Faster Enabledness-Updates for the Reachability Graph Computation

Torsten Liebke and Christian Rosenke

Universität Rostock, Institut für Informatik, Germany
{torsten.liebke,christian.rosenke}@uni-rostock.de

Abstract. The reachability graph entirely describes the state space and the transitional dynamics of a place/transition Petri net N. Many challenges in model checking reduce to searching the reachability graph. Therefore, this is a time critical operation deserving elaborate speed-up efforts in every detail. To enumerate the neighborhood of a reached state m is *the* fundamental subtask during the graph search. It requires to determine exactly the set $T(m)$ of enabled transitions in state m. In the past, we already avoided to compute $T(m)$ from scratch in every reached state m, as not much changes between consecutive states. Instead, we prepared a data structure $DI(N)$ allowing the production of $T(m)$ by briefly updating $T(m')$ when migrating from state m' to m. Although saving up much time during the actual graph search, the preprocessing of $DI(N)$ has been an unpleasantly costly investment until this point. In this paper, we introduce a new, generally much faster method to compute $DI(N)$. We implemented it in the award wining model checker *LoLA 2* and compared it to the previous approach. In our experiments we used the *model checking contest (MCC)* as a benchmark. The new method is in almost all cases at least one order of magnitude faster.

Keywords: Model Checking · Preprocessing · Enabledness-Updates.

1 Introduction

In model checking, the main task is to build the, possibly reduced, state space of a place/transition Petri net N. One of the critical points while building the state space, is to determine whether a transition is enabled or not. I.e. in each state the list of enabled transitions, which can lead to new states, must be computed. The aim is to avoid checking in every state every transition for enabledness. To avoid this, we only check the enabledness information of all transitions in the initial state. Then, whenever a fired transition t leads to a new state, we only want to update the enabledness information of transitions that are possibly disabled or enabled by the firing transition t. To this end, in the preprocessing, before building the state space, we compute the look-up data structure $DI(N)$, which basically is a list of possibly disabled/enabled transitions for each transition firing. The advantage is that the lists of $DI(N)$ are in general way smaller than all transitions and, thus, faster to process.

The problem of speeding-up the computation of the state space using enabling tests was already studied more than 25 years ago [7, 4]. The problem also arises playing the token game while simulating the Petri net [1]. The method described in [1] is based on the definition of linear enabling functions and the classification of transitions into five categories. The issue with this approach is that the Petri net has to be transformed and so-called silent transitions and preemptive transitions have to be added to the Petri net. In [4] a method is presented mainly to work in the context of unfolding algebraic nets. The authors compared their implementation with the tool LoLA [6] and came to the conclusion that LoLA is faster but needs more memory. Since 2010 LoLA 2 [9] uses a more advanced method, which we describe in this work. This method performs well in practice. But still, the overhead is unpleasantly costly and sometimes the approach needs several minutes or more for certain models in the yearly model checking contest (MCC) [2] to compute $DI(N)$.

In this work, we introduce a new and faster method for computing $DI(N)$. The key to the performance gain is an indexed graph data structure and a rigorous reduction of costly copying operations. As a sidenote, we like to point out that this approach is also used to speed up the computation of conflicting transitions in partial order reduction techniques [3, 5, 8] used to reduce the state space.

2 Terminology

Definition 1 (Place/Transition Net). *A* place/transition net*, P/T net for short, consists of a finite set of* places P*, a finite set of* transitions T *where* $P \cap T = \emptyset$*, a set of* arcs $F \subseteq (P \times T) \cup (T \times P)$*, a* weight function $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ *where* $W(x,y) = 0$ *if and only if* $(x,y) \notin F$*, and an* initial marking m_0*, where a* marking *is a mapping* $m : P \longrightarrow \mathbb{N}$.
For a node $x \in P \cup T$*,* $^\bullet x = \{y \mid (y,x) \in F\}$ *and* $x^\bullet = \{y \mid (x,y) \in F\}$ *are its pre- resp. post-set.*

The behavior of a P/T net is defined by the *transition rule*.

