# State Space Analysis using Symmetries on Decision Diagrams

Maximilien Colange, Fabrice Kordon, Yann Thierry-Mieg
*LIP6, CNRS UMR 7606, Université P. & M. Curie*
*4, place Jussieu, 75005 Paris, France*
*Maximilien.Colange@lip6.fr, Fabrice.Kordon@lip6.fr,*
*Yann.Thierry-Mieg@lip6.fr*

Souheib Baarir
*LIP6, CNRS UMR 7606 and*
*Université Paris Ouest Nanterre La Défense*
*200, avenue de la République, Nanterre, France*
*Souheib.Baarir@lip6.fr*

***Abstract*—Two well-accepted techniques to tackle combinatorial explosion in model-checking are exploitation of symmetries and the use of reduced decision diagrams. Some work showed that these two techniques can be stacked in specific cases.**

**This paper presents a novel and more general approach to combine these two techniques. Expected benefits of this combination are:**

- **in symmetry-based reduction, the main source of complexity resides in the canonization computation that must be performed for each new encountered state; the use of shared decision diagrams allows one to canonize sets of states at once.**
- **in decision diagram based techniques, dependencies between variables induce explosion in representation size; the manipulation of canonical states allows to partly overcome this limitation.**

**We show that this combination is experimentally effective in many typical cases.**

***Keywords*-Symmetries, Decision Diagrams, State Space Analysis**

## I. Introduction

Formal verification of concurrent systems, while promising push-the-button technology to check the correctness of systems, rapidly encounters the state space explosion problem.

Among many techniques proposed to fight this problem, symbolic approaches based on symmetries [1], [2] and BDD [3] have proven successful in practice.

**Symmetries.** If we are given a symmetry group $G$ over states and the transition relation, we can build a quotient graph of equivalence classes (also called orbits) of states, that may be exponentially smaller than the full state graph [4]. This quotient graph preserves many properties of interest such as reachability and linear temporal logic provided the property is itself symmetric with respect to $G$.

To build such a graph, the approach most commonly used [1], [5] consists in using a canonical representative of each orbit. However, an orbit may be of exponential size with respect to the number of elements in the state vector. Thus, the computation of a canonical representative of an orbit has exponential worst case complexity in time and/or memory (if the orbit is actually built).

This work was supported by the Délégation Générale pour l'Armement.

Junttila [5] proposes a general definition of this approach for systems whose states are integer vectors and symmetry groups are arbitrary permutation groups. Using the Schreier-Sims representation [6] of permutation groups, he proposes an algorithm effective in practice to compute a representative of an equivalence class.

However, the proposed algorithm only deals with explicit encoding of the state space. Thus, the problem remains hard since the algorithm must be applied on each individual state. This prevents a direct implementation on top of symbolic data structures such as decision diagrams (DD).

**Decision Diagrams.** Reduced Ordered BDD (ROBDD) were introduced by [7] to compactly represent boolean functions over boolean domains such as large circuits. Since their first use for model-checking [3], many variants of decision diagrams have been proposed. They all allow to manipulate large sets of states symbolically. The DD size can be exponentially smaller than the size of the represented set. Thanks to dynamic programming, algorithms manipulating DD are usually polynomial in the representation size.

Unfortunately, algorithms that manipulate classical explicit data structures must be redesigned to take advantage of DD. This is not always possible, particularly if the algorithm involves separate treatments for every state.

**Combining Symmetries and Decision Diagrams.** Indeed, initial attempts to combine a symbolic representation of sets of states with a computation of a quotient graph met mitigated success. The problem, identified in [8], is that the orbit relation –allowing to map states to their representative– has exponential size when represented as a BDD, whichever the variable order chosen.

Variations such as using several representatives of an orbit, can be more effective but do not fully exploit the symmetry group.

A slightly different approach to build a quotient graph [2], is to use an abstract representation of orbits. This also allows to exploit symmetries of the transition relation, however this approach can only deal with specific groups of symmetries, and cannot easily be generalized to arbitrary permutation groups.

In practice, this approach can often be successfully com-

bined with symbolic representation of sets of states, as shown in [9] for symmetries limited to the full permutation group, or for the specific framework of Symmetric Nets (a.k.a. Well-Formed Petri nets) in [10] [11]. However, this approach is limited to specific symmetry groups and lacks generality.

**Contribution.** We propose an algorithm allowing to work with arbitrary symmetry groups, that can be effectively implemented on top of symbolic data structures. Given a total ordering on states, the smallest state in an orbit is considered as its canonical representative.

Instead of directly representing the orbit relation, we introduce a "monotonic" function that, given a state $s$, returns a state $s'$ in the same orbit such that $s' < s$, if such an element $s'$ exists. By repeatedly applying such a monotonic function in a fixpoint, we achieve the same effect as if we were using the orbit relation, without ever having to explicitly compute and represent it.

Because this function operates over sets of states, it avoids individual representative computations for each state, thus leading to a general and efficient algorithm to combine the use of symmetries with symbolic data structures.

**Outline.** Section II defines the required notions on symmetries and decisions diagrams. Then, section III details how symmetries can be represented on top of decision diagrams. An example and some benchmarks are also provided here before a conclusion in section IV.

## II. Preliminaries

This section defines the notions of quotient graph and the type of decision diagrams we use in section III.

### A. Quotient graph, definitions

We recall here the theory of symmetry reduction for state space analysis. These definitions are adapted from [5].

**Definition 1.** *Transition system*
*A transition system is a tuple $(S, \Delta, S_0)$ such that:*

- *$S$ is a finite set of states,*
- *$\Delta \subseteq S \times S$ is the transition relation,*
- *$S_0 \subseteq S$ is the set of initial states.*

Transitions for $(s_1, s_2) \in \Delta$ are noted $s_1 \to s_2$. Symmetries of transition systems are defined using a bisimilarity relation between states.

