

Improving Static Variable Orders via Invariants^{*}

Gianfranco Ciardo¹, Gerald Lüttgen², and Andy Jinqing Yu¹

¹ Dept. of Computer Science and Engineering, UC Riverside, CA 92521, USA
{ciardo, jqyu}@cs.ucr.edu

² Dept. of Computer Science, University of York, York YO10 5DD, U.K.
luettgen@cs.york.ac.uk

Abstract. Choosing a good variable order is crucial for making symbolic state-space generation algorithms truly efficient. One such algorithm is the MDD-based Saturation algorithm for Petri nets implemented in SMART, whose efficiency relies on exploiting event locality.

This paper presents a novel, static ordering heuristic that considers place invariants of Petri nets. In contrast to related work, we use the functional dependencies encoded by invariants to *merge* decision-diagram variables, rather than to eliminate them. We prove that merging variables always yields smaller MDDs and improves event locality, while eliminating variables may increase MDD sizes and break locality. Combining this idea of merging with heuristics for maximizing event locality, we obtain an algorithm for static variable order which outperforms competing approaches regarding both time-efficiency and memory-efficiency, as we demonstrate by extensive benchmarking.

1 Introduction

Petri nets [26] are a popular formalism for specifying concurrent and distributed systems, and much research [32] has been conducted in the automated analysis of a Petri net’s state space. Many analysis techniques rely on generating and exploring a net’s reachable markings, using algorithms based on *decision diagrams* [10, 29] or *place invariants* [17, 31, 34, 35].

While decision diagrams have allowed researchers to investigate real-world nets with thousands of places and transitions, their performance crucially depends on the underlying *variable order* [1, 23]. Unfortunately, finding a good variable order is known to be an NP-complete problem [2]. Thus, many heuristics for either the static or the dynamic ordering of variables have been proposed, which have shown varying degree of success; see [18] for a survey.

In the state-space exploration of Petri nets, place invariants find use in approximating state spaces [28], since every reachable state must by definition satisfy each invariant, and in compactly storing markings by exploiting *functional dependencies* [6, 19, 27]. This latter use of invariants is also considered when encoding places with decision-diagram variables, as it eliminates some variables, offering hope for smaller decision diagrams during state-space exploration [17].

^{*} Work supported in part by the National Science Foundation under grants CNS-0501747 and CNS-0501748 and by the EPSRC under grant GR/S86211/01.

The contributions of this paper are twofold. First, we show that *eliminating variables* based on invariance information may actually increase the sizes of decision diagrams, whence the above ‘hope’ is misplaced. Instead, we show that *merging variables* is guaranteed to lead to smaller decision diagrams. While our merging technique is obviously not applicable for *Binary* Decision Diagrams (BDDs), it is compatible with techniques using *Multi-way* Decision Diagrams (MDDs), such as SMART’s *Saturation algorithm* for computing reachable markings [10]. In addition, merging variables improves *event locality*, i.e., it decreases the span of events over MDD levels, rather than worsening it as is the case with variable elimination. This is important since algorithms like Saturation become more efficient as event locality is increased.

Second, we propose a new *heuristic* for static variable ordering which is suitable for Saturation. This heuristic combines previous ideas, which only took the height and span of events into account [39], with variable merging based on linear place invariants. We implement our heuristic into SMART [9], generating the invariants with GreatSPN [7], and show via extensive benchmarking that this heuristic outperforms approaches that ignore place invariants, with respect to both time-efficiency and memory-efficiency. Indeed, the benefits of our heuristic are greatest for practical nets, including large instances of the *slotted-ring network* [30] and the *kanban system* [40], which have been tractable only using ad-hoc variable orderings and mergings found through our intuition and extensive experimentation. This shows that exploiting invariants is key for optimizing the performance of symbolic state-exploration techniques, provided one uses invariance information for merging variables and not for eliminating them.

2 Preliminaries

In this section we briefly cover the class of Petri nets considered, *self-modifying nets*, and their two main analysis approaches, reachability and invariant analysis. Then, we discuss decision diagrams and how they can encode sets of markings and the transformations that transitions perform on markings. Finally, we survey a range of symbolic state-space generation algorithms, from the simple breadth-first iteration to our own Saturation algorithm.

2.1 Petri nets and self-modifying nets

We consider *self-modifying nets with inhibitor arcs*, described by a tuple of the form $(\mathcal{P}, \mathcal{T}, \mathbf{F}^-, \mathbf{F}^+, \mathbf{F}^\circ, \mathbf{m}^{init})$, where

- \mathcal{P} and \mathcal{T} are sets of *places* and *transitions* satisfying $\mathcal{P} \cap \mathcal{T} = \emptyset$ and $\mathcal{P} \cup \mathcal{T} \neq \emptyset$. A *marking* $\mathbf{m} \in \mathbb{N}^{\mathcal{P}}$ assigns a number of *tokens* \mathbf{m}_p to each place $p \in \mathcal{P}$.
- $\mathbf{F}^-: \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \rightarrow \mathbb{N}$, $\mathbf{F}^+: \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \rightarrow \mathbb{N}$, and $\mathbf{F}^\circ: \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \rightarrow \mathbb{N} \cup \{\infty\}$ are $|\mathcal{P}| \times |\mathcal{T}|$ *incidence matrices*; $\mathbf{F}_{p,t}^-$, $\mathbf{F}_{p,t}^+$, and $\mathbf{F}_{p,t}^\circ$ are the *marking-dependent* [8, 41] cardinalities of the *input*, *output*, and *inhibitor arcs* between p and t .
- \mathbf{m}^{init} is the *initial marking*.

The evolution of the net from a marking \mathbf{m} is governed by the following rules, keeping in mind that the cardinality of any arc is evaluated in the current marking, i.e., prior to the firing of any transition:

Enabling: Transition t is enabled in marking \mathbf{m} if, for each place p , the input arc is satisfied, $\mathbf{m}_p \geq \mathbf{F}_{p,t}^-(\mathbf{m})$, and the inhibitor arc is not, $\mathbf{m}_p < \mathbf{F}_{p,t}^o(\mathbf{m})$.

Firing: Firing enabled transition t in marking \mathbf{m} leads to marking \mathbf{n} , where, for each place p , $\mathbf{n}_p = \mathbf{m}_p - \mathbf{F}_{p,t}^-(\mathbf{m}) + \mathbf{F}_{p,t}^+(\mathbf{m})$. We write $\mathcal{N}_t(\mathbf{m}) = \{\mathbf{n}\}$, to stress that, for general discrete-state formalisms, the *next-state function* \mathcal{N}_t for event t , applied to a single state \mathbf{m} , returns a set of states. Then, we can write $\mathcal{N}_t(\mathbf{m}) = \emptyset$ to indicate that t is not enabled in marking \mathbf{m} .

2.2 Reachability analysis and invariant analysis

The two main techniques for Petri net analysis are *reachability analysis* and *invariant analysis*. The former builds and analyzes the *state space* of the net (or *reachability set*), defined as $\mathcal{M} = \{\mathbf{m} : \exists d, \mathbf{m} \in \mathcal{N}^d(\mathbf{m}^{init})\} = \mathcal{N}^*(\mathbf{m}^{init})$, where we extend the next-state function to arbitrary sets of markings $\mathcal{X} \subseteq \mathbb{N}^{\mathcal{P}}$, $\mathcal{N}_t(\mathcal{X}) = \bigcup_{\mathbf{m} \in \mathcal{X}} \mathcal{N}_t(\mathbf{m})$, write \mathcal{N} for the union of all next-state functions, $\mathcal{N}(\mathcal{X}) = \bigcup_{t \in \mathcal{T}} \mathcal{N}_t(\mathcal{X})$, and define multiple applications of the next-state function as usual, $\mathcal{N}^0(\mathcal{X}) = \mathcal{X}$, $\mathcal{N}^d(\mathcal{X}) = \mathcal{N}(\mathcal{N}^{d-1}(\mathcal{X}))$, and $\mathcal{N}^*(\mathcal{X}) = \bigcup_{d \in \mathbb{N}} \mathcal{N}^d(\mathcal{X})$.

Invariant analysis is instead concerned with deriving *a priori* relationships guaranteed to be satisfied by any reachable marking, based exclusively on the net structure. In nets where the arcs have a constant cardinality independent of the marking, i.e., ordinary Petri nets with or without inhibitor arcs [26], much work has focused on the computation of *p-semiflows* [14, 15], i.e., non-zero solutions $\mathbf{w} \in \mathbb{N}^{\mathcal{P}}$ to the linear set of “flow” equations $\mathbf{w} \cdot \mathbf{F} = 0$, where $\mathbf{F} = \mathbf{F}^+ - \mathbf{F}^-$. Since any linear combination of such solutions is also a solution, it suffices to consider a set of *minimal* p-semiflows from which all others can be derived through non-negative linear combinations. A semiflow \mathbf{w} specifies the constraint $\sum_{p \in \mathcal{P}} \mathbf{w}_p \cdot \mathbf{m}_p = C$ on any reachable marking \mathbf{m} , where the constant $C = \sum_{p \in \mathcal{P}} \mathbf{w}_p \cdot \mathbf{m}_p^{init}$ is determined by the initial marking. When marking-dependent arc multiplicities are present, linear p-semiflows [8], or even more general relationships [41], may still exist. However, invariant analysis provides necessary, not sufficient, conditions on reachability; a marking \mathbf{m} might satisfy all known invariants and still be unreachable.

