Decision Diagrams for Petri Nets: which Variable Ordering?

Elvio Gilberto Amparore[1], Susanna Donatelli[1], Marco Beccuti[1], Giulio Garbi[1], Andrew Miner[2]

[1] Università di Torino, Dipartimento di Informatica, Italy
{amparore,susi,beccuti}@di.unito.it
[2] Iowa State University, Iowa, USA
asminer@iastate.edu

Abstract. The efficacy of decision diagram techniques for state space generation is known to be heavily dependent on the variable order. Ordering can be done a-priori (static) or during the state space generation (dynamic). We focus our attention on static ordering techniques. Many static decision diagram variable ordering techniques exist, but it is hard to choose which method to use, since only fragmented information about their performance is available. In the work reported in this paper we used the models of the Model Checking Contest 2016 edition to provide an extensive comparison of 14 different algorithms, in order to better understand their efficacy. Comparison is based on the size of the decision diagram of the generated state space. The state space is generated using the Saturation method provided by the Meddly library.

Keywords: decision diagrams, static variable ordering, heuristic optimization, saturation.

1 Introduction

Binary Decision diagram (BDD) [10] is a well-known data structure that was extensively used in industrial hardware verification thanks to its ability of encoding complex boolean functions on very large domains. In the context of discrete event dynamic systems in general, and of Petri nets in particular, BDDs and its extensions (e.g. Multi-way Decision Diagram -MDD) were proposed to efficiently generate and store the state space of complex systems. Indeed, symbolic state space generation techniques exploit Decision Diagrams (DDs) because they allow to encode and manipulate entire sets of states at once, instead of storing and exploring each state explicitly.

The size of DD representation is known to be strongly dependent on variable orders: a good ordering can significantly change the memory consumption and the execution time needed to generate and encode the state space of a system. Unfortunately finding the optimal variable ordering is known to be NP-complete [9]. Therefore, efficient DD generation is usually reliant on various heuristics for the selection of (sub)optimal orderings. In this paper we will only consider *static* variable ordering, i.e. once the variable ordering l is selected, the MDD construction starts without the possibility of changing l. In the literature several papers were published to study the topic of variable ordering. An overview of these works can be found in [25], and in the recent work in [19]. In particular the latter work considers a new set of variable ordering algorithms, based on Bandwidth-reduction methods [27], and observes that they can be successfully applied for variable ordering. We also consider the work published in [18] (based on the ideas in [28]), which are state-of-the-art variable ordering methods specialized for Petri nets.

The motivation of this work was to understand how these different algorithms for variable orderings behave. Also, we wanted to investigate whether the availability of structural information on the Petri net model could make a difference. As far as we know there is no extensive comparison of these specific methods.

In particular we have addressed the following research objectives:

1. Build an environment (a *benchmark*) in which different algorithms can be checked on a vast number of models.
2. Investigate whether structural information like P-semiflows can be exploited to define better algorithms for variable orderings.
3. What are the right metrics to compare variable ordering algorithms in the most fair manner.

To achieve these objectives we have built a benchmark in which 14 different algorithms for variable orderings have been implemented and compared on state space generation of the Petri nets taken from the models of the Model Checking context (both colored and uncolored), 2016 edition [16]. The implementation is part of RGMEDD [6], the model-checker of GreatSPN [5], and uses MDD saturation [12]. The ordering algorithms are either taken from the literature (both in their basic form and with a few new variations) or they were already included in GreatSPN.

Figure 1, left, depicts the benchmark workflow. Given a net system $\mathcal{S} = (\mathcal{N}, m_0)$ all ordering algorithms in $\mathcal{A}$ are run (box 1), then the reachability set RS_l of the system is computed *for each* ordering $l \in \mathcal{L}$ (box 2) and algorithms are ranked according to some MDD metrics $\mathrm{MM}(RS_l)$, (box 3). The best algorithm a^* is then the best algorithm for solving the PN system $\mathcal{S} = (\mathcal{N}, m_0)$ (box 4) and its state space RS_l could be the one used to check properties.

This workflow allows to: 1) provide indications on the best performing algorithm for a given model and 2) compare the algorithms in $\mathcal{A}$ on a large set of models to possibly identify the algorithm with the best average performances. The problem of defining a ranking among algorithms (or of identifying the "best" algorithm) is non-trivial and will be explored in Section 3.

Figure 1, right, shows a high level view of the approach used to compare variable ordering algorithms in the benchmark. Columns represents algorithms, and rows represents *model instances*, that is to say a Petri net model with an associated initial marking. A square in position (j, k) represents the state space generation for the j^{th} model instance done by GreatSPN using the variable

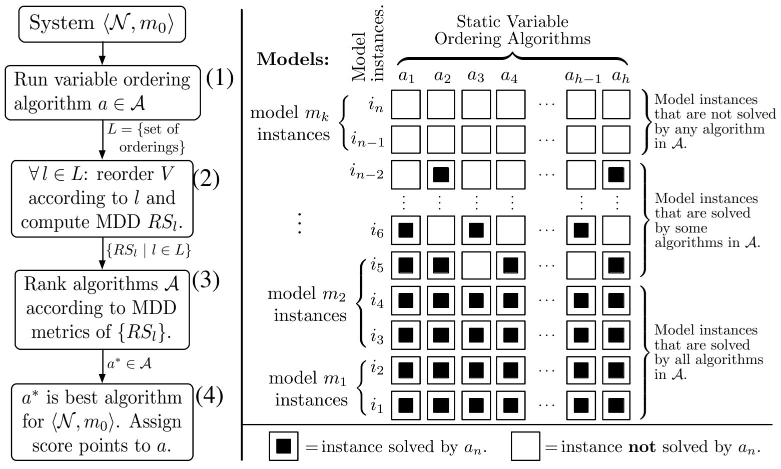

System $\langle\mathcal{N}, m_0\rangle$
Run variable ordering algorithm $a \in \mathcal{A}$ (1)
$L = \{$set of orderings$\}$
$\forall l \in L$: reorder V according to l and compute MDD RS_l. (2)
$\{RS_l \mid l \in L\}$
Rank algorithms $\mathcal{A}$ according to MDD metrics of $\{RS_l\}$. (3)
$a^* \in \mathcal{A}$
a^* is best algorithm for $\langle\mathcal{N}, m_0\rangle$. Assign score points to a. (4)
Models:
Model instances.
Static Variable Ordering Algorithms
a_1 a_2 a_3 a_4 a_{h-1} a_h
model m_k instances
model m_2 instances
model m_1 instances
i_n i_{n-1} i_{n-2} i_6 i_5 i_4 i_3 i_2 i_1
Model instances that are not solved by any algorithm in $\mathcal{A}$.
Model instances that are solved by some algorithms in $\mathcal{A}$.
Model instances that are solved by all algorithms in $\mathcal{A}$.
■ = instance solved by a_n.
□ = instance **not** solved by a_n.

Fig. 1. Workflow for analysis and testing of static variable ordering algorithms.

ordering computed by algorithm a_k. A black square indicates that the state space was generated within the given time and memory limits.

In the analysis of the results from the benchmark we shall distinguish among model instances for which no variable ordering was good enough to allow GreatSPN to generate the state space (only white squares on the model instance row, as for the first two rows in the figure), model instances for which at least one variable ordering was good enough to generate the state space (at least one black square in the row), and model instances in which GreatSPN generates the state space with all variable orderings (all black squares in the row), that we shall consider "easy" instances.