**Definition 2.** *Symmetry*
*Let $\mathcal{K} = (S, \Delta, S_0)$ be a transition system. A symmetry of $\mathcal{K}$ is a permutation $g$ over $S$ such that:*

- *$g.S_0 = S_0$*
- *$g$ is congruent with respect to the transition relation: $\forall s_1, s_2 \in S, s_1 \to s_2 \Leftrightarrow g.s_1 \to g.s_2$*

$G$, the set of all symmetries of $\mathcal{K}$, is a group because:

- the composition is associative,
- the composition of two symmetries and the inverse of a symmetry are still symmetries.

**Definition 3.** *Equivalence relation $\equiv_G$*
*Two states $s_1, s_2 \in S$ are said to be symmetric, denoted $s_1 \equiv_G s_2$, if there is a $g \in G$ such that $g.s_1 = s_2$. $\equiv_G$ is an equivalence relation over $S$. $[x]_G$ denotes the equivalence class (also called orbit) of $x$ under $\equiv_G$.*

We may now define the abstraction of a transition system using $\equiv_G$.

**Definition 4.** *Reduced transition system*
*$\tilde{\mathcal{K}} = (\tilde{S}, \tilde{\Delta}, \tilde{S}_0)$ is a reduction of $\mathcal{K}$ w.r.t. $G$ if and only if:*

- *$\tilde{S} \subseteq S$, $\forall s \in S, \exists \tilde{s} \in \tilde{S} : s \equiv_G \tilde{s}$,*
- *$\tilde{S}_0 \subseteq \tilde{S}$ and $\forall g \in G, g.\tilde{S}_0 \subseteq S_0$,*
- *$\tilde{\Delta} \subseteq \tilde{S} \times \tilde{S}$,*
- *if $\tilde{s}_1 \in \tilde{S}$ and $(\tilde{s}_1, s_2) \in \Delta$, then there exists $\tilde{s}_2 \in \tilde{S}$ such that $\tilde{s}_2 \equiv_G s_2$ and $(\tilde{s}_1, \tilde{s}_2) \in \tilde{\Delta}$,*
- *if $(\tilde{s}_1, \tilde{s}_2) \in \tilde{\Delta}$ then there exists $s_2 \in S$ such that $s_2 \equiv_G \tilde{s}_2$ and $(\tilde{s}_1, s_2) \in \Delta$.*

A reduction, $\tilde{\mathcal{K}}$ of $\mathcal{K}$ w.r.t. $G$ preserves the reachability property and, under appropriate conditions, linear temporal formulae [12], [4]. Hence, the verification can be done on $\tilde{\mathcal{K}}$. Note that this definition allows to use several representatives per orbit, generalizing the notion of quotient graph. This approach using several representatives yields a larger reduced structure but may be faster to build [5].

An abstract algorithm to compute $\tilde{\mathcal{K}}$ is presented on figure 1. Let `repr` be a function that maps an element $s \in S$ onto its representative $\tilde{s} \in [s]_G$. Let `succ` be the function that maps any state $s$ to its successors: $\texttt{succ}(s) = \{s' | s \to s'\}$.

$\tilde{S} := \texttt{repr}(S_0)$
$\tilde{\Delta} := \emptyset$
**repeat**
  **for** $s \in \tilde{S}$ **do**
    $\tilde{S}' := \texttt{repr}(\texttt{succ}(s))$
    $\tilde{\Delta} := \tilde{\Delta} \cup \{(s, \tilde{s}') | \tilde{s}' \in \tilde{S}'\}$
    $\tilde{S} := \tilde{S} \cup \tilde{S}'$
  **end for**
**until** a fixpoint is reached

Figure 1. The algorithm to generate $\tilde{\mathcal{K}}$

The size of $\tilde{S}$ depends on the function `repr`, as $\tilde{S} = \texttt{repr}(S)$, with two extreme cases:

- if `repr` is the identity, then $\tilde{S} = S$ and $\tilde{\mathcal{K}} = \mathcal{K}$.
- if `repr` maps all elements of an orbit onto the same unique element, then $\tilde{S}$ is in bijection with $S_{/G}$, and the size of $\tilde{S}$ is minimal.

Computing such a unique representative is however exponential in time for the worst case: the canonization problem is equivalent to graph isomorphism. This class of complexity is not known to have a polynomial solution [8], [5].

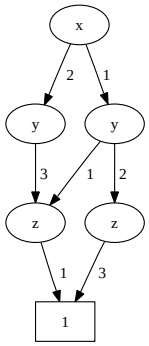



Figure 2. This DDD represents the set of sequences of assignments: $\{(x := 2; y := 3; z := 1;), (x := 1; y := 1; z := 1;), (x := 1; y := 2; z := 3;)\}$.

### B. Decision diagrams

Shared Decision Diagrams (DD) 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: nodes of the decision tree are unique in memory thanks to a canonical representation, the number of paths through the diagram (states) can be exponential in the representation size (nodes in the DD), equality of two sets can be tested in constant time, using caches most operations manipulating a DD are polynomial in the representation size, the effectiveness of the encoding strongly depends on the chosen variable ordering [13].

In this paper we rely on Data Decision Diagrams (DDD, defined in [14]), which extend classical BDD in two respects: *1)* variables are considered to have an integer domain instead of a Boolean one, and, *2)* operations over DDD are encoded using homomorphisms instead of the usual fashion where another decision diagram with two variables per variable of the state signature is used.