In this paper, we use invariants to improve (symbolic) state-space generation. We assume to be given a self-modifying net with inhibitor arcs (or a similar discrete-state model whose next-state function is decomposed according to a set of asynchronous events), and a set \mathcal{W} of linear invariants, each of the form $\sum_{p \in \mathcal{P}} \mathbf{W}_{v,p} \cdot \mathbf{m}_p = C_v$, guaranteed to hold in any reachable marking \mathbf{m} . Then,

- $Support(v) = \{p \in \mathcal{P} : \mathbf{W}_{v,p} > 0\}$ is the *support* of the v^{th} invariant.
- $\mathbf{W} \in \mathbb{N}^{|\mathcal{W}| \times |\mathcal{P}|}$ describes the set of invariants. In addition, observe that the case $|Support(v)| = 1$ is degenerate, as it implies that the marking of the place p in the support is fixed. We then assume that p is removed from

the net after modifying it appropriately, i.e., substituting the constant \mathbf{m}_p^{init} for \mathbf{m}_p in the marking-dependent expression of any arc and removing any transition t with $\mathbf{F}_{p,t}^- > \mathbf{m}_p^{init}$ or $\mathbf{F}_{p,t}^o \leq \mathbf{m}_p^{init}$. Thus, each row of \mathbf{W} contains at least two positive entries.

- The marking of any one place $p \in \text{Support}(v)$ can be expressed as a function of the places in $\text{Support}(v) \setminus p$ through inversion, i.e., in every reachable marking \mathbf{m} , the relation $\mathbf{m}_p = (C_v - \sum_{q \in \mathcal{P} \setminus \{p\}} \mathbf{W}_{v,q} \cdot \mathbf{m}_q) / \mathbf{W}_{v,p}$ holds.

We say that a set of non-negative integer variables \mathcal{V}' is *functionally dependent* on a set of non-negative integer variables \mathcal{V}'' if, when the values of the variables in \mathcal{V}'' are known, the values of the variables in \mathcal{V}' are uniquely determined. In our linear Petri-net invariant setting, \mathcal{V}' and \mathcal{V}'' correspond to the markings of two sets of places \mathcal{P}' and \mathcal{P}'' , and functional dependence implies that the submatrix $\mathbf{W}_{\mathcal{W}', \mathcal{P}'}$ of \mathbf{W} , obtained by retaining only columns corresponding to places in \mathcal{P}' and rows corresponding to invariants having support in $\mathcal{P}' \cup \mathcal{P}''$, i.e., $\mathcal{W}' = \{v \in \mathcal{W} : \text{Support}(v) \subseteq \mathcal{P}' \cup \mathcal{P}''\}$, has rank $|\mathcal{P}'|$. This fundamental concept of functional dependence is at the heart of our treatment, and could be generalized to the case of nonlinear invariants where not every place in $\text{Support}(v)$ can be expressed as a function of the remaining places in the support. To keep presentation and notation simple, we do not discuss such invariants.

2.3 Decision diagrams

The state-space generation algorithms we consider use *quasi-reduced ordered multi-way decision diagrams* (MDDs) [22] to store *structured* sets, i.e., subsets of a *potential set* $\hat{\mathcal{S}} = \mathcal{S}_K \times \dots \times \mathcal{S}_1$, where each *local set* \mathcal{S}_l , for $K \geq l \geq 1$, is of the form $\{0, 1, \dots, n_l - 1\}$. Formally, an MDD over $\hat{\mathcal{S}}$ is a directed acyclic edge-labeled multi-graph such that:

- Each node p belongs to a *level* in $\{K, \dots, 1, 0\}$, denoted $p.lvl$.
- There is a single *root* node r^* at level K or 0 .
- Level 0 may only contain the *terminal* nodes 0 and 1 .
- A node p at level $l > 0$ has n_l outgoing edges, labeled from 0 to $n_l - 1$. The edge labeled by $i \in \mathcal{S}_l$ points to node q at level $l - 1$ or 0 ; we write $p[i] = q$.
- Given nodes p and q at level l , if $p[i] = q[i]$ for all $i \in \mathcal{S}_l$, then $p = q$.
- The edges of a node at level $l > 0$ cannot all point to 0 or all point to 1 .

An MDD node p at level l encodes, with respect to level $m \geq l$, the set of tuples $\mathcal{B}(m, p) = \mathcal{S}_m \times \dots \times \mathcal{S}_{l+1} \times (\bigcup_{i \in \mathcal{S}_l} \{i\} \times \mathcal{B}(l - 1, p[i]))$, letting $\mathcal{X} \times \mathcal{B}(0, 0) = \emptyset$ and $\mathcal{X} \times \mathcal{B}(0, 1) = \mathcal{X}$. If $m = l$, we write $\mathcal{B}(p)$ instead of $\mathcal{B}(l, p)$. Fig. 1 contains an example where $K = 4$, showing the composition of the sets \mathcal{S}_l (a), the MDD (b), and the set of tuples encoded by it (c). Here, as in [11], we employ a *dynamic* MDD variant where the sets \mathcal{S}_l are not fixed but are grown as needed, so that the MDD can be used to encode arbitrary (but finite) subsets of \mathbb{N}^K . The only overhead in such a data structure is that, since a set \mathcal{S}_l may grow and change the meaning of an edge spanning level l and pointing to node 1 , only edges pointing

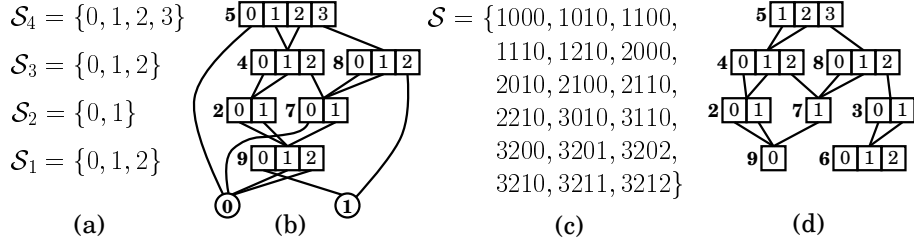


Fig. 1. An example of an MDD and the set of 4-tuples it encodes.

to node 0 are allowed to span multiple levels, while node 1 can be pointed only by edges from nodes at level 1. Fig. 1(d) shows how this requires the insertions of nodes **3** and **6** along edge 2 from **8**; we employ a simplified representation style where terminal nodes and edges to 0 are omitted.

For our class of nets, it might be difficult (and it is generally impossible) to compute an upper bound on the marking of a place. To store a set of reachable markings during symbolic state-space generation, we could then use dynamic MDDs over $\mathbb{N}^{|\mathcal{P}|}$, so that a marking \mathbf{m} is simply a tuple encoded in the MDD. However, this simplistic approach has several drawbacks:

- Even if the (current) bound B_p on the marking of a place p is tight, i.e., there is a reachable marking \mathbf{m} with $\mathbf{m}_p = B_p$, the local set \mathcal{S}_p might have “holes”, i.e., no reachable marking \mathbf{n} might have $\mathbf{n}_p = c$, for some $0 \leq c < B_p$. This may waste memory or computation during MDD manipulation.
- If many different markings for p are possible, \mathcal{S}_p and thus the nodes at level p might be too large, again decreasing the efficiency of MDD manipulations. It might then be better to *split* a place over multiple MDD levels. This is actually necessary if the implementation uses BDDs [3], which are essentially our MDDs restricted to the case where each \mathcal{S}_l is just $\{0, 1\}$.
- On the other hand, our symbolic algorithms can greatly benefit from “event locality” which we will discuss later. To enhance such locality, we might instead want to *merge* certain places into a single MDD level.
- If some invariants are known, we can avoid storing some of the $|\mathcal{P}|$ components of the marking, since they can be recovered from the remaining ones.