In the analysis it is also important to distinguish whether we evaluate ordering algorithms w.r.t. all possible instances or on a representative set of them. Figure 1, right, highlights that instances are not independent, since they are often generated from the same "model" that is to say the same Petri net $\mathcal{N}$ by varying the initial marking m_0 or some other parameter (like the cardinality of the color classes). As we shall see in the experimental part, collecting measures over all instances, in which all instances have the same weight, may lead to a distortion of the observed behaviour, since the number of instances per model can differ significantly. A measure "per model" is therefore also considered.

This work could not have been possible without the models made available by the Model Checking Contest, the functions of the Meddly MDD library and the GreatSPN framework. We shall now review them in the following.

Model Checking Contest. The Model Checking Contest[16] is a yearly scientific event whose aim is to provide a comparison among the different available verification tools. The 2016 edition employed a set of 665 PNML instances generated from 65 (un)colored models, provided by the scientific community. The participating tools are compared on several examination goals, i.e. state space, reachability, LTL and CTL formulas. The MCC team has designed a score system to evaluate tools that we shall employ as one of the considered benchmark metrics for evaluating the algorithms, as evaluating the orderings can be reduced to evaluating the same tool, GreatSPN, in as many variations as the number of ordering algorithms considered.

Meddly library. Meddly (Multi-terminal and Edge-valued Decision Diagram LibrarY) [8] is an open-source library implementation of Binary Decision Diagrams (BDDs) and several variants, including Multi-way Decision Diagrams (MDDs, implemented "natively") and Matrix Diagrams (MxDs, implemented as MDDs with an identity reduction rule). Users can construct one or more forests (collections of nodes) over the same or different domains (collections of variables). Several "apply" operations are implemented, including customized and efficient relational product operations and saturation [12] for generating the set of states (as an MDD) reachable from an initial set of states according to a transition relation (as an MxD). Saturation may be invoked either with an already known ("pre-generated") transition relation, or with a transition relation that is built "on the fly" during saturation [13], although this is currently a prototype implementation. The transition relation may be specified as a single monolithic relation that is then automatically split [23], or as a relation partitioned by levels or by events [14], which is usually preferred since the relation for a single Petri net transition tends to be small and easy to construct.

GreatSPN framework. GreatSPN is a well-known collection of tools for the design and analysis of Petri net models [5, 6]. The tools are aimed at the qualitative and quantitative analysis of Generalized Stochastic Petri Net (GSPN) [1] and Stochastic Symmetrical Net (SSN) through computation of structural properties, state space generation and analysis, analytical computation of performance indices, fluid approximation and diffusion approximation, symbolic CTL model checking, all available through a common graphical user interface [4]. The state space generation [7] of GreatSPN is based on Meddly. In this paper we use the collection of variable ordering heuristics implemented in GreatSPN. This collection has been enlarged to include all the variable ordering algorithms described in Section 2.

The paper is organized as follows: Section 2 reviews the considered algorithms; Section 3 describes the benchmark (models, scores and results); Section 4 considers a parameter estimation problem for optimizing one of the best methods found (Sloan) and Section 5 concludes the paper outlining possible future directions of work.

2 The set $\mathcal{A}$ of variable ordering algorithms

In this section we briefly review the algorithms considered by the benchmark. Although our target model is that of Petri nets, we describe the algorithms in a more general form (as some of them were not defined specifically for Petri nets). We therefore consider the following high level description of the model.

Let V be the set of *variables*, that translates directly to MDD levels. Let E be the set of events in the model. Events are connected to *input* and *output* variables. Let $V^{\text{in}}(e)$ and $V^{\text{out}}(e)$ be the sets of input and output variables of event e, respectively. Let $E^{\text{in}}(v)$ and $E^{\text{out}}(v)$ be the set of events to which variable v participates as an input or as an output variable, respectively. For some models, structural information is known in the form of disjoint variables partitions named *structural units*. Let Π be the set of structural units. With $\Pi(v)$ we denote the unit of variable v. Let $V(\pi)$ be the set of vertices in unit $\pi \in \Pi$. In this paper we consider three different types of structural unit sets. Let Π_{PSF} be the set of units corresponding to the P-semiflows of the net, obtained ignoring the place multiplicity. On some models, structural units are known because they are obtained from the composition of smaller parts. We mainly refer to [17] for the concept of Nested Units (NU). Let Π_{NU} be this set of structural units, which is defined only for a subset of models. Finally, structural units can be derived using a clustering algorithm. Let Π_{Cl} be such set of units. We will discuss clustering in section 2.5.

Following this criteria, we subdivide the set of algorithms $\mathcal{A}$ into: The set $\mathcal{A}_{\text{Gen}}$, that do not use any structural information; The set $\mathcal{A}_{\text{PSF}}$ that requires Π_{PSF}; The set $\mathcal{A}_{\text{NU}}$ that require Π_{NU}. Since clustering can be computed on almost any model, we consider method that use Π_{Cl} as part of $\mathcal{A}_{\text{Gen}}$.

In our context, the set of MDD variables V corresponds to the place set of the Petri net, and the set of events is the transition set of the Petri net. Let $l : V \to \mathbb{N}$ be a variable order, i.e. an assignment of consecutive integer values to the variables V.

2.1 Force-based orderings

The FORCE heuristic, introduced in [3], is a n-dimensional graph layering technique based on the idea that variables form an hyper-graph, such that variables connected by the same event are subject to an "attractive" force, while variables not directly connected by an event are subject to a "repulsive" force. It is a simple method that can be summarized as follows:

Algorithm 1 Pseudocode of the FORCE heuristic.

```
Function FORCE:
    Shuffle the variables randomly.
    repeat $K$ times:
        for each event $e \in E$:
            compute center of gravity $cog_e = \frac{1}{|e|} \sum_{v \in e} l(v)$
        for each variable $v \in V$:
            compute hyper-position $p(v) = \frac{1}{E(v)} \sum_{e \in E(v)} cog_e$
        Sort vertices according the their $p(v)$ value.
        Weight obtained variable order using metric function $f(l)$.
    return the variable order that had the best metric among
            the $K$ generated permutations.
```

Algorithm 1 gives the general skeleton of the FORCE algorithm. One of the most interesting part of this method is that it can be seen as a *factory* of variable orders, that generates K different orders. The algorithm starts by shuffling the variables set, then it iterates K times trying to achieve a convergence. The version of FORCE that we tested uses $K = 200$. In our experience, FORCE does not converge stably to a single permutation on most models, and it generates a new permutation at every iteration.

After having produced a candidate variable order l, a metric function $f(l) \rightarrow \mathbb{R}$ is used to estimate the *quality* of that order. The chosen order is then the one that maximises (or minimises) the target metric function. In our benchmark we tested the following metric functions:

- *Point-Transition Spans*: the PTS score is given by the sum of all distances between the center of gravities cog_e and their connected variables. The formula is the following: $\mathrm{PTS}(l) = \frac{1}{|E|\cdot|V|} \sum_{e \in E} \left(\sum_{v \in V(e)} \left| p(v) - cog_e \right| \right)$.
- *Normalized Event Span*: the NES score [26] is based on the idea of measuring the sum of the *event spans* of each event e. The event span $span_l(e)$ of event e for the variable order l is defined as the largest distance in l between every pair of variables $v, v' \in V(e)$. The metric is then defined as a normalized sum of these spans: $\mathrm{NES}(l) = \frac{1}{|E|} \sum_{e \in E} \frac{span(e)+1}{|V|}$.
- *Weighted Event Span*: WES [26] is a modified version of NES where events closer to the top of the variable order weigh more. This modification is based on the assumption that the Saturation algorithm performs better when most events are close to the bottom of the order. The formula of WES^i is: $\mathrm{WES}^i(l) = \frac{1}{|E|} \sum_{e \in E} \frac{span(e)+1}{|V|} \cdot \left(\frac{2 \cdot top(e)}{|V|} \right)^i$, with $i = 1$.

