Decision diagrams for Petri nets:
a comparison of variable ordering algorithms

Decision diagrams for Petri nets: a comparison of variable ordering algorithms

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

[1] Università di Torino, Dipartimento di Informatica
{amparore,susi,beccuti}@di.unito.it
[2] Iowa State University
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 performance information is available. In the work reported in this paper we used the models of the Model Checking Contest 2017 edition to conduct an extensive comparison of 18 different algorithms, in order to better understand their efficacy. Comparison is based on the size of the decision diagram of the reachable state space, which is generated using the Saturation method provided by the Meddly library.

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

1 Introduction

A binary decision diagram (BDD) [12] is a well-known data structure that has been 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 various extensions (e.g. Multi-way Decision Diagrams, or MDDs) 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 intermediate and final sizes of DD representations are known to be strongly dependent on the choice of variable order: 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 an optimal variable ordering is known to be NP-complete [11]. 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 [28], and more recently in [21]. In particular the latter work considers a new set of variable ordering algorithms, based on Bandwidth-reduction methods [29], and observes that they can be successfully applied to variable ordering. We also consider the work published in [20] (based on the ideas in [30]), 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 of 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. Develop metrics to compare variable ordering algorithms in the most fair manner.

To achieve these objectives we have built a benchmark in which 18 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), 2017 edition [23]. The implementation is part of RGMEDD [6], the model-checker of GreatSPN [5], and uses MDD saturation [14]. 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 workflow we have followed in the benchmark.

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 performance. 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 represent algorithms, and rows represent *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

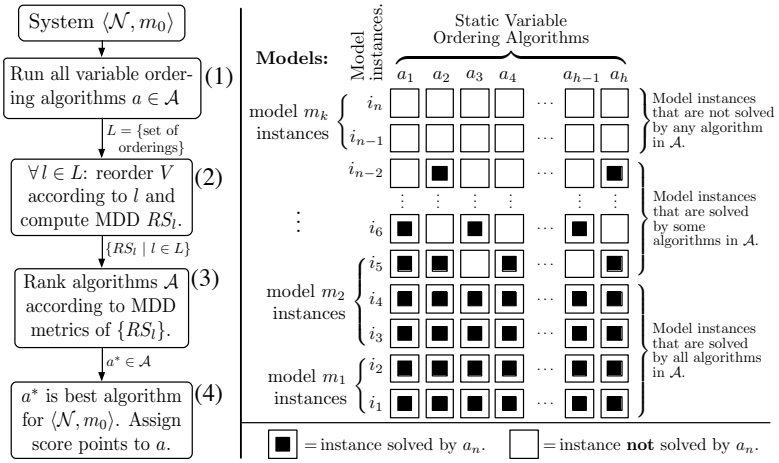

Fig. 1. Workflow for analysis and testing of static variable ordering algorithms.

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 [23] is a yearly scientific event whose aim is to provide a comparison among the different available verification tools. The 2017 edition employed a set of 817 PNML instances generated from 75 (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) [10] 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 [14] 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 [15], although this is currently a prototype implementation. The transition relation may be specified as a single monolithic relation that is then automatically split [26], or as a relation partitioned by levels or by events [16], 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 [9] 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); and Section 4 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 category 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 variable 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 the model is composed of smaller parts. We mainly refer to [18] 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 the set of such units. We will discuss clustering in Section 2.6.

Following these criteria, we subdivide the set of algorithms $\mathcal{A}$ into $\mathcal{A}_{\text{Gen}}$, the set of algorithms that do not use any structural information; $\mathcal{A}_{\text{PSF}}$, the set of algorithms that require Π_{PSF}; and $\mathcal{A}_{\text{NU}}$, the set of algorithms that require Π_{NU}. Since clustering can be computed on almost any model, we consider methods 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 Breadth-first and Depth-first orderings

Breadth-first and Depth-first search orderings (`BFS` and `DFS`) are two of the simplest possible variable ordering heuristics. They consist of a traversal of the net graph, starting from the first place, recording the visited places in breadth and depth order. These two methods usually show poor performance, but are included nonetheless in our tests since they are sometimes employed for BDD generation.