For simplicity, and since we employ MDDs, we ignore the issue of splitting a place over multiple levels, but assume the use of $K \leq |\mathcal{P}|$ *indexing* functions that map submarkings into natural numbers. Given a net, we partition its places into K subsets $\mathcal{P}_K, \dots, \mathcal{P}_1$, so that a marking \mathbf{m} is written as the collection of the K submarkings $(\mathbf{m}_K, \dots, \mathbf{m}_1)$. Then, \mathbf{m} can be mapped to the tuple of the corresponding K submarking indices $(\mathbf{i}_K, \dots, \mathbf{i}_1)$, where $\mathbf{i}_l = \psi_l(\mathbf{m}_l)$ and $\psi_l : \mathbb{N}^{|\mathcal{P}_l|} \rightarrow \mathbb{N} \cup \{\text{null}\}$ is a partial function. In practice, each ψ_l only needs to map the set \mathcal{M}_l of submarkings for \mathcal{P}_l known to be reachable so far, to the range $\{0, \dots, |\mathcal{M}_l| - 1\}$ of natural numbers. We can then define ψ_l *dynamically*:

- Initially, set $\mathcal{M}_l = \{\mathbf{m}_l^{init}\}$ and $\psi_l(\mathbf{m}_l^{init}) = 0$, i.e., map the only known submarking for \mathcal{P}_l , the initial submarking, to the first natural number.

- For any other $\mathbf{m}_l \in \mathbb{N}^{|\mathcal{P}_l|} \setminus \mathcal{M}_l$, i.e., any submarking not yet discovered, set $\psi_l(\mathbf{m}_l)$ to the default value null.
- When a new submarking \mathbf{m}_l for \mathcal{P}_l is discovered, add it to \mathcal{M}_l and set $\psi_l(\mathbf{m}_l) = |\mathcal{M}_l| - 1$.

This mapping can be efficiently stored in a search tree and offers great flexibility in choosing the MDD variables $(\mathbf{x}_K, \dots, \mathbf{x}_1)$ corresponding to the possible values of the indices at each level. We can have as little as a single variable (when $K=1$, $\mathcal{S}_1 = \mathcal{S}$ and we perform an explicit reachability set generation), or as many as $|\mathcal{P}|$ variables, so that each place corresponds to a different level of the MDD.

2.4 Symbolic algorithms to generate the state space of a net

We now focus on building the state space of a net using MDDs, i.e., on computing $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ corresponding to \mathcal{M} . Since the functions ψ_l , $K \geq l \geq 1$, provide a bijection between markings and K -tuples, knowledge of \mathcal{S} implies knowledge of \mathcal{M} . As they manipulate sets of tuples, not individual tuples, all symbolic state-space generation algorithms are some variation of the following:

“Build the MDD encoding \mathcal{S} , defined as the smallest solution to the fixpoint equation $\mathcal{S} = \mathcal{S} \cup \mathcal{N}(\mathcal{S})$ subject to $\mathcal{S} \supseteq \mathcal{X}_{init}$ ”,

where the next-state function \mathcal{N} is now applied to tuples instead of markings.

Of course, \mathcal{N} is also encoded using either MDDs or related formalisms. The most common choice is a $2K$ -level MDD with *interleaved* levels for the *current* variables \mathbf{x} and the *next* variables \mathbf{x}' , i.e., if $\mathbf{i}' \in \mathcal{N}(\mathbf{i})$, there is a path $(i_K, i'_K, \dots, i_1, i'_1)$ from the root of the MDD encoding \mathcal{N} to node 1. In our asynchronous context, a *disjunctive partition* [4] can be used, where each transition $t \in \mathcal{T}$ is encoded as a separate $2K$ -level MDD. This is the case in the standard breadth-first algorithm *Bfs* shown in Fig. 2. Function *Union* returns the root of the MDD encoding the union of the sets encoded by the arguments (all encoded as K -level MDDs), while function *Image* returns the root of the MDD encoding the set of states reachable in one application of the second argument (a $2K$ -level MDD) from any state encoded by the first argument (a K -level MDD); both functions are easily expressed in recursive form. In the figure, we identify sets and relations with the MDDs encoding them; thus, for example, $\mathcal{N}_t[i][i']$ denotes the node in the MDD encoding \mathcal{N}_t which is reached by following the edge labeled i from the root and then the edge labeled i' from the resulting node.

To improve over breadth-first iterations, we have proposed algorithms [12, 13] that exploit *chaining* [33] and *event locality*. Chaining is based on the observation that the number of symbolic iterations might be reduced if the application of asynchronous events (transitions) is compounded sequentially; see *BfsChaining* in Fig. 2. While the search order is not strictly breadth-first anymore, the set of known states at the d^{th} iteration of the *repeat-until* loop is guaranteed to be at least as large with chaining as without.

However, the efficiency of symbolic state-space generation is determined not just by the *number* of iterations but also by their *cost*, i.e., by the size of the

<pre> mdd Bfs(mdd \mathcal{X}_{init}) 1 $\mathcal{S} \leftarrow \mathcal{X}_{init}$; 2 repeat 3 $\mathcal{X} \leftarrow \emptyset$; 4 foreach $t \in \mathcal{T}$ do 5 $\mathcal{X} \leftarrow Union(\mathcal{X}, Image(\mathcal{S}, \mathcal{N}_t))$; 6 $\mathcal{S} \leftarrow Union(\mathcal{S}, \mathcal{X})$; 7 until \mathcal{S} does not change; 8 return \mathcal{S}; </pre>	<pre> mdd BfsChaining(mdd \mathcal{X}_{init}) 1 $\mathcal{S} \leftarrow \mathcal{X}_{init}$; 2 repeat 3 foreach $t \in \mathcal{T}$ do 4 $\mathcal{S} \leftarrow Union(\mathcal{S}, Image(\mathcal{S}, \mathcal{N}_t))$; 5 until \mathcal{S} does not change; 6 return \mathcal{S}; </pre>
<pre> void Saturation(mdd \mathcal{X}_{init}) 1 for $l = 1$ to K do 2 foreach node p at level l on a path from \mathcal{X}_{init} to 1 do 3 Saturate(p); • update p in place </pre>	
<pre> void Saturate(mdd p) 1 $l \leftarrow p.lvl$; 2 repeat 3 choose t s.t. $Top(t) = l, i \in \mathcal{S}_l, i' \in \mathcal{S}_l$ s.t. $p[i] \neq 0$ and $\mathcal{N}_t[i][i'] \neq 0$; 4 $p[i'] \leftarrow Union(p[i'], ImageSat(p[i], \mathcal{N}_t[i][i']))$; 5 until p does not change; </pre>	
<pre> mdd ImageSat(mdd $q, mdd2 f$) 1 if $q = 0$ or $f = 0$ then return 0; 2 $k \leftarrow q.lvl$; • $f.lvl = k$ as well 3 $s \leftarrow$ new level-k node with edges set to 0; 4 foreach $i \in \mathcal{S}_k, i' \in \mathcal{S}_k$ s.t. $q[i] \neq 0$ and $f[i][i'] \neq 0$ do 5 $s[i'] \leftarrow Union(s[i'], ImageSat(q[i], f[i][i']))$; 6 Saturate($s$); 7 return s. </pre>	

Fig. 2. Breadth-first, chaining, and Saturation state-space generation.

MDDs involved. In practice, chaining has been shown to be quite effective in many asynchronous models, but its effectiveness can be greatly affected by the order in which transitions are applied. Event locality can then be used to define a good ordering heuristic [12], as we will explain next.

Given a transition t , we define $\mathcal{V}_M(t) = \{x_l : \exists \mathbf{i}, \mathbf{i}' \in \widehat{\mathcal{S}}, \mathbf{i}' \in \mathcal{N}_t(\mathbf{i}) \wedge i_l \neq i'_l\}$ and $\mathcal{V}_D(t) = \{x_l : \exists \mathbf{i}, \mathbf{j} \in \widehat{\mathcal{S}}, \forall k \neq l, i_k = j_k \wedge \mathcal{N}_t(\mathbf{i}) \neq \emptyset \wedge \mathcal{N}_t(\mathbf{j}) = \emptyset\}$, i.e., the variables that can be modified by t , or that can disable t , respectively. Moreover, we let $Top(t) = \max\{l : x_l \in \mathcal{V}_M(t) \cup \mathcal{V}_D(t)\}$ and $Bot(t) = \min\{l : x_l \in \mathcal{V}_M(t) \cup \mathcal{V}_D(t)\}$.

We showed experimentally in [12] that applying the transitions $t \in \mathcal{T}$ in an order consistent with their value of Top , from 1 to K , results in effective chaining. Locality is easily determined for our nets since the enabling and firing effect of a transition t depend only on its input, output, and inhibitor places, plus any place appearing in the cardinality expression of the corresponding arcs.