In addition to the evaluation metrics, structural information of the model can be used to establish additional *center of gravity*. We tested three variations of the FORCE method, which are:

- `FORCE-*`: Events are used as centers of gravity, as described in Algorithm 1, where `*` is one among `PTS`, `NES` or `WES`.
- `FORCE-P`: P-semiflows are used as center of gravities instead of the events. The method is only tested for those models that have P-semiflows. It uses the WES metric to select between the K generated orderings.
- `FORCE-NU`: Structural units are used as center of gravities, along with the events. This intuitively tries to keep together those variables that belong to the same structural unit. Again, WES is used for the metric function.

The set $\mathcal{A}$ of algorithms considered in the benchmark includes: `FORCE-PTS`, `FORCE-NES` and `FORCE-WES` in $\mathcal{A}_{\mathrm{Gen}}$; the method `FORCE-P` in $\mathcal{A}_{\mathrm{PSF}}$; and the method `FORCE-NU` in $\mathcal{A}_{\mathrm{NU}}$, for a total of 5 variations of this method.

2.2 Bandwidth-reduction methods

Bandwidth-reduction(BR) methods are a class of algorithms that try to permute a sparse matrix into a band matrix, i.e. where most of the non-zero entries are confined in a (hopefully) small band near the diagonal. It is known [11] that reducing the event span in the variable order generally improves the compactness of the generated MDD. Therefore, event span reduction can be seen as an equivalent of a bandwidth reduction on a sparse matrix. A first connection between bandwidth-reduction methods and variable ordering has been tested in [19] and in [22] on the model bipartite graph. In these works the considered BR methods where: The Reverse Cuthill-Mckee [15]; The King algorithm [20]; and The Sloan algorithm [27]. The choice was motivated by their ready availability in the Boost-C++ and in the ViennaCL library. In particular, Sloan, which is the state-of-the-art method for bandwidth reduction, showed promising performances as a variable ordering method. In addition, Sloan almost always outperforms [19] the other two BR methods, but for completeness of our benchmark we have decided to replicate the results. We concentrate our review on the Sloan method only, due to its effectiveness and its parametric nature.

The goal of the Sloan algorithm is to condensate the entries of a symmetric square matrix $\mathbf{A}$ around its diagonal, thus reducing the matrix *bandwidth* and *profile* [27]. It works on symmetric matrices only, hence it is necessary to impose some form of translation of the model graph into a form that is accepted by the algorithm. The work in [22] adopts the symmetrization of the dependency graph of the model, i.e. the input matrix $\mathbf{A}$ for the Sloan algorithm will have $(|V|+|E|)\times(|V|+|E|)$ entries. We follow instead a different approach. The size of $\mathbf{A}$ is U, with $|V| \leq U \leq |V|+|E|$. Every event e generates entries in $\mathbf{A}$: when $|V^{\text{in}}(e)|\times|V^{\text{out}}(e)| < T$, with T a threshold value, all the cross-product of $V^{\text{in}}(e)$ vertices with the $V^{\text{out}}(e)$ vertices are nonzero entries in $\mathbf{A}$. If instead $|V^{\text{in}}(e)|\times|V^{\text{out}}(e)| \geq T$, a pseudo-vertex v_e is added, and all $V^{\text{in}}(e)\times\{v_e\}$ and $\{v_e\}\times V^{\text{out}}(e)$ entries in $\mathbf{A}$ are set to be nonzero. Usually U will be equal to V, or just slightly larger. This choice better represents the variable–variable interaction matrix, while avoiding degenerate cases where highly connected event could generate a dense matrix $\mathbf{A}$. In our implementation, T is set to 100. The matrix is finally made symmetric using: $\mathbf{A}' = \mathbf{A} + \mathbf{A}^T$. As we shall see in section 3, the computational cost of Sloan remains almost always bounded.

Algorithm 2 Pseudocode of the Sloan algorithm.

```
Function Sloan:
    Select a vertex $u$ of the graph.
    Select $v$ as the most-distant vertex to $u$ with a graph visit.
    Establish a gradient from 0 in $v$ to $d$ in $u$ using a depth-first visit.
    Initialize visit frontier $Q = \{v\}$
    repeat until $Q$ is empty:
        Remove from the frontier $Q$ the vertex $v'$ that minimizes $P(v')$.
        Add $v'$ to the variable ordering $l$.
        Add the unexplored adjacent vertices of $v'$ to $Q$.
```

A second relevant characteristic of Sloan is its *parametric priority function* $P(v')$, which guides variable selection in the greedy strategy. A very compact pseudocode of Sloan is given in Algorithm 2. A more detailed one can be found in [21]. The method follows two phases. In the first phase it searches a pseudo-diameter of the $\mathbf{A}$ matrix graph, i.e. two vertices v, u that have an (almost) maximal distance. Usually, an heuristic approach based on the construction of the *root level structure* of the graph is employed. The method then performs a visit, starting from v, exploring in sequence all vertices in the visit frontier Q that maximize the priority function:

$$P(v') \; = \; -W_1 \cdot \mathit{incr}(v') \; + \; W_2 \cdot \mathit{dist}(v, v')$$

where $\mathit{incr}(v')$ is the number of unexplored vertices adjacent to v', $\mathit{dist}(v, v')$ is the distance between the initial vertex v and v', and W_1 and W_2 are two integer weights. The weights control how Sloan prioritises the visit of the local cluster (W_1) and how much the selection should follow the gradient (W_2). Since the two weights control a linear combination of factors, in our analysis we shall consider only the ratio $\frac{W_1}{W_2}$. In Section 4 we will concentrate on the best selection of this ratio. We use the name `SLO`, `CM` and `KING` to refer to the Sloan, Cuthill-Mckee and King algorithms in the $\mathcal{A}_{\text{Gen}}$ set. GreatSPN uses the Boost-C++ implementations of these methods.

2.3 P-semiflows chaining algorithm

In this subsection we propose a new heuristic algorithm exploiting the Π_{PSF} set of structural units obtained by the P-semiflows computation. A P-semiflow is a positive, integer, left annuler of the incidence matrix of a Petri net, and it is known that, in any reachable marking, the sum of tokens in the net places, weighted by the P-semi-flow coefficients, is constant and equal to the weighted sum of the initial marking (P-invariant). Its main idea is to maintain the places shared between two Π_{PSF} units (i.e. P-semiflows) as close as possible in the final MDD variable ordering, since their markings cannot vary arbitrarily. The pseudo-code is reported in Algorithm 3. The algorithm takes as input the Π_{PSF} set and returns as output a variable ordering (stored in the ordered l). Initially, the π_i unit sharing the highest number of places with another unit is removed by Π_{PSF} and saved in π_{curr}. All its places are added to l.

Then the main loop runs until Π_{PSF} becomes empty. The loop comprises the following operations. The π_j unit sharing the highest number of places with π_{curr} is selected. All the places of π_j in l, which are not currently C (i.e. the list of currently discovered common places) are removed. The common places between π_i and π_j not present in C are appended to l. Then the places present only in π_j are added to l. After these steps, C is updated with the common places in π_i and π_j, and π_j is removed by Π_{PSF}. Finally π_{curr} becomes π_j, completing the iteration. This algorithm is named `P` and belongs to the $\mathcal{A}_{\text{PSF}}$ set.

