Extreme Symmetries in Complex Distributed Systems: the Bag-Oriented Approach*

M. Colange[1], L.M. Hillah[2], F. Kordon[1], and P. Parutto[1]

[1] LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6
4, place Jussieu, F-75252 Paris Cedex 05, France
Maximilien.Colange@lip6.fr, Fabrice.Kordon@lip6.fr
[2] LIP6, CNRS UMR 7606 and Université Paris Ouest Nanterre La Défense
200, avenue de la République, F-92001 Nanterre Cedex, France
Lom-Messan.Hillah@lip6.fr

Abstract. Model checking is widely used as an automatic exhaustive verification technique to check properties of complex systems. However, it is difficult to operate in the context of today's emerging systems that combine distribution (and asynchronous communications) together with a large size (and a hierarchical composition of components – and thus, of specifications).
This paper combines existing techniques tackling the known combinatorial explosion of model checking. To achieve this, we exploit the structure of such distributed systems (symmetries and hierarchical composition), thus allowing a better compression factor and calculus factorization in favorable cases. We present these techniques and assess their impact on some benchmark examples.

Keywords: Symmetric Nets with Bags, formal method, model checking, state space generation, Symmetries-based techniques, Hierarchical Set Decision Diagrams.

1 Introduction

Context Model checking is now widely used as an automatic and exhaustive verification technique to check properties of complex systems. However, this approach suffers from an intrinsic combinatorial explosion issue that must be tackled. One trend is to take full advantage of the characteristics of the class of system being analyzed. A first example of this was, in the 1990's, the exploitation of the characteristics of hardware systems [7].

Today's emerging complex systems have two main characteristics. Firstly, they are more and more distributed: numerous entities, often sharing the same code (only the context differs), communicate asynchronously. Secondly, these entities are often hierarchically organized: systems are composed of systems (SoS *for Systems of Systems*).

Moreover, these emerging systems handle more and more critical functions and need to be trusted. Their complexity prevents traditional test or simulation based approaches to reach a satisfactory level of confidence, formal methods, such as those based on state space analysis, must be operated. However, the asynchronous nature of

* This work was partially supported by a grant from the Direction Générale pour l'Armement.

such systems, as well as their size, increase the combinatorial explosion of these analysis techniques, thus preventing their use in satisfactory conditions.

Problem One issue in tackling the combinatorial explosion related to state space analysis is to combine existing reduction techniques. Among them, several are commonly involved in the analysis of distributed systems:

1. Partial Orders: this technique aims at fighting the interleaving introduced when several execution flows are running in parallel. When several paths lead from state A to state B in the state space, only one is stored. This technique may reduce the explored state space by orders of magnitude in favorable cases [16, 3].
2. Symmetries: this technique aims at identifying the possible permutations of actors in a system (i.e. all clients are identical up to a permutation of identity). Instead of building the explicit state space, symmetries allow to compute a *quotient* state space that can be exponentially smaller in favorable cases. This technique was introduced for Petri nets in the early 1990's [6] and then adapted in a more general case [23]. A variant, called "counter abstraction" [2], also allow to consider as a whole groups of processes.
3. Locality: this technique aims at exploiting the locality of the system's evolution. Typically, when one process evolves, the set of variables to be changed is very small compared to the state of the full system. This locality property allows to share the representation of common parts in the state space. The use of appropriate data structures to represent the state space such as decision diagrams leads to exponential gains in favorable cases [5].

Preliminary experiments [28, 10] have demonstrated that the two last techniques can be combined, despite the fact that they appear to be independent. The idea is to achieve this in a more efficient way and reduce the need for the partial order reduction by using a dedicated modeling technique, since it seems difficult to combine the three techniques together.

Contribution This paper proposes a method for structuring symmetries in a system model by means of bags. The idea is to make a link between potentially useful modeling constructs and efficient model checking mechanisms in order to improve the combination of reduction techniques. The objective is twofold:

– Reducing again the potential interleaving, thus decreasing the need for partial order techniques,
– Exploiting the hierarchical structure of symmetries for a better encoding of the quotient state space into decision diagrams, and thus, increasing the combined efficiency.

Altogether, these combined techniques should help us provide efficient state space generation and state space analysis for distributed systems.

To achieve this, we rely on the use of bags to structure data carried out in a system. Thus, we use Symmetric nets with Bags (SNB) [18] that are a compact and readable dialect of colored Petri nets, allowing structured specification of complex systems. However, we claim it can be generalized to other notations dedicated to concurrent systems as soon as a structural analysis, allowing to detect permutations and a hierarchical structuring, can be performed.

We call this approach *"extreme symmetries"* because we make an intensive use of these in different ways, as it is explained in the discussion part of section 2.1.

Contents Section 2 presents basic definitions of the formal concepts the presented method relies on. Then, section 3 details the main principles of our contribution. Finally, section 4 presents an early performance evaluation of the method by means of selected examples.

2 Definitions

This section provides the definitions needed in this paper: Symmetric Nets with Bags (SNB) and Decision Diagrams (DD). The goal of this section is to provide an overview for the understanding of the paper only. However, there are references to formal and precise definitions.

Symmetric Nets with Bags are used to model systems with the possibility to structure data in a new way where hierarchical information can be exploited to tackle complexity. Decision Diagrams is a commonly accepted technique to represent state spaces in a very compact way. In this paper, we focus on a specific class of decision diagrams: Set Decision Diagrams (SDD), that can be composed hierarchically.

2.1 Symmetric Nets with Bags

Petri Nets [15] They are a well-known formalism for the modeling of asynchronous systems. Basically, a Petri Net (or P/T net) consists in a set of places and a set of transitions. Places contain tokens, and transitions move tokens from place to place. More precisely, when it *fires*, a transition consumes tokens from its input places, and produces tokens to its output places, thus reaching a new marking (vector of tokens in places). By applying a fixed point on the firing relation, one can theoretically generate the full state space for finite systems.

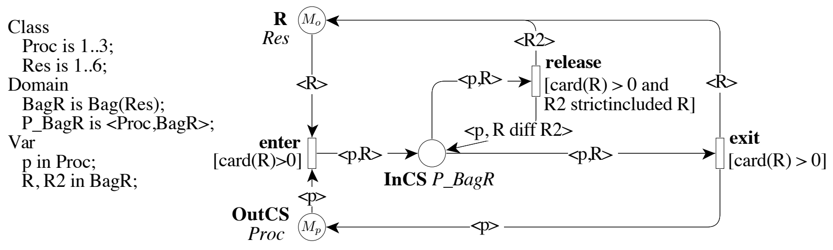
Colored Petri nets [21, 22] extend P/T nets by adding typed data in tokens. Colored Petri nets may have a better expressive power (sometimes leading to undecidability) but the specification is smaller, thus increasing readability.

Among several variants of colored Petri nets, Symmetric Nets (SN) [6] define a simple typing mechanism (discrete types, cartesian product) allowing the exploitation of symmetries in a system. The expressive power of SNs is the same as P/T nets.

Symmetric Nets with Bags (SNB) This is a recent extension of SN [18]. SNBs propose a new mechanism allowing to avoid some interleaving, by enabling multisets (or bags, see definition 3 in section 3.3) of colors in tokens. SNBs allow to express the same symmetries as SNs do, thus enabling similar exploitation of symmetries.

The SNB presented in Fig. 1 models a simple deadlock-free resource manager based on the global allocation of all required resources before entering a critical section [26]. There are two discrete types of color respectively describing the processes ids (Proc) and the critical resources (Res). Then, the type representing the set of bags of resources is defined (BagR), as well as its cartesian product with processes ids (P_BagR).