A DDD is a data structure for representing a set of sequences of assignments of the form $\omega_1 := v_1; \omega_2 := v_2; \ldots; \omega_n := v_n$, also noted $\omega_1 \xrightarrow{v_1} \omega_2 \xrightarrow{v_2} \cdots \omega_n \xrightarrow{v_n} \mathbf{1}$, where $\omega_i$ are variables and $v_i$ are integer values. We assume no implicit variable ordering and the same variable can occur several times in an assignment sequence (though with some constraints, see [14]). We define the terminal $\mathbf{1}$ to represent the empty assignment sequence, that terminates any valid sequence. The terminal $\mathbf{0}$ represents the empty set of assignment sequences.

**Definition 5** (DDD)**.** *Let* Var *be a set of variables, and for any* $\omega$ *in* Var, *let* $\mathrm{Dom}(\omega) \subseteq \mathbb{N}$ *be the domain of* $\omega$*. The set* $\mathbb{D}$ *of DDD is defined inductively by:*
$\delta \in \mathbb{D}$ *if either* $\delta \in \{\mathbf{0}, \mathbf{1}\}$ *or* $\delta = \langle \omega, arc \rangle$ *with* $\omega \in Var$*, and* $arc : \mathrm{Dom}(\omega) \to \mathbb{D}$ *is a mapping where only a finite subset of* $\mathrm{Dom}(\omega)$ *maps to other DDD than* $\mathbf{0}$*.*

*By convention, edges that map to the DDD* $\mathbf{0}$ *are not represented.*

For instance, consider the DDD shown in figure 2. Each path in the DDD thus corresponds to a sequence of assignments. In this work, we use DDD to represent states in $\mathbb{N}^n$, thus each assignment sequence represents a system state.

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

DDD also offer a concatenation $\delta_1 \cdot \delta_2$ which replaces terminal $\mathbf{1}$ of $\delta_1$ by $\delta_2$. This corresponds to a cartesian product. Basic and inductive homomorphisms are also introduced to define application specific operations. A more detailed description of DDD homomorphisms can be found in [14].

A basic homomorphism is a mapping $\Phi : \mathbb{D} \mapsto \mathbb{D}$ satisfying $\Phi(\mathbf{0}) = \mathbf{0}$ and $\forall \delta, \delta' \in \mathbb{D}, \Phi(\delta \cup \delta') = \Phi(\delta) \cup \Phi(\delta')$. Many basic homomorphisms are hard-coded. The sum $+$ operation between two homomorphisms ($\forall \delta \in \mathbb{D}, (\Phi_1 + \Phi_2)(\delta) = \Phi_1(\delta) \cup \Phi_2(\delta)$) 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 $\forall \delta \in \mathbb{D}, c(\delta) \subseteq \delta$. This allows to represent Boolean conditions, as $c$ selects states satisfying a given condition; thus the negation of $c$ is $\overline{c}(\delta) = \delta \setminus c(\delta)$. As a shorthand for "if-then-else", we use $\mathtt{IfThenElse}(c, h_1, h_2) = h_1 \circ c + h_2 \circ \overline{c}$, where $h_1$ and $h_2$ are homomorphisms.

The **fixpoint** $h^\star$ of a homomorphism, defined as $h^\star(\delta) = h^k(\delta)$ where $k$ is the smallest integer such that $h^k(\delta) = h^{k+1}(\delta)$, is also a homomorphism provided a finite $k$ 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 DDD $s_0$ 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 [15].

## III. Symmetries and Symbolic Structures

In this section we will develop our ideas about how to combine Symmetries and Symbolic Structures in a general framework.

### A. Assumptions

**States** We make the assumption that the system's states S are vectors of integers, of fixed size $n$: $S \subseteq \mathbb{N}^n$.

**Symmetries** We consider symmetries that permute the indexes: $\forall g \in G, \forall v = (v_1, v_2, \ldots, v_n) \in S, g.v = (v_{g.1}, v_{g.2}, \ldots, v_{g.n})$. The group of all permutations over a set of size $n$ is denoted by $\mathcal{S}_n$.

We then manipulate symmetry groups as sets of permutations. Conversely, given a set of permutations $H$, let $\langle H \rangle$ denote the group generated by $H$.

States are totally ordered. We use lexicographic ordering, noted $<$. The **canonical representative** $\hat{s}$ of an orbit $[s]_G$ is defined as its smallest element (with respect to $<$). Thus, $\forall s \in S, \hat{s} = min[s]_G$.

### B. Symbolic Symmetry algorithm

Given these premises, we use the algorithm of figure 3 to canonize a set of states.

```
set_canonize(H ⊆ S_n, S ⊆ ℕ^n):
repeat
  for g ∈ H do
    S' := {s | s ∈ S, g.s < s}
    S := S ∪ g.S'
    S := S \ S'
  end for
until S no longer evolves
return S
```

Figure 3. Symbolic algorithm to canonize a set of states.

This algorithm iterates over the permutations of $H$, applying each one only to the states that it reduces. If the permutations in $H$ are permutations of the symmetry group $G$ of the system, we are ensured that at each step of the algorithm, each state is either left as is, or mapped to a strictly smaller state belonging to its orbit. Since each orbit has a minimum (its canonical representative) this algorithm is guaranteed to converge.

Admittedly, the algorithm might visit each state of an orbit (in decreasing order, one by one), yielding worst case exponential complexity. Since the problem is equivalent to graph isomorphism, this is not surprising. In practice however, with an appropriate choice of a small set of permutations in $H$, this algorithm can be quite effective.

Let us note that the order in which the permutations of $H$ are considered in the "for" loop (or equivalently, in the composition $\bigcirc_{g \in H}$) does not impact correctness, but may impede performance.