Recognizing locality, however, offers great potential beyond suggesting a good chaining order. If $Top(t) = l$ and $Bot(t) = k$, any variable x_m outside this range, i.e., above l or below k , is not changed by the firing of transition t . When computing the image in line 4 of *BfsChaining*, we can then access only MDD nodes at level l or below and update *in-place* only MDD nodes at level l , without having

to access the MDD from the root. While *Kronecker* [25] and *matrix diagram* [24] encodings have been used to exploit these *identity transformations* in \mathcal{N}_t , the most general and efficient data structure appears to be a decision diagram with special reduction rules [13]. In this paper, we assume that \mathcal{N}_t is encoded with an MDD over just the current and next variables between $Top(t)$ and $Bot(t)$ included, instead of a $2K$ -level MDD. If there are no marking-dependent arc cardinalities, the structure of this MDD is quite simple, as we just need to encode the effect of every input, inhibitor, and output arc connected to t ; the result is an MDD with just one node per ‘unprimed’ level. For general self-modifying nets, a “localized” explicit enumeration approach may be used [13], although a completely symbolic approach might be preferable.

We can now introduce our most advanced algorithm, *Saturation*, also shown in Fig. 2. An MDD node p at level l is *saturated* [10] if

$$\forall t \in \mathcal{T}, Top(t) \leq l \Rightarrow \mathcal{B}(K, p) \supseteq \mathcal{N}_t(\mathcal{B}(K, p)).$$

To saturate node p once its descendants are saturated, we compute the effect of firing t on p for each transition t such that $Top(t) = l$, recursively saturating any node at lower levels which may be created in the process, and add the result to $\mathcal{B}(p)$ using in-place updates. Thus, *Saturation* proceeds saturating the nodes in the MDD encoding the initial set of states bottom-up, starting at level 1 and stopping when the root at level K is saturated.

Only saturated nodes appear in the *operation cache* (needed to retrieve the result of an *ImageSat* or *Union* call, if it has already been issued before with the same parameters) and the *unique table* (needed to enforce MDD canonicity by recognizing duplicate nodes). Since nodes in the MDD encoding the final \mathcal{S} are saturated by definition, this unique property – not shared by any other approach – is key to a much greater efficiency. Indeed, we have experimentally found that both memory and run-time requirements for our *Saturation* approach are usually several orders of magnitude smaller than for the traditional symbolic breadth-first exploration, when modeling asynchronous systems.

3 Structural invariants to improve symbolic algorithms

Structural invariants have been proposed for *variable elimination*. For example, [17] suggests an algorithm that starts with an empty set of boolean variables (places of a safe Petri net) and examines each place in some arbitrary order, adding it as new (lower) level of the BDD only if it is not functionally dependent on the current set of variables. This greedy elimination algorithm reduces the number of levels of the BDD, with the goal of making symbolic state-space generation more efficient. However, we argue that this invariant-based elimination severely hurts locality and is generally a bad idea, not only for *Saturation*, but even for the simpler BFS iterations (if properly implemented to exploit locality). To see why this is the case, consider a transition t with an input or inhibitor arc from a place p , i.e., $p \in \mathcal{V}_D(t)$. If p is in the support of invariant v and is eliminated because all other places in $Support(v)$ already correspond

to BDD levels, the marking of p can indeed be determined from the marking of each place $q \in \text{Support}(v) \setminus \{p\}$. However, this not only removes p from $\mathcal{V}_D(t)$ but also adds $\text{Support}(v) \setminus \{p\}$ to it. In most cases, the *span* of transition t , i.e., the value of $\text{Top}(t) - \text{Bot}(t) + 1$, can greatly increase, resulting in a more costly image computation for \mathcal{N}_t .

The solution we present in Sec. 3.1, enabled by our use of MDDs instead of BDDs, is to perform *variable merging*. This achieves the same goal of reducing the number of levels (actually resulting in more levels being eliminated, since it considers groups of variables at a time, not just individual ones as in [17]), without negatively affecting locality and actually improving it for a meaningful class of nets. Then, having at our disposal the invariants, we turn to the problem of variable ordering, and show in Sec. 3.2 how our previous idea of minimizing the sum of the top levels affected by each transition [39] can be extended to take invariants into account as well, treating an invariant analogously to a transition and its support as if it were the set of places “affected” by the invariant.

3.1 Using structural invariants to merge state variables

As one of our main contributions, we first present and prove a general theorem stating that *merging two MDD variables* based on functional dependence guarantees to reduce the size of an MDD. In contrast, we show that placing variables in the support of an invariant close to each other without merging them, as suggested by many static and dynamic variable reordering techniques [19, 21], may actually increase the size of the MDD. We then adopt our merging theorem to improve Petri-net state-space encodings with place invariants, and present a greedy algorithm to iteratively merge MDD variables, given a set of place invariants obtained from a structural analysis of a net. Thus, our goal is to determine both a merging of the MDD variables and an ordering of these merged variables.

Variable merging based on functional dependence. To discuss what happens to the size of an MDD when merging variables based on functional dependence, one must take into account both the number of nodes and their sizes. To be precise, and to follow what is done in an efficient “sparse node” implementation, the size of an MDD node is given by the number of its non-zero edges, i.e., the number of outgoing edges that do not point to node 0. Thus, since a node has always at least one non-zero edge, the size of a node for variable x_l can range from one to $|\mathcal{S}_l|$. First, we recall a theorem on the number of MDD nodes required to encode a boolean function f .

Theorem 1 [38] Using the variable order (x_K, \dots, x_1) , the number of MDD nodes for variable $x_l \in \mathcal{S}_l$ (for $K \geq l \geq 1$) in the MDD encoding of a boolean function $f(x_K, \dots, x_1)$ equals the number of different subfunctions obtained by fixing the values of x_K, \dots, x_{l+1} to all the possible values $i_K \in \mathcal{S}_K, \dots, i_{l+1} \in \mathcal{S}_{l+1}$.

In the following, we denote by $f[\overset{x_{k_1}}{i_{k_1}}, \dots, \overset{x_{k_n}}{i_{k_n}}]$ the subfunction obtained from f by fixing the value of x_{k_1}, \dots, x_{k_n} to i_{k_1}, \dots, i_{k_n} , and we use the same symbol for a boolean function and its MDD encoding, once the variable order is given.

Theorem 2 Consider an MDD f encoding a set $\mathcal{X} \subseteq \mathcal{S}_K \times \dots \times \mathcal{S}_1$ with variable order $\pi = (x_K, \dots, x_1)$. If x_m is functionally dependent on $\{x_K, \dots, x_k\}$, with $k > m$, then define a new variable $x_{k,m} \equiv x_k n_m + x_m$, where $n_m = |\mathcal{S}_m|$, having domain $\mathcal{S}_{k,m} = \{0, \dots, |\mathcal{S}_k \times \mathcal{S}_m| - 1\}$. Let the MDD g encode $\Lambda(\mathcal{X})$ with variable order $\bar{\pi} = (x_K, \dots, x_{k+1}, x_{k,m}, x_{k-1}, \dots, x_{m+1}, x_{m-1}, \dots, x_1)$, where $\Lambda(x_K, \dots, x_1) = (x_K, \dots, x_{k+1}, x_{k,m}, x_{k-1}, \dots, x_{m+1}, x_{m-1}, \dots, x_1)$.

Then, (1) $f_{[i_k, i_m]}^{[x_k, x_m]} \equiv g_{[i_k n_m + i_m]}^{[x_k, x_m]}$; and (2) g requires strictly fewer MDD nodes and non-zero edges than f .

Proof. Property (1) follows directly from the definition of $x_{k,m}$, Λ , and g . Let $\bar{\nu}_l$ and ν_l be the number of nodes corresponding to variable x_l in g and f , respectively. Analogously, let $\bar{\epsilon}_l$ and ϵ_l be the number of non-zero edges leaving these nodes. To establish Property (2), we prove that $\bar{\nu}_{k,m} = \nu_k$ and $\bar{\epsilon}_{k,m} = \epsilon_k$, $\bar{\nu}_l = \nu_l$ and $\bar{\epsilon}_l = \epsilon_l$ for $x_l \in \{x_K, \dots, x_{k+1}, x_{m-1}, \dots, x_1\}$, and $\bar{\nu}_l \leq \nu_l$ and $\bar{\epsilon}_l \leq \epsilon_l$ for $x_l \in \{x_{k-1}, \dots, x_{m+1}\}$. These relations, in addition to the fact that f contains $\nu_m > 0$ additional nodes corresponding to x_m (each of them having exactly one non-zero edge, because of the functional dependence), show that g is encoded using at least $\nu_m = \epsilon_m$ fewer nodes and edges than f . We now prove these relations by considering the different possible positions of variable x_l in $\bar{\pi}$.