Initially, marking M_r in R contains the available resources of the system (there can be several copies of some resources, i.e. several tokens of the same value) and M_p



```
Class
  Proc is 1..3;
  Res is 1..6;
Domain
  BagR is Bag(Res);
  P_BagR is <Proc,BagR>;
Var
  p in Proc;
  R, R2 in BagR;
```

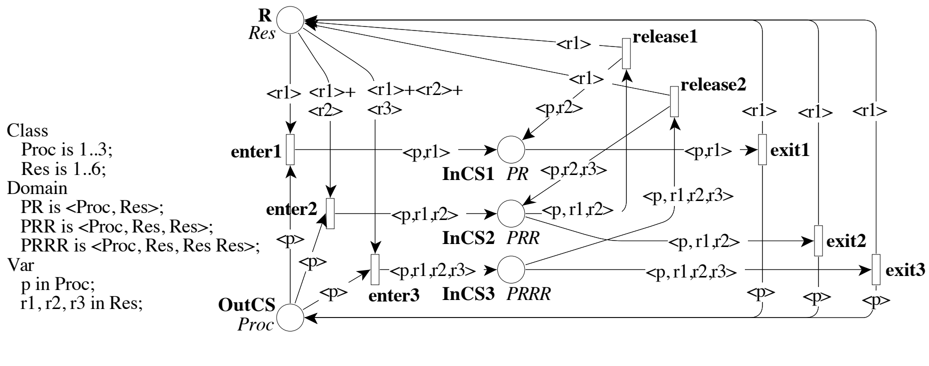
Fig. 1: An example of SNB: the Resource Manager. $M_p = \langle \mathsf{Proc}.all \rangle$ and $M_r = n \times \langle \mathsf{Res}.all \rangle$, with n being a positive value. $\mathsf{Class}.all$ is the function that generates generates one token per possible value in Class.

in outCS represents all the processes (they are represented by their identity) that are initially out of the critical section. Transition enter assigns to a process p a *bag* of resources R. As indicated by the guard of the transition, a process is assigned at least one resource. Place InCS holds the processes using at least one resource (in the critical section). Transition release allows to release resources. However, its guard prevents from releasing all the resources, which is done when firing transition exit.

SNB do not extend the expressive power of SN but lead to a more compact model as illustrated below.

SNB versus SN This system could be modeled using SN too. However, the resulting model is then more complex. Two strategies to unfold a SNB into an equivalent SN are considered.

The first one relies on the *unfolding* of places and transitions in which bags occur. It is illustrated in Fig. 2. Let us detail the process for transition enter when $1 < card(\mathsf{R}) \leq N$. First, the cartesian product P_BagR is replaced by N cartesian products (one per possible cardinality of R in enter bindings). Transition enter and place InCS must also be duplicated N times since tokens and bindings are typed by the new cartesian products types. The main problem of this modeling technique is that an upper bound of the bag cardinality must be known a priori (here, $N = 3$ was chosen). Also, changing the model



```
Class
  Proc is 1..3;
  Res is 1..6;
Domain
  PR is <Proc, Res>;
  PRR is <Proc, Res, Res>;
  PRRR is <Proc, Res, Res Res>;
Var
  p in Proc;
  r1, r2, r3 in Res;
```

Fig. 2: Unfolding the model of Fig. 1 for $1 < card(\mathsf{R}) \leq N, with\, N = 3$.

parameters (i.e. type definition) or guards has an impact on the model's structure, thus leading to uneasy maintenance.

The second strategy is to "pump" resources (generally one by one). For instance the exit transition would be replaced by a sub-model having several places and the transitions ensuring that all resources used by a process are released. The main drawback of this approach is that it changes the semantics of the model by introducing more states and interleaving in the state space. Furthermore, it requires either the use of inhibitor arcs, or to preserve and manage the number of allocated resources to p. It is also hard to scale up, as for the previous strategy.

The SNB model is therefore much more compact and scalable than its SN equivalent. Indeed, modification on the model parameters does not require structural changes.

Benefits of SNBs Bags in SNBs allow to abstract complex constructs such as the aforementioned pumping scheme. The encapsulation of bags in tokens also allows a better structuring of the model. This is illustrated in our example where they encode quite clearly the allocation of resources to a process. Bags in SNB thus offer to the modeler two tools: an abstraction mechanism, along with a structuring mechanism, that may be combined.

Moreover, by avoiding situations that generate interleaving, bags reduce the need for partial order techniques that are difficult to stack on top of decision diagrams and symmetries without paying the price of not using this technique.

This is what we call *"extreme symmetries"*: a way to structure symmetries in the system specification to enable the activation of efficient encoding of the state space generation and analysis (in the decision diagram meaning of it). This structuring information is transparently provided by the modeler instead of being guessed by the model checker.

2.2 Decision Diagrams

Principle Shared Decision Diagrams (DD) [4] are a data structure to compactly represent sets. There are many variants of decision diagrams used for model-checking, but they all rely on the same underlying principles: *i)* nodes of the decision tree are unique in memory thanks to a canonical representation; *ii)* the number of paths through the diagram (states) can be exponential in the representation size (nodes in the DD); *iii)* using caches, most operations manipulating a DD are polynomial in the representation size; *iv)* the effectiveness of the encoding strongly depends on the chosen variable ordering [9].

Set Decision Diagrams (SDD) In this paper we rely on Hierarchical Set Decision Diagrams (SDD, defined in [14]), which extend classical BDD in two respects: *1)* variables are considered to have a set domain instead of a Boolean one; *2)* operations over SDD are encoded using homomorphisms instead of the usual fashion where another decision diagram with two variables per variable of the state signature is used. Definitions are taken almost verbatim from [29].

A SDD is a data structure for representing a set of sequences of assignments of the form $\omega_1 \in s_1; \omega_2 \in s_2; \cdots; \omega_n \in s_n$, also noted $\omega_1 \xrightarrow{s_1} \omega_2 \xrightarrow{s_2} \cdots \omega_n \xrightarrow{s_n} \mathbf{1}$, where ω_i are variables and s_i are sets. These sets can themselves be represented by SDD: in that case, we think of SDD as hierarchical decision diagrams. We assume no implicit variable ordering and the same variable can occur several times in an assignment sequence. We define the terminal **1** to represent the empty assignment sequence, terminating any valid sequence. The terminal **0** represents the empty set of assignment sequences. Let *Var* be a set of variables, and for any ω in *Var*, let $\mathrm{Dom}(\omega)$ be the domain of ω, that may be infinite.

Definition 1 (SDD). *The set* $\mathbb{S}$ *of SDD is defined inductively by* $\delta \in \mathbb{S}$ *if either:*

- $\delta \in \{\mathbf{0}, \mathbf{1}\}$ *or*
- $\delta = \langle \omega, \pi, \alpha \rangle$ *with:*
  - $\omega \in$ *Var,*
  - $\pi = \{s_0; \ldots; s_n\}$ *a finite partition of* $\mathrm{Dom}(\omega)$
  - α *an injective mapping from* π *to* $\mathbb{S}$

By convention, paths terminated by the SDD **0** *are not represented.*

Let us note that SDD or other variants of DD can be used as the domain of variables, thus introducing hierarchy in the data structures.

Example of State Encoding let us illustrate the use of SDD with a simple example: the encoding of two states in the model of Fig. 1:

$$S_0 = \mathsf{InCS}(\emptyset) + \mathsf{OutCS}(\langle 1_p \rangle + \langle 2_p \rangle + \langle 3_p \rangle) + \mathsf{R}(\langle 1_r \rangle + \langle 2_r \rangle + \langle 3_r \rangle + \langle 4_r \rangle + \langle 5_r \rangle + \langle 6_r \rangle)$$
$$S_1 = \mathsf{InCS}(\langle 1_p, \{1_r, 2_r, 3_r, 4_r, 5_r, 6_r\} \rangle) + \mathsf{OutCS}(\langle 2_p \rangle + \langle 3_p \rangle) + \mathsf{R}(\emptyset)$$

S_0 is the initial state where all resources are available and all processes out of the critical section. S_1 is a state where process 1 is in the critical section and uses all resources. Figure 3 shows a possible encoding of these two states. Let us first provide some notation convention in this figure:

- $1_p, 2_p, 3_p$ (respectively $1_r, 2_r, 3_r, 4_r, 5_r, 6_r$) correspond to the values in Proc (respectively Res),
- double lines correspond to the encoding of the marking structure, single lines to a piece of marking and dotted lines to a hierarchical relation,