Actually, the choice of $H$ is critical to overall performance of this algorithm. If $H = G$, then this algorithm converges after a single iteration of the outer loop ("repeat"). In other words, for each state, $H$ contains the permutation that maps it to its representative. However, this means that, on the worst case, the size of $H$ is exponential in $n$. This is congruent with the observations of [8] in which the orbit relation is shown to be exponential in representation size.

A contrario, when $H$ is small, many iterations may be necessary for the algorithm to converge, but each element of $H$ is likely to reduce larger subsets $S'$. Since the complexity of applying a permutation to a set of states is related to the representation size (in DDD nodes) and not to the number of states in the set, manipulating larger sets lowers the overall complexity.

**Monotonic$_<$ Property.** To obtain minimality, we would like to choose $H$ such that $\texttt{set\_canonize}(H,S) = \{min[s]_G | s \in S\}$.

In essence this means we require that any state $s$ that is not the minimum of its orbit $[s]_G$ can be reduced (according to $<$) by applying a permutation of $H$.

**Definition 6** (*monotonic$_<$*)**.** *Let $G$ be a subgroup of $\mathcal{S}_n$. $H \subseteq G$ is monotonic$_<$ w.r.t. $G$ if and only if:*

- $\forall s \in S, (\exists g \in G, g.s < s \implies \exists h \in H, h.s < s)$.

In algorithm 3, when states can no longer be reduced by any permutation of $H$, by definition of the *monotonic$_<$* property, the states in $S$ are the canonical representatives of the input states. When $H$ is *monotonic$_<$* w.r.t. $G$, the algorithm returns the set of canonical representatives of the input states.

If $H$ is not *monotonic$_<$* w.r.t. $G$, the algorithm behaves like the one of figure 1 when several representatives are used.

### C. Symbolic encoding

States being elements of $\mathbb{N}^n$ are naturally represented as a DDD of $n$ variables. Note that by assumption, the system size is fixed in number of variables. For systems requiring to dynamically allocate variables, a pool size bound must then be known a priori. However we are allowed to use integers with a priori unknown bounds as variables. This feature of DDD is exploited here, but the algorithm could work with boolean variables and any type of Decision Diagrams. Labels of states, if we consider a Kripke structure instead of a transition system, can be encoded as additional state variables.

To encode algorithm of figure 3 using homomorphisms, we define for any permutation $g \in \mathcal{S}_n$ :

- $\texttt{reduces}(g)$, a selector homomorphism to retain states that are reduced by $g$, i.e. $\texttt{reduces}(g)(S) = \{s | s \in S, g.s < s\}$,
- $\texttt{apply}(g)$, a homomorphism to apply $g$ to each state of a set, i.e. $\texttt{apply}(g)(S) = \{g.s | s \in S\}$

The full algorithm is then expressed by the equation:

$$\texttt{set\_canonize}(H) = \left(\bigcirc_{g \in H} \texttt{IfThenElse}(\texttt{reduces}(g), \texttt{apply}(g), Id)\right)^{\star}$$

Since convergence is ensured by the fact each orbit has a minimum, the fixpoint $^{\star}$ is well-defined. The homomorphism $\texttt{set\_canonize}(H)$ can be applied to any set of states, yielding their canonical representatives when $H$ is *monotonic$_<$*.

**Apply and Reduces.** The homomorphism `reduces`, given that we are using lexicographic order, and that states are in $\mathbb{N}^n$, is expressed as a composition of variable comparisons. For instance, consider the permutation $g = (2,3,1,4)$ of $\mathcal{S}_4$. We have $g^{-1} = (3,1,2,4)$. Hence $g$ reduces $s = (s_1, s_2, s_3, s_4)$ iff

$$s_{g^{-1}.1} < s_1 \vee (s_{g^{-1}.1} = s_1 \wedge (s_{g^{-1}.2} < s_2 \vee (s_{g^{-1}.2} = s_2 \wedge (\ldots))))$$

This general formula is instantiated for this specific $g$ in the following way:

$s_3 < s_1 \vee (s_3 = s_1 \wedge (s_1 < s_2 \vee (s_1 = s_2 \wedge s_2 < s_3)))$

Let us note that since position 4 is invariant by $g$, there are only three nested variable comparisons. Subsequent conditions are trivially simplified away. This condition is expressed using a selector homomorphism allowing comparison (by $<$ and $=$) of the value of two variables of a state. The full condition homomorphism is expressed using composition $\circ$ for $\wedge$ and the sum $+$ for $\vee$.

The homomorphism `apply` is built as a composition of transpositions of adjacent elements noted $\tau_{i,i+1}$. The original DDD definition [14] includes a general homomorphism to swap arbitrary variables of a DDD. Transposition of adjacent variables is a particular case of this.

We compute a path with the minimal number of these transpositions necessary to achieve the desired effect and compose them to build `apply`. For instance, with $g = (2,3,1,4)$ of $\mathcal{S}_4$,

$$g = \tau_{2,3} \circ \tau_{1,2}$$

Let us note that the DDD homomorphism framework allows to easily define these complex operations, hence the implementation using libDDD [16] is straightforward. As a beneficial side effect, since a given transposition $\tau$ can occur in several permutations, various permutations may benefit from the cache for transpositions.

Our algorithm can be implemented using other decision diagrams libraries, although swap and comparison of variables may not be offered natively.

Note that the same algorithmic bricks can be used to compute the orbit of states, using the equation:

$$\mathit{orbit}(H) = (\bigcirc_{g \in H}(\texttt{apply}(g) + Id))^\star$$

If $\langle H \rangle = G$, applying $\mathit{orbit}(H)$ to a set of states $S$ returns the set $\bigcup_{s \in S}[s]_G$.