2.2 Force-based orderings

The Force heuristic, introduced in [3], is a n-dimensional graph layering technique based on the idea that variables form a 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. Events and variables are positioned over a real-valued line, and then sorted to get the ordering.

Algorithm 1 gives the general skeleton of the Force algorithm. The algorithm starts by shuffling the variable set, then it iterates trying to achieve a convergence of a metric. Usually, different initial orders produce different final orders, so

Algorithm 1 Pseudocode of the Force heuristic.

Function `Force`:
  Shuffle the variables randomly.
  repeat:
    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.
    Compute $PTS^{(i)} = \sum_{e \in E} \sum_{v \in e} \big| cog_e - p(v) \big|$.
  until series of $PTS^{(i)}$ values monotonically decreases.
  return the variable order that had the smallest $PTS^{(i)}$ value.

Force can be seen as a *factory* of variable orders. In addition, starting from a non-random initial order usually produces a better order. The metric is the total distance between the transition points and the variable points, known as *Point-Transition Spans* (PTS).

Structural information of the model can be used to establish additional *centers of gravity*. We tested the following three variations of the Force method:

- `Force`: Events are used as centers of gravity, as described in Algorithm 1.
- `Force-P`: P-semiflows are used as centers of gravity, along with the Petri net events. The method is tested only for those models that have P-semiflows.
- `Force-NU`: Structural units are used as centers of gravity, along with the events. This intuitively tries to keep together those variables that belong to the same structural unit. Again, this variation can be used only for those models that have Nested Units.

The set $\mathcal{A}$ of algorithms considered in the benchmark includes: `Force` in $\mathcal{A}_{\text{Gen}}$; the method `Force-P` in $\mathcal{A}_{\text{PSF}}$; and the method `Force-NU` in $\mathcal{A}_{\text{NU}}$, for a total of three variations of this method.

2.3 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 [13] 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 [21] and in [25] on the model bipartite graph. In these works the considered BR methods are:

- `CM`, `CM2` and `ACM`: The Reverse Cuthill-Mckee [17]. We test three implementations of this method: The one implemented in the Boost-C++ library (CM), the one implemented in the ViennaCL library (CM2), and the Advanced Cuthill-Mckee (ACM);

– `KING`: The King algorithm [22];
– `SLO` and `SLO-16`: The Sloan algorithm [29], with two parametric variations. The first version uses $W_1 = 1$ and $W_2 = 2$ (default values), while the second version uses $W_1 = 1$, $W_2 = 16$. Further information on these two variants can be found in [7].
– `GPS`: the Gibbs-Poole-Stockmeyer algorithm [19].

The choice was motivated by their ready availability in the Boost-C++ and ViennaCL libraries. In particular, Sloan, which is the state-of-the-art method for bandwidth reduction, showed promising performance as a variable ordering method. Sloan almost always outperforms [21] the other BR methods, but for completeness of our benchmark we have decided to test all of them. 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 condense the entries of a symmetric square matrix $\mathbf{A}$ around its diagonal, thus reducing the matrix *bandwidth* and *profile* [29]. 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 [25] 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 $|V^{\text{in}}(e)| \times |V^{\text{out}}(e)| < T$, where T is a threshold value, all entries in the cross product $V^{\text{in}}(e) \times V^{\text{out}}(e)$ are set to nonzero 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 a highly connected event could generate a dense matrix $\mathbf{A}$. In our implementation, the threshold T is set to 100. The matrix is finally made symmetric using: $\mathbf{A}' = \mathbf{A} + \mathbf{A}^{\text{T}}$. 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 traversal.
  Establish a gradient from 0 in v to d in u using a breadth-first traversal.
  Initialize traversal 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 [24]. The method follows two phases. In the first phase it determines a pseudo-diameter of the $\mathbf{A}$ matrix graph, i.e. two vertices v, u that have an (almost) maximal distance. Usually, a heuristic approach based on the construction of the *root level structure* of the graph is employed. The method then performs a traversal, starting from v, exploring in sequence all vertices in the traversal frontier Q that maximize the priority function:

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

where $incr(v')$ is the number of unexplored vertices adjacent to v', $dist(v, v')$ 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 prioritizes the traversal 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 $\frac{W_1}{W_2}$. Two ratios are tested: $\frac{W_1}{W_2} = \frac{1}{2}$, named `SLO`, and $\frac{W_1}{W_2} = \frac{1}{16}$, named `SLO-16`. An analysis of the parametric variations of Sloan for variable ordering selection can be found in [7].

