_AND_CHECK given below. The if-statement condition at line 3 of the DFS procedure must naturally replaced by UNFIRE_AND_CHECK(ITH($S[h], i$), m).

```
FUNCTION UNFIRE_AND_CHECK (stored_marking sm, marking m)
1  if  $sm.type = explicit$  then $r \leftarrow sm.m = m$
2  else
3     let $m'$ be such that $m'[sm.t\rangle m$
4     if  $m'(p) < 0$ for some place $p$  then $r \leftarrow false$
5     else $r \leftarrow$ UNFIRE_AND_CHECK($sm.pred, m'$)
6     end if
7  end if
8  return $r$
```

A state collapsing compression scheme for colored Petri nets The state collapsing method [1] and its improvement, the recursive indexing method [2] are two state compression methods. Both are based on an assumption particularly adapted to software model checking. Thus, this technique is implemented in Bogor [20], JPF [21] and Spin [22]. This hypothesis is the following: even when the number n of syntactically possible states for a process is huge, the number m of states effectively reached by this process is usually much smaller. The idea is then to represent local process states on $log_2(m)$ bits instead of $log_2(n)$ bits. These $log_2(m)$ bits form an index of a table which is shared by all the global states and in which the actual local states are stored on $log_2(n)$ bits. Such a strategy allows to save a significant amount of memory when $n >> m$.

For colored Petri nets, we can adapt this principle on the basis of two observations:

1. Given a place, some token values within its domain may never be held in it.
2. Given a transition, some of its instances may never be firable.

This is so because the types, i.e., color domains, of the places and transitions of the net are usually over-approximations made by the user of the possible values really met during the search.
The first observation can help us to define an efficient compression function for the markings which are stored explicitly. If we note m the number of bits used to store a collapsed item of a color domain, each token value of a place p can be represented with m bits instead of using $log_2(|C(p)|)$ bits.
The second observation is useful to compress Δ-markings. By using the same parameter m, a collapsed Δ-marking will fit in

$$1 + \log_2(H) + \log_2(L) + \log_2(|T|) + m$$

bits. The first bit is used to indicate to the decoder the type of marking to decode. The “pointer” to the predecessor of the Δ-marking in the hash table is a couple (h, i) where h is the slot of the hash table which contains the predecessor, and i is its position in the slot list. If we note H the size of the hash table and L the maximal length of a slot list, $\log_2(H) + \log_2(L)$ bits are sufficient to encode this couple. At last, $\log_2(|T|) + m$ is the number of bits used to encode a collapsed transition instance (t, c_t). Without collapsing the vector, $\log_2(|T|) + \log_2(|C(t)|)$

bits are required to encode this couple.
Since m and L cannot be known before the search terminates, they must be fixed by the user. When the supplied values are not sufficient, an error is reported at the run time to the user who, in turn, can change these parameters and rerun the search. On all the experimentations we made, $L = 2^8$ and $m = 16$ were both sufficient to encode any Δ-marking. With these values each Δ-marking can be represented with approximatively 64 bits, which is the number of bits recommended in [18] to store a hash signature of a state descriptor. Thus, by combining our method with a state collapsing compression scheme we obtain a reliable storage method that performs a reduction similar to the one obtained by an unreliable method such as hashcompact.

Analysis of the method We address now the following question: "what can we expect from the method in terms of memory reduction ?". If we note N_e the number of markings stored explicitly, N_δ the number of Δ-markings, $N = N_e + N_\delta$, V_e the average size of markings stored explicitly, and V_δ the average size of Δ-markings, the total amount of memory required to store the state space will be $N_e \times V_e + N_\delta \times V_\delta$.
Now if we suppose that the proportion of markings met at depths 0, k_δ, ..., $n.k_\delta$ is $\frac{N}{k_\delta}$, we can approximate it by $(\frac{1}{k_\delta} \times N).V_e + (\frac{k_\delta - 1}{k_\delta} \times N).V_\delta$.
By collapsing Δ-markings, we have seen that V_δ becomes a constant value, typically between 8 and 12 bytes. So, for large values of k_δ, e.g., ≥ 50, the memory allocated weakly depends on the size of the state vector and can be approximated by $N \times V_\delta$.