**Illustrative example.** Let us detail the run of the algorithm on a small illustrative example. Figure 4 shows the intermediate DDD produced by the application of `set_canonize`$(H)$ to a system of three variables. With $G = \mathcal{S}_3$ as symmetry group, we choose $H = \{\tau_{1,2}, \tau_{2,3}\}$, which is *monotonic*$_<$ w.r.t. $G$, as will be proved in III-D. We focus on the inner loop in algorithm 3. Each step corresponds to the application of an element $g$ of $H$ to the states reduced by $g$ in the current DD. At the end of the algorithm, another iteration is necessary to check for convergence.

As we can see through this toy example, each step of the algorithm simultaneously reduces several states. In a single step, each permutation reduces all the states it can, even if they belong to different orbits.

States that belong to the same orbit are progressively collapsed onto their representative. Because of sharing of sub-structures, notice that states $(2,1,3)$ and $(2,3,1)$ in 4(a) are collapsed onto $(2,1,3)$ in 4(b). $(2,1,3)$ is not a canonical representative, but it is smaller than $(2,3,1)$. At this step, the two states are merged, allowing to share any subsequent canonization step. In general, each step –with complexity polynomial in the DD size– might merge exponentially many states. This contrasts with explicit approaches that canonize all these states individually.

### D. Finding a *monotonic*$_<$

As previously explained, whatever the choice of $H \subseteq G$ the algorithm of Fig. 3 is still valid. On the other hand, the choice of $H$ is critical to its efficiency. Ideally $H$ should be *monotonic*$_<$ w.r.t. $G$ to obtain maximal reduction, and heuristically for decision diagram based implementations, $H$ should be as small as possible.

In the general case, the computation of a set $H$ *monotonic*$_<$ w.r.t. $G$ that is of minimal size, is in $O(n^n)$ with a brute force algorithm.

Efficient data structures to store groups of permutations such as the Schreier-Sims representation [6] could provide a candidate to define $H$. However, the generating set they provide is not *monotonic*$_<$ in general. Even when it is, its size can be much larger than necessary. For instance, the Schreier-Sims representation of the full group of permutations $\mathcal{S}_n$ is quadratic in $n$, whereas a *monotonic*$_<$ set of size $n$ exists.

We provide in this section an appropriate set $H$ for common symmetry groups.

**Proposition 1.** *The set of adjacent transpositions is monotonic*$_<$ *for* $\mathcal{S}_n$.

*Proof:* Let $s = (s_1, \ldots, s_n) \in \mathbb{N}^n$ be a state, such that $\exists g \in \mathcal{S}_n, g.s < s$. This means that $s$ is not sorted, and therefore, there exists an index $i$ such that $s_i > s_{i+1}$. Thus $s' = \tau_{i,i+1}.s = (s_1, \ldots, s_{i+1}, s_i, \ldots, s_n) < s$. ■

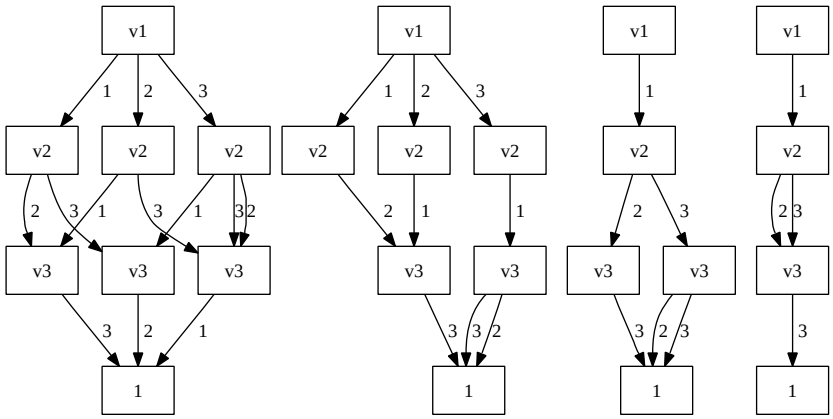
**Proposition 2.** *Let $r$ be the rotation $(2,3,\ldots,n,1)$, and $G = \langle r \rangle = \{id, r, r^2 \ldots r^{n-1}\}$. Then $G$ is the only monotonic*$_<$ *set w.r.t. $G$.*

*Proof:* For $0 < i \leq n$, let $s = (1,2,\ldots i-1,0,i+1,\ldots n)$. Then the only rotation in $G$ that reduces $s$ is $r^i$. ■

These two groups are the most frequently encountered groups of symmetries in the literature, as they occur naturally in many symmetric systems. This gives us *monotonic*$_<$ sets of size $n$ for these two groups. The two properties above are still true when considering groups that act on a subset of the system variables.

When the symmetries of the system arise from several symmetry groups (i.e. symmetries of subsystems), we choose to use the union of their respective *monotonic*$_<$ sets.

Let $G = \langle E \cup F \rangle$, and $H_E$, $H_F$ be *monotonic*$_<$ sets w.r.t. $E$ and $F$ respectively. $H_G = H_E \cup H_F$ is *monotonic*$_<$ w.r.t. $G$ if $E$ and $F$ act on disjoint sets of variables. Otherwise, we are not ensured that $H_G$ is *monotonic*$_<$ w.r.t. $G$, but it can still be used as a good candidate set for the algorithm.