Algorithm 3 Pseudocode of the P-semiflows chaining algorithm.

```
Function P-semiflows($\Pi_{\mathrm{PSF}}$):
    $l = \varnothing$ is the ordered list of places.
    $C = \varnothing$ is the set of current discovered common places.
    Select a unit $\pi_i \in \Pi_{\mathrm{PSF}}$ s.t. $\max_{\{i,j\} \in |\Pi_{\mathrm{PSF}}|} \pi_i \cap \pi_j$ with $i \neq j$
    $\Pi_{\mathrm{PSF}} = \Pi_{\mathrm{PSF}} \setminus \{\pi_i\}$
    $\pi_{curr} = \pi_i$
    Append $V(\pi_{curr})$ to $l$
    repeat until $\Pi_{\mathrm{PSF}}$ is empty:
        Select a unit $\pi_j \in \Pi_{\mathrm{PSF}}$ s.t. $\max_{j \in |\Pi_{\mathrm{PSF}}|} \pi_{curr} \cap \pi_j$
        Remove $(l \cap V(\pi_j)) \setminus C$ to $l$
        Append $V(\pi_{curr} \cap \pi_j) \setminus C$ to $l$
        Append $V(\pi_j) \setminus (C \cap V(\pi_{curr}))$ to $l$
        Add $V(\pi_{curr} \cap \pi_j)$ to $C$
        $\pi_{curr} = \pi_j$
        $\Pi_{\mathrm{PSF}} = \Pi_{\mathrm{PSF}} \setminus \{\pi_j\}$
    return $l$
```

2.4 The Noack and the Tovchigrechko greedy heuristics algorithms

The Noack [24] and the Tovchigrechko [28] methods are greedy heuristics that build up the variable order sequence by picking, at every iteration, the variable that minimizes an objective function. A detailed description can be found in [18]. A pseudo-code is given in Algorithm 4.

Algorithm 4 Pseudocode of the Noack/Tovchigrechko heuristics.

```
Function NOACK-TOV:
    $S = \varnothing$ is the set of already selected places.
    for $i$ from 1 to $|V|$:
        compute weight $W(v) = f(v, S)$ for each $v \notin S$.
        find $v$ that maximizes $W(v)$.
        $l(i) = v$.
        $S \leftarrow S \cup \{v\}$.
    return the variable order $l$.
```

The main difference between the Noack and the Tovchigrechko methods is the weight function $f(v, S)$, defined as:

$$f_{\mathrm{Noack}}(v, S) = \sum_{\substack{e \in E^{\mathrm{out}}(v) \\ k_1(e) \wedge k_2(e)}} \big(g_1(e) + z_1(e)\big) \quad + \sum_{\substack{e \in E^{\mathrm{in}}(v) \\ k_1(e) \wedge k_2(e)}} \big(g_2(e) + c_2(e)\big)$$

$$f_{\mathrm{Tov}}(v, S) = \sum_{\substack{e \in E^{\mathrm{out}}(v) \\ k_1(e)}} g_1(e) \quad + \sum_{\substack{e \in E^{\mathrm{out}}(v) \\ k_2(e)}} c_1(e) \quad + \sum_{\substack{e \in E^{\mathrm{in}}(v) \\ k_1(e)}} g_2(e) \quad + \sum_{\substack{e \in E^{\mathrm{in}}(v) \\ k_2(e)}} c_2(e)$$

where the sub-terms are defined as:

$$g_1(e) = \frac{\max\big(0.1,\ |S\cap V^{\text{in}}(e)|\big)}{|V^{\text{in}}(e)|}, \qquad g_2(e) = \frac{1+|S\cap V^{\text{in}}(e)|}{|V^{\text{in}}(e)|}$$
$$c_1(e) = \frac{\max\big(0.1,\ 2\cdot|S\cap V^{\text{out}}(e)|\big)}{|V^{\text{out}}(e)|}, \qquad c_2(e) = \frac{\max\big(0.2,\ 2\cdot|S\cap V^{\text{out}}(e)|\big)}{|V^{\text{out}}(e)|}$$
$$z_1(e) = \frac{2\cdot|S\cap V^{\text{out}}(e)|}{|V^{\text{out}}(e)|}, \quad k_1(e) = |V^{\text{in}}(e)| > 0, \quad k_2(e) = |V^{\text{out}}(e)| > 0$$

Few technical information is known about the criteria that were followed for the definition of the f_{Noack} and f_{Tov} functions. An important characteristic is that both functions have different criteria for input and output event conditions, i.e. they do not work on the symmetrized event relation, like the Sloan method.

The set $\mathcal{A}$ of algorithms considered in the benchmark includes four variations of these two algorithms: the variants `NOACK` and `TOV` are the standard implementations, as described in [18]. In addition, we tested whether these methods could benefit from an additional weight function that favours structural units. The two methods `NOACK-NU` and `TOV-NU` employ this technique. In this case, function $f(v,S)$ is modified to add a weight of $0.75 \cdot |S \cap V(\Pi(v))|$ to variable v. In our experiments we tried various values for the weight, but we generally observed that these modified methods do not benefit from this additional information.

2.5 Markov Clustering heuristic

The heuristic `MCL` is based on the idea of exploring the effectiveness of clustering algorithms to improve variable order technique. The hypothesis is that in some models, it could be beneficial to first group places that belong to connected clusters. For our tests we selected the Markov Cluster algorithm [29]. The method works as a modified version of Sloan, were clusters are first ordered according to their average gradient, and then places belonging to the same clusters will appear consecutively on the variable ordering, following the cluster orders. This method is named `MCL` and belongs to the $\mathcal{A}_{\text{Gen}}$ set.

3 The Benchmark

The considered model instances are that of the Model Checking Contest, 2016 edition [16], which consists of 665 PNML models. We discarded 257 instances that our tool was not capable to solve in the imposed time and memory limits, because either the net was too big or the RS MDD was too large under any considered ordering. Thus, we considered for the benchmark the set $\mathcal{I}$ made of 386 instances, belonging to a set $\mathcal{M}$ of 62 models. These 386 instances run for the 14 tested algorithms for 75 minutes, with 16GB of memory and a decision diagram cache of 2^{26} entries. In the 386 instances of $\mathcal{I}$ two sub-groups are identified: The set $\mathcal{I}_{\text{PSF}} \subset \mathcal{I}$ of instances for which P-semiflows are available, with 328 instances generated from a subset $\mathcal{M}_{\text{PSF}}$ of 55 models; The set $\mathcal{I}_{\text{NU}} \subset \mathcal{I}$ of instances for which nested units are available, with 65 instances generated from a subset $\mathcal{M}_{\text{NU}}$ of 15 models.

The overall tests were performed on OCCAM [2] (Open Computing Cluster for Advanced data Manipulation), a multi-purpose flexible HPC cluster designed and maintained by a collaboration between the University of Torino and the Torino branch of the National Institute for Nuclear Physics. OCCAM counts slightly more than 1100 cpu cores including three different types of computing nodes: standard Xeon E5 dual-socket nodes, large Xeon E5 quad-sockets nodes with 768 GB RAM, and multi-GPU NVIDIA nodes. Our experiments took 221 hours of computation using 3 standard Xeon E5 dual-socket nodes.