Case 1: $x_l \in \{x_{m-1}, \dots, x_1\}$. Since $f_{[i_k, i_m]}^{[x_k, x_m]} \equiv g_{[i_k n_m + i_m]}^{[x_k, x_m]}$, we let f_1 and g_1 be

$$\begin{aligned} f_1 &= f_{[i_k, i_m]}^{[x_k, x_m]} [i_K, \dots, i_{k+1}, i_{k-1}, \dots, i_{m+1}, i_{m-1}, \dots, i_{l+1}] \\ g_1 &= g_{[i_k n_m + i_m]}^{[x_k, x_m]} [i_K, \dots, i_{k+1}, i_{k-1}, \dots, i_{m+1}, i_{m-1}, \dots, i_{l+1}] \end{aligned}$$

and conclude that $f_1 \equiv g_1$. Recall that the number of nodes of variable x_l in f is the number of different subfunctions $f_{[i_k, \dots, i_{l+1}]}^{[x_k, \dots, x_{l+1}]}$, for all possible i_k, \dots, i_{l+1} . Since f and g have the same set of such subfunctions, we must have $\bar{\nu}_l = \nu_l$. To see that $\bar{\epsilon}_l = \epsilon_l$ as well, simply observe that each pair of corresponding MDD nodes, e.g., f_1 and g_1 , must have the same number of non-zero edges, since $f_1 \equiv g_1$ implies $f_1[i_l] \equiv g_1[i_l]$ for any $i_l \in \mathcal{X}_l$, and the edge i_l is non-zero if and only if $f_1[i_l] \neq 0$.

Case 2: $x_l \in \{x_{k-1}, \dots, x_{m+1}\}$. Consider two different nodes of x_l in g , encoding two different subfunctions g_1 and g_2 which obviously satisfy $g_1 \neq 0$ and $g_2 \neq 0$:

$$g_1 \equiv g_{[i_k, \dots, i_{k+1}, i_k n_m + i_m, i_{k-1}, \dots, i_{l+1}]}^{[x_k, \dots, x_{l+1}]} \quad g_2 \equiv g_{[j_k, \dots, j_{k+1}, j_k n_m + j_m, j_{k-1}, \dots, j_{l+1}]}^{[x_k, \dots, x_{l+1}]}$$

Then, define f_1 and f_2 as follows, which obviously satisfy $f_1 \neq 0$ and $f_2 \neq 0$, too:

$$f_1 \equiv f_{[i_k, \dots, i_{k+1}, i_k, i_{k-1}, \dots, i_{l+1}]}^{[x_k, \dots, x_{l+1}]} \quad f_2 \equiv f_{[j_k, \dots, j_{k+1}, j_k, j_{k-1}, \dots, i_{l+1}]}^{[x_k, \dots, x_{l+1}]}$$

We prove by contradiction that f_1 and f_2 must be different and therefore encoded by two different nodes of variable x_l in f . Since x_m is functionally dependent on $\{x_K, \dots, x_k\}$ and the value of (x_K, \dots, x_k) is fixed to (i_K, \dots, i_k) for f_1 and to (j_K, \dots, j_k) for f_2 , there must exist unique values i_m and j_m such that $f_1[i_m] \neq 0$ and $f_2[j_m] \neq 0$. If f_1 and f_2 were the same function, we would have $i_m = j_m$ and $f_1[i_m] \equiv f_2[j_m]$. From Property (1), we then obtain $g_1 \equiv f_1[i_m] \equiv f_2[j_m] \equiv g_2$,

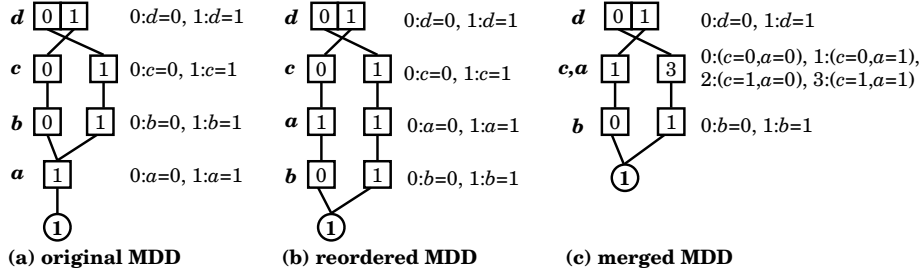


Fig. 3. An example where moving support variables closer increases the MDD size.

which is a contradiction. Thus, distinct nodes of g must correspond to distinct nodes of f , i.e., $\bar{\nu}_l \leq \nu_l$. Again, to see that $\bar{\epsilon}_l \leq \epsilon_l$, observe that the MDD nodes encoding f_1 and g_1 must have the same number of non-zero edges because, for all $i_l \in \mathcal{S}_l$, $g_1[x_{i_l}^{i_l}] \equiv f_1[x_{i_m}^{i_l}][x_{i_l}^{i_l}]$. Furthermore, if multiple nodes in f correspond to the same node of g , i.e., if $\bar{\nu}_l < \nu_l$, we also have $\bar{\epsilon}_l < \epsilon_l$.

Case 3: $x_l \in \{x_K, \dots, x_{k+1}, x_{k,m}\}$. Observe that $g \equiv \Lambda(f)$ and $g[x_{i_K}^{x_K}, \dots, x_{i_{l+1}}^{x_{l+1}}] \equiv \Lambda_l(f[x_{i_K}^{x_K}, \dots, x_{i_{l+1}}^{x_{l+1}}])$, where Λ_l is defined analogously to Λ , i.e., $\Lambda_l(x_l, \dots, x_1) = (x_l, x_{k+1}, x_{k+m}, x_{k-1}, \dots, x_{m+1}, \dots, x_{m-1}, \dots, x_1)$. As for Case 1, we can prove that $\bar{\nu}_l = \nu_l$ and $\bar{\epsilon}_l = \epsilon_l$ by observing that g and f must have the same subfunctions, and the MDD nodes encoding these subfunctions must have the same number of non-zero edges. \square

Intuitively, merging variable x_m with x_k is not that different from moving it just below x_k in the variable order, the commonly suggested approach for BDDs to help reduce the number of nodes [19, 21]. However, the example in Fig. 3 illustrates that the latter can instead increase the BDD size. Fig. 3(a) shows an example of MDDs that encodes a boolean function with initial variable order (d, c, b, a) , satisfying the invariant $a + c + d = 2$. Fig. 3(b) shows the result of reordering the MDD to put variables a , c , and d close to each other, by swapping variables b and a . Note that the number of nodes in the second MDD increases from six to seven, and the number of non-zero edges from seven to eight. Fig. 3(c) shows instead the result of merging variables a and c , where the number of nodes decreases from six to five and the number of non-zero edges from seven to six, in accordance with Thm. 2. The meaning of the elements of \mathcal{S}_l in terms of the value of the variables assigned to level l is shown to the right of each MDD. We stress that this reduction in the number of nodes can only be achieved if the MDDs are implemented natively, not as the interface to BDDs implemented in [22, 36]; this is apparent since Fig. 3(b) is exactly the BDD that would be built if the MDD of Fig. 3(c) was implemented using BDDs.

Focusing now on Petri nets, we restate Thm. 2 in Petri net terminology and use place invariants to determine functional dependence.

Theorem 3 Consider a Petri net with an ordered partition π of \mathcal{P} into the sets $(\mathcal{P}_K, \dots, \mathcal{P}_1)$ and mappings $\psi_{\mathcal{P}_l} : \mathbb{N}^{\mathcal{P}_l} \rightarrow \mathbb{N} \cup \{\text{null}\}$, for $K \geq l \geq 1$. Let the

```

Partition InvariantBasedMerging (Invariants  $\mathbf{W}_{\mathcal{W},\mathcal{P}}$ )
1  $K \leftarrow |\mathcal{P}|$ ;
2  $\pi \leftarrow (\{p_K\}, \dots, \{p_1\})$ ;           • Initialize the partition with one place per class
3 repeat
4   for  $m = K - 1$  to 1 do
5      $k = \text{LevelToMerge}(K, m, \pi, \mathbf{W}_{\mathcal{W},\mathcal{P}})$ ;
6     if  $k > m$  then
7        $\pi \leftarrow (\pi_K, \dots, \pi_{k+1}, \pi_k \cup \pi_m, \pi_{k-1}, \dots, \pi_{m+1}, \pi_{m-1}, \dots, \pi_1)$ ;
8        $K \leftarrow K - 1$ ;                       • The partition is now one class smaller
9   until  $\pi$  does not change;
10 return  $\pi$ ;

int LevelToMerge (int  $L$ , int  $m$ , Partition  $(\mathcal{Q}_L, \dots, \mathcal{Q}_1)$ , Invariants  $\mathbf{W}_{\mathcal{W},\mathcal{P}}$ )
1 foreach  $k = L$  downto  $m + 1$  do
2    $\mathcal{W}' \leftarrow \{v \in \mathcal{W} \mid \text{Support}(v) \subseteq \mathcal{Q}_m \cup \bigcup_{l=k}^L \mathcal{Q}_l\}$ ;
3   if  $|\mathcal{Q}_m| = \text{Rank}(\mathbf{W}_{\mathcal{W}',\mathcal{Q}_m})$  then
4     return  $k$ ;
5 return  $m$ ;

```

Fig. 4. A greedy algorithm to iteratively merge MDD variables using Thm. 3.