5 Experimental results

The storage method introduced in this paper, as well as the state collapsing scheme presented in the previous section have both been implemented in Helena [23], an explicit model checker for high level Petri nets. This section presents the results of some experiments that have been made with Helena.
All the experiments described in this section have been made on a Pentium 4, 2.8 Ghz with 2 Gb of RAM. The search method used was a depth first search and the two optimizations previously presented were both turned on.

Experimentation on academic examples We first consider the six following examples frequently used in benchmarks: the distributed database system, the sieves of Eratosthene, the mutual exclusion algorithm of Peterson, a simple mutual exclusion protocol, the leader election protocol of Chang and Roberts, and lastly the dining philosophers. All these models can be found in the Helena distribution available at `http://helena.cnam.fr`. The results of our experimentations are reported in table 1. For each model, its name and its parameter are given in the first column. Four searches were done with k_δ in the set $\{5, 10, 20, 50\}$. For each of these runs, the relative performance in time and in space with respect to a classical DFS without state compression are reported respectively in columns T and S. The last row of the table contains the average values observed. The collapse method was not helpful for these simple models and therefore disabled.

Table 1. Results obtained for academic examples

		$k_\delta = 5$		$k_\delta = 10$		$k_\delta = 20$		$k_\delta = 50$	
	States	T	S	T	S	T	S	T	S
Dbm, 12	2 125 765	109%	24%	119%	16%	171%	5%	173%	4%
Eratos, 70	3 177 699	123%	40%	148%	33%	198%	29%	340%	27%
Peterson, 4	3 407 946	131%	64%	151%	60%	191%	58%	316%	57%
Mutex, 17	5 701 632	130%	58%	148%	53%	176%	50%	249%	48%
Leader, 16	10 475 430	137%	46%	176%	39%	231%	37%	329%	32%
Dining, 13	14 741 195	137%	54%	155%	48%	181%	45%	218%	44%
		128%	**56%**	**150%**	**42%**	**191%**	**37%**	**271%**	**35%**

We observe that the best compression ratio provided by our method are obtained for the two models which exhibit the largest state vectors, namely the distributed database system, and the sieves of Eratosthene. The best reduction is obtained for the distributed database system with $k_\delta = 50$. In this case, the average size of the state vector can be reduced from 148 bytes to 8 bytes. Let us note that, for this model, the size of the search stack is bounded by $2 \cdot N$. This explains why the results obtained for $k_\delta = 20$ and $k_\delta = 50$ are so close. In the opposite, for models with small state vectors, the gains obtained are quite smaller. For instance, for the Peterson model, the state vector of markings stored explicitly hardly reaches 16 bytes. Thus, we cannot expect our method to perform a reduction ratio better than $40\% - 60\%$. It appears that our method is not quite adapted for the storage of state spaces with small state vectors (< 20 bytes). That is due to the fact that the overhead needed to store a Δ-marking as well as extraneous data used to store markings, mainly pointers and additional bits added to vectors to fit in a machine word, are too important to enable a satisfactory reduction of the state space. Finally, the increase of the execution time remains acceptable for low values of k_δ but tends to grow with k_δ.