Scores. Typically, the most important parameter that measures the performance of variable ordering is the MDD peak size, measured in nodes. The peak size represents the maximum MDD size reached during the computation, and it therefore represents the memory requirement. It is also directly related to the time cost of building the MDD. For these reasons we preferred to use the peak size alone rather than a weighted measure of time, memory and peak size, that would make the result interpretation more complex. The peak size is, however, a quantity that is strictly related to the model instance. Different instances will have unrelated peak sizes, often with different magnitudes. To treat instances in a balanced way, some form of normalized score is needed. We consider three different score functions: for all of them the score of an algorithm is first normalized against the other algorithms on the same instance, which gives a score per instance, and then summed over all instances. Let i be an instance, solved by algorithms $\mathcal{A} = \{a_1, \dots, a_m\}$ with peak nodes $P_i = \{p_{a_1}(i), \dots, p_{a_m}(i)\}$. The scores of an *algorithm a for an instance i* are:

- The *Mean Standard Score of instance i* is defined as: $\mathrm{MSS}_a(i) = \frac{p_a(i) - \mu_{\mathcal{A}}(i)}{\sigma_{\mathcal{A}}(i)}$, where $\mu_{\mathcal{A}}(i)$ and $\sigma_{\mathcal{A}}(i)$ are the mean and standard deviations for instance summed over the all algorithms that solve instance i.
- The *Normalized Score for instance i* is defined as: $\mathrm{NS}_a(i) = 1 - \frac{\min\{p \in P_i\}}{p_a(i)}$, which just rescales the peak nodes taking the minimum as the scaling factor.
- The *Model Checking Contest score*[1] *for instance i* defined as: $\mathrm{MCC}_a(i) = 48$ if a terminates on i in the given time bound, plus 24 if $p_a(i) = \min\{p \in P_i\}$.

The final scores used for ranking algorithms over a set $\mathcal{I}' \subseteq \mathcal{I}$ is then determined as the sum over $\mathcal{I}'$ of the scores per instance:

- The *Mean Standard Score of algorithm a*: $\mathrm{MSS}_a = \frac{1}{|\mathcal{I}'|} \sum_{i \in \mathcal{I}'} \mathrm{MSS}_a(i)$
- The *Normalized Score of algorithm a*: $\mathrm{NS}_a = \frac{1}{|\mathcal{I}'|} \sum_{i \in \mathcal{I}'} \mathrm{NS}_a(i)$
- The *Model Checking Contest score of a*: $\mathrm{MCC}_a = \frac{1}{|\mathcal{I}'|} \sum_{i \in \mathcal{I}'} \mathrm{MCC}_a(i)$

MSS requires a certain amount of samples to be significant. Therefore, we apply it only for those model instances were all our tested algorithms terminated in the time bound. For those instances where not all algorithm terminated, we apply NS. We decided to test the MCC score to check if it is a good or a biased metric, when compared to the standard score.

[1] We actually use a simplified version where answer correctness is not considered.

Table 1. Performance of the ordering algorithms using the MCC2016 models.

		Num. instances				Average scores			
Algorithm a	$\mathcal{A}$	applied	solved	optimal	uniq.	NS_a	MSS^*_a	NS^*_a	MCC_a
`FORCE-PTS`	$\mathcal{A}_{\text{Gen}}$	386	259	23	1	0.699	-0.071	0.552	33.63
`FORCE-NES`	$\mathcal{A}_{\text{Gen}}$	386	267	27	0	0.657	**-0.275**	0.505	34.88
`FORCE-WES1`	$\mathcal{A}_{\text{Gen}}$	386	263	23	0	0.663	-0.228	0.505	34.13
`CM`	$\mathcal{A}_{\text{Gen}}$	386	290	69	2	0.586	-0.076	0.449	40.35
`KING`	$\mathcal{A}_{\text{Gen}}$	386	284	36	2	0.635	-0.032	0.504	37.55
`SLO`	$\mathcal{A}_{\text{Gen}}$	386	**340**	71	**5**	0.534	-0.125	0.471	**46.69**
`NOACK`	$\mathcal{A}_{\text{Gen}}$	386	312	51	0	0.535	-0.052	**0.425**	41.96
`TOV`	$\mathcal{A}_{\text{Gen}}$	386	325	**75**	2	**0.524**	-0.043	0.435	45.07
`MCL`	$\mathcal{A}_{\text{Gen}}$	386	250	27	4	0.767	0.676	0.640	32.76
`FORCE-P`	$\mathcal{A}_{\text{PSF}}$	328	230	33	0	0.604	-0.292	0.435	36.07
`P`	$\mathcal{A}_{\text{PSF}}$	328	258	26	3	0.729	0.623	0.656	39.00
`FORCE-NU`	$\mathcal{A}_{\text{NU}}$	65	38	11	1	0.711	-0.364	0.506	31.01
`NOACK-NU`	$\mathcal{A}_{\text{NU}}$	65	47	3	0	0.777	0.049	0.692	34.70
`TOV-NU`	$\mathcal{A}_{\text{NU}}$	65	45	1	0	0.773	0.122	0.672	32.86

Table 1 describes the general summary of the benchmark results. For each algorithm, we report again its requirement class ($\mathcal{A}_{\text{Gen}}, \mathcal{A}_{\text{PSF}}, \mathcal{A}_{\text{NU}}$). The table reports the number of instances where: the algorithm is applied, the algorithm finishes in the time and memory bounds, the number of times the algorithm finds the best ordering among the others, and the number of times the algorithm is the only one capable of solving an instance. The last four columns report: the NS score on all instances NS_a; the MSS score on completed instances (MSS^*_a); the NS score on completed instances (NS^*_a); and the MCC_a score per instance. Remember that for NS and MSS, a lower score is better, while for MCC a higher score is better.

From the table, it emerges that `TOV` and `SLO` are the methods that perform better, with a clear margin. Other methods, like `FORCE-P` and `FORCE-NU` have the worst average behaviour, even if they perform well when they are capable of solving an instance. Considering the column of "optimal" instances, some methods seem to perform well, like `CM`, but as we shall see later in this section, this is a bias caused by the uneven number of instances per model (i.e. some models have more instances than others). To our surprise, both `MCL` and `P` methods show only a mediocre average performance even if there is a certain number of instances where they perform particularly well.

In general, most instances are solved by more than one algorithm. In the rest of the section we go into model details, first considering the performance of the algorithms, and then by ranking the methods considering either instances($\mathcal{I}$, $\mathcal{I}_{\text{PSF}}$, $\mathcal{I}_{\text{NU}}$) or models($\mathcal{M}$, $\mathcal{M}_{\text{PSF}}$, $\mathcal{M}_{\text{NU}}$).

Efficiency of variable ordering algorithms. Table 2 reports the times required to generate the variable ordering. The table reveals several problems and inef-

Table 2. Time distribution for the tested variable ordering methods, ordered by $\mathbb{E}[t]$.

Method Name	t<0.01	0.01<t<1	1<t<10	t>10	d.n.f.	$\mathbb{E}[t]$	$\sigma[t]$
`KING`	559	44	1	0	0	0.006	0.056
`CM`	558	45	1	0	0	0.006	0.059
`MCL`	522	80	2	0	22	0.016	0.129
`FORCE-NES`	371	220	13	0	4	0.090	0.423
`FORCE-WES1`	367	222	15	0	4	0.095	0.432
`FORCE-PTS`	368	221	15	0	4	0.098	0.442
`SLO`	510	87	5	2	0	0.211	3.602
`FORCE-NU`	409	131	38	26	0	0.708	2.235
`TOV`	434	142	21	7	0	0.854	9.092
`NOACK`	435	142	20	7	0	0.869	9.200
`NOACK-NU`	433	144	19	8	0	0.879	9.505
`TOV-NU`	430	146	20	8	0	0.927	9.849
`FORCE-P`	217	281	78	28	74	1.016	2.474
`P`	510	54	14	26	103	14.621	159.891

ficiencies of some of these methods in our implementation. In particular, our implementation of the Noack and Tovchigrechko methods are simple prototypes, and have at least the cost of $O(|V|^2)$. The `P` algorithm also shows a severe performance problem, with a cost that does not match its theoretical complexity. Surprisingly, the cost of Sloan is quite high. The library documentation states that its cost is $O(\log(m) \cdot |E|)$ with $m = \max\{\mathit{degree}(v) \mid v \in V\}$, but the numerical analysis seems to suggest that the library implementation could have some hidden inefficiency.