Definition 2 (Transition rule of a P/T net). *Let* $N = [P, T, F, W, m_0]$ *be a P/T net. Transition* $t \in T$ *is* enabled *in marking* m *if, for all* $p \in P$*,* $W(p,t) \leq m(p)$*. The set of all enabled transitions in a marking* m *is denoted as* $T(m)$*. If* t *is enabled in* m*,* t *can* fire*, producing a new marking* m' *where, for all* $p \in P$*,* $m'(p) = m(p) - W(p,t) + W(t,p)$*. This firing relation is denoted as* $m \xrightarrow{t} m'$*.*

Using the transition rule, a P/T net induces the *reachability graph*, also called the *state space* of the P/T net.

Definition 3 (Reachability graph of a P/T net). *The* reachability graph *(M, E) of a P/T net N has a set of vertices M that comprises all markings that are reachable by any firing sequence from the initial marking of N. Every element $m \xrightarrow{t} m'$ of the firing relation ($t \in T$) defines an edge E from m to m' annotated with t.*

3 Preprocessing decreasing and increasing transitions

To speed up building the reachability graph, it is very valuable to preprocess the candidate transitions that can become en- or disabled after firing transitions. In fact, if a transition t fires leading from marking m to m', it may, in the course of this, disable a previously activated transition t', if t consumes from a place $p \in {}^\bullet t \cap {}^\bullet t'$. More precisely, we would like to capture the situation where both, t and t', are enabled in m, hence, (i) $m(p) \geq \max\{W(p,t), W(p,t')\}$, and where t' is not enabled in m', anymore, hence, (ii) $W(p,t') > m'(p) = m(p) - W(p,t) + W(t,p)$. As preprocessing is unaware of the specific markings m and m', we need to combine (i) and (ii) for the condition

$$W(p,t') < \max\{W(p,t), W(p,t')\} - W(p,t) + W(t,p).$$

In such a case, we say that t *decreases* t' or that t' is *decreased by* t and denote this as the binary relation $t \searrow t'$. In the case of $W(p,t) \leq W(p,t')$, the decreasing condition reduces to $W(t,p) < W(p,t)$ and, otherwise, it simply becomes $W(t,p) < W(p,t')$. Taking everything together, we get the following definition:

Definition 4 (Decreasing transitions). *We call a transition $t' \in T$* decreased by *a transition t, if there exists a place $p \in {}^\bullet t \cap {}^\bullet t'$ with $W(t,p) < W(p,t)$ and $W(t,p) < W(p,t')$ and denote this by $t \searrow t'$.*

On the other hand, if $m \xrightarrow{t} m'$, it may also happen that a previously disabled transition t' becomes enabled in m', namely if t produces token on a place $p \in t^\bullet \cap {}^\bullet t'$. In this simpler situation we have

$$0 < W(p,t') \leq m'(p) = m(p) - W(p,t) + W(t,p),$$

which leads to the following definition, right away:

Definition 5 (Increasing transitions). *We call a transition $t' \in T$* increased by *a transition t, if there exists a place $p \in t^\bullet \cap {}^\bullet t'$ with $W(t,p) > W(p,t)$ and $0 < W(p,t')$ and denote this by $t \nearrow t'$.*

By breaking down the concepts of decreasing and increasing to binary relations, we are able to describe all information as graphs.

Definition 6 (Decrease and increase graph). *If N is a P/T net with transition set T then we call the directed graph $D(N) = (T, \searrow)$ with node set T and arc set $\searrow$ the* decrease graph *of N and the directed graph $I(N) = (T, \nearrow)$ with node set T and arc set $\nearrow$ the* increase graph *of N. The pair of decrease- and increase graph is subsequently denoted as $DI(N)$.*