2.4 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 in 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.

2.5 The Noack and the Tovchigrechko greedy heuristics algorithms

The Noack [27] and the Tovchigrechko [30] methods are greedy heuristics that build up the variable order sequence by picking, at every iteration, the variable

Algorithm 3 Pseudocode of the P-semiflows chaining algorithm.

Function `P-chaining(`Π_{PSF}`)`:

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

that minimizes an objective function. A detailed description can be found in [20]. 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:

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

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

where the sub-terms are defined as:

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

Not much 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 Noack and Tovchigrechko heuristics will be called `NOACK` and `TOV` in the benchmark.

2.6 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 [31]. The method works as a modified version of Sloan, where clusters are first ordered according to their average gradient, and then places belonging to the same cluster 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.

2.7 Gradient-Π ordering

The Gradient-Π heuristic is a new heuristic that mixes a set of structural information Π with a gradient-like approach similar to the Sloan method. A detailed description can be found in [8]. We tested two variations of this method:

- `Grad-P`: the set Π is the set Π_{PSF} of P-semiflows of the net.
- `Grad-NU`: the set Π is the set Π_{NU} of Nested Units of the net.

A pseudo-code is given in Algorithm 5.

Gradient-Π shares with the P-chaining method the idea of ordering the variables taking one invariant at a time. The main differences are that 1) the structural units are ordered according to a *score*(π) function that is based on the gradient, and 2) the variables inside each unit π are again ordered in gradient order.

3 The Benchmark

The considered model instances are that of the Model Checking Contest, 2017 edition [23], which consists of 817 PNML files. We discarded several instances

Algorithm 5 Pseudocode of the Gradient-Π heuristics.

Function `Gradient-`$\Pi(v_0, \Pi)$:
  Select v as the most-distant vertex to v_0 with a graph traversal.
  Establish a gradient from 0 in v to d in v_0 using a breadth-first traversal.
  $l \leftarrow \{\}$
  while exists at least one $\pi \in \Pi$ with $\pi \setminus S \neq \varnothing$:
    for each element $\pi \in \Pi$ with $\pi \setminus S \neq \varnothing$:
      Compute $score(\pi) = \sum_{v \in \pi \cap S} grad(v) \ - \sum_{v \in \pi \setminus S} grad(v)$
    Let $\pi_{\max}$ be the element with maximum $score(\pi)$ value.
    Append variables in $(\pi_{\max} \setminus S)$ to l in ascending gradient order.
    $S \leftarrow S \cup \pi_{\max}$.
  Append all variables in $(V \setminus S)$ to l in ascending gradient order.
  return l.

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 393 instances, belonging to a set $\mathcal{M}$ of 69 models. These 393 instances run for the 18 tested algorithms for 20 minutes, with 4GB of memory and a decision diagram cache of 2^{26} entries. In the 393 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 315 instances generated from a subset $\mathcal{M}_{\text{PSF}}$ of 62 models; The set $\mathcal{I}_{\text{NU}} \subset \mathcal{I}$ of instances for which nested units are available, with 109 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 contains 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.