Experimentation on models extracted from programs In a second step, we made experiments on two colored Petri nets automatically translated from concurrent Ada programs by the tool Quasar [24]. Both programs make use of advanced features of Ada tasking such as dynamic task creation. The first one is a client / server program with dynamic creation of servers to handle the requests of the clients. The second one is an Ada implementation of the sieves of Eratosthene to find all the prime numbers between 2 and N.
The results of our experimentations are reported in table 2. For both programs, we made four series of tests: without any compression technique, i.e., no state collapsing and no Δ encoding, (column *No comp.*), with the state collapsing method enabled (column *Collapse*), with the Δ-markings method enabled (column Δ), and finally, with both methods enabled (column Δ + *Collapse*). For

Table 2. Results obtained for programs

	No comp.	Collapse	Δ			Δ + Collapse		
			$k_\delta = 10$	$k_\delta = 20$	$k_\delta = 50$	$k_\delta = 10$	$k_\delta = 20$	$k_\delta = 50$
Client / server program								
N=4, 10 running tasks, 34 731 states								
M	9.45	1.37	1.89	1.42	1.15	0.36	0.30	0.26
T	00:00:02	00:00:03	00:00:03	00:00:03	00:00:06	00:00:03	00:00:04	00:00:07
V	285.17	41.26	56.93	42.71	34.61	11.01	9.10	8.00
N=5, 12 running tasks, 635 463 states								
M	205.63	28.37	36.80	21.94	20.96	6.84	4.98	4.85
T	00:00:51	00:01:54	00:01:11	00:01:44	00:02:54	00:01:18	00:01:52	00:03:04
V	339.31	46.82	60.73	36.20	34.59	11.29	8.21	8.00
N=6, 14 running tasks, 13 805 931 states								
M	-	684.41	1 035.15	962.489	447.73	176.90	167.85	105.33
T	-	00:26:04	00:00:00	00:44:20	01:36:33	00:38:43	00:48:59	01:38:17
V	-	51.98	78.62	73.10	34.01	13.44	12.75	8.00

Eratosthene program								
N=20, 9 running tasks, 3 599 634 states								
M	698.74	214.51	130.70	100.72	78.67	46.15	37.28	30.75
T	00:07:10	00:08:05	00:09:37	00:12:05	00:20:25	00:11:07	00:14:11	00:22:34
V	203.54	62.49	38.06	29.34	22.92	13.44	10.86	8.96
N=25, 10 running tasks, 24 884 738 states								
M	-	-	933.75	676.24	539.47	334.79	260.289	220.77
T	-	-	01:30:13	01:53:50	03:06:11	01:45:05	02:07:02	03:22:54
V	-	-	39.35	28.49	22.73	14.11	10.97	9.30
N=30, 11 running tasks, 96 566 610 states								
M	-	-	-	-	-	1 331.37	1 026.89	843.32
T	-	-	-	-	-	10:21:50	12:25:40	17:17:31
V	-	-	-	-	-	14.46	11.15	9.16

each run, row M reports the size in megabytes of the state space, row T reports the execution time of the search in the form hh:mm:ss, and row V reports the average size of the state vector in bytes. Some searches ran out of memory. This is indicated by a "-" in the table.

The reduction of the size of the state space for these models is much more impressive than for the simple models previously considered. Even for the lowest value of k_δ the reduction observed is significant. Without any compression technique enabled, the search is limited to state spaces with a few millions of states. State collapsing provides good results for both examples for a slight increase of the run time, but used without our method it is limited to state spaces with 10^7 states. For the Eratosthene program, our method clearly outperforms its competitor in terms of memory usage, even for the lowest value of k_δ. Finally, we observe that the best results are naturally obtained when combining both methods. For the more aggressive compression scheme (Δ + Collapse, $k_\delta = 50$), the search only consumed 2% to 5% of the space required without any state compression, reducing the average size of the state vector to less than 10 bytes,

whatever the model is. Such a drastic reduction comes without an unbearable cost: for the compression scheme mentioned, the execution time only tripled on all the examples.
The results observed confirm the analysis made in the previous section: for high values of k_δ the average size of the state vector is no more related to the model, and it tends to a limit which is the size of a collapsed vector (8 bytes in these examples).