3.1 Results of the benchmark

Figure 2 shows the benchmark results, separated by instance class and algorithm class. The plots on the left (1, 3, and 5) report the results on the instances that are solved by all algorithms in the given time and memory limits ("Easy instances" hereafter), while those on the right report the results for all instances solved by at least one algorithm ("All instances" hereafter). In the left plots we report the three tested metrics, while on the right plots we discard the MSS, since the available samples for each instance may vary and could be too low for the Gaussian assumption. To fit all scores in a single plot we have rescaled the score values in the $[0, 1]$ range.

Algorithms are sorted by their NS score, best one on the left. The top row (plot 1 and 2) considers the $\mathcal{A}_{\text{Gen}}$ methods on all $\mathcal{I}$ instances. The center row considers the $\mathcal{A}_{\text{Gen}} \cup \mathcal{A}_{\text{PSF}}$ methods on the $\mathcal{I}_{\text{PSF}}$ instances. The bottom row considers the $\mathcal{A}_{\text{Gen}} \cup \mathcal{A}_{\text{NU}}$ methods on the $\mathcal{I}_{\text{NU}}$ instances. Plots 1, 3, and 5 consider 187, 145 and 11 instances, respectively, which are the "easy" instances.

All three scores seem to provide (almost) consistent rankings. The only exception is plot 5. We suspect that this discrepancy can be explained by the small number of available instances (i.e. 11). The MCC metric is not very significant

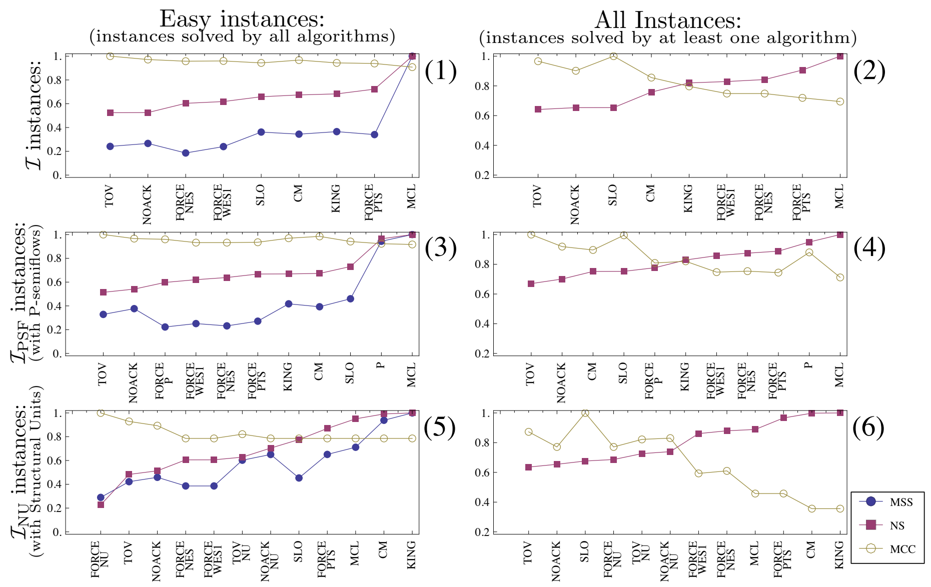


Fig. 2. Benchmark results, weighted by instance.

when we consider only the subset of instances solved by all algorithms, since it does not reward those methods that complete more often that the others. It is instead more significant for the right plots, which consider also the instances where algorithms do not finish.

In general, we can observe that FORCE methods seem to provide better variable orderings when the easy instances are considered (left plots). When we instead consider all the instances, FORCE methods appear to be less efficient. This suggests that FORCE methods could not scale well on larger instances.

We observe a marginally favorable behaviour of the two methods `FORCE-P` and `FORCE-NU` for the models with P-semiflows and structural units, compared to the standard FORCE method. The general trend that seems to emerge from these plots is that `TOV`, `NOACK` and `SLO` have the best average performance. This is particularly evident in plot 2, which refers to the behaviour on "all" instances.

One problem of this benchmark setup is that the MCC model set is made by multiple parametric instances of the same model, and the number of model instances vary largely (some models have just one, others have up to 40 instances). Usually, the relative performances of each algorithm on different instances of the same model are similar. Thus, an instance-oriented benchmark is biased toward those models that have more instances. Therefore, we consider a modified version of the three metrics MSS, NS and MCC, where the value is normalized against the number of instances $\mathcal{I}_m$ of each considered model $m \in \mathcal{M}'$. Therefore, MSS_a is redefined as: $\mathrm{MSS}_a = \frac{1}{|\mathcal{M}'|} \sum_{m \in \mathcal{M}'} \frac{1}{|\mathcal{I}_m|} \sum_{i \in \mathcal{I}_m} \mathrm{MSS}_a(i)$, and NS and MCC are modified analogously.

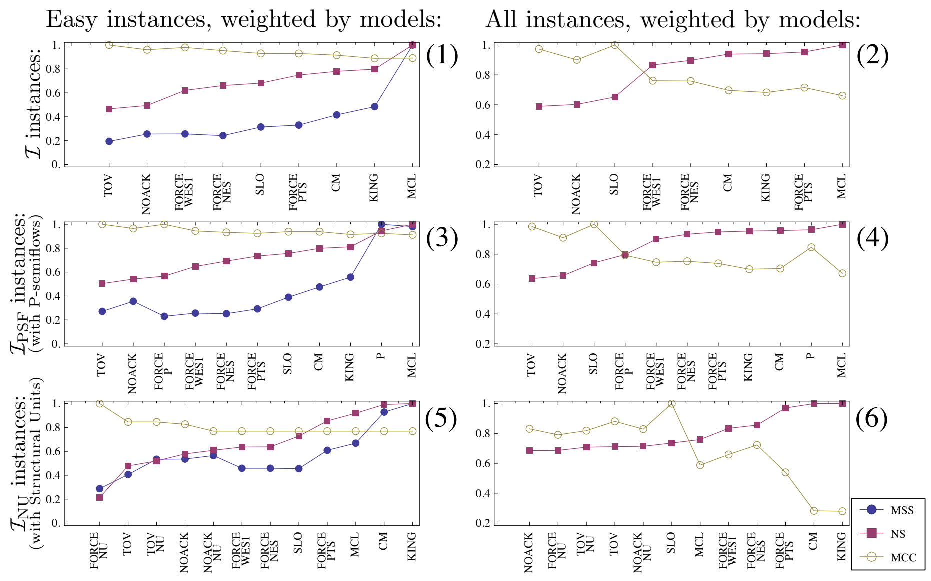


Fig. 3. Benchmark results, weighted by model.

Figure 3 shows the benchmark results weighted by models. Plots 1, 3, and 5 consider 48, 42 and 5 models, respectively, which are those models which have at least an "easy" instance. The main difference observed is the performance of the Cuthill-Mckee method, which on model-average does not seem to perform well. This is explained by the fact that `CM` performs well on three models (BridgeAndVehicle, Diffusion2D and SmallOperatingSystem) that have a large number of instances. But when we look from a model point-of-view, the performance of `CM` drops down.

As we have already stated before, both `MCL` and `P` methods are consistently the worst methods. In addition, neither `TOV-NU` nor `NOACK-NU` do not seem to take advantage of the prioritization on structural units, when compared to their base counterparts.

We may also observe that while `TOV` has the highest NS and MSS scores, `SLO` has the highest MCC score. To investigate this behaviour we look at the score distributions of the algorithms in Figure 4.