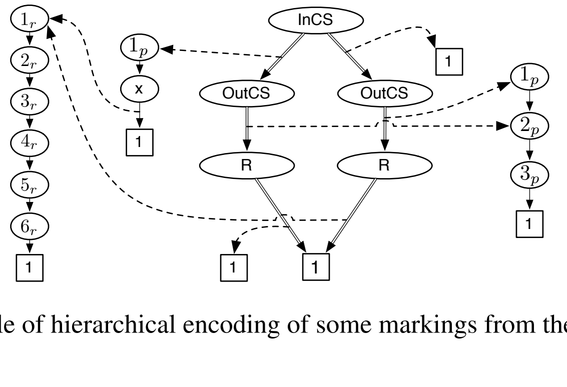


Fig. 3: Example of hierarchical encoding of some markings from the net of Fig. 1

The main part of this SDD has two paths: the left one encodes S_1, the right one encodes S_0. The encoding of S_1 must be read as follows: place InCS holds a composed token represented by another SDD on the left. This SDD refers itself to a second one that represents the bag containing one occurrence of each element in Res. Then, continuing the path, place outCS holds two tokens: 2_p and 3_p. Finally, place R is empty (the underlying SDD is reduced to its terminal). A similar interpretation can be performed for S_0.

Figure 3 illustrates several types of sharing with SDD. First, as for traditional decision diagrams, common nodes are represented only once (let us note that the terminal node is represented several times to make the figure clearer but there is only one occurrence in memory). Second, sub-SDD introduce a new type of sharing. Typically, the marking of R in S_0 and the bag contained in the token of InCS in S_1 are represented once in a sub-SDD. Similarly the rightmost SDD encodes two markings: $\{\langle 1_p\rangle, \langle 2_p\rangle, \langle 3_p\rangle\}$ and $\{\langle 2_p\rangle, \langle 3_p\rangle\}$ that share a common part.

SDD operations SDD support standard set operations: $\cup$, $\cap$, $\setminus$. The semantics of these operations are based on the sets of assignment sequences that the SDD represent.

SDD also offer a concatenation $\delta_1 \cdot \delta_2$ which replaces terminal **1** of δ_1 by δ_2. This corresponds to a cartesian product. Basic and inductive homomorphisms are also introduced to define application-specific operations. A more detailed description of SDD homomorphisms can be found in [13].

A basic homomorphism is a mapping $\Phi : \mathbb{S} \mapsto \mathbb{S}$ satisfying $\Phi(\mathbf{0}) = \mathbf{0}$ and $\forall \delta, \delta' \in \mathbb{S}, \Phi(\delta \cup \delta') = \Phi(\delta) \cup \Phi(\delta')$. Many basic homomorphisms are hard-coded. The sum $+$ operation between two homomorphisms ($\forall \delta \in \mathbb{S}, (\Phi_1 + \Phi_2)(\delta) = \Phi_1(\delta) \cup \Phi_2(\delta)$) and the composition of two homomorphisms $\circ$ ($\Phi_1 \circ \Phi_2(\delta) = \Phi_1(\Phi_2(\delta))$) are themselves homomorphisms.

A homomorphism c is a *selector* iff. $\forall \delta \in \mathbb{S}, c(\delta) \subseteq \delta$. This allows to represent boolean conditions, as c selects states satisfying a given condition; thus the negation of c is $\bar{c}(\delta) = \delta \setminus c(\delta)$. As a shorthand for "if-then-else", we use $\texttt{IfThenElse}(c, h_1, h_2) = h_1 \circ c + h_2 \circ \bar{c}$, where h_1 and h_2 are homomorphisms.

This mechanism is generalized with a variant called "multi-linear" homomorphisms. Such a homomorphism splits a SDD into several parts, for which it applies a specific operation.

Multi-linear homomorphisms are particularly useful when one wants to change the value of a variable x depending on the value of a variable y that has not been read yet (e.g. in tokens containing bags). Since SDD represent a set, several values for y may exist. Multi-linear homomorphisms split the SDD into several subsets $s_1 \ldots s_n$, one for each value of y. Thus, for each i, all the elements of s_i have the same value for y and can therefore be applied the same update of variable x. Thus, the main part (x) remains unchanged and this reduces the number of temporary SDD nodes to be merged later; this reduced the known “peak-effect” of decision diagrams.

The *fixpoint* $h^\star$ of a homomorphism, defined as $h^\star(\delta) = h^k(d)$ where k is the smallest integer such that $h^k(\delta) = h^{k+1}(\delta)$, is also a homomorphism provided k exists.

Besides providing a high level way of specifying a system’s transition relation, homomorphisms can be used to express many model checking algorithms directly. For instance, given a SDD s_0 representing initial states and a homomorphism *succ* representing the transition relation, we can obtain reachable states by the equation $Reach = (succ + Id)^{\star}(s_0)$.

Specifying model checking problems as homomorphisms allows the software library to enable automatic rewritings that yield much better performances, such as the saturation algorithm [19].

Discussion on SDD When hierarchical structuring is possible, SDD allow a better sharing than traditional "flat" decision diagrams. The main reason is that hierarchy introduced flexibility in the encoding, thus reducing the known effect of variable ordering on the performances of this technique. They also enable partial reuse of local encoding patterns (as shown in Fig. 3)

Finally, the homomorphism notion and the associated rewriting techniques allow an intensive use of caches and the activation of efficient resolution algorithms such as automatic saturation [19]. Therefore, SDD are a good candidate to be stacked with the symmetry-based optimizations brought by SNB.

In this specific framework, multi-linear homomorphisms are of particular interest to canonize marking. This is because the structure of the marking may not respect the locality of the operations where decision diagrams are usually very efficient. Such homomorphisms could help to reduce the drawback of this lack of locality.

3 Formal Analysis of Extreme Symmetric Systems

Formal analysis consists in verifying expected properties of a system modeled in an appropriate formalism. We focus here on state-based analysis. Thus, expected properties usually are specified as reachability formulas, deadlock detection, LTL or CTL formulas, etc. over the model state space. The drawback of these approaches is the so-called combinatorial explosion of the number of states that hinders analysis.

3.1 Existing Approaches for the Analysis of Symmetric Systems

Several approaches have been proposed to tackle this combinatorial explosion. We focus on two of them, namely symmetry reduction, and decision diagrams. We then associate these two techniques with bag-based modeling.

Symmetries Concurrent systems often exhibit symmetries: the typical example consists in n identical processes that behave asynchronously. A state in such systems is then characterized by the states of these n processes, up to a renaming of the processes: the processes are all behaviorally equivalent.

Formally, two components are said to be symmetrical if they can be permuted without changing the behavior of the system. In most systems, there are several groups of components with similar behavior: each component can be permutable with any component in the same groups. In SNB, such behavioral groups are defined as *equivalence classes* on C.

Definition 2 (Color Equivalence Classes). *Let us consider a discrete data type (i.e. a color in a SNB) C. Two colors in C are symmetrical if the behavior of the system is not affected when they are swapped.*

The "is symmetrical to" is an equivalence relation over C. Its equivalence classes $C_1, \ldots, C_n$ partition C in the following way:

$$C = \dot{\bigcup_{i=1..N}} C_i$$

The symmetries on the colors naturally extend to symmetries of system states. The relation "is symmetrical to" is also an equivalence relation over the set of system states, whose equivalence classes are also called *symbolic states*. The *quotient* state space is defined as the quotient of the reachability graph by this equivalence relation [6]. In favorable cases, the quotient state space is exponentially smaller than the state space.

Decision Diagrams We mentioned decision diagrams in section 2.2 as a compact structure. They were first used in model checking [5] to successfully handle large state spaces. Several variants have been used since then for the efficient representation of large state spaces.