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 instead of weighted measures of time, memory and peak size, that would make interpretation of the results more complex. The peak size is, however, a quantity that is strictly related to the model instance. Different instances of the same model 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 $\mathcal{A} = \{a_1, \ldots, a_m\}$ with peak nodes $P_i = \{p_{a_1}(i), \ldots, p_{a_m}(i)\}$. The scores of an *algorithm a for an instance i* are:

- The *Mean Standard Score of instance i* is defined as: $\mathrm{MSS}_a(i) = \frac{\mu_{\mathcal{A}}(i) - p_a(i)}{\sigma_{\mathcal{A}}(i)}$, where $\mu_{\mathcal{A}}(i)$ and $\sigma_{\mathcal{A}}(i)$ are the mean and standard deviations for instance I summed over all algorithms that solved instance i .
- The *Normalized Score for instance i* is defined as: $\mathrm{NS}_a(i) = \frac{\min\{p \in P_i\}}{p_a(i)}$, which just rescales the peak nodes taking the minimum as the scaling factor.
- The *Model Checking Contest score*[1] *for instance i* is 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 score 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. The set of instances where all algorithms finish is named *"Easy instances"* hereafter. We use MSS^*_a, NS^*_a and MCC^*_a to denote that the score is computed on the set of Easy instances only. If we instead consider all the instances, where some algorithms could not finish in time, we may apply only the NS and the MCC score. We decided to test the MCC score to check if it is a good or a biased metric, when compared to the standard score.

Score normalization: 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 per model vary largely. Some models have just one instance, while other models have up to 40 instances. Usually, the relative performance of each algorithm on different instances of the same model are similar, according to our experience. Thus, an instance-oriented benchmark is biased toward those models that have more instances. Therefore, we consider two benchmark settings for the computation of the scores:

- *Scores "By Instance"*: each model instance has the same weight, so those models with many instances will be more important. This reflects more closely the MCC score model.
- *Scores "By Model"*: each score 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:

$$\overline{\mathrm{MSS}}_a = \frac{1}{|\mathcal{M}'|} \sum_{m \in \mathcal{M}'} \frac{1}{|\mathcal{I}_m|} \sum_{i \in \mathcal{I}_m} \mathrm{MSS}_a(i)$$

  Analogously, $\overline{\mathrm{NS}}_a$ and $\overline{\mathrm{MCC}}_a$ are by-model versions of the NS and MCC scores.

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

In our opinion, the normalization of the by-model scores reflects more closely the idea of "average behaviour" of a variable ordering heuristic when applied to a new, unknown problem.

Assumptions: The computations are carried out by the GreatSPN model checker, which uses saturation by-events to generate the MDD representation of the state space. Since we consider MDD peak size for score computation, this is relevant, since other RS algorithms could produce different scores. We assume also that the model selection made for the MCC model set is somewhat "fair". In principle this is true, in the sense that the models have been selected using various criteria (variety, interest, case studies, ...) that have nothing to do with the performance of the saturation algorithm.

Table 1. Performance of the ordering algorithms using the MCC2017 models.