ordered partition $\bar{\pi}$ be the one obtained by merging \mathcal{P}_k and \mathcal{P}_m into $\mathcal{P}_{\{k,m\}}$, with $k > m$, resulting in the order $(\mathcal{P}_K, \dots, \mathcal{P}_{\{k,m\}}, \mathcal{P}_{k-1}, \dots, \mathcal{P}_{m+1}, \mathcal{P}_{m-1}, \dots, \mathcal{P}_1)$ and the same mappings as before, except for the new $\psi_{\{\mathcal{P}_k, \mathcal{P}_m\}} : \mathbb{N}^{\mathcal{P}_{\{k,m\}}} \rightarrow \mathbb{N} \cup \{\text{null}\}$ to replace ψ_k and ψ_m , which satisfies $\psi_{\{\mathcal{P}_k, \mathcal{P}_m\}}(\mathbf{m}_{\mathcal{P}_k}, \mathbf{m}_{\mathcal{P}_m}) = \text{null}$ if and only if $\psi_k(\mathbf{m}_{\mathcal{P}_k}) = \text{null}$ or $\psi_m(\mathbf{m}_{\mathcal{P}_m}) = \text{null}$. Then, if \mathcal{P}_m is functionally dependent on $\bigcup_{K \geq l \geq k} \mathcal{P}_l$, the MDD encoding of any nonempty set of markings \mathcal{X} requires strictly fewer nodes and edges with $\bar{\pi}$ than with π .

Proof. The proof is a specialization of the one of Thm. 2, noting that, there, we used the mapping $x_{k,m} = x_k n_m + x_m$ for simplicity. In reality, any mapping where $x_{k,m}$ can uniquely encode any *reachable* combination of x_k and x_m may be employed. This is necessary in practice when using dynamic MDDs, where the sets \mathcal{S}_l , i.e., the bounds on the net places, are not known a priori. \square

Greedy algorithm to merge MDD variables. Based on Thm. 3, Fig. 4 illustrates a greedy algorithm to merge as many MDD variables as possible, given a set of place invariants, while guaranteeing that the number of nodes and non-zero edges can only decrease.

For a Petri net, procedure *InvariantBasedMerging* in Fig. 4 takes a set of linearly independent place invariants, in the form of a matrix $\mathbf{W}_{\mathcal{W},\mathcal{P}}$, as input and assumes one place per variable in the initial MDD variable order (line 2). The procedure then traverses each level m of the MDD, from top to bottom according to the given partition π , and calls procedure *LevelToMerge* to compute the highest level k such that the m^{th} partition class π_m functionally depends on $\mathcal{P}' = \bigcup_{K \geq l \geq k} \mathcal{P}_l$. It does so by determining the set \mathcal{W}' of invariants whose support is a subset of $\pi_m \cup \mathcal{P}'$, and by performing Gaussian elimination on submatrix $\mathbf{W}_{\mathcal{W}',\pi_m}$ to check whether it has full column rank (line 3 of *LevelToMerge*). If such level k exists, then π_m is merged with π_k , otherwise the partition remains

unchanged. Procedure *InvariantBasedMerging* repeats this merging process until no more merging is possible, then it returns the final partition π .

The procedure has polynomial complexity, since it computes $O(|\mathcal{P}|^3)$ matrix ranks in the worst case. In practice, due to the sparsity of matrix \mathbf{W} , the performance is excellent, as discussed in Sec. 4. We leave a discussion of whether it achieves the smallest possible number of MDD levels, without increasing the number or size of the nodes according to Thm. 3, to future work.

3.2 Using structural invariants to order state variables

It is well-known that the variable order can greatly affect the efficiency of decision diagram algorithms, and that computing an optimal order is an NP-complete problem [2]. Thus, symbolic model-checking tools must rely on heuristics aimed at finding either a good order *statically*, i.e., prior to starting any symbolic manipulation, or at improving the order *dynamically*, i.e., during symbolic manipulation.

Focusing on static approaches, our locality-based encoding suggests that variable orders with small span $Top(t) - Bot(t) + 1$ for each transition t are preferable, both memory-wise when encoding \mathcal{N}_t , and time-wise when applying \mathcal{N}_t to compute an image. Furthermore, since Saturation works on the nodes in a bottom-up fashion, it prefers orders where most spans are situated in lower levels. In the past, we have then considered the following static heuristics [39]:

- **SOS**: Minimize the sum of the transition spans, $\sum_{t \in \mathcal{T}} (Top(t) - Bot(t) + 1)$.
- **SOT**: Minimize the sum of the transition tops, $\sum_{t \in \mathcal{T}} Top(t)$.
- **Combined SOS/SOT**: Minimize $\sum_{t \in \mathcal{T}} Top(t)^\alpha \cdot (Top(t) - Bot(t) + 1)$.

The combined heuristic encompasses SOS and SOT, since the parameter α controls the relative importance of the size of the span vs. its location. When $\alpha = 0$, the heuristic becomes SOS, as it ignores the location of the span, while for $\alpha \gg 1$, it approaches SOT. For the test suite in [39], $\alpha = 1$ works generally well, confirming our intuition about the behavior of Saturation, namely that Saturation tends to perform better when both the size and the location of the spans are small.

We now propose to integrate the idea of an ordering heuristic based on transition locality with the equally intuitive idea that an order where variables in the support of an invariant are “close to each other,” is preferable [29]. However, given the lesson of the previous section, we also wish to apply our greedy merging heuristic. There are four ways to approach this:

- For each possible permutation of the places, apply our merging heuristic. Then, evaluate the score of the chosen objective function (among the three above), and select the permutation that results in the minimum score. Of course, this results in the optimal order with respect to the chosen objective function, but the approach is not feasible except for very small nets.
- Choose one of the objective functions and, assuming one place per level, compute an order that produces a low score. Note that this is not necessarily

the minimum score, as this is itself an NP-complete problem. Then, apply either our greedy merging heuristic, or a modified version of it that ensures that the achieved score is not worsened.

- Given an initial ordering of the places, use our greedy merging heuristic. Then, compute an order that produces a low, not necessarily minimal, score for the chosen objective function, subject to the constraints of Thm. 3, to keep the node size linear. For example, if $\mathbf{m}_a + \mathbf{m}_b + \mathbf{m}_c = N$ in every marking \mathbf{m} , if places a and b are not covered by any other invariant, if a and b have been merged together, and if they are at a level below that of c , then we cannot move them above c . If we did, a node encoding \mathbf{m}_a and \mathbf{m}_b could have $O(N^2)$ nonzero edges, since $\mathbf{m}_a + \mathbf{m}_b$ is not fixed until we know \mathbf{m}_c .
- Consider an invariant just like a transition, i.e., modify the chosen objective function to sum over both transitions and invariants, where the support of an invariant is treated just like the dependence list of a transition. Once the order is obtained, apply our greedy merging heuristic.

We adopt the last approach in conjunction with the SOT objective function, for several reasons. First, it is very similar in spirit to our original ordering approach, yet it adds novel information about invariants to guide the heuristic. Second, we have reasonably fast heuristics to solve SOT (indeed we even have a $\log n$ approximation algorithm for it), while the heuristics for SOS are not as fast, and those for the combined SOS/SOT problem are even slower. More importantly, when applying our greedy merging algorithm after the variable ordering heuristic, the span of an event is changed in unpredictable ways that do not preserve the optimality of the achieved score.

A fundamental observation is that, if place p is in the support of invariant v , any transition t that *modifies* p must also modify at least one other place q in the support of v . Thus, if p and p' are the lowest and highest places of the support of v according to the current MDD order, merging p with the second lowest place r in the support will not change the fact that p' is still the highest place in the support of v . Analogously, the highest place p'' determining $Top(t)$ is at least as high as q , which is at least as high as r ; thus, again, p'' will still determine the value of $Top(t)$. Of course, the levels of p' and p'' are decreased by one, simply because the level of p , below them, is removed. Unfortunately, the same does not hold when p only *controls* the enabling or firing of a transition t , i.e., if there is an inhibitor arc from p to t or if p appears in the marking-dependent cardinality expression of arcs attached to t . In that case, merging p to a higher level k might increase the value of $Top(t)$ to k . Thus, for standard Petri nets with no inhibitor arcs and for the restricted self-modifying nets considered in [8], merging is guaranteed to improve the score of SOT, although it does not necessarily preserve optimality.

One danger of treating invariants like transitions in the scoring heuristic is that the number of invariants can be exponentially large, even when limiting ourselves to minimal ones (i.e., those whose support is not a superset of any other support). In such cases, the invariants would overwhelm the transitions and the resulting order would *de facto* be based almost exclusively on the invariants. To

avoid this problem, we compute a set of linearly independent invariants and feed only those to our heuristic for SOT; clearly, this set will contain at most $|\mathcal{P}|$ elements, whence it is of the same order as $|\mathcal{T}|$ in practical models.