(a) contains the orbit of $(1,2,3)$ and the state $(3,3,1)$. (b) after $\tau_{2,3}$ in inner loop (c) after $\tau_{1,2}$ (d) after $\tau_{2,3}$

Figure 4. Using $G = \mathcal{S}_3$, we obtain $H = \{\tau_{1,2}, \tau_{2,3}\}$. `set_canonize`$(H)$ applied to (a) successively gives (b), (c), (d). (d) is the set of canonical representatives $\{(1,2,3),(1,3,3)\}$.

| | | # generated states | | | time (s) | | | Memory (MB) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | **Scale** | **LoLa** | **DD** | **DD-Sym** | **LoLa** | **DD** | **DD-Sym** | **LoLa** | **DD** | **DD-Sym** |
| Soft. Product Line | 80 | 486 | $3.8685 \cdot 10^{25}$ | 486 | 540 | 135 | 193 | 364 | 273 | 558 |
| Soft. Product Line | 100 | 606 | $4.0564 \cdot 10^{31}$ | 606 | 1,488 | 238 | 399 | 558 | 411 | 964 |
| Soft. Product Line | 120 | 726 | $4.2535 \cdot 10^{37}$ | 726 | 3,284 | 389 | 776 | 795 | 589 | 1,305 |
| Soft. Product Line | 140 | — | $4.4601 \cdot 10^{43}$ | 846 | — | 581 | 1,417 | — | 800 | 1,311 |
| Soft. Product Line | 160 | — | $4.6768 \cdot 10^{49}$ | 966 | — | 844 | 2,472 | — | 1,014 | 1,314 |
| Soft. Product Line | 180 | — | $4.9040 \cdot 10^{55}$ | — | — | 1,167 | — | — | 1,256 | — |
| Soft. Product Line | 200 | — | $5.1422 \cdot 10^{61}$ | — | — | 1,587 | — | — | 1,560 | — |
| clients servers | 5 | 22,840 | 805,284 | 192 | 1 | 4 | 1.4 | 14 | 122 | 58 |
| clients servers | 6 | 425,646 | 11,368,449 | 448 | 27 | 18 | 3.8 | 234 | 383 | 140 |
| clients servers | 7 | 3,630,511 | 157,169,826 | 1,024 | 452 | 67 | 8.8 | 1,971 | 1,232 | 267 |
| clients servers | 8 | — | 2,130,740,721 | 2,295 | — | 306 | 19 | — | 3,200 | 521 |
| clients servers | 12 | — | — | 42,926 | — | — | 350 | — | — | 4,004 |
| SaleStore | 5 | 4,456 | 71,238 | 106 | 0.1 | 0.93 | 0.32 | 3.9 | 36 | 17 |
| SaleStore | 10 | 1,410,608 | 184,554,369 | 496 | 111 | 37 | 3.2 | 689 | 708 | 99 |
| SaleStore | 15 | — | 207,629,747,172 | 1,186 | — | 692 | 12.5 | — | 4,190 | 303 |
| SaleStore | 20 | — | — | 2,176 | — | — | 30 | — | — | 722 |
| SaleStore | 30 | — | — | 5,056 | — | — | 154 | — | — | 2,455 |
| SaleStore | 40 | — | — | 9,136 | — | — | 495 | — | — | 4,194 |

Table I
PERFORMANCES OF STATE SPACE GENERATION USING LOLA, PLAIN DD AND THE COMBINATION OF DD WITH SYMMETRIES.

Other types of symmetries on data values, such as $v = (obj_1, obj_2 \ldots ref_1, ref_2)$, and $g.v = (obj_{g.1}, obj_{g.2} \ldots g.ref_1, g.ref_2)$ can be integrated into our algorithm seamlessly. This symmetry is of interest as it corresponds to the case where $obj_1$ and $obj_2$ contain similar objects and $ref_1, ref_2$ are references to these objects, that need to be reindexed if we exchange the positions of the two objects. This case is encountered when canonizing the memory (heap in particular) of a concurrent system.

### *E. Assessment*

In this section, we assess our algorithm on some examples. We compare our approach to an implementation of Junttila's algorithm for symmetry reduction and to symbolic model checking without the use of symmetries.

The tool LoLa [17] uses a Schreier-Sims representation of the symmetry group and produces a reduced system with potentially several representatives per orbit. It works with explicit data structures, thus its memory consumption grows linearly with the number of representative states. LoLa is a well maintained and mature software.

On the other hand, we compare our algorithm to libITS [16], [18], a model checker implemented using DDD, but no symmetries. Up to the addition of symmetries, it uses the same encoding (states, transition relation) as our prototype DD-Sym.

Table I compares the size of the produced state space (time and memory consumption during its elaboration), for these three tools. Experiments were run on a Xeon 64 bits at 2.6 GHz processor with a time limitation of 1 hour and memory limit of 5Gbytes. The following models (shown in annex) were processed:

**Software Product Line [19].** It is a model extracted and then adapted from a case study concerning a software configuration process. Features and configuration options are

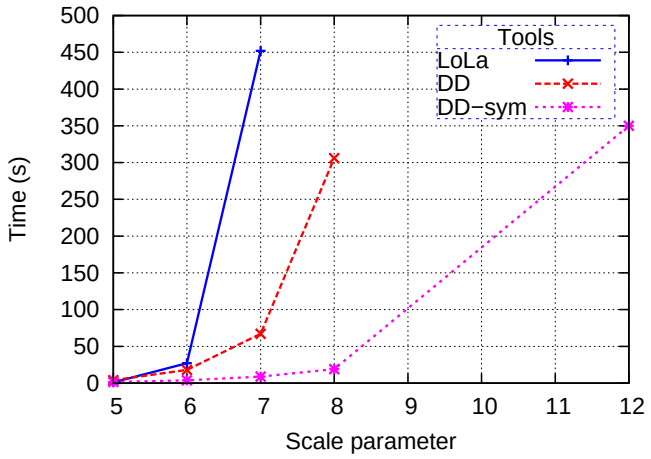



Figure 5. Clients servers model: time to build the state space.