Symmetry reduction and decision diagrams can be used together. Although previous works have shown their efficiency, decision diagrams still require specific algorithms because their optimal use is not straightforward. [8] has shown that the traditional approach for the representation of complex operations on decision diagrams fails at providing an efficient solution to the computation of a quotient state space. However, the notion of homomorphism presented in section 2.2 appears to be a promising way to overcome this obstacle, as several investigations show [13, 28, 10]. Nevertheless, the design of algorithms for symmetry reduction using decision diagrams is still a challenging problem.

3.2 Using Bags Information to Optimize State Space Generation

Bags provide an abstraction mechanism, especially for subsystems that generate interleaving. They thus decrease the need for partial order techniques that are incompatible with decision diagrams. This can be observed on transition enter in Fig. 1 and its unfolding presented in Fig 2. In the SNB, there is only one possible transition to be fired while, in the second model, several exist. So, if the number of symbolic states in the quotient state space remains the same, the number of firings is dramatically reduced in the case of SNB [10]. Then, we avoid several type of situations where partial order techniques could be operated.

At this stage, several techniques can be activated, based on the information carried out by bags, as provided in SNB. We list these techniques before showing how they are applied to implement the transition relation.

Technique 1: dedicated representation of guards together with the definition of Symmetric nets, a dedicated representation for guards was introduced [6]. The objective was to preserve the information about equivalence classes in color types and thus, to enable the implementation of the so-called *symbolic firing* of transitions.

For instance, an expression like $v < V$ (where v represents a variable and V a constant in the color class C) implicitly defines two color equivalence classes. The first one C_1 contains all the values of C that are smaller than V, and C_2 contains the other values of C. Once the color equivalence classes have been computed, such an expression can be rewritten $v \in C_1$.

This rewriting can be generalized to any relation between a variable and a constant provided that equivalence classes are computed. Guards are then expressed using a disjunction of membership test to selected equivalence classes. For instance, inequalities between two variables leads to a partition of C into $N = |C|$ singleton subclasses (i.e. such an expression breaks all the symmetries) while $=$ and $\neq$ preserve the equivalence relation.

Such a representation is painful to explicit by the modeler but it can be automatically computed on Symmetric Nets [27]. Extensions of this algorithm can be provided, considering extra operators to manage bags cardinalities.

Technique 2: deducing a hierarchical representation from the bag structure This idea has been introduced in a first reachability analysis tool for SNB [10] and its principle is roughly presented in the example for marking encoding in Fig. 3. It can also be extended to the manipulation of bags.

Technique 3: recursive unfolding this technique is efficient when a system (or parts of a system) can be defined recursively [19]. Let us sketch its principle on the dining philosophers problem. Instead of considering symmetries "horizontally" (e.g. all philosophers share the same behavior), the idea is to consider them "vertically". Then, T_n the table of n can be decomposed in a recursive way:

$$
\begin{array}{llll}
T_n = 2\times & T_{\frac{n}{2}} & + \, interactions & = 2 \times \frac{1}{2} tables \\
\quad = 2\times & \overbrace{2 \times T_{\frac{n}{4}} + interactions} & + \, interactions & = 4 \times \frac{1}{4} tables \\
\quad = 2\times\ 2\times & \overbrace{2 \times T_{\frac{n}{8}} + interactions} + interactions & + \, interactions & = 8 \times \frac{1}{8} tables \\
\quad \vdots & & & \quad \vdots
\end{array}
$$

until T_2 (the "elementary" table with 2 philosophers) is reached. This technique, when it applies for regular systems, proved to be extremely efficient thanks to a recursive hierarchical encoding (which is possible with SDD). We show, later in this paper, that bags can be encoded recursively in a similar way to the example provided here.

Technique 4: anonymization this technique was introduced to deal with the computation of a hierarchical order to encode a state space with SDD [20]. The principle is to reuse similar patterns with a new interpretation. For instance, if we consider two sequences of affectations ($x = 4 \rightarrow y = 2 \rightarrow \mathbf{1}$) and ($t = 4 \rightarrow u = 2 \rightarrow \mathbf{1}$), one can observe that x, y for the first one and t, u for the second one can be considered as "contextual information", thus reducing those two patterns to a single one.

This technique, associated with a hierarchical representation, can dramatically reduce the number of different SDD patterns, and thus, lead to a more compact storage of the state space.

These four techniques are exploited to elaborate an efficient representation in memory of the state space, as well as performant computation of the quotient state space.

```
Class                                Var
    Res is Res1 = [1..3] ∪           r1 in Res1;
           Res2 = [4..6];            r2 in Res2;
```

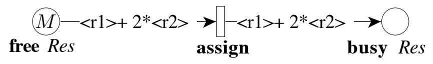


Fig. 4: illustration of the symbolic firing, $M = \langle \mathsf{Res}_1.all \rangle + 4 \times \langle \mathsf{Res}_2.all \rangle$

3.3 Computing the Transition Relation in SNB

We detail here how efficient algorithms dedicated to SNB can be deduced from the techniques identified in the previous section.

The Transition Relation Models usually are specified in terms of a transition system, with initial states and a generic transition relation. The verification of a property then consists in an exploration of the state space. Algorithm 1 is typical of the state space generation for a model given as a set of initial states and the transition relation. Once the state space has been generated, several properties (reachability, deadlock detection, LTL, CTL formulae . . .) can be checked against it.

Depending on the type of property to be verified, this algorithm can be tweaked for improved performance. For instance, reachability and LTL formulae can be checked on-the-fly (the algorithm returns as soon as the property is verified or a counter-example is found). We do not focus on such optimizations but on the “core generation” of the state space instead.

The Transition Relation in SN The transition relation is quite straightforward in this case. [6] introduces a framework for an efficient use of symmetries in SN. Formally, colors are separated into *color equivalence classes* that express the possible symmetries, as explained earlier.

Let us illustrate this framework with the SN of Fig. 4 that illustrates the affectation of resources in a system. There are two types of resources: Res_1 (one is hold at a time) and Res_2 (two copies are hold simultaneously).

Since all resources in Res_1 (resp. Res_2) are symmetrical, there are several symmetrical bindings for the variables r1 and r2 that lead to symmetrical markings.

The color equivalence classes are partitioned into dynamic subclasses, depending on the marking. For instance, Res_1 could be split into Z_1, the free resources (tokens in place free), and Z_2 the busy ones (tokens in place busy). A symbolic marking is thus expressed in terms of such dynamic subclasses. Similarly, binding the variables of a transition to dynamic subclasses rather than explicit values allows to capture several symmetrical bindings at once. For instance, considering that $\mathsf{Res}_1 = Z_1 \cup Z_2$ and $\mathsf{Res}_2 = Z_3 \cup Z_4$, r1

```
Require: a model $M$ given as a set of initial states $S_0$ and the transition relation $Next$
  $S \leftarrow S_0$
  repeat
    $S \leftarrow S \cup Next(S)$
  until a fixpoint is reached
  return $S$
Ensure: $S$ is the state space of $M$
```

Algorithm 1: The state space generation algorithm

and r2 have two possible bindings each, leading to four symbolic bindings. This number is to be compared to the nine possible explicit bindings (9 in the example).

Variables can only be bound to dynamic subclasses Z such that $card(Z) = 1$. This may lead to a preprocessing of the symbolic marking, called *splitting* in order to obtain such dynamic subclasses. Similarly, once the symbolic firing occurred, a postprocessing operation called *grouping* recomputes the dynamic subclasses.

Let us note that, in this case, both the description of markings and the description of bindings are represented using equivalence classes as basic representation of values. The larger the color equivalence classes, the fewer values are evaluated during the firing of transitions, and the fewer states in the quotient state space.

The Transition Relation in SNB [18] extends the notion of symbolic markings and symbolic firing to SNB. The sole difference between the SN and SNB transition relations is the binding of bag variables, and the efficient computation of all the possible bindings. In SNB, classical variables are instantiated as in SN, and bag variables are instantiated with bags over dynamic subclasses. The same processes of splitting and grouping the dynamic subclasses, adapted for bags, occur.

Finding all the bindings of a transition with bag variables is always reducible to the enumeration of bags over a finite domain (the dynamic subclasses) with bounded cardinality. A naive enumeration would however suffer from the interleaving that was supposed to be avoided. We propose an appoach for the efficient computations of such bindings.