4 Experimental results

We have implemented our static variable ordering ideas based on place invariants in the verification tool SMART [9], which supports Petri nets as front-end and reads an invariant matrix generated by the Petri-net tool GreatSPN [7]. This section reports our experimental results on a suite of asynchronous Petri net benchmarks for symbolic state-space generation.

We ran our experiments on a 3GHz Pentium workstation with 1GB RAM. Benchmarks *mmgt*, *dac*, *sentest*, *speed*, *dp*, *q*, *elevator*, and *key* are safe Petri nets taken from Corbett [16]. Benchmarks *knights* (board game model), *fms* and *kanban* [40] (manufacturing models), and *slot* [30], *courier* [42], and *ralep* [20] (protocol models) are Petri nets (without marking-dependent arcs, since GreatSPN does not accept this extension) from the SMART distribution.

Results. The first five columns of Table 1 show the model name and parameters, and the number of places ($\#P$), events ($\#T$) and place invariants computed by GreatSPN ($\#I$). The remaining columns are grouped according to whether the static variable order, computed via a fairly efficient logarithmic approximation for **SOT**, uses just the place-event matrix (**Event**) or the combined place-event+invariant matrix (**Event+Inv**). The approximation uses a randomized procedure, whence different parameters for the same model may result in different performance trends. For example, with **Event**, merging reduces the runtime of *courier* from 251 to 68sec when the parameter is 40, but has negligible effect when the parameter is 20.

The time for static variable ordering is stated in column **Time Ord**. For each group, we further report results according to whether variable merging is employed; method **No Merge** just uses the static order and therefore has one MDD variable per place of the Petri net, while **Merge** starts from the static order and merges variables using the proposed greedy algorithm of Fig. 4.

In addition, we state the run-time, peak, and final memory usage if the state-space generation with Saturation completes within 30 minutes. For **Merge**, we also report the number of MDD variables merged ($\#M$). The run-time spent on merging variables is not reported separately because it is quite small, always less than 5% of the total run-time, for any of the models. The time needed by GreatSPN to compute the invariants is shown in column **Time Inv**.

Discussion. From Table 1, we see the effectiveness of the new static variable ordering by comparing the two **No Merge** columns for **Event** and **Event+Inv**. The latter performs much better than the former on *mmgt*, *fms*, *slot*, *courier*, and *kanban*, slightly worse on *elevator* and *knights*, and similarly on the remaining benchmarks. The run-time for variable order computation is normally a small percentage of the run-times. The same can be said for invariant computation,

Table 1. Experimental results (Time in sec, Memory in KB; “>1800” means that run-time exceeds 1800 sec or memory exceeds 1GB).

Model	N	#P	#T	#I	Event								Event+Inv								
					Time	No Merge			Merge			Time	Time	No Merge			Merge				
					Ord	Time	Peak	Final	Time	Peak	Final	#M	Inv	Ord	Time	Peak	Final	Time	Peak	Final	#M
<i>mmgt</i>	3	122	172	47	4.0	1.85	1575	83	1.78	1485	77	12	0.02	3.0	0.96	838	46	0.93	829	43	12
<i>mmgt</i>	4	158	232	48	6.0	18.05	20645	295	17.10	19294	280	14	0.02	5.0	4.71	5385	142	4.78	5375	132	14
<i>dac</i>	15	105	73	183	1.0	0.24	30	26	0.20	21	19	28	0.02	1.0	0.24	28	27	0.18	20	19	28
<i>sentest</i>	75	252	102	3315	2.0	0.55	49	44	0.25	20	17	157	1.07	2.0	0.53	49	45	0.23	20	17	157
<i>sentest</i>	100	327	127	5665	5.0	0.71	70	65	0.32	27	24	207	3.48	5.0	0.94	64	61	0.3	24	22	208
<i>speed</i>	1	29	31	10	0.0	0.07	48	6	0.06	32	4	10	0.01	0.0	0.09	44	6	0.07	29	4	10
<i>dp</i>	12	72	48	48	0.0	0.16	19	15	0.09	9	7	36	0.01	1.0	0.17	18	15	0.09	9	7	36
<i>q</i>	1	163	194	492	5.0	0.84	715	349	0.72	619	294	27	0.09	5.0	0.77	524	336	0.65	442	280	29
<i>elevator</i>	3	326	782	693	28.0	47.06	6570	1620	45.39	6532	1412	9	1.87	21.0	49.73	7403	1654	47.71	7359	1443	9
<i>key</i>	2	94	92	774	0.0	0.26	86	72	0.23	90	58	16	0.45	0.0	0.25	91	71	0.26	102	58	15
<i>key</i>	3	129	133	5491	3.0	0.54	231	161	0.53	210	145	18	127.11	2.0	0.51	211	146	0.46	196	136	18
<i>knights</i>	5	243	401	91	2.0	9.20	3321	60	7.03	2138	39	25	0.03	2.0	12.37	4084	60	9.5	2584	39	25
<i>fms</i>	20	38	20	27	0.0	2.58	1388	334	2.76	1371	317	3	0.01	0.0	0.39	189	66	0.5	180	57	3
<i>fms</i>	40	38	20	27	0.0	26.34	10480	1786	27.20	10418	1724	3	0.01	0.0	2.28	755	250	2.57	749	244	3
<i>fms</i>	80	38	20	27	0.0	93.59	19159	9068	110.20	18923	8831	3	0.01	0.0	31.16	9420	1383	32.7	9301	1263	3
<i>slot</i>	20	160	160	42	2.0	>1800	-	-	>1800	-	-	-	0.01	2.0	1.57	1658	122	1.35	1213	90	41
<i>slot</i>	40	320	320	82	12.0	>1800	-	-	>1800	-	-	-	0.03	8.0	10.96	11802	481	8.56	8540	353	81
<i>courier</i>	20	45	34	13	0.0	7.35	6985	871	7.20	6816	775	13	0.01	1.0	4.14	2693	267	3.89	2441	229	13
<i>courier</i>	40	45	34	13	0.0	251.06	108562	4260	68.22	39397	1126	13	0.01	1.0	25.38	12282	1127	24.99	11413	994	13
<i>courier</i>	80	45	34	13	1.0	>1800	-	-	>1800	-	-	-	0.01	1.0	191.34	57385	5902	187.28	50540	5212	13
<i>kanban</i>	20	16	16	6	0.0	307.66	51522	33866	192.56	44343	26687	4	0.01	0.0	0.93	513	55	1.23	443	45	4
<i>kanban</i>	40	16	16	6	0.0	539.11	134734	49478	402.62	113223	45753	4	0.01	0.0	7.61	3043	240	8.91	2777	206	4
<i>kanban</i>	80	16	16	6	0.0	>1800	-	-	>1800	-	-	-	0.01	0.0	85.02	20404	1249	99.07	19367	1124	4
<i>ralep</i>	7	91	140	21	2.0	22.73	25767	3424	20.64	21552	2526	21	0.01	3.0	23.00	27704	3349	22.82	24258	2613	21
<i>ralep</i>	8	104	168	24	2.0	99.79	90499	9166	106.03	80117	6572	24	0.01	6.0	85.20	88486	7872	81.5	66893	6006	24
<i>ralep</i>	9	117	198	27	2.0	359.68	238313	17531	429.62	223186	12605	27	0.01	3.0	361.09	232590	15463	387.61	196297	11891	27

with the exception of two models, *sentest* and *key*, where GreatSPN computes a large number of (non-minimal) invariants and requires more run-time than state-space generation itself (pathologically so for *key* with parameter 3). However, it must be stressed that the run-times for state-space generation are the ones obtained using our heuristic; if we were to use random or even just not as good orders, the state-space generation run-times would be much worse.

To see the effectiveness of invariants-based variable merging, one can compare the **No Merge** and **Merge** columns of Table 1, for either **Event** or **Event+Inv**. Merging almost always substantially improves the peak and final memory usage and results in comparable or better run-time performance, with up to a factor of three improvement in memory and time.

Even if merging is guaranteed to reduce the size of a given MDD, applying this idea is still a heuristic. This is because it changes the value of *Top* for the transition in the net in such a way that Saturation may apply them in a different order, resulting in a larger peak size (in our benchmarks, this happens only for *key* with parameter 2). Overall, though, we believe that our ordering and merging heuristics can pave the way to a fully automated static ordering approach. This has a very practical impact, as it does not require one to come up with a good order, and it reduces or eliminates altogether the reliance on dynamic variable reordering which is known to be quite expensive in practice.

