A Semi-Explicit Method to Store State Spaces in a Compact Way

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 and algorithms based on an explicit states enumeration. Thus, techniques which allow to represent the states efficiently are precious for these tools. We present in this work 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 states may not be explicitly represented in the state space. Experiments show that very compact representations are obtained, with an acceptable increase of the run time.

1 Introduction

Model checking is a powerful and automatic tool for the verification of finite state systems. It consists in enumerating all the possible configurations (or states) or executions of the system to track the ones which do not match the specification. In order to avoid redundant computations or to prevent from entering cycles, model checking algorithms have to keep a trace of the visited states in a state space (or reachability set). To achieve this goal, two strategies exist : symbolic and explicit model checking.
In symbolic model checking [1–4], the reachability set is stored in a data structure, e.g., a binary decision diagram (BDD) or a variant, which enables the sharing of data between states. The transition relation is also represented symbolically and the reachability set is generated incrementally. At each step a set of states is processed and the set of newly reachable states is added to the reachability set. Symbolic model checking has already been successfully applied to the verification of systems with 10^{150} states [5].
In explicit state model checking, states of the system are processed one by one. Each new encountered state is added to the reachability set and represented explicitly in a data structure, e.g. a hash table, which disables the sharing of data. For this reason, the limited amount of memory is the major bottleneck of explicit model checking. Indeed, due to the state explosion problem inherent to the concurrent execution of several components, the number of possible configurations can be far too large for a complete enumeration. Explicit model checking is typically limited to the verification of systems with 10^6-10^8 states. To

attack this problem, numerous techniques have been proposed to represent the state space in a compact way. Examples include state collapsing [6], recursive indexing [7], sharing trees [8], state compaction [9], difference compression [10], very tight hashing [11], and representations based on firing counts vector [12]. Each method tries to provide efficient compression ratios without an unbearable increase of the execution time.
It is commonly accepted that symbolic model checking fits better for the verification of hardware systems which exhibit regular state spaces whereas explicit model checking has advantages for the verification of software. Possible reasons are that software are usually less structured than hardware and make heavy use of structured data types which are not adapted for a symbolic representation.

This work proposes a new method to store the state space. Though it belongs to the second family, we qualify it as semi-explicit since states may not be explicitly represented in the state space. Colored Petri nets is the formalism used to illustrate our purpose. However, the method proposed is not specific to colored Petri nets and we believe that it could easily be adapted to other formalisms. Thus we will try to formulate our ideas in very general terms.

The remainder of this work is organized as follows. Section 2 recalls some basic concepts on Colored Petri nets. Our storage method is presented in section 3. A set of experimental results are presented in Section 4. At last we conclude in Section 5 and we briefly present some directions that could take our future works.

2 Colored Petri nets

We give in this section some basic concepts and notations on colored Petri nets. A more complete definition can be found in [13]. 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 naturally. States of colored Petri nets are also called markings.

Definition 1. *A colored Petri net is a tuple $\langle P, T, \Sigma, C, W^-, W^+, m_0\rangle$ where P is a finite set of* **places***; T is a finite set of* **transitions** *such that $P \cap T = \emptyset$; Σ, the* **colours 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***, associates to each $p \in P$ an element of $Bag(C(p))$.*

A couple (e, c) with $e \in P \cup T$ and $c \in C(e)$ is called an instance of e. The term token is used to denote an item of $\cup_{p \in P} C(p)$. Color domains will be defined as cartesian products of finite sets, and color mappings of incidence matrixes will be noted as linear combinations of tuples of expressions. For instance if we let $C = \{1,\ldots,N\}$ and $x, y \in C$ then $\langle x,1\rangle + 2.\langle y,2\rangle$ is a mapping from $C \times C$ to $Bag(C \times C)$ which maps each couple $\langle x,y\rangle$ to one token $\langle x,1\rangle$ and two tokens $\langle y,2\rangle$. The set of syntactically possible markings of a net N is noted $\mathbb{M}_N$ and is defined as the set of mappings which map each $p \in P$ to an element of $Bag(C(p))$. The firing rule, i.e., transition relation, and the state space of colored Petri nets are formally defined below.