A bag (or multiset) is a set where there can be several instances of some elements.

Definition 3. *Bag Let $C = \{c_1 \dots c_n\}$ be a finite set. A bag b over C is a formal sum $b = \Sigma_{i=1}^{n} a_i c_i$ where $a_i \in \mathbb{N}$ is the multiplicity of the element c_i.*
The cardinal of b is $|b| = \Sigma_{i=1}^{n} a_i$, and the support of b is the set of elements with non-zero multiplicity: $Supp(b) = \{c_i | a_i > 0\}$.
The multiplicity of c_i may also be denoted by $b(c_i)$.

$Bag(C)$ denotes the set of multisets over C. $Bag_n(C)$ denotes the set of multisets over C having a cardinality of n. The union, intersection and difference on sets extend naturally to multisets:

- $b_1 \cup b_2 = \Sigma_{i=1}^{n} (b_1(c_i) + b_2(c_i)) c_i$
- $b_1 \cap b_2 = \Sigma_{i=1}^{n} \min(b_1(c_i), b_2(c_i)) c_i$
- $b_1 - b_2 = \Sigma_{i=1}^{n} \max(b_1(c_i) - b_2(c_i), 0) c_i$
- $b_1 \subset b_2$ if and only if $b_1(c) \leq b_2(c)$ for all $c \in C$

Note that when all the multiplicity are 0 or 1, then the bag is a set, and that all definitions above fall back to classical set definitions. Further optimizations can be obtained when the encountered bags are actually restricted to be sets, but are not detailed here as they mainly rely on classical computations over sets.

Application of technique 1 is then trivial (canonization of guards). However, it required some extensions for bags. In order to compute the set of bags having cardinal n over a set C, one may use recursive definitions concerning either the cardinality of, or the support of the bags, thus applying technique 2.

Property 1 *Recursion over the cardinal*
$Bag_{m+n}(C) = Bag_m(C) \uplus Bag_n(C) = \{b_1 \cup b_2 | b_1 \in Bag_m(C), b_2 \in Bag_n(C)\}$

Property 2 *Recursion over the support*
$Bag_n(C_1 \cup \ldots \cup C_k) = (\bigcup_{i=1}^{k} Bag_n(C_i)) \cup Bag_n^*(C_1, \ldots, C_k)$ *where* Bag_n^* *represents the bag of cardinality n over mixed supports (i.e. involving several* C_i*).*

Properties 1 and 2 allow to represent a set of bags of a given cardinality in terms of sets of bags of smaller cardinality or smaller support.

When colors classes are split into equivalence classes, the use of property 2 reduces the problem in terms of generating sets of bags over color classes. Then, we use the recursion over bags cardinality. All together, these properties allow for an efficient divide-and-conquer generation strategy.

For instance, by carefully choosing the parameters n and m in property 1, one can represent $Bag_n(C)$ in $O(log(n))$ SDD nodes. This leads to the type of recursive encoding over the structure of the bags as done by technique 3 (its principle is sketched in Fig. 5). Thus, as soon as we detect an upper bound for the cardinality of a bag over C, its representation is easily elaborated based on this scheme.

Finally, technique 4 (Anonymization) can be applied to increase the sharing of common patterns. This can be applied to the markings structure (see the example presented in Fig. 3) or to the net structure (as illustrated in [20]).

It is also applicable to the recursive bag representation as illustrated in Fig. 6. The idea is to have a representation of the bag cardinalities for a generic class $\mathfrak{C}$ that can be mapped to any C or $C_i \subset C$. Then, values in $\mathfrak{C}$ are only referenced by their position and the maximum cardinality of $\mathfrak{C}$ is the one of the largest type (or equivalence class). When a reference is made to $\mathfrak{C}$, the associated context is expressed using a reference to the effective class $\mathfrak{C}$ is mapped to. This relation is done at runtime when effective markings or transition bindings in the SNB are computed. By applying this principle to the recursion over bag cardinalities in Fig. 5 we get the reduced representation of Fig. 6. This technique also applies to the recursion over bag support.

All together, these techniques allow to contain the combinatorial explosion of the state space in terms of memory, as well as CPU consumption, since less calculus are needed.

Overview of the state space representation The representation of a SNB state space is divided in three parts (see Fig. 7):

1. the structure of the system states,
2. a sets of bags "heap",

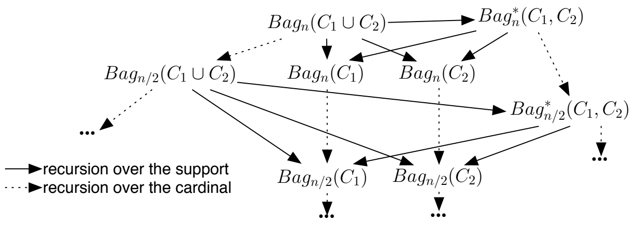


Fig. 5: Recursion over the definition of bags

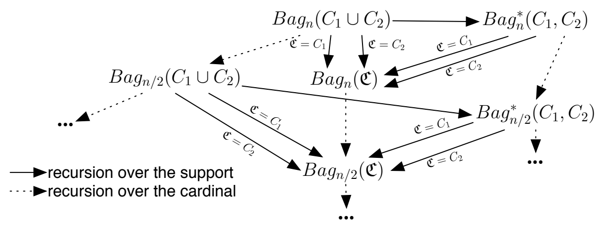


Fig. 6: Optimized representation of bags (anonymization over bag cardinalities)

3. representation of bag values.

Structure of the system states encodes the structure of states as they are expressed in the SNB. A naive way to encode this is to list the net places in a given order. However, the hierarchy supported by SDD allows a better reuse of patterns describing parts of the SNB structure.

The representation of tokens in places refers to a "heap" storing sets of bags as a unique and shared data structure.

Sets of bags "heap" encodes the bags of color that can be referenced to represent tokens in places. Once again, the representation can be hierarchical, especially when bags are hold in colored tokens.

The representation of bags in tokens refers to common representation of bag.

Representation of bag values recursively encodes the bags referenced in the state space.

The four techniques mentioned in section 3.2 can be activated there. In particular, anonymization allows a better reuse of representation patterns in any part of the state space.

Implementation of the transition relation As explained earlier, one of the main characteristics of SDDs is the ability for developers to use dedicated operations (homomorphisms). To take full advantage from SDDs, the transition relation is encoded with homomorphisms. There are four steps in the transition relation [18]:

1. splitting dynamic subclasses,
2. binding the variables of each transition to splitted dynamic subclasses,
3. firing the transition,
4. canonizing the symbolic markings to group and rename dynamic subclasses.

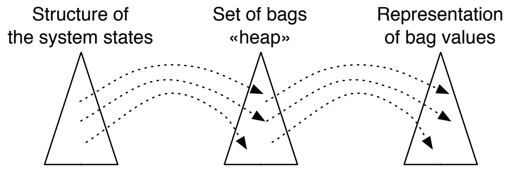


Fig. 7: Structure of the state space representation

```
Class
  People is 1..P;
  Gift is 1..G;
Domain
  BagPeople is bag (People);
  BagGift is bag(Gift);
  PeopleBagGift is <People, BagGift>;
```

```
Var
  p in People;
  BP in BagPeople;
  BG in BagGift;
```

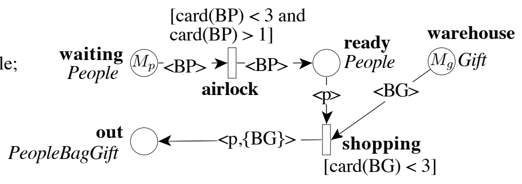


Fig. 8: The SaleStore example modeled with a SNB. $M_p = \langle \mathsf{People}.all \rangle$ and $M_g = \langle \mathsf{Gifts}.all \rangle$, $\mathsf{P} = \mathsf{G} = n$ is the scaling parameter of this example.