		By instances							**By models**						
Algorithms		Instances			Average scores				Models			Average scores			
Name	$\mathcal{A}$	N	solv.	best	NS_a	MSS^*_a	NS^*_a	MCC_a	N	solv.	best	$\overline{NS}_a$	$\overline{MSS^*_a}$	$\overline{NS^*_a}$	$\overline{MCC}_a$
`Force`	$\mathcal{A}_{\text{Gen}}$	393	289	41	0.37	-0.16	0.21	37.80	69	27.31	9.89	0.79	-0.39	0.44	41.34
`BFS`	$\mathcal{A}_{\text{Gen}}$	393	188	12	0.10	0.46	0.07	23.69	69	5.80	1.34	0.52	1.03	0.11	25.50
`DFS`	$\mathcal{A}_{\text{Gen}}$	393	187	3	0.08	0.41	0.07	23.02	69	5.11	0.62	0.50	0.76	0.12	24.34
`CM`	$\mathcal{A}_{\text{Gen}}$	393	263	42	0.31	-0.13	0.18	34.69	69	16.35	2.81	0.68	-0.25	0.31	33.79
`CM2`	$\mathcal{A}_{\text{Gen}}$	393	274	46	0.27	0.08	0.13	36.27	69	16.30	7.82	0.70	0.27	0.21	36.46
`ACM`	$\mathcal{A}_{\text{Gen}}$	393	275	46	0.27	0.08	0.13	36.40	69	16.25	7.82	0.70	0.27	0.21	36.49
`GPS`	$\mathcal{A}_{\text{Gen}}$	393	276	46	0.27	0.08	0.13	36.52	69	16.31	7.82	0.70	0.27	0.21	36.53
`KING`	$\mathcal{A}_{\text{Gen}}$	393	249	27	0.28	-0.13	0.18	32.06	69	15.43	2.16	0.66	-0.21	0.30	32.62
`SLO`	$\mathcal{A}_{\text{Gen}}$	393	327	36	0.39	-0.13	0.20	42.14	69	24.76	4.65	0.88	-0.32	0.37	43.99
`SLO-16`	$\mathcal{A}_{\text{Gen}}$	393	**331**	43	0.40	-0.12	0.19	43.05	69	26.24	8.31	0.89	-0.32	0.39	45.56
`NOACK`	$\mathcal{A}_{\text{Gen}}$	393	290	37	0.37	-0.12	0.21	37.68	69	29.18	6.87	0.77	-0.32	0.45	39.55
`TOV`	$\mathcal{A}_{\text{Gen}}$	393	293	46	0.38	-0.12	0.21	38.60	69	29.30	8.90	0.80	-0.32	0.46	41.59
`MCL`	$\mathcal{A}_{\text{Gen}}$	393	223	13	0.20	0.11	0.12	28.03	69	13.00	2.38	0.64	0.16	0.22	31.38
Algorithms that require P-semiflows:															
`P-chain`	$\mathcal{A}_{\text{PSF}}$	315	234	18	0.22	0.05	0.14	37.03	62	13.30	4.35	0.80	0.24	0.26	40.09
`GradP`	$\mathcal{A}_{\text{PSF}}$	315	260	**73**	**0.55**	-0.18	**0.25**	45.18	62	**32.98**	**13.97**	0.87	-0.40	**0.51**	47.27
`ForceP`	$\mathcal{A}_{\text{PSF}}$	315	255	35	0.39	**-0.19**	0.22	41.52	62	23.19	6.44	0.82	**-0.41**	0.43	42.01
Algorithms that require Nested Units:															
`GradNU`	$\mathcal{A}_{\text{NU}}$	109	92	39	0.66	-0.07	0.11	**49.10**	15	10.19	6.67	**0.94**	-0.21	0.34	**55.82**
`ForceNU`	$\mathcal{A}_{\text{NU}}$	109	85	1	0.23	-0.07	0.05	37.65	15	2.35	0.09	0.80	-0.18	0.12	38.45

Results: 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 average results for the "By instances" and "By Models" normalization. For the "By instances" group, it reports the number of instances where the algorithm is applied, the number of solved instances in the time and memory bounds, and the number of times the algorithm finds the best ordering

among the others. The next four columns report the NS score on all instances NS_a, the MSS score on the easy instances (MSS^*_a), the NS score on the easy instances (NS^*_a), and the MCC_a score on all instances. The structure is repeated for the "By model" normalization. First the number of applied models is reported, followed by the number of models solved by each algorithm (which can be fractional, since when an algorithm solves only some instances of a model, it gets a fractional score) and the number of best orders found. Finally, the scores are repeated, normalized by model.

From the table emerges that the Sloan algorithm has the best average performance on the MCC models, with a clear margin. It is then followed by the `TOV/NOACK` heuristics, and the `Force` heuristics. Methods that exploit structural information also show very positive results. Gradient-P (which is applied only to 315 instances out of 393) is particularly effective in finding the best variable orders most of the time. Similarly, Gradient-NU has again very positive scores.

Other methods, like `BFS` and `DFS` have usually a bad average behaviour, even if they perform well on a small subset of instances. Considering the column of "best" instances, some methods seem to perform well, like `CM`, but this is a bias caused by the uneven number of instances per model (i.e. some models have more instances than others). In fact, the behaviour of the `CM` algorithm when weighted by-models is much more modest. To our surprise, the `P-chaining` method shows only a mediocre average performance even though there are several instances where it performs 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($\mathcal{I}$, $\mathcal{I}_{\mathrm{PSF}}$, $\mathcal{I}_{\mathrm{NU}}$) or models($\mathcal{M}$, $\mathcal{M}_{\mathrm{PSF}}$, $\mathcal{M}_{\mathrm{NU}}$).