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 $m \in \mathbb{M}_N$.*

- *(t, c_t) is* **firable***, i.e., enabled, 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 $\{m_0\} \cup \{m \in \mathbb{M}_N \mid \exists m' \in \mathbb{R}_N, t \in T, c_t \in C(t) \mid m'[(t,c_t)\rangle m\}$*

As an example, let us consider figure 1. This one illustrates a colored Petri net of the dining philosophers problem for N philosophers numbered from 0 to $N-1$. Forks are also numbered this way. Places are represented by ovals, transitions by lines. Arcs and their valuations represent the incidence matrixes W^- and W^+. Color domains of places can be easily deduced from the inscriptions which label their connected arcs. For transitions, color domains are given by the variables which appear on their connected arcs : each variable corresponds to a component of the color domain.
When a philosopher p wants to eat, he first takes a seat. Then he successively takes his left (p) and his right ($p+1$) fork. Once his meat finished, he simultaneously puts down his two forks and leaves the table. Initially, all the philosophers are thinking, forks are all free and the number of free chairs is $N-1$.

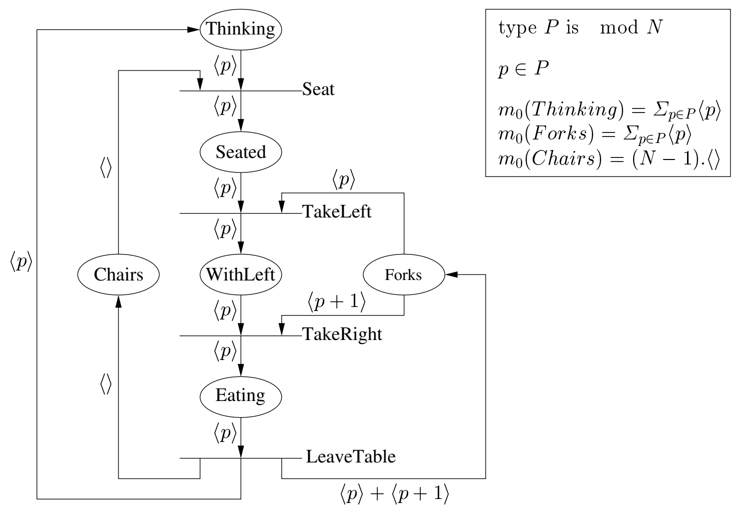

Fig. 1. Model of the dining philosophers

3 The Δ-markings storage method

This section describes the Δ-marking method and is organized as follows. Firstly we present the general idea of the method. In a second time we give a depth first search algorithm based on it. A direct optimization of the algorithm is then described. The last part of the section details the encoding of state vectors, i.e., bits vectors in which states are encoded before their insertion into the state space.

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 by only storing a reference on a predecessor of the marking coupled with the firable transition instance which leads from the predecessor to the marking. Such markings are called Δ-markings. Markings stored in the usual way are said to be stored explicitly whereas Δ-markings are said to be stored symbolically. This underlying idea is illustrated by figure 2. During the exploration of the state space, each marking at a depth d such that $d \mod k_\delta = 0$ (with k_δ a user defined parameter) is stored explicitly. These ones are the grayed markings on the figure. All other markings are stored symbolically. Each one points on one of its predecessors (doted arc). This link states that this marking is obtained from the predecessor by the firing of some instance. It implies that the symbolic representation of a marking is not unique since a marking may be reachable from several markings. Storing a reference on a marking and a transition instance should obviously lead to better state representations, especially when the modeled system exhibits large state vectors.