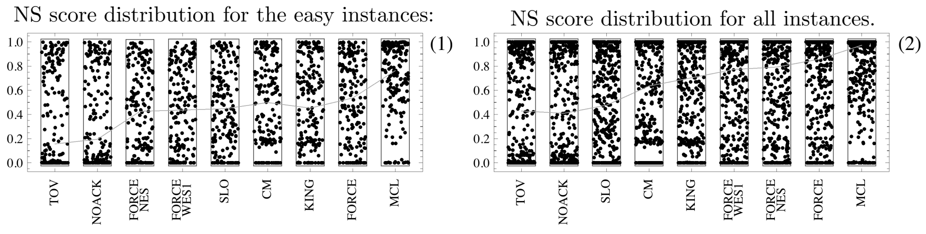


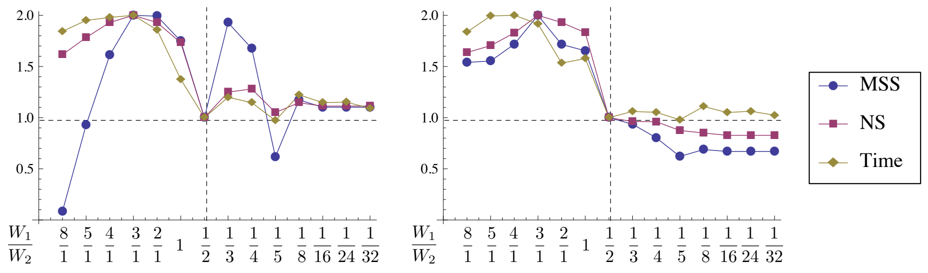
Fig. 4. Score distributions for the by Instance case.

Plot 1 and 2 of Figure 4 show the NS score distributions on the "easier" and "all" instances. The y coordinate of each dot represents the score obtained by an algorithm for an instance. Each box width is chosen large enough to observe the point density. In both plots `TOV` and `NOACK` have a behaviour that is more polarized than `SLO`, which means that usually either they produce an almost-optimal variable ordering, or they have a higher chance of failing at producing one. On the other hand, `SLO` seems to have a more balanced behaviour on most instances, as highlighted by the MCC score, even if it provides the best variable ordering fewer times than `TOV`. This happens because the MCC score is designed to give almost identical points to all algorithms that complete (except for the best algorithm, which takes extra points), regardless of the "quality" of the variable ordering. Therefore, the method that *on-average* performs better will have a higher MCC score.

4 Parameter Estimation of the Sloan priority function weights

Since Sloan seems to provide very good variable orders on average, we have investigated whether the parametric priority function $P(v')$ can be tuned, since the implementation of Sloan available in the library has been developed for a different purpose than ordering MDD variables.

Figure 5 shows the MSS, NS and Time lines of the Sloan algorithm applied to 271 instances. The results have been obtained considering the subset of $\mathcal{I}$ that terminate in less than 10 minutes. These instances belong to 56 different models. The lines are rescaled, and centered around the $W_1 = 1, W_2 = 2$ point, which are the standard coefficients of the priority function.



(**A**) Scores per instances. (**B**) Scores per models.

Fig. 5. Performance trends of Sloan method for different weight values.

We experiment a parameter space of the $\frac{W_1}{W_2}$ ratio in the range going from $\frac{8}{1}$ to $\frac{1}{32}$. The figure shows that the ratio of $\frac{1}{2}$, which is the standard value for the algorithm as implemented in the libraries, is not necessarily the best choice on average. As before, the per-instance plots shows slightly different results than the per-model plot. The per-model plot shows that, on average, a higher value of W_2 is beneficial to find a better variable ordering.

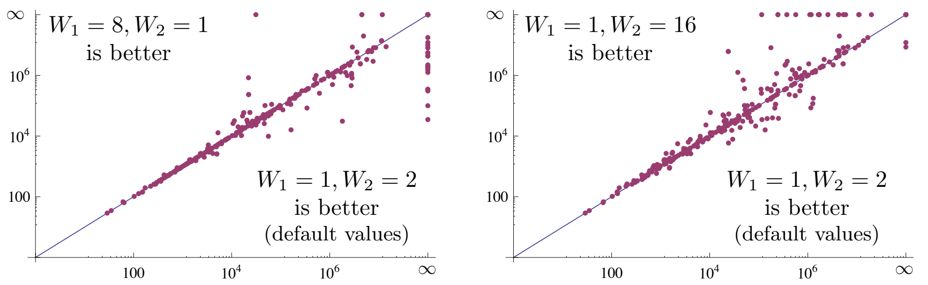


Fig. 6. Comparison of 3 variations of the weights in the Sloan priority function $P(v')$.

We now consider the $\frac{8}{1}$, $\frac{1}{2}$ and $\frac{1}{16}$ versions for a more detailed analysis on the full set of 386 instances with a time limit of 75 minutes, depicted in Figure 6. Each point represents an instance, where its x and y coordinates are the MDD peak values computed using the compared parameter variations. Coordinates are represented on a log-log scale. Instances that did not finish are considered as having an infinite peak node value, and are represented on the top (resp. right) sides of the plot. Most instances stay close to the diagonal, which means that there is almost no difference between the two parametric variations on that instance. Some instance instead show different MDD peaks, and are mainly concentrated toward the variations with $W_2 > W_1$. We shall now refer to the parametric variation of Sloan with $W_1 = 1, W_2 = 16$ as `SLO:1/16`.

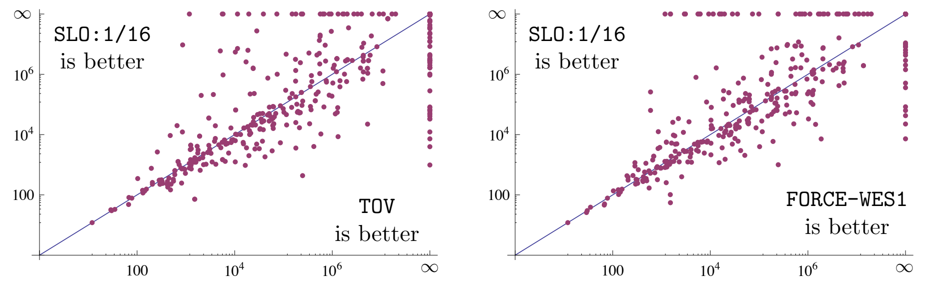


Fig. 7. Comparison of Sloan against Tovchigrechko and FORCE-WES1 methods.

Figure 7 shows the comparison of `SLO:1/16` against `TOV` and `FORCE-WES1`. Unlike the previous case, the plots show that many instances are far from the diagonal, which means that the three algorithms perform well on different instances.

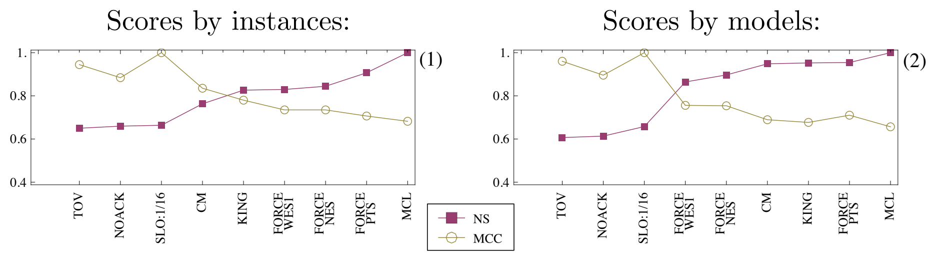


Fig. 8. Benchmark results after the addition of the optimized Sloan method.