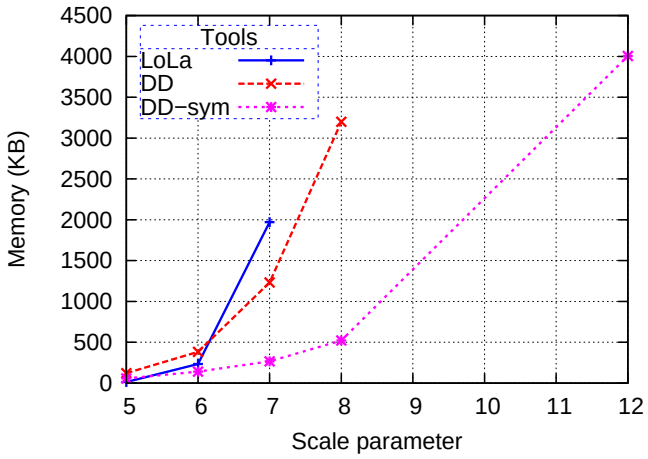



Figure 7. Clients servers model: memory to build the state space.

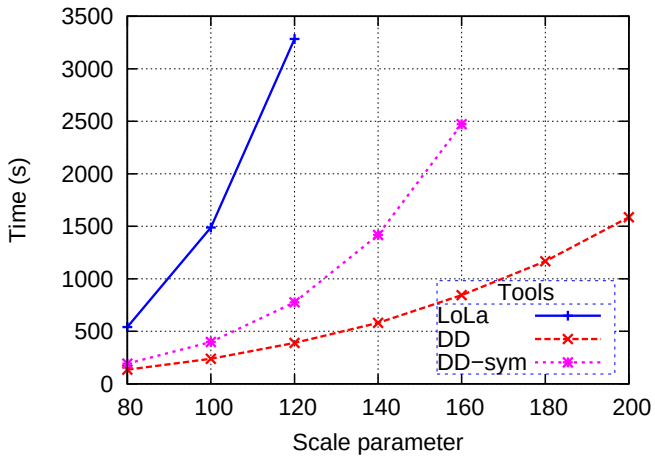



Figure 6. Software Product Line model: time to build the state space.

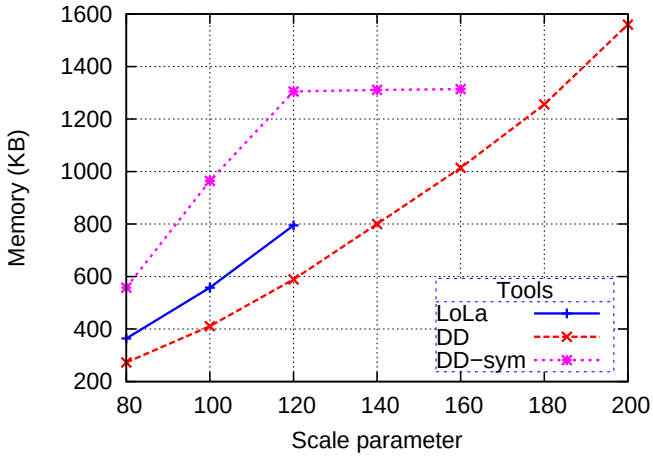



Figure 8. Software Product Line model: memory to build the state space.

fully symmetric domains, that do not interact directly. Thus, the union of their respective *monotonic*$_<$ sets is *monotonic*$_<$ w.r.t. the symmetry group of the model. LoLa and DD-Sym both compute the quotient graph, with one representative per orbit. The symmetry group exhibited by this model is particularly simple; this means the canonization procedure has a relatively low complexity. The classical DD implementation has the best performance on this example, and LoLa the worst. However, DD-Sym's memory consumption does not grow beyond 1.3Gbyte, at the point where the DD garbage collector activates. This means that DD-Sym, on this model, does not compute intermediate structures whose size exceeds 1.3Gbyte, and could actually run within a memory confined to 1.3Gbyte. This limit is paid in time, as the garbage collection frees the DD caches. In deed, DD-Sym fails earlier than the classical DD implementation, due to time confinement. On the other hand, the classical DD implementation, that uses the same garbage collection mechanism, has a much bigger memory peak.

**Clients servers [10].** It models a simple remote procedure call protocol between $n$ clients and $n$ servers sharing a common communication channel. Clients (resp. servers) are considered indistinguishable up to their identity. Thus, we have a full symmetry group on clients and a full symmetry group on servers. We use the union of their respective *monotonic*$_<$ sets, which is not itself *monotonic*$_<$ w.r.t. the symmetries of the whole system. However, on this model, DD-Sym scales better than the other tools. Although the reduction factor is good, the multiple representatives approach in LoLa still retains too many states to cope with larger scale parameters. DDDs are able to handle up to 2 billion states, but fail earlier than our prototype.

**SaleStore [11].** It models a shopping mall where clients can shop for gifts. Clients and gifts form two fully symmetric domains, that interact when a client buys some gifts. Similarly to the clients servers model, we use the union of

the *monotonic*$_<$ sets. Again, the number of states in LoLa's representation grows very fast; it fails before the purely symbolic approach. DD-Sym allows the symbolic approach to scale up to much larger model parameters, as the number of representatives grows very slowly.

**Discussion on the Results of Table I and figures 5 to 8.** The three tools do not compute the same representation of the state space, hence they don't always find the same number of states. The classical DD tool computes the full transition system without any symmetry reduction, while both Lola and DD-Sym compute a quotient structure and the number of states shown is actually the number of orbit representatives computed.

Both DD-Sym and Lola use an algorithm which might lead to several representatives of an orbit being represented. LoLa's strategy to compute several representatives for each orbit reduces the cost of canonization, and is supposed to be a good trade-off between the time-consuming canonization and the size of the quotient graph. Our own algorithm may produce several representatives if the provided set $H$ is not *monotonic*$_<$ w.r.t. $G$.