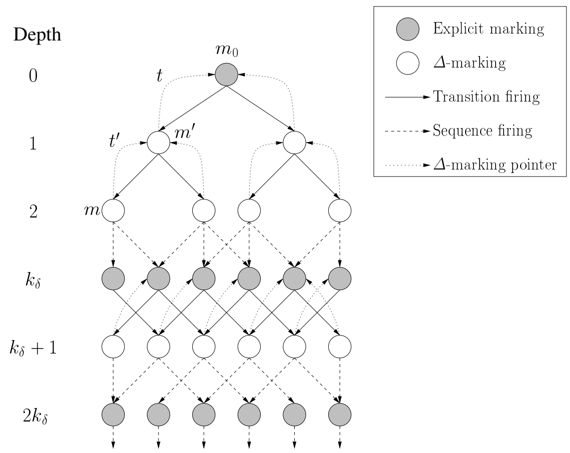

Fig. 2. A state space with Δ-markings

The algorithm We propose a search algorithm based on our method. Let us point out that we arbitrarily chose 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 3. For sake of simplicity, no distinction is done between a transition and a transition instance.
Markings are stored in a hash table S. Its size is noted H. To manage collisions, each slot of the hash table contains a list of markings. A hash function *hash* maps each marking to the range $[1..H]$. Each time a marking m is encountered we compare it to each marking m' of the list contained in slot $hash(m)$ to check whether it is new or not. If m' is stored explicitly, the comparison is straightforward. In the case of a Δ-marking, m' contains both a reference on another marking m_r of the hash table and the transition t such that $m_r[t\rangle m'$. If m_r is stored explicitly, we apply to it the firing of t to have an explicit view of m'. If m_r is a Δ-marking we look at the marking pointed by m_r and so on until an explicit marking is found. When such a marking is found, we apply to it the correct firings sequence to recover m'. Such an operation is called a *reconstitution* and the sequence which enables to reconstitute a Δ-marking is called a *reconstituting sequence*. Once the marking reconstituted, the comparison becomes straightforward. Let us go back to figure 2 and suppose, for instance, that we have to reconstitute marking m. To do so, we will have to backtrack first to m' and then to m_0 and apply to it the reconstituting sequence $t.t'$. 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 on its predecessor in the search stack. To achieve this, three additional parameters are passed to the depth first search procedure `DFS` : 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 on its predecessor in the search stack and each k_δ^{th} marking in the search stack is stored explicitly.

The choice of the parameter k_δ of the method has a great impact on the performances of the algorithm. Choosing a high value of k_δ should naturally lead to more condensed state spaces but also slow the reconstitutions, and thus the search. On the other hand, a low value of k_δ involves a small run time overhead but bigger state spaces.

Updating the predecessors of Δ-markings Lines $6-11$ of procedure `DFS` constitutes an optimization of the algorithm. The idea, illustrated by figure 4, 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 $m_0[t_0\rangle m_1[t_1\rangle \ldots [t_{n-1}\rangle m_n$ and that m_0 is its only marking stored explicitly by the algorithm. Each $m_i \neq m_0$ points on m_{i-1}. When m_n has to be reconstituted, we have to backtrack to m_0 and apply to it the reconstituting

```
PROCEDURE DFS (marking m, int depth, stored_marking pred, transition t)
h ← HASH(m)
(* loop on the markings of slot h to check whether m is new or not *)
for i ∈ [1..LENGTH(S[h])] do
   sm ← ITH(S[h], i)
   if m = RECONSTITUTE(sm) then
      (* update the predecessor if a shorter path to an explicit marking is found *)
      if depth ≠ 0 ∧ depth < sm.depth then
         sm.pred ← pred
         sm.t ← t
         UPDATE_ITH(S[h], i, sm)
      endif
      return
   endif
endfor
(* add the marking to the state space *)
if depth = 0 then
   new.type ← explicit
   new.m ← m
else
   new.type ← delta
   new.pred ← pred
   new.t ← t
endif
new.depth ← depth
ADD(S[h], new)
(* explore all the successors of m *)
for t ∈ ENABLED(m) do
   (* fire t to obtain a new marking m′ *)
   ∀p ∈ P, m′ ← m − W⁻(p, t) + W⁺(p, t)
   DFS(m′, (depth + 1) mod k_δ, new, t)
endfor

FUNCTION RECONSTITUTE (stored_marking sm)
(* reconstitute stored marking sm *)
if sm.type = explicit then
   result ← sm.m
else
   (* reconstitute the predecessor *)
   m ← RECONSTITUTE(sm.pred)
   (* the result is obtained by firing the transition on the reconstituted predecessor *)
   ∀p ∈ P, result ← m − W⁻(p, sm.t) + W⁺(p, sm.t)
endif
return result

PROCEDURE EXPLORE_STATE_SPACE ()
(* initialize the state space and launch the exploration of the initial marking *)
INIT(S)
DFS(m_0, 0, nil, nil)
```