6 Discussion and perspectives

The two main techniques that are employed by model checkers to tackle the state explosion problem are partial order methods, e.g., persistent sets, and symmetry based reductions.
It is straightforward to see that the Δ-markings method is fully compatible with the first ones. Indeed, partial order methods are based on a selective search algorithm which only selects at each state a subset of the enabled transitions to generate the immediate successors of the state. It is therefore fully independent from the representation of the state space used.
Concerning symmetry reduction, the problem is more difficult. Computing symmetries often requires to permute threads or objects in the state vector. This means that the difference between two states is no more as simple as a single transition instance. Some other informations may thus also be saved in a Δ-marking to represent symmetries. Consequently, the algorithm could need more space, and also more time since the reconstitutions could be slowed. The problem of an efficient combination of the two methods is therefore an open problem that we are currently addressing.

When model checking problems of industrial sizes, it is preferable to not keep in memory all the visited states. Some techniques designed to achieve this have been presented in section 2. At first glance, it seems hard to combine our method with a "partial storage" method. This is so because Δ-markings point to some other states of the state space, disallowing, a priori, the possibility to not store every state in the reachability set. However, we see a possibility to combine our method with Geldenhuys's stratified caching [10]. The underlying idea of this caching policy is to systematically store and keep in the cache all the states met at given depths. Strata are thus classified as available and unavailable. States belonging to available strata may be replaced by new ones during the search, while the other ones must remain in the cache. The goal is to place an upper bound on the extra work realized by limiting the lengths of redundant exploration paths.
A solution to combine both methods is to allow the replacement of markings for which we are sure that no other Δ-marking points to it. When the first replacement occurs these markings are those who do not belong to the DFS stack and which have been met at depths $k_\delta - 1$, $2.k_\delta - 1$, $\ldots$, $n.k_\delta - 1$. Indeed all the successors of these markings are stored explicitly in the state space, or

point to another marking. Once all these states of this available strata have been replaced, the next available strata are those at depths $k_\delta - 2$, $2.k_\delta - 2$, ..., $n.k_\delta - 2$ and so on. By this way, we can guarantee that, at each step of the algorithm, each Δ-marking points to a marking which has not been replaced. Figure 4 illustrates our purpose. Parameter k_δ is set to 3. In the configuration depicted, markings at depth 2 have already been replaced by the algorithm. Markings at depth 1 are now eligible for replacement. Markings m and m' will become eligible as soon as they leave the stack.

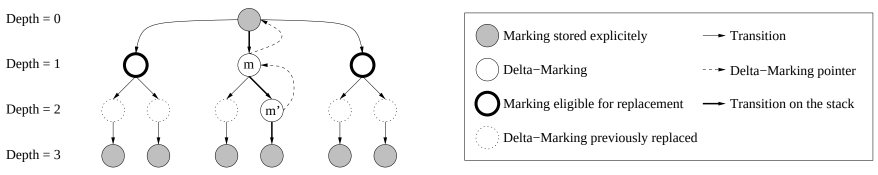

Fig. 4. Combining stratified caching and the Δ-markings method

7 Conclusions

We have presented in this work the Δ-markings method to store the state space of colored Petri nets. The basic idea of this method is to store a large set of states of the system in a non explicit way by only storing references on other states. Some optimizations which considerably reduce the run time penalty caused by the method have also been presented. In addition, our method can be combined with a state collapsing compression scheme to push the compression one step further. This simple idea reveals to be very useful in practice. The results of our experiments have shown the efficiency of our approach. On all examples, even for models issued from the translation of concurrent Ada programs, our experiments show that the average size of the state vector is close to 10 bytes. Furthermore, our technique does not increase the execution time in an unacceptable way.
Another appreciable property of our method is that it gives to the user the ability to specify the desired compression ratio, through the parameter k_δ.

References

1. Visser W. Memory efficient state storage in spin. In *SPIN'1996*, pages 21–35.
2. Holzmann G.J. State compression in spin : Recursive indexing and compression training runs. In *SPIN'1997*.
3. Geldenhuys J. and Valmari A. A nearly memory-optimal data structure for sets and mappings. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN'2003*, volume 2648 of *LNCS*, pages 136–150. Springer.
4. Grègoire J.C. State space compression in spin with getss. In *SPIN'1996*, pages 90–108.
5. Parreaux B. Difference compression in spin. In *SPIN'1998*.