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 Easy instances, while those on the right report the results for all the instances. In the left plots we report the three tested metrics, while on the right plots we discard the MSS metric, 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 $\mathcal{A}_{\mathrm{Gen}}$ methods on all $\mathcal{I}$ instances. The center row considers the $\mathcal{A}_{\mathrm{Gen}} \cup \mathcal{A}_{\mathrm{PSF}}$ methods on the $\mathcal{I}_{\mathrm{PSF}}$ instances. The bottom row considers the $\mathcal{A}_{\mathrm{Gen}} \cup \mathcal{A}_{\mathrm{NU}}$ methods on the $\mathcal{I}_{\mathrm{NU}}$ instances. Plots 1, 3, and 5 consider 152, 122 and 15 instances, respectively, which are the "easy" instances. The Easy instances are in a certain sense a biased set, since instances are dropped (not easy) every time any algorithm fails in generating a reasonable variable order. However, from these plots it is possible to observe that the trend of the NS score is close to that of the MSS score. Therefore, we will mainly consider the NS score only, since it can be computed on the whole set of instances.

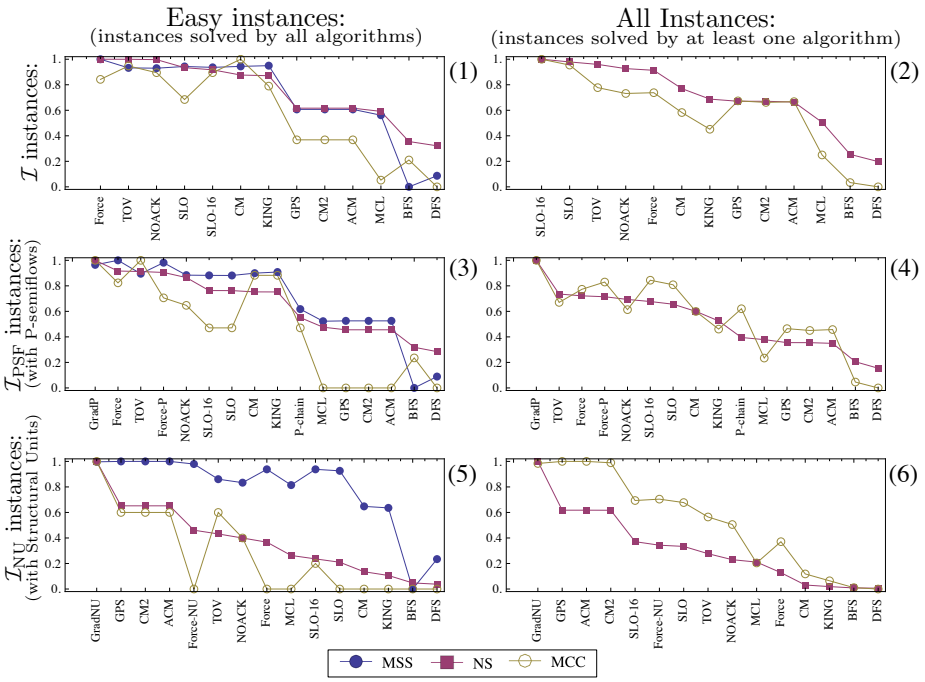

Fig. 2. Benchmark results, weighted by instance.

Plots 5 and 6 use fewer samples than the others, and the algorithm ranking is slightly different. We suspect that this discrepancy can be explained by the small number of available instances. Therefore, we mainly focus our attention on the other plots. The Sloan methods and the Gradient-Π methods appear to have the best performance, in terms of both the NS score and the MCC score. When we look at the average behaviour on all instances (plot 2), we may also observe that `TOV/NOACK` and `Force` have close-to-best performance. This shows that being capable of using the structural information of the model for the purpose of variable ordering can be an effective strategy.