Fig. 3. A depth first search algorithm based on the Δ-markings storage method

sequence $t_0.t_1 \ldots t_{n-1}$. 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 on m_n (the set S on the figure). Experimental results of section 4 show that we can hope for a speed improvement proportional to the parameter k_δ which is not surprising.

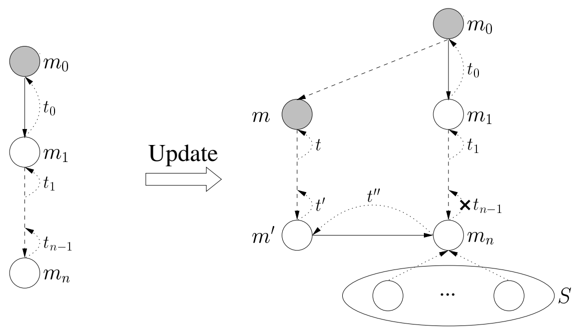

Fig. 4. Updating the predecessor of m_n

Encoding state vectors In order to save memory, markings are encoded as bits vectors before their insertion into the state space. To have an accurate idea of the benefits we can expect from the Δ-markings method we detail here the encoding of Δ-markings. Our encoding scheme for explicit markings is a slight modification of the one used in the tool Maria [14] and will not be described here.

To ensure a correct reconstitution, each Δ-marking must contain two different fields. The first one is the "address" of the predecessor in the hash table. It is a couple (h, i) where h is the slot of the hash table which contains the predecessor, and i is the position in the slot list of the predecessor. If H is the size of the hash table then $log_2(H)$ bits are sufficient to encode h. The number of bits required to store i depends on the number of collisions that will occur during the search and is thus a direct function of H. By default, we can use $I = 8$ binary digits to store i. The second element of a Δ-marking is the transition instance fired from the predecessor to reconstitute the marking. It is a couple (t, c_t) with $t \in T$ and $c_t \in C(t)$. Encoding this couple is straightforward. $log_2(|T|)$ binary digits are needed to encode t while c_t requires $log_2(|C(t)|)$ binary digits. A last bit is necessary to indicate to the decoder the type of marking to decode. To summarize, the number of binary digits required to encode a Δ-marking obtained by the firing of transition t is

$$1 + log_2(H) + I + log_2(|T|) + log_2(|C(t)|)$$

We easily see that the size of a Δ-marking is a logarithmic function of the size of the color domains whereas the size of an explicit marking is usually at least a linear function of this size. For instance, if we consider the dining philosophers of figure 1, the overhead of $1 + log_2(H) + I + log_2(|T|)$ is only penalizing when $N \leq 4$. Below this value, our method offers no advantage and will rather slow the search. On the other hand, for $N = 16$ philosophers, $H = 2^{20}$, the average sizes of an explicit marking and a Δ-marking are respectively 195 and 36 bits.

Encoding a color of a transition t with $log_2(|C(t)|)$ bits makes the assumption that a transition can potentially be fired with all the possible colors within its domain. This is often not the case in practical models and it could be considered in future works to combine our method with Visser's state collapsing method [6] to reduce the number of binary digits required to store a transition color.

4 Experimental results

This section presents a set of experimental results. The first example considered is the distributed database system (taken from [14]), a well known example of the colored Petri nets literature. In a second example, we consider the dining philosophers model of figure 1. All the experiments presented in this section have been realized on a Pentium 4 with 2.5 Ghz and a main memory of 2 Gb. Times reported are expressed in seconds.