Each step is encoded as a homomorphism, and the transition relation is the composition of these four homomorphisms. Each step is theoretically independent, but it is of interest to propagate some information from one step to the next one for optimization purposes. In our implementation for instance, the two first steps are almost merged, in order to compute as few bindings as possible.

Building these operations on top of SDDs allows to profit from the shared structure. For instance, splitting the dynamic subclasses in the marking of a place is done only once for all the markings that share it. This significantly optimizes each step, especially the costly canonization.

The presented data structures are generated on the fly by the homomorphisms when needed. Thus, the SDD representation of the system acts like a cache itself.

4 Assessment

In this section, we take several examples for which the use of SNB is of interest for modeling purpose. We then provide some performances compared to the reference tool on Symmetric Nets: GreatSPN [17].

4.1 The Examples

We selected three examples that illustrate the interest of bag-based modeling as well as the interest of bags in the optimization of state space based analysis: the deadlock-free resource manager, the salestore, and the distributor. The two first models are “toy examples” emphasizing the use of bags in tokens. The last model also benefits from the use of bags but it was designed from a case study found in the litterature.

The resource manager model This is the model presented in Fig. 1 (see section 2.1). However, for the need of performances analysis, we constrained it to let processes have a maximum of three resources in their critical section. To do so, we changed the guard of transition enter into `[card(R) > 0 and card(R) < 4]`. The scaling parameter of this example is n in the initial marking M_r.

This model shows how bags can help to preserve symmetries. In order to discriminate between the transition exit (where a process and all its allocated resources are released) and the transition release (where some resources are released, but the process remains in he critical section), the SN unfolding breaks a few symmetries. This explains the difference of the number of symbolic states found by Crocodile and by GreatSPN in Table 1 (section 4.2). This is also an example of the interest of the bags in such models.

The salestore model This model was introduced in [10] and is shown in Fig. 8. People enter the sale store through an airlock (transition airlock) with a capacity of two (of course, a single person may enter too). Then, people may buy items (at most two but possibly zero if none fits their need) and leave with the acquired items. Let us note that this example has two scalable parameters: P, the number of involved people in the system and G, the number of possible gifts in the warehouse. In our example, the model has been explicitly constrained: the airlock has a maximum capacity of 2 people and each customer cannot leave with more than 2 gifts.

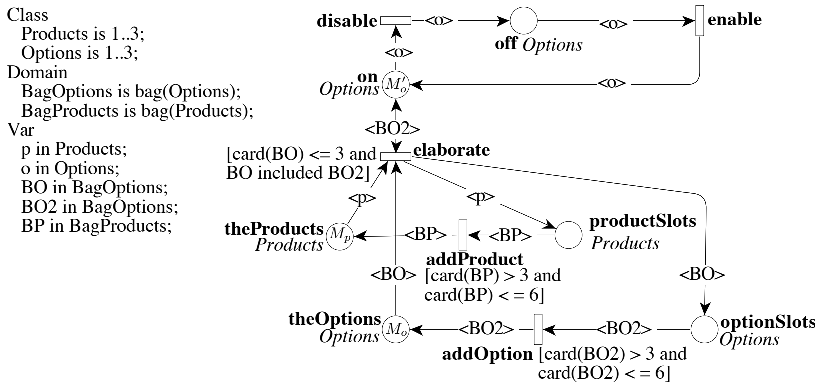
The distributor model This model of a coffee dispenser machine and optional features (e.g. milk, sugar, etc.) was introduced in [25] as a Feature Petri Net. We present in Fig.9 a SNB version of this model. The machine dispenses products (place theProducts) like coffee or tea. When brewing (transition elaborate) one of these products, it may add options (place theOptions) like milk or sugar, on demand.

The machine is refilled with products and options according to the conditions on transitions addProduct and addOption respectively. Options may be enabled (transition enable) or disabled individually (transition disable) and dynamically.

The original work in [25] presenting Feature Petri nets is intended to model the behavior of Software Product Lines (SPL). That approach was proposed as a means to *"ensure that all products*[3] *meet their specifications without having to check each product individually"*. A modular modeling framework is then proposed to incrementally build larger feature nets from smaller ones. The Feature nets are based on P/T nets. New features are thus added as new net fragments.

Our model is an adaptation of this example, showing SNB suitability to model a SPL: the specification is much more compact since, instead of adding pieces of Nets,

[3] i.e. an assembly of selected features to build a specific model (e.g. {Coffee, Tea, Milk}).



```
Class
  Products is 1..3;
  Options is 1..3;
Domain
  BagOptions is bag(Options);
  BagProducts is bag(Products);
Var
  p in Products;
  o in Options;
  BO in BagOptions;
  BO2 in BagOptions;
  BP in BagProducts;
```

Fig. 9: The distributor example modeled with a SNB. $M_p = x \times \langle \mathsf{Products}.all \rangle$, $M_o = x \times \langle \mathsf{Options}.all \rangle$ and $M_o' = \langle \mathsf{Options}.all \rangle$, x is the scaling parameter of this example.

```
Class
  People is 1..P;
  Gift is 1..G;
Domain
  PeopleGift is <People, Gift>;
  PeopleGiftGift is <People, Gift, Gift>;
Var
  p1,p2 in People;
  g1,g2 in Gift;
```

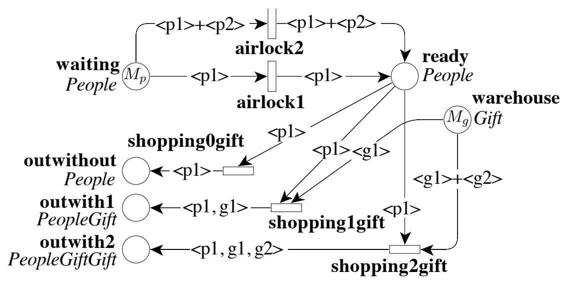


Fig. 10: The unfolded equivalent SN from the model of Fig. 8

only the definition of color types is changed. Moreover, thanks to the use of bags, scalability over the sets of features and their multiplicity in the machine configurations is guaranteed.

4.2 Performances

A first version of Crocodile was implemented and compared to GreatSPN [17] in [10]. GreatSPN is a well-known model checker for various classes of Petri Nets. Among its various capabilities, we only use its quotient state space generation features. It works with SN only, and does not use decision diagrams [4].

This first study revealed that the combination of symmetries and decision diagrams is of interest: our tool was more performant than GreatSPN. However, at this stage, the management of bags was not optimized at all. This impeded performances when bags were used in tokens, while sets in tokens were appropriately handled.

For the *resource manager* and the *salestore*, we compare Crocodile2 to GreatSPN once again. We compute the quotient state space of the SNB and the unfolded SN with Crocodile2, and the quotient state space of the unfolded SN with GreatSPN. The unfolded SN of the *resource manager* is presented in Fig. 2 and the one of the *salestore* is presented in Fig. 10.

Since the *distributor* is too complex to unfold, we only compute the quotient graph with Crocodile2 from the SNB version. The idea is to show its capability to scale-up well. All experiments were run on a Xeon 64 bits at 2.6 GHz processor.

Table 1 reports these experiments. It displays the following information: value of the scaling parameter, number of explicit states in the system, number of symbolic states found by Crocodile2 and GreatSPN (they should be the same), time and memory to compute the quotient state space in the various versions we processed. Gray cells show that no experiment has been done for this configuration. EDNF means "execution did not finished" (more than 4 hours of processing).

[4] A prototype version of GreatSPN uses several variants of decision diagrams [1]: multi-way DD (MDD), multi-terminal MDD (MTMDD), and edge-valued MDD (EV+ MDD). None of these are hierarchical and they encode Stochastic P/T nets so far. Their results also show significant gain from the original version. However, we could not use this version of GreatSPN against our prototype.