Figure 3 shows the benchmark results weighted "By models". Plots 1, 3, and 5 consider 56, 50 and 6 models, respectively, which are those models which have at least one "easy" instance. Some methods have a different ranking due to the bias introduced by the uneven number of model instances. For example, the performance of the `CM` method appears to be worse when weighting "By models". 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 on average it does not produce very good variable orders. Again, the ranking has the Sloan method on top for the $\overline{\text{NS}}$ score on plots 2 and 4. The Gradient-Π methods show a more modest NS score, even if they are still on top of the rankings. However, the MCC score of these two methods are very high, suggesting their effectiveness (plot 4 and 6). Again, the

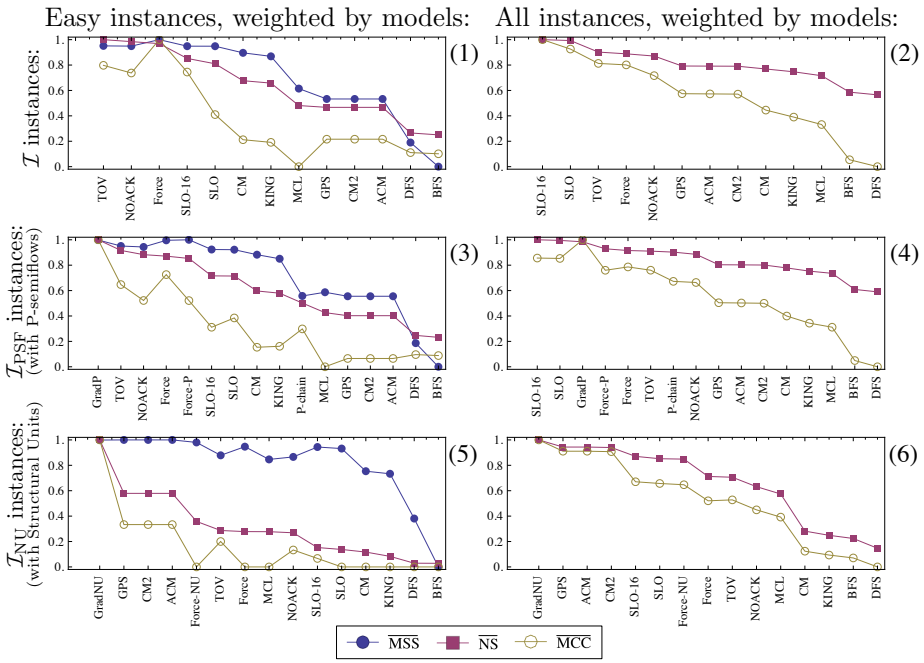

Fig. 3. Benchmark results, weighted by model.

number of Easy instances is very small and does not allow to make conclusions on the results of plots 1, 3 and 5.

As stated before, the ranking of the NS score is not the same as the MCC score. To better investigate this behaviour we look at the NS score distributions of the algorithms in Figure 4.

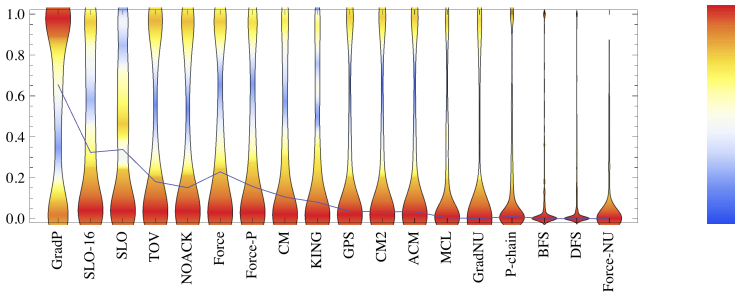

Fig. 4. NS score distributions for the "By Instance" case, on all models.

Figure 4 shows the NS score distributions on the 393 model instances. The bar of each algorithm a shows the distribution of the NS scores obtained by a for all the instances. From the distribution, it is clear that most algorithms

have very polarized behaviours (either they produce a very good order, or they fail). However, Sloan has a more continuous distribution, meaning that on many instances it is capable of producing a reasonable variable order, even if it is not the best among the generated ones. Therefore, a method that *on-average* performs reasonably well will have an high MCC score, like `SLO-16` in Fig. 2(2). Also, `Grad-P` has a very high chance of finding the best solution among the tested ones, showing that it is a very promising heuristic.