The distributed database system (figure 5) We consider in this example a set of N database managers which communicate to maintain consistent replica of a database. When a manager updates its local copy of the database, he sends requests to other managers for updating their local copy. As soon as a manager receives such a request he starts the update of its copy. Its update finished, each manager acknowledges the initiating manager. This process finishes when

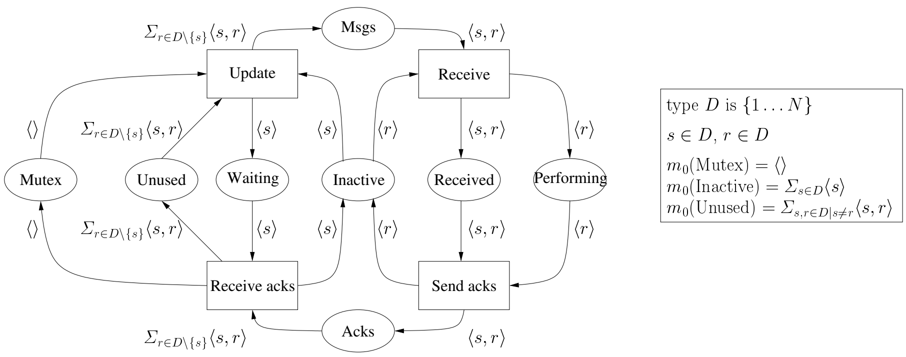

Fig. 5. The distributed database system

the initiating manager collects all the acknowledgments. Managers can be either `Inactive`, either `Waiting` for acknowledgments, either `Performing` an update. Places `Msgs`, `Received`, `Acks` and `Unused` simulate communication channels between sites. Thus, $N.(N-1)$ tokens are distributed upon these places at each marking. At last the correctness of the protocol is ensured by place `Mutex` which guarantees that two managers can not concurrently update their local copy.

Table 1 presents some results obtained for this model with $k_\delta = 1$, i.e., by storing all states explicitly. For several values of the number of managers N, the table reports the number of reachable states (column *States*), the time measured for the exploration of the whole reachability set (column *Time*) and the size of the state space in bytes (column *Space*). This size includes additional pointers used to store markings but does not include the hash table itself.

N	*States*	*Time*	*Space*
8	17 497	2	1 037 592
9	59 050	9	5 268 788
10	196 831	40	21 257 168
11	649 540	172	83 140 760
12	2 125 765	770	322 639 856
13	6 908 734	3 696	1 213 125 024

Table 1. Results obtained for the "explicit" state space generation

The Δ-markings method has been applied on this example for values of k_δ in the set $\{2, 5, 10, 20, 50\}$ and the results have been reported in table 2. For each of these values, the table reports the relative performance in space (column S) and time (column T) with respect to the results of table 1. It is important to notice that, for this example, the maximum size of the stack search is $2.N$. Consequently, for values of k_δ greater than $2.N$, the only marking stored explicitly is the initial marking.

	$k_\delta = 2$		$k_\delta = 5$		$k_\delta = 10$		$k_\delta = 20$		$k_\delta = 50$	
N	S (%)	T (%)	S (%)	T (%)	S (%)	T (%)	S (%)	T (%)	S (%)	T (%)
8	56.92	150.00	30.41	150.00	24.09	200.00	13.49	250.00	13.49	250.00
9	55.16	155.56	27.47	177.78	23.35	188.89	8.97	233.33	8.97	233.33
10	53.95	150.00	26.07	185.00	22.17	200.00	7.41	245.00	7.41	245.00
11	53.12	158.14	25.02	194.77	19.52	208.14	6.26	254.07	6.25	254.12
12	52.68	156.88	24.19	194.16	16.03	211.17	5.40	255.71	5.25	256.36
13	52.31	158.33	23.56	197.89	12.85	219.81	5.06	261.88	4.44	263.80

Table 2. Relative performance of the Δ-markings method for several values of k_δ

The results obtained are quite encouraging. Even for small values of k_δ the reduction of the size of the state space is significant. The distributed database system is typically the kind of model for which our method offers an interesting alternative compared to a classical explicit storage. This can be simply explained by the fact that the model exhibits large state vectors. If we compare the size obtained for $N = 10$ with $k_\delta = 20$ (1 574 712 bytes) with the one obtained by Maria ([14], table 1 : 18 006 540 bytes) we observe a reduction factor of more than 11. Moreover, for the highest values of k_δ, it seems that extraneous data, i.e. pointers, seem to dominate the total memory requirements. Lastly, the increase of the execution time remains acceptable and never exceeds a factor of 3. Nevertheless, we must observe that this increase tends to grow with N for a same value of k_δ. The reason is that the number of tokens in places grows with N, leading to slower reconstitutions.