Having preprocessed $DI(N)$, building the reachability graph of a net N can be sped up. In fact, whenever we traverse an edge $m \xrightarrow{t} m'$ from a node m to a new node m' by firing a transition t, we have to determine the set of reachability arcs that are incident to m'. In other words, this means to compute the set $T(m')$ of transitions in N that are enabled in m'. However, as we come from m, we are already in possession of $T(m)$ and, probably, the difference between $T(m)$ and $T(m')$ is not too big. This is where $DI(N)$ helps us to make just a few updates of $T(m)$ in order to get $T(m')$. We begin with the decrease graph $D(N)$ and obtain the neighborhood $d(t)$ of t, which consists of all transitions that may become disabled by firing t. Secondly, we also compute the neighborhood $i(t)$ of t in $I(N)$ for the transitions that may become enabled after firing t. Afterwards, the job of computing $T(m')$ reduces to the following update:

$$\begin{aligned} T(m') = T(m) &- \{t' \in d(t) \mid \exists p \in {}^\bullet t' : W(p,t') > m'(p)\} \\ &+ \{t' \in i(t) \mid \forall p \in {}^\bullet t' : W(p,t') \leq m'(p)\} \end{aligned}$$

As this update considers only the transitions of $d(t) \cup i(t)$, which is usually much smaller than the whole T, investing into the computation of $DI(N)$ pays out.

A second application field of $DI(N)$ is partial order reduction [3, 5, 8], which is based on the observation that concurrent and independent running processes contribute extensively to the state explosion problem, while having only little influence on the property preservation of individual processes. In essence, while building the reachability graph, in each found marking, they all compute a *subset of transitions* and only fire the enabled transitions in it to explore more states. Hence, the state space is reduced. At the heart of the computation of such subsets of transitions, lies the finding of *conflicting transitions*. Two transitions are in conflict if the firing of one of them can disable the other one. We do not want to get to involved into partial order reduction here and settle with the plain proposition, that finding conflicting transitions can be reduced to the concepts of decreasing transitions. More precisely, partial order subsets can formally be defined relative to the reverse graph of $D(N)$.

4 The former computation of the decrease-increase-graph

For any ordering $p_1, p_2, \ldots, p_n$ of the places of the net N, we define for every $i \in \{0, \ldots, n\}$ the subnet N_i that, while containing all transitions T of N, only consists of the places $P_i = \{p_1, \ldots, p_i\}$ and the arcs that go between T and P_i. The former approach to the computation of $DI(N) = DI(N_n)$ is to start with $DI(N_0)$ and then consider the place sequence $p_1, \ldots, p_n$ in order to successively obtain $DI(N_i)$ for all $i \in \{1, \ldots, n\}$.

It is easy to see that $DI(N_{i+1})$ is just $DI(N_i)$ plus the edges induced by p_{i+1}. This happens, as the decreasing and the increasing relations between transitions are defined only existentially over the place set. In other words, if $DI(N_i)$ has a $\searrow$-edge or $\nearrow$-edge, respectively, between two transitions t, t' then considering an additional place p_{i+1} cannot revoke the existence of place $p \in \{p_1, \ldots, p_i\}$ that justified the aforesaid edge between t, t'. For that reason it makes sense to define $\searrow(p_{i+1})$ and $\nearrow(p_{i+1})$, the edges of $DI(N_{i+1})$ that are additionally introduced by the consideration of p.

Definition 7 (Incremental edges). *For all places p of N, the set $\searrow(p) = \{(t,t') \mid W(t,p) < W(p,t), W(t,p) < W(p,t')\}$ is called* decreasing edges *of p and the set $\nearrow(p) = \{(t,t') \mid W(t,p) > W(p,t), 0 < W(p,t')\}$ is the* increasing edges *of p.*

The set $\nearrow(p)$ can be represented in a very simple way: There are always two transition sets $T^0, T^1 \subseteq T$ such that $\nearrow(p) = \{(t,t') \mid t \in T^0, t' \in T^1\}$. In fact, $T^0 = \{t \mid W(t,p) > W(p,t)\}$ and $T^1 = p^\bullet$. We capture this property in the following definition.