4 Conclusions and Future works

In this paper we presented a comparative benchmark of 18 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 among the generic methods, the Sloan method, the Tovchigrechko/Noack methods and the Force method have the best performance, and their effectiveness covers 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 [21] and [25]. We conjecture that a key ingredient of the Sloan method is the gradient. This conjecture has been used in [8] to design the new heuristic Gradient-Π, which is an hybrid between Sloan and P-chain. This heuristic proves to have very good performance. We conjecture that other algorithms, like `TOV` or `Force`, could be improved by using a superimposed gradient. When the net has some structural information, like P-semiflows or Nested Units, we observed that only some specialized algorithms could take a significant advantage from it. Surprisingly, the P-chaining method (one of the original heuristics of GreatSPN) showed poor performance when compared to more modern algorithms like Gradient-Π. However, the general results of other heuristics (Force-P, Gradient-Π) allow us to conclude that structural information is useful in deriving good variable orders. Of course, exploitation of structural information cannot be a general technique, since it is not available for all models.

We also tested three different scoring metrics. 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, i.e. MCC favours the average behaviour over finding 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 ranking similar to the one reported in [25]. 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. In addition, the tested model set was different (106 instances). However, the final observations drawn in that paper are close to the ones we get from our tests, confirming the effectiveness of the Sloan method for variable ordering.

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. Amparore, E.G., Donatelli, S., Beccuti, M., Garbi, G., Miner, A.: Decision diagrams for Petri nets: which variable ordering? In: Petri Net Performance Engineering conference (PNSE). pp. 31–50. CEUR-WS (2017)
8. Amparore, E.G., Susanna, D., Marco, B.: Gradient-based variable ordering of decision diagrams for systems with structural units. Automated Technology for Verification and Analysis (ATVA) (2017)
9. 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)
10. 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)
11. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. 45(9), 993–1002 (Sep 1996)
12. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35, 677–691 (August 1986)
13. 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)
14. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state-space generation. In: TACAS'01. pp. 328–342 (2001)

15. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: In Proc. of TACAS 2003. pp. 379–393. LNCS 2619, Springer (apr 2003)
16. 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)
17. 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)
18. 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)
19. Gibbs, N.E., Poole, Jr, W.G., Stockmeyer, P.K.: An algorithm for reducing the bandwidth and profile of a sparse matrix. SIAM Journal on Numerical Analysis 13(2), 236–250 (1976)
20. 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)
21. Kamp, E.: Bandwidth, profile and wavefront reduction for static variable ordering in symbolic model checking. Tech. rep., University of Twente (June, 2015)
22. 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)
23. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Berthomieu, B., Ciardo, G., Colange, M., Dal Zilio, S., Amparore, E., Beccuti, M., Liebke, T., Meijer, J., Miner, A., Rohr, C., Srba, J., Thierry-Mieg, Y., van de Pol, J., Wolf, K.: Complete Results for the 2017 Edition of the Model Checking Contest. http://mcc.lip6.fr/2017/results.php (June 2017)
24. Kumfert, G., Pothen, A.: Two improved algorithms for envelope and wavefront reduction. BIT Numerical Mathematics 37(3), 559–590 (1997)
25. Meijer, J., van de Pol, J.: Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In: 8th NASA Formal Methods, 2016, Minneapolis. pp. 255–271. Springer, Cham (2016)
26. Miner, A.S.: Implicit GSPN reachability set generation using decision diagrams. Performance Evaluation 56(1-4), 145–165 (mar 2004)
27. Noack, A.: A ZBDD package for efficient model checking of Petri nets (in German). Ph.D. thesis, BTU Cottbus, Department of CS (1999)
28. Rice, M., Kulhari, S.: A survey of static variable ordering heuristics for efficient BDD/MDD construction. Tech. rep., University of California (2008)
29. 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)
30. Tovchigrechko, A.: Model checking using interval decision diagrams. Ph.D. thesis, BTU Cottbus, Department of CS (2008)
31. Van Dongen, S.: A cluster algorithm for graphs. Inform. systems 10, 1–40 (2000)