The dining philosophers (figure 1) For this problem, the gain is less impressive than for the first example as the size of explicit markings is quite small. The reduction factor of the state space never exceeds 2 even for highest values of k_δ. It appears that our method is not quite adapted for the exploration and storage of state spaces with small state vectors. 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.
The goal of this example is to show how the optimization presented in section 3 can significantly reduce the run time requirements. Table 3 reports, for several values of the parameter N (the number of philosophers), the number of reachable states (column $States$) and the execution time (column $Time$) for the generation of the "explicit" reachability set, i.e. without Δ-markings. In addition, we have measured the execution times of the generation of the reachability set with the Δ-markings method for k_δ in the set $\{10, 50, 100, 1000\}$ and reported for each of theses values the relative performances without (column T) and with (column T^*) the optimization with respect to results of column $Time$. A timeout was set to three hours after which the search was interrupted. It is indicated by a $-$ in the table. It is not surprising to observe that the best optimizations are obtained for high values of k_δ. Indeed, the farer a Δ-marking is from an explicit marking, the better updates we can observe. Consequently, this optimization makes it possible to use the Δ-markings method with high values of k_δ.

			$k_\delta = 10$		$k_\delta = 50$		$k_\delta = 100$		$k_\delta = 1000$	
N	$States$	$Time$	T (%)	T^* (%)	T (%)	T^* (%)	T (%)	T^* (%)	T (%)	T^* (%)
10	321 667	19	184.21	163.16	415.79	210.53	673.68	252.63	6 100.00	584.21
11	1 153 351	71	195.60	187.32	446.48	223.94	746.48	264.79	6 684.51	616.90
12	4 126 351	285	199.30	171.22	447.37	224.21	750.88	264.96	-	600.00
13	14 741 195	1133	205.12	176.08	459.40	231.69	769.81	270.08	-	615.89

Table 3. Execution times for several values of k_δ with and without optimization

5 Conclusions and perspectives

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. A set of experiments have shown the efficiency of our approach. Furthermore, our technique does not increase the execution time in an unacceptable way. We have also given an optimization which enables to considerably reduce the run time penalty caused by the method.

Future works should take the three following directions :

1. Combine our method with other techniques. We think that a combination with state collapsing [6] should be particularly adapted to reduce even more the size of Δ-markings.
2. Explore a set of possible optimizations that have not been deeply studied yet and thus not pointed out in this paper.
   A first optimization would be to use some caching techniques in the reconstitution process in order to avoid backtracking to an explicit marking at each reconstitution. An efficient implementation of the reconstitution function is crucial for the performance of our method. Moreover, it will enable to use the Δ-markings method with higher values of k_δ, leading to more condensed state spaces.
   A second optimization would be to use the invariants of the net to store Δ-markings more compactly. Indeed, these invariants can be helpful to detect some redundancies in the models. For instance, if we consider the transition `Send Acks` in the distributed database system we observe that the instantiation value of variable s can be deduced from the value of variable r, i.e. at a given marking a site r can not acknowledge two different sites. Thus this variable can be omitted when storing a Δ-marking obtained by the firing of this transition. Making use of the invariants of the net to obtain better encoding schemes is not a new idea and it has been investigated in [12, 15, 16] in which they use ordinary Petri nets. The computation of colored invariants is a tough challenge. However, in our sense, a user with a knowledge of the analyzed system can provide a model checking tool with some basic invariants of the net which correspond to the represented entities : processes, critical sections, etc.
   At last, we will consider modifying the algorithm presented in section 3 in order to support dynamic variations of the parameter k_δ. We could imagine to start the algorithm with a very low value of k_δ in order to fasten the search, and to increase it as the amount of available memory decreases.
3. Include this storage method in Quasar [17]. Quasar is a tool for the verification of concurrent Ada software based on colored Petri nets. We believe that the Δ-markings method should be particularly efficient within the scope of the verification of software as these make heavy use of structured data types and usually exhibit large state vectors.