5 Related work

Most work on developing heuristics for finding good variable orders has been carried out in the context of digital-circuit verification and BDDs. Our focus in this paper is on *static* ordering, i.e., on finding a good ordering *before* constructing decision diagrams. In circuit verification, such approaches are typically based on a circuit’s topological structure and work on the so-called *model connectivity graph*, by either searching, evaluating, or decomposing this graph. Grumberg, Livne, and Markovitch [18] present a good survey of these approaches and propose a static ordering based on “experience from training models”. Dynamic grouping of boolean variables into MDD variables is proposed in [36]; however, invariants are not used to guide such grouping.

Our approach to static variable ordering with respect to Petri nets is unique in that it considers *place invariants* and proposes *variable merging* instead of variable elimination. It must be pointed out that Pastor, Cortadella, and Roig mention in [29] that they “choose the ordering with some initial support from the structure of the Petri net (the P-invariants of the net)”; however, no details are given. More fundamentally, though, our work here shows that ordering using invariants is simply not as effective as ordering *and* merging using invariants.

Invariants are one popular approach to analyzing Petri nets [32, 34]. With few exceptions, e.g., work by Schmidt [35] that utilizes transition invariants and research by Silva and his group [5] on performance throughput bounds, most researchers focus on place invariants. On the one hand, place invariants can help in identifying upper bounds of a Petri net’s reachable markings. Indeed, place

invariants provide necessary but not sufficient conditions on the reachability of a given marking. This in turn can benefit state-space generation algorithms, as is demonstrated, e.g., by Pastor, Cortadella, and Peña in [28].

On the other hand, place invariants can be used to reduce the amount of memory needed for storing a single marking [6, 35], by exploiting the functional dependencies described by each invariant. When storing sets of markings via decision diagrams, this eliminates some decision-diagram variables. To determine which exact places or variables should be dropped, Davies, Knottenbelt, and Kritzinger present a heuristic in [17]. In that paper they also propose an ad-hoc heuristic for the static variable ordering within BDDs, based on finding pairs of similar subnets and interleaving the corresponding places’ bit-vectors.

General functional dependencies have also been studied by Hu and Dill [19]. In contrast to work in Petri nets where generated invariants are known to be correct, Hu and Dill do not assume the correctness of given functional dependencies, but prove them correct alongside verification. Last, but not least, we shall mention the approach to static variable ordering taken by Semenov and Yakovlev [37], who suggest to find a “close to optimal ordering” via net unfolding techniques.

6 Conclusions and future work

This paper demonstrated the importance of considering place invariants of Petri nets when statically ordering variables for symbolic state-space generation. Previous work focused either solely on optimizing event locality [39], or on eliminating variables based on invariance information [17]. The novel heuristic proposed in this paper enhances the former work by exploiting place invariants for *merging* variables, instead of eliminating them as is done in all related research. While merging is not an option for BDDs, it is suitable for MDD-based approaches, including our Saturation algorithm [10]. We proved that merging MDD variables always reduces MDD sizes, while eliminating variables may actually enlarge MDDs. In addition, for standard Petri nets, merging never breaks event locality and often improves it, thus benefiting Saturation.

The benchmarking conducted by us within SMART [9] showed that our heuristic outperforms related static variable-ordering approaches in terms of time-efficiency *and* memory-efficiency. Most importantly, this is the case for practical examples, such as large instances of the slotted-ring network and the kanban system which had been out of reach of existing state-space exploration technology before. Hence, using invariants in variable-ordering heuristics is crucial, but it must be done correctly. In particular, the widespread practice of eliminating variables based on invariance information is counter-productive and should be abandoned in favor of merging variables.

Future work should proceed along two directions. On the one hand, we wish to explore whether our greedy merging algorithm is optimal, in the sense that it reduces an MDD to the smallest number of MDD variables according to our

merging rule. On the other hand, we intend to investigate whether place invariants are also beneficial in the context of *dynamic* variable ordering.

Acknowledgments. We wish to thank the anonymous reviewers for their constructive comments and suggestions.

References

1. A. Aziz, S. Taşiran, and R. K. Brayton. BDD variable ordering for interacting finite state machines. In *DAC*, pp. 283–288. ACM Press, 1994.
2. B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. on Computers*, 45(9):993–1002, 1996.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, 1986.
4. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Very Large Scale Integration*, pp. 49–58, 1991. IFIP Transactions, North-Holland.
5. J. Campos, G. Chiola, and M. Silva. Ergodicity and throughput bounds of Petri nets with unique consistent firing count vectors. *IEEE Trans. Softw. Eng.*, 17(2):117–126, 1991.
6. G. Chiola. Compiling techniques for the analysis of stochastic Petri nets. In *Modeling Techniques and Tools for Computer Performance Evaluation*, pp. 11–24. Plenum Press, 1989.
7. G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaud. GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation*, 24(1&2):47–68, 1995.
8. G. Ciardo. Petri nets with marking-dependent arc multiplicity: Properties and analysis. In *ICATPN*, LNCS 815:179–198, 1994.
9. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Performance Evaluation*, 63:578–608, 2006.
10. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *TACAS*, LNCS 2031:328–342, 2001.
11. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *TACAS*, LNCS 2619:379–393, 2003.
12. G. Ciardo, R. Marmorstein, and R. Siminiceanu. The Saturation algorithm for symbolic state space exploration. *STTT*, 8(1):4–25, 2006.
13. G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *CHARME*, LNCS 3725:146–161, 2005.
14. J. Colom and M. Silva. Improving the linearly based characterization of P/T nets. In *Advances in Petri Nets*, LNCS 483:113–145, 1991.
15. J. M. Colom and M. Silva. Convex geometry and semiflows in P/T nets: A comparative study of algorithms for the computation of minimal p-semiflows. In *ICATPN*, LNCS 483:74–95, 1989.
16. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–180, 1996.
17. I. Davies, W. Knottenbelt, and P. S. Kritzinger. Symbolic methods for the state space exploration of GSPN models. In *TOOLS*, LNCS 2324:188–199, 2002.
18. O. Grumberg, S. Livne, and S. Markovitch. Learning to order BDD variables in verification. *J. Art. Int. Res.*, 18:83–116, 2003.

19. A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *DAC*, pp. 266–271. ACM Press, 1993.
20. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. In *Foundations of Computer Science*, pp. 150–158. IEEE Press, 1981.
21. S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Variable ordering and selection for FSM traversal. In *ICCAD*, pp. 476–479. ACM Press, 1991.
22. T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
23. K. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. PhD thesis, Carnegie-Mellon Univ., 1992.
24. A. S. Miner. Implicit GSPN reachability set generation using decision diagrams. *Performance Evaluation*, 56(1-4):145–165, 2004.
25. A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *ICATPN '99*, LNCS 1639:6–25, 1999.
26. T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, 1989.
27. E. Pastor and J. Cortadella. Efficient encoding schemes for symbolic analysis of Petri nets. In *DATE*, pp. 790–795. IEEE Press, 1998.
28. E. Pastor, J. Cortadella, and M. Peña. Structural methods to improve the symbolic analysis of Petri nets. In *ICATPN*, LNCS 1639:26–45, 1999.
29. E. Pastor, J. Cortadella, and O. Roig. Symbolic analysis of bounded Petri nets. *IEEE Trans. Computers*, 50(5):432–448, 2001.
30. E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In *ICATPN*, LNCS 815:416–435, 1994.
31. P. Ramachandran and M. Kamath. On place invariant sets and the rank of the incidence matrix of Petri nets. In *Systems, Man, and Cybernetics*, pp. 160–165. IEEE Press, 1998.
32. W. Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag, 1998.
33. O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *ICATPN*, LNCS 935:374–391, 1995.
34. S. Sankaranarayanan, H. Sipma, and Z. Manna. Petri net analysis using invariant generation. In *Verification: Theory and Practice*, LNCS 2772:682–701, 2003.
35. K. Schmidt. Using Petri net invariants in state space construction. In *TACAS*, LNCS 2619:473–488, 2003.
36. F. Schmiedle, W. Günther and R. Drechsler. Dynamic Re-Encoding During MDD Minimization. In *ISMVL*, pp. 239–244. IEEE Press, 2000.
37. A. Semenov and A. Yakovlev. Combining partial orders and symbolic traversal for efficient verification of asynchronous circuits. Techn. Rep. CS-TR 501, Newcastle Univ., 1995.
38. D. Sieling and I. Wegener. NC-algorithms for operations on binary decision diagrams. *Parallel Processing Letters*, 3:3–12, 1993.
39. R. Siminiceanu and G. Ciardo. New metrics for static variable ordering in decision diagrams. In *TACAS*, LNCS 3920:90–104, 2006.
40. M. Tilgner, Y. Takahashi, and G. Ciardo. SNS 1.0: Synchronized Network Solver. In *Manufacturing and Petri Nets*, pp. 215–234, 1996.
41. R. Valk. Generalizations of Petri nets. In *MFCS*, LNCS 118:140–155, 1981.
42. C. M. Woodside and Y. Li. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In *PNPM*, pp. 64–73. IEEE Press, 1991.