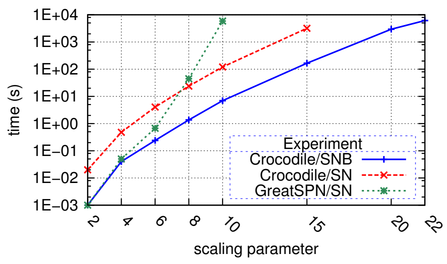
scaling parameter	# Explicit States	# Symbolic States Crocodile2 on SNB	# Symbolic States GreatSPN on SN	Time (s) Crocodile2 SNB	Time (s) Crocodile2 SN	Time (s) GreatSPN SN	Memory (KB) Crocodile2 SNB	Memory (KB) Crocodile2 SN	Memory (KB) GreatSPN SN
Resource manager									
02	1.8×10^{01}	4	8	ε	0.02	ε	80	80	80
04	2.2×10^{03}	12	38	0.04	0.48	0.05	80	3 112	80
06	7.2×10^{05}	27	116	0.24	4	0.67	4 008	4 716	1 156
08	4.5×10^{08}	53	289	1	23	44	8 188	8 816	9 076
10	4.8×10^{11}	94	621	6	120	5 892	19 120	16 820	851 596
15	8.2×10^{19}	295	**EDNF**	165	3 185	**EDNF**	245 212	101 860	**EDNF**
20	7.7×10^{28}	717		2 941	**EDNF**		3 059 216	**EDNF**	
22	4.3×10^{32}	973		6 131			4 192 604		
23	3.5×10^{34}	**EDNF**		**EDNF**			**EDNF**		
Salestore									
02	2.9×10^{01}	13	13	0.01	0.01	ε	80	80	80
04	4.2×10^{03}	60	60	0.13	0.14	0.04	2 864	3 128	80
06	1.5×10^{06}	180	180	0.94	0.97	0.63	5 720	6 028	1132
08	9.4×10^{08}	425	425	5	4	40	12 292	14 108	9 056
10	9.7×10^{11}	861	861	23	18	4 798	32 056	34 352	851 452
15	1.6×10^{20}	3 336	**EDNF**	500	242	**EDNF**	222 764	221 192	**EDNF**
20	1.3×10^{29}	9 196		5 010	2 157		1 076 816	910 444	
22	7.0×10^{32}	12 948		10 003	4 596		1 811 008	1 398 944	
23	5.6×10^{34}	**EDNF**		**EDNF**	6 339		**EDNF**	1 730 348	
24	4.5×10^{36}				9 357			2 419 028	
25	3.8×10^{38}				**EDNF**			**EDNF**	
Distributor									
01	2.2×10^{04}	64		0.11			2 740		
02	1.4×10^{06}	560		0.84			3 936		
03	1.6×10^{07}	2 400		4			7 064		
04	8.9×10^{07}	7 700		14			13 760		
05	3.4×10^{08}	20 384		43			26 768		
06	1.0×10^{09}	47 040		108			46 224		
07	2.5×10^{09}	97 920		236			71 436		
08	5.7×10^{09}	188 100		472			111 144		
09	1.1×10^{10}	338 800		918			171 040		
10	2.2×10^{10}	578 864		1 596			239 684		
11	3.8×10^{10}	946 400		2 894			420 620		
12	6.5×10^{10}	1 490 580		4 553			699 408		
13	1.0×10^{11}	2 273 600		7 763			1 070 832		
14	1.6×10^{11}	**EDNF**		**EDNF**			**EDNF**		

Table 1: Performances of state space generation using Crocodile2 and GreatSPN.

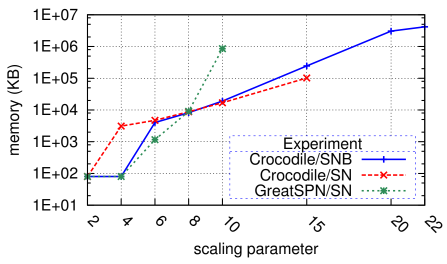
Figure 11 provides charts showing the evolution of the required CPU and memory for processing the quotient state space.

A first observation is that both tools compute the same number of symbolic states for the Salestore model, a proof that our algorithm reaches the minimum quotient state space with SNB. However, this is not the case for the resource manager and this is due to the unfolding that breaks some hierarchical symmetries as we already mentioned in section 4.1.

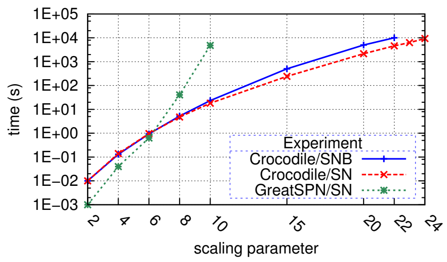
We also observe that, for small values of the scaling parameter, greatSPN outperforms Crocodile2 both in time and memory consumption. This is typical of the involved techniques (both decision diagram-based and symmetries-based) that have an "initial cost" due to the management of data structures, that is not compensated in the case of small state spaces. Then, curves cross when the gain in memory and CPU compensates this overhead.



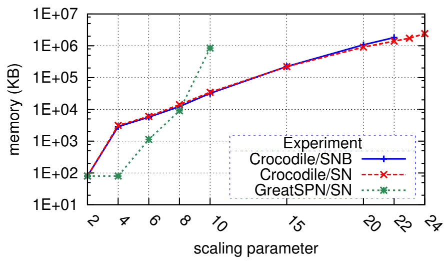
(a) Resource manager, time measures



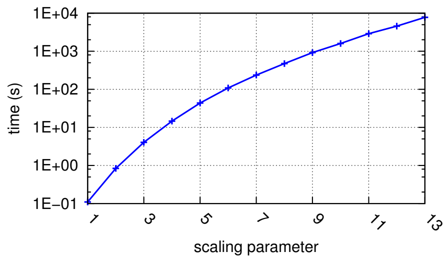
(b) Resource manager, memory measures



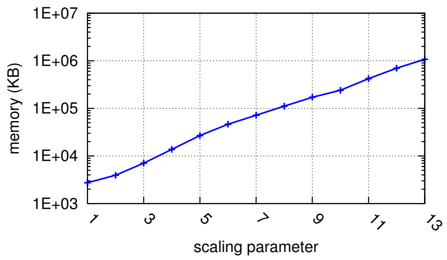
(c) Salestore, time measures



(d) Salestore, memory measures



(e) Distributor, time measures



(f) Distributor, memory measures

Fig. 11: Time and Memory required to generate the quotient state space for the examples

For the two models we also compare with GreatSPN, we reach a stage where greatSPN exceeds the maximum computation time. As observed in [10], the combinatorial explosion of firings is greater for GreatSPN since it canonizes more states than Crocodile2. When both are working on SN, Crocodile2 benefits from the fact that decision diagrams allow to fire and canonize a set of states at the same time. SNB bring a strong reduction of the interleaving that should increase efficiency. In fact the smaller number of symbolic states for the SNB version of the resource manager leads to an increased efficiency of the state space generation in time (see Fig. 11a).

For those two models, we observe no increase of performances between the SNB version and the unfolded SN one (for Crocodile2). This seems mostly due to the fact that multi-linear homomorphisms have been recently implemented in libDDD and probably require some optimization when associated with hierarchical representations. We guess (this must be investigated) that there are some side effects on the cache management that impede CPU gains and memory consumption.

The distributor model is difficult to model easily with SN while its structure would depend on the numbers of products and options which is the scaling parameter (this SNB structure remains constant). However, our tool is able to scale up quite well with respect to the number of generated states.

4.3 Discussion

Evaluation of Reachability Properties So far, our prototype only provides analysis of reachability properties. Such properties are constraints that can be checked during state space generation. This does not bring extra complexity (just a constant due to the property evaluation). Evaluation of a reachability property is done using the following schema:

- translation of the property into constraint c on the symbolic markings (expressed as a SDD),
- for each new symbolic state s, compare the canonical representation of s with c (since both are SDDs, this is a fast operation).

So far, once a state verifying the property is found, the tool must reexecute the state space generation algorithm to store the list of symbolic firings leading to the identified state. Thus, verification of a reachability property may lead to building twice the state space in the worst case. This complexity is compensated by the gain in the state space generation.