References

1. McMillan K.L. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
2. Couvreur J.-M., Encrenaz E., Paviot-Adet E., Poitrenaud D., and Wacrenier P.-A. Data decision diagrams for petri net analysis. In *Proc. of the 23rd International Conference on Applications and Theory of Petri Nets*, volume 2360 of *LNCS*, pages 1–101. Springer, 2002.
3. Ciardo G. and Miner A.S. Efficient reachability set generation and storage using decision diagrams. In *Proc. of the 20th International Conference on Applications and Theory of Petri Nets*, volume 1630 of *LNCS*, pages 6–25. Springer, 1999.
4. Pastor E., Roig O., Cortadella J., and Badia R. M. Petri net analysis using boolean manipulation. In *Proc. of the 15th International Conference on Application and Theory of Petri Nets 1994*, volume 815 of *LNCS*, pages 416–435. Springer, 1994.
5. Eisner C. and Peled D. Comparing symbolic and explicit model checking of a software system. In *Proc. of the 9th International SPIN Workshop on Model Checking of Software*, volume 2318 of *LNCS*, pages 230–239. Springer, 2002.
6. Visser W. Memory efficient state storage in spin. In *Proc. Second Spin Workshop*, Rutgers University, New Jersey, USA, August 1996.
7. Holzmann G.J. State compression in spin : Recursive indexing and compression training runs. In *Proc. Third Spin Workshop*, 1997.
8. Grègoire J.C. State space compression in spin with getss. In *Proc. Second Spin Workshop*, Rutgers University, New Jersey, USA, August 1996.
9. Geldenhuys J. and de Villiers P.J.A. Runtime efficient state compaction in spin. In *Proc. of the fifth Spin Workshop*, volume 1680 of *LNCS*, pages 12–21. Springer, 1999.
10. Parreaux B. Difference compression in spin. In *Proc. Fourth Spin Workshop*, 1998.
11. Geldenhuys J. and Valmari A. A nearly memory-optimal data structure for sets and mappings. In *Proc. of the 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 136–150, 2003.
12. Chiola G. and Carvajal-Schiaffino R. A reachability graph construction algorithm based on canonical transition firing count vectors. In *Proceedings of 9th International Workshop on Petri Nets and Performance Models*, pages 113–122, 2001.
13. Jensen K. Coloured petri nets: A high level language for system design and analysis. In *Advances in Petri Nets*, volume 483 of *LNCS*, pages 342–416. Springer, 1991.
14. Marko Mäkelä. Condensed storage of multi-set sequences. In Kurt Jensen, editor, *Practical Use of High-Level Petri Nets*, number 547 in DAIMI report PB, pages 111–125. University of Århus, Denmark, June 2000.
15. Pastor E. and Cortadella J. Efficient encoding schemes for symbolic analysis of petri nets. In *Proc. of the conference on Design, automation and test in Europe*, pages 790–795. IEEE Computer Society, 1998.
16. Schmidt K. Using petri net invariants in state space construction. In *Proc. of the 9th International Conference : Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 473–488. Springer, 2003.
17. Evangelista S., Kaiser C., Pradat-Peyre J.F., and Rousseau P. Quasar : a new tool for analyzing concurrent programs. In *Proc. of the Ada-Europe International Conference on Reliable software technologies*, volume 2655 of *LNCS*, pages 166–181. Springer, 2003.
18. Leroy D. and Wolper P. Reliable hashing without collision detection. In *Computer Aided Verification : Proc. of the 5th International Conference CAV'93*, volume 697 of *LNCS*, pages 59–70. Springer, 1993.