Finally, Figure 8 shows the benchmark results after the substitution of `SLO` with `SLO:1/16`. The plots report both the NS and MCC metrics by instances (left) and by models (right) on "All" instances. `SLO:1/16` performs better than `SLO` in 71 instances (∼13 models) with a MDD peak reduction of 37% (24% for models), while `SLO` beats `SLO:1/16` in 66 instances (∼18 models) with a peak reduction of 56% (20% for models). For 200 instances (∼28 models) the variable ordering remains unchanged. Of the 386 tested models, `SLO` solves 340 of them, and in 71 cases it provides the best ordering among the other algorithms. Instead, `SLO:1/16` solves 341 instances, being optimal in 81 cases. These data show the favorable performance of `SLO:1/16`. This also confirms the observations of Figure 6, i.e. the set of models where Sloan works well remains almost unaltered, but `SLO:1/16` has a higher chance of finding a better ordering.

5 Conclusions and Future works

In this paper we presented a comparative benchmark of 14 variable ordering algorithms. Some of these algorithms are popular among Petri net based model checkers, while others have been defined to investigate the use of structural information for variable orderings. We observed that the Tovchigrechko/Noack methods and the Sloan method have the best performances, and their effectiveness cover different subsets of model instances. While the methods of Tovchigrechko/Noack were designed for Petri net models, the method of Sloan is a standard algorithm for bandwidth reduction of matrices, whose effectiveness for variable ordering was pointed out only recently in [19] and [22]. We observed that Sloan for variable ordering generation can be improved by changing the priority function weights. The parameter estimation on Sloan seems to suggest that the gradient (controlled by W_2) has a higher impact than keeping a low matrix profile (controlled by W_1). We conjecture that other algorithms, like `TOV`, `FORCE` or `P`, could be improved by using a super-imposed gradient. We did not observe significant improvements for those algorithms that use structural information of the net, and in some cases (`TOV` and `NOACK`) we even observed a degradation in performance. It is therefore unclear whether and how this type of structural information can be used to improve variable orderings. We also tested three different metric scores. We observed an agreement between the MSS and the NS score, which is nice since NS can be used even when few algorithms complete.

We also observed that MCC is a good score, that favours a different aspect than MSS/NS (average behaviour over better ordering). The use of a per-model weight on the scores has helped in identifying a bias in the benchmark results. We think that some form of per-model weight is necessary when using the MCC model set.

It should be noted that we observed a different ranking than the one reported in [22]. That paper deals with metrics for variable ordering (without RS construction), but in the last section the authors report a small experimental assessment similar to our benchmark. In that assessment Tovchigrechko was not tested, and the best algorithms were mostly Sloan and FORCE. For Sloan, they used the default parameter setting with a rather different symmetrization of the adjacency matrix, while it is not clear what variation of FORCE was used. This, together with the fact that the tested model set was different (106 instances), makes it difficult to draw any definite conclusions.

Acknowledgement.

We would like to thank the MCC team and all colleagues that collaborated with them for the construction of the MCC database of models, and the Meddly library developers.

References

1. Ajmone Marsan, M., Conte, G., Balbo, G.: A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. ACM Transactions on Computer Systems 2, 93–122 (May 1984)
2. Aldinucci, M., Bagnasco, S., Lusso, S., Pasteris, P., Vallero, S., Rabellino, S.: The Open Computing Cluster for Advanced data Manipulation (OCCAM). In: 22nd Int. Conf. on Computing in High Energy and Nuclear Physics. San Francisco (2016)
3. Aloul, F.A., Markov, I.L., Sakallah, K.A.: FORCE: A fast and easy-to-implement variable-ordering heuristic. In: Proc. of GLSVLSI. pp. 116–119. ACM, NY (2003)
4. Amparore, E.G.: Reengineering the editor of the greatspn framework. In: PNSE@ Petri Nets. pp. 153–170 (2015)
5. Amparore, E.G., Balbo, G., Beccuti, M., Donatelli, S., Franceschinis, G.: 30 Years of GreatSPN, chap. In: Principles of Performance and Reliability Modeling and Evaluation: Essays in Honor of Kishor Trivedi, pp. 227–254. Springer, Cham (2016)
6. Amparore, E.G., Beccuti, M., Donatelli, S.: (stochastic) model checking in GreatSPN. In: Ciardo, G., Kindler, E. (eds.) 35th Int. Conf. Application and Theory of Petri Nets and Concurrency, Tunis. pp. 354–363. Springer, Cham (2014)
7. Baarir, S., Beccuti, M., Cerotti, D., Pierro, M.D., Donatelli, S., Franceschinis, G.: The GreatSPN tool: recent enhancements. Performance Eval. 36(4), 4–9 (2009)
8. Babar, J., Miner, A.: Meddly: Multi-terminal and edge-valued decision diagram library. In: Quantitative Evaluation of Systems, International Conference on. pp. 195–196. IEEE Computer Society, Los Alamitos, CA, USA (2010)
9. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. 45(9), 993–1002 (Sep 1996)

10. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35, 677–691 (August 1986)
11. Burch, J.R., Clarke, E.M., Long, D.E.: Symbolic model checking with partitioned transition relations. In: IFIP TC10/WG 10.5 Very Large Scale Integration. pp. 49–58. North-Holland (1991)
12. Ciardo, G., L uttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state-space generation. In: TACAS'01. pp. 328–342 (2001)
13. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: In Proc. of TACAS 2003. pp. 379–393. LNCS 2619, Springer (apr 2003)
14. Ciardo, G., Yu, A.J.: Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In: Proc. CHARME. pp. 146–161. LNCS 3725, Springer (2005)
15. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proc. of the 1969 24th National Conference. pp. 157–172. ACM, New York (1969)
16. F. Kordon et all: Complete Results for the 2016 Edition of the Model Checking Contest. `http://mcc.lip6.fr/2016/results.php` (June 2016)
17. Garavel, H.: Nested-Unit Petri Nets: A structural means to increase efficiency and scalability of verification on elementary nets. In: 36th Int. Conf. Application and Theory of Petri Nets, Brussels. pp. 179–199. Springer, Cham (2015)
18. Heiner, M., Rohr, C., Schwarick, M., Tovchigrechko, A.A.: MARCIE's secrets of efficient model checking. In: Transactions on Petri Nets and Other Models of Concurrency XI. pp. 286–296. Springer, Heidelberg (2016)
19. Kamp, E.: Bandwidth, profile and wavefront reduction for static variable ordering in symbolic model checking. Tech. rep., University of Twente (June, 2015)
20. King, I.P.: An automatic reordering scheme for simultaneous equations derived from network systems. Journal of Numerical Methods in Eng. 2(4), 523–533 (1970)
21. Kumfert, G., Pothen, A.: Two improved algorithms for envelope and wavefront reduction. BIT Numerical Mathematics 37(3), 559–590 (1997)
22. Meijer, J., van de Pol, J.: Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In: bth NASA Formal Methods, 2016, Minneapolis. pp. 255–271. Springer, Cham (2016)
23. Miner, A.S.: Implicit GSPN reachability set generation using decision diagrams. Performance Evaluation 56(1-4), 145–165 (mar 2004)
24. Noack, A.: A ZBDD package for efficient model checking of Petri nets (in German). Ph.D. thesis, BTU Cottbus, Department of CS (1999)
25. Rice, M., Kulhari, S.: A survey of static variable ordering heuristics for efficient BDD/MDD construction. Tech. rep., University of California (2008)
26. Siminiceanu, R.I., Ciardo, G.: New metrics for static variable ordering in decision diagrams. In: 12th Int. Conf. TACAS 2006. pp. 90–104. Springer, Heidelberg (2006)
27. Sloan, S.W.: An algorithm for profile and wavefront reduction of sparse matrices. International Journal for Numerical Methods in Engineering 23(2), 239–251 (1986)
28. Tovchigrechko, A.: Model checking using interval decision diagrams. Ph.D. thesis, BTU Cottbus, Department of CS (2008)
29. Van Dongen, S.: A cluster algorithm for graphs. Inform. systems 10, 1–40 (2000)