Towards evaluation of CTL formulas CTL formulas can be evaluated on symmetric systems provided that, either it respects the system symmetries, or the equivalence relation is computed including constraints of both the model and the property (this may degrade the model symmetries).

Crocodile2 is implemented on top of `libITS` [24], that provides access to SDD via high-level structuring mechanisms (synchronizations and hierarchy). This library supports the evaluation of CTL formulas when atomic propositions they refer to are expressed in a symbolic way.

CTL evaluation heavily relies on the transition relation of the system. This is why we focus on the efficiency of its implementation since it is a key issue to provide efficient CTL analysis.

Usability of Bags in Tokens One could have some skepticism about the usability of SNB. In fact, they are good to capture some dynamic aspects that are commonly found in distributed systems when a variable number of resources is handled by an actor of the system. This is typically the case for resources in the resource manager model (Fig. 1). As shown in section 2.1, modeling of such parts with SN requires to manually bound the number of handled resources. On top of the fact that symmetries are more difficult to capture, this makes the model more complex, and each state to be handled more difficult to encode in memory.

A domain that is very suitable for SNB-based modeling is games where players carry out a variable number of objects or features. The resulting model is much simpler and its analysis benefits from the use of Bags.

Moreover, the handling of SN being included in the handling of SNB, Crocodile2 remains a good tool to perform analysis on such models.

5 Conclusion

This paper presents a method that links a modeling concept recently introduced in Petri nets, the use of bags in tokens, to some efficient state space generation techniques. This modeling concept helps the modeler increase the structuring of symmetries in a specification in a relatively "natural" manner. This bag concept (as introduced in SNB) is of particular interest when associating a variable number of items to an entity (e.g. the critical resources in the resource manager model – see Fig. 1). Such a modeling issue often occurs when modeling distributed systems. This structuring information is reused in the back-end of a model checker tool to tackle the combinatorial explosion.

One main interest of the proposed modeling concept is to reduce the need for partial order techniques. Another originality is to increase the efficiency of the combined use of symmetries-based techniques, together with hierarchical decision diagrams. This association of techniques is of interest when performing state space-based analysis of complex and distributed systems.

We have implemented the presented strategies in a tool, Crocodile2, which outperforms the previous version thanks to the intensive use of efficient data representation techniques and operations. This tool is to be integrated in the *CosyVerif* verification environment [12].

Early assessment of this method by means of SNB models shows increased performance of reachability analysis versus a reference tool like GreatSPN. These promising results strengthen the idea that, in order to tackle complex distributed systems analysis, combined techniques must be activated together with enabling model-level optimized constructs. However, this requires some structural analysis capabilities such as the ones provided by Petri nets.

So far, we have experimented this association on Symmetric Nets with Bags (SNB). A further objective is to generalize this concept in order to apply it to other types of notations dedicated to classes of systems exhibiting symmetries such as peer-to-peer applications or Software Product Lines. A first study in that direction [11] showed interesting results.

Another objective is the optimization of the multi-linear homomorphisms that aim at tackling the cost of non-locality when using decision diagrams. Such an improvement would benefit to all application based on decision diagrams.

References

1. Babar, J., Beccuti, M., Donatelli, S., Miner, A.S.: Greatspn enhanced with decision diagram data structures. In: 31st International Conference on Petri Nets and Other Models of Concurrency (ICATPN 2010). LNCS, vol. 6128, pp. 308–317. Springer (2010)

2. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: 21^{th} International Conference on Computer Aided Verification (CAV). LNCS, vol. 5643, pp. 64–78. Springer (2009)
3. Bosnacki, D., Holzmann, G.J.: Improving spin's partial-order reduction for breadth-first search. In: Model Checking Software, 12th International SPIN Workshop. LNCS, vol. 3639, pp. 91–105. Springer (2005)
4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35(8), 677–691 (Aug 1986)
5. Burch, J.R., Clarke, E.M., Mcmillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} States and beyond. Information and computation 98(2), 142–170 (1992)
6. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: Stochastic well-formed coloured nets for symmetric modelling applications. IEEE Transactions on Computers 42(11), 1343–1360 (Nov 1993)
7. Clarke, E.M.: The birth of model checking. In: 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer (2008)
8. Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9(1), 77–104 (1996)
9. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
10. Colange, M., Baarir, S., Kordon, F., Thierry-Mieg, Y.: Crocodile: a Symbolic/Symbolic tool for the analysis of Symmetric Nets with Bag. In: 32nd International Conference on Petri Nets and Other Models of Concurrency (ICATPN 2011). LNCS, vol. 6709, pp. 338–347. Springer, Newcastle, UK (June 2011)
11. Colange, M., Kordon, F., Thierry-Mieg, Y., Baarir, S.: State Space Analysis using Symmetries on Decision Diagrams. In: 12^{th} International Conference on Application of Concurrency to System Design (ACSD'2012). pp. 164–172. IEEE Computer Society, Hamburg, Germany (June 2012)
12. Cosyverif: a verification environment: `http://www.cosyverif.org` (2012)
13. Couvreur, J., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.: Data decision diagrams for Petri net analysis. Application and Theory of Petri Nets 2002 pp. 129–158 (2002)
14. Couvreur, J., Thierry-Mieg, Y.: Hierarchical decision diagrams to exploit model structure. Formal Techniques for Networked and Distributed Systems-FORTE 2005 pp. 443–457 (2005)
15. Girault, C., Valk, R.: Petri Nets for Systems Engineering. Springer Verlag - ISBN: 3-540-41217-4 (2003)
16. Godefroid, P., Wolper, P.: A partial approach to model checking. In: Logic in Computer Science, 1991. LICS '91., Proceedings of Sixth Annual IEEE Symposium on. pp. 406 –415 (july 1991)
17. GreatSPN: Petri nets suite: `http://www.di.unito.it/~greatspn` (2012)
18. Haddad, S., Kordon, F., Petrucci, L., Pradat-Peyre, J., Treves, L.: Efficient state-based analysis by introducing bags in petri nets color domains. In: American Control Conference, 2009. ACC'09. pp. 5018–5025. IEEE (2009)
19. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Building efficient model checkers using hierarchical set decision diagrams and automatic saturation. Fundamenta Informaticae 94(3-4), 413–437 (September 2009)
20. Hong, S., Kordon, F., Paviot-Adet, E., Evangelista, S.: Computing a Hierarchical Static order for Decision Diagram-Based Representation from P/T Nets. Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) V, 121–140 (January 2012)
21. Jensen, K.: Coloured Petri nets and the invariant-method. Theor. Comput. Sci. 14, 317–336 (1981)

22. Jensen, K., Kristensen, L.: Coloured Petri Nets : Modelling and Validation of Concurrent Systems. Springer Verlag - ISBN: ISBN 978-3-642-00283-0 (2009)
23. Junttila, T.: On the Symmetry Reduction Method for Petri Nets and similar formalisms. Ph.D. thesis, Helsinki University of Technology, Espoo, Finland (2003)
24. libits: `http://move.lip6.fr/software/DDD` (2012)
25. Muschevici, R., Proença, J., Clarke, D.: Modular Modelling of Software Product Lines with Feature Nets. In: 9th International Conference on Software Engineering and Formal Methods (SEFM). LNCS, vol. 7041, pp. 318–333. Springer (2011)
26. Tanenbaum, A.: Operating Systems: Design and Implementation. Prentice Hall (1987)
27. Thierry-Mieg, Y., Dutheillet, C., Mounier, I.: Automatic symmetry detection in well-formed nets. In: Proc. of ICATPN 2003. LNCS, vol. 2679, pp. 82–101. Springer Verlag (June 2003)
28. Thierry-Mieg, Y., Ilié, J., Poitrenaud, D.: A symbolic symbolic state space representation. Formal Techniques for Networked and Distributed Systems–FORTE 2004 pp. 276–291 (2004)
29. Thierry-Mieg, Y., Poitrenaud, D., Hamez, A., Kordon, F.: Hierarchical Set Decision Diagrams and Regular Models. In: 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09). LNCS, vol. 5505, pp. 1–15. Springer, York, UK (2009)