6. Geldenhuys J. and de Villiers P.J.A. Runtime efficient state compaction in spin. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *SPIN'1999*, volume 1680 of *LNCS*, pages 12–21. Springer.
7. Pastor E. and Cortadella J. Efficient encoding schemes for symbolic analysis of petri nets. In *Conference on Design, Automation and Test*, pages 790–795. IEEE Computer Society, 1998.
8. Schmidt K. Using petri net invariants in state space construction. In Hubert Garavel and John Hatcliff, editors, *TACAS'2003*, volume 2619 of *LNCS*, pages 473–488. Springer.
9. Godefroid P., Holzmann G.J., and Pirottin D. State-space caching revisited. In Gregor von Bochmann and David K. Probst, editors, *CAV'1992*, volume 663 of *LNCS*, pages 178–191. Springer.
10. Geldenhuys J. State caching reconsidered. In Susanne Graf and Laurent Mounier, editors, *SPIN'2004*, volume 2989 of *LNCS*, pages 23–38. Springer.
11. Parashkevov A.N. and Yantchev J. Space efficient reachability analysis through use of pseudo-root states. In Ed Brinksma, editor, *TACAS'1997*, volume 1217 of *LNCS*, pages 50–64. Springer.
12. Christensen S., Kristensen L.M., and Mailund T. A sweep-line method for state space exploration. In Tiziana Margaria and Wang Yi, editors, *TACAS'2001*, volume 2031 of *LNCS*, pages 450–464. Springer.
13. Mailund T. and Westergaard M. Obtaining memory-efficient reachability graph representations using the sweep-line method. In Kurt Jensen and Andreas Podelski, editors, *TACAS'2004*, volume 2988 of *LNCS*, pages 177–191. Springer.
14. Schmidt K. Automated generation of a progress measure for the sweep-line method. In Kurt Jensen and Andreas Podelski, editors, *TACAS'2004*, volume 2988 of *LNCS*, pages 192–204. Springer.
15. Behrmann G., Larsen K.G., and Pelánek R. To store or not to store. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV'2003*, volume 2725 of *LNCS*, pages 433–445. Springer.
16. Godefroid P. Model checking for programming languages using verisoft. In *POPL'1997*, pages 174–186.
17. Holzmann G.J. *Design and validation of computer protocols.* Prentice Hall, 1991.
18. Leroy D. and Wolper P. Reliable hashing without collision detection. In Costas Courcoubetis, editor, *CAV'1993*, volume 697 of *LNCS*, pages 59–70. Springer.
19. Jensen K. Coloured petri nets: A high level language for system design and analysis. In Grzegorz Rozenberg, editor, *ATPN'1989*, volume 483 of *LNCS*, pages 342–416. Springer.
20. Dwyer M.B., Hatcliff J., Iosif R., and Robby. Space-reduction strategies for model checking dynamic software. *Electronic Notes in Theoretical Computer Science*, 89(3), 2003.
21. Lerda F. and Visser W. Addressing dynamic issues of program model checking. In Dragan Bosnacki and Stefan Leue, editors, *SPIN'2001*, volume 2057 of *LNCS*, pages 80–102. Springer.
22. Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
23. Evangelista S. High level petri nets analysis with helena. In Gianfranco Ciardo and Philippe Darondeau, editors, *ATPN'2005*, volume 3536 of *LNCS*, pages 455–464. Springer.
24. Evangelista S., Kaiser C., Pradat-Peyre J.F., and Rousseau P. Quasar : a new tool for analyzing concurrent programs. In Jean-Pierre Rosen and Alfred Strohmeier, editors, *Ada-Europe'2003*, volume 2655 of *LNCS*, pages 168–181. Springer.