In order to control when the tested tools achieve full reduction, we have processed small instances of the models with another tool that is guaranteed to perform full reduction. In practice, both LoLa and DD-Sym compute a single representative per orbit for the Software Product Line model, achieving full reduction. For the two other models, the sets $H$ we use to canonize are not *monotonic*$_<$. In spite of this, DD-Sym only computes a single representative per orbit for the SaleStore model. On the client-server model, neither LoLa nor DD-Sym achieve maximum reduction, but LoLa computes many more representative than we do.

LoLa fails on the Software Product Line model, due to the time confinement, although it consumes 33% memory more than the DD classical implementation. As previously explained, DD-Sym's memory consumption reaches a maximum when the garbage collector activates. DD-Sym then fails on bigger instances due to time confinement, but would consume less memory than the classical DD implementation if the time limit were higher.

For the two other models, DD-Sym exhibits the best performances, and LoLa the worst. DD-Sym handles the models for higher scaling parameters, and its time and memory consumption grow slower than the two other tools. LoLa handles fewer instances and its time and memory consumption are higher than the pure-DD tool.

The great number of states handled by the DD tool with a reasonable amount of memory shows the strength of decision diagrams to compress large state sets. Moreover, this assessment fully validates our novel approach, as it favorably compares to both Junttila's algorithm and the classical DD approach. We thus have designed a way to combine the two symbolic approaches so that their respective optimizations can stack.

These results, while preliminary, are encouraging. It allows our DD checker to scale better for some symmetric models. It also favorably compares to explicit symmetry-based methods.

DD-Sym will be integrated into the ITS framework, and extended to use Hierarchical Set Decision Diagrams [18].

## IV. Conclusion

We have presented a novel approach to combine symmetries with symbolic data structures. It relies on the choice of an appropriate subset of symmetries, that allows to compute a reduced state space without needing to represent the orbit relation. Our algorithm supports arbitrary symmetry groups. Even if a *monotonic*$_<$ set cannot be computed easily, we provide an approximation that works well in practice for commonly encountered symmetries. We ensure correctness even if the provided set of symmetries does not respect the *monotonic*$_<$ property; this simply yields a larger state space.

Although our experiments are so far limited, we show that this approach can improve a method that only uses decision diagrams.

We are currently investigating the definition of *monotonic*$_<$ sets for other symmetries, such as those encountered when considering memory addresses and pointers.

Another perspective in the context of local symmetries [20] [21] involves adaptation of the set used for canonization during the state space construction.

## References


[1] C. Norris Ip and D. Dill, "Better verification through symmetry," *Formal methods in system design*, vol. 9, no. 1, pp. 41–75, 1996.

[2] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "Stochastic well-formed colored nets and symmetric modeling applications," *Computers, IEEE Transactions on*, vol. 42, no. 11, pp. 1343–1360, 1993.

[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: $10^{20}$ States and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.

[4] E. Clarke, E. Emerson, S. Jha, and A. Sistla, "Symmetry reductions in model checking," in *Computer Aided Verification*. Springer, 1998, pp. 147–158.

[5] T. Junttila, "On the symmetry reduction method for Petri nets and similar formalisms," Ph.D. dissertation, Helsinki University of Technology, Espoo, Finland, 2003.

[6] C. Sims, "Computation with permutation groups," in *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*. ACM, 1971, pp. 23–28.

[7] R. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[8] E. Clarke, R. Enders, T. Filkorn, and S. Jha, "Exploiting symmetry in temporal logic model checking," *Formal Methods in System Design*, vol. 9, no. 1, pp. 77–104, 1996.

[9] E. Emerson and T. Wahl, "On combining symmetry reduction and symbolic representation for efficient model checking," *Correct Hardware Design and Verification Methods*, pp. 216–230, 2003.

[10] Y. Thierry-Mieg, J. Ilié, and D. Poitrenaud, "A symbolic symbolic state space representation," *Formal Techniques for Networked and Distributed Systems–FORTE 2004*, pp. 276–291, 2004.

[11] M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg, "Crocodile: a symbolic/symbolic tool for the analysis of symmetric nets with bag," *Applications and Theory of Petri Nets*, pp. 338–347, 2011.

[12] K. Ajami, S. Haddad, and J. Ilie, "Exploiting symmetry in linear time temporal logic model checking: One step beyond," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 52–67, 1998.

[13] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[14] J. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P. Wacrenier, "Data decision diagrams for Petri net analysis," *Application and Theory of Petri Nets 2002*, pp. 129–158, 2002.

[15] A. Hamez, Y. Thierry-Mieg, and F. Kordon, "Hierarchical set decision diagrams and automatic saturation," *Applications and Theory of Petri Nets*, pp. 211–230, 2008.

[16] MoVe team. The `libddd` home page. [Online]. Available: http://move.lip6.fr/software/DDD

[17] Universität Rostock. The `LoLa` home page. [Online]. Available: http://www.informatik.uni-rostock.de/tpp/lola/

[18] Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon, "Hierarchical set decision diagrams and regular models," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 5505, pp. 1–15, 2009.

[19] R. Muschevici, D. Clarke, and J. Proenca, "Feature petri nets," in *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)*, vol. 2, 2010.

[20] E. Emerson and R. Trefler, "From asymmetry to full symmetry: New techniques for symmetry reduction in model checking," *Conference on Correct Hardware Design and Verification Methods*, pp. 142–156, 1999.

[21] T. Wahl, "Adaptive symmetry reduction," in *Computer Aided Verification*. Springer, 2007, pp. 393–405.