Definition 8 (Homogenous pair). *A pair (T^0, T^1) of transition subsets of T is called* $\nearrow$-homogenous *if $t \nearrow t'$ for all $t \in T^0$ and all $t' \in T^1$. This is also denoted as $T^0 \nearrow T^1$. Analogously, it is called* $\searrow$-homogenous *if $t \searrow t'$ for all $t \in T^0$ and all $t' \in T^1$, which is denoted as $T^0 \searrow T^1$.*

If (T^0, T^1) is exactly the $\nearrow$-homogenous pair of $\nearrow(p)$, this is made explicit by writing $T^0 \nearrow_p T^1$.

The set $\searrow(p)$, in turn, is generally not describable as a single $\searrow$-homogenous pair. That is why we used to fall back on the set

$$\dot{\searrow}(p) = \{(t,t') \mid W(t,p) < W(p,t), 0 < W(p,t')\}.$$

As, obviously, $\searrow(p) \subseteq \dot{\searrow}(p)$, this set is weaker but sufficient for the anticipated purpose. Moreover, $T^0 = \{t \mid W(t,p) < W(p,t)\}$ and $T^1 = {}^\bullet p$ provide a $\dot{\searrow}$-homogenous pair for $T^0 \dot{\searrow}_p T^1$, which stands for $\dot{\searrow}(p) = \{(t,t') \mid t \in T^0, t' \in T^1\}$. In practice, we almost always have $\searrow(p) = \dot{\searrow}(p)$, which justifies this simplification.

The graphs of every $DI(N_i)$ are represented by a list of homogenous pairs each. More precisely, $I(N_i)$ is given by a list of $\nearrow$-homogenous pairs. This basically corresponds to a compressed adjacency list representation, where all transitions $t_1, t_2, \dots$ with the same adjacency list T^1 are together in $T^0 = \{t_1, t_2, \dots\}$ and we get $T^0 \nearrow T^1$.

Therefore, in iteration $i+1$, we have to go through all homogenous pairs $T^0_j \nearrow T^1_j$ of $I(N_i)$ and make updates according to $T^0 \nearrow_{p_{i+1}} T^1$ in order to obtain $I(N_{i+1})$. More precisely, in $I(N_{i+1})$ every pair $T^0_j \nearrow T^1_j$ is replaced by the new pairs

$$(T^0_j \setminus T^0) \nearrow T^1_j \quad \text{and} \quad (T^0_j \cap T^0) \nearrow (T^1_j \cup T^1)$$

unless they are empty.

Equivalently, every graph $D(N_i)$ is implemented as a list of $\dot{\searrow}$-homogenous pairs, which have to be modified according to $T^0 \dot{\searrow}_{p_{i+1}} T^1$ for the next step $D(N_{i+1})$.

In our implementation, we use numbers to represent transitions and keep the pair items T^0_j and T^1_j as ordered lists. This make the computation of $T^0_j \setminus T^0$,

$T_j^0 \cap T^0$, and $T_j^1 \cup T^1$ linear time operations, which is fairly efficient. Nevertheless, we are forced to touch every pair, even though most of them are probably not intersected by T^0 or T^1. Hence, in worst case, the computation of $DI(N)$ takes $\mathcal{O}(|P| \cdot |T|^2)$ time, as, for every place, we need to consider $\mathcal{O}(|T|)$ homogenous pairs in $DI(N_i)$ and process each of them in linear time $\mathcal{O}(|T|)$.

5 Sped up computation of the decrease-increase-graph

In our former approach, a lot of time is wasted in updating the adjacency lists. In every iteration, all lists have to be touched while most of them are not even relevant and, in case of an actual update, many copying operations occur. The speed-up idea is to create a more efficient way of determining the necessary updates. Moreover, in order to omit unnecessarily copying arrays around, the actual creation of adjacency lists is postponed until after the iteration of all places. If we want to build the graph $G(N)$, which is either $I(N)$ or $D(N)$, our new approach works in three steps:

1. For a given ordering $p_1, \ldots, p_n$ of the places, we generate the corresponding list of homogenous pairs $T_1^0 \to T_1^1, T_2^0 \to T_2^1, \ldots,$ where $\to$ stands for $\nearrow$ in case of the computation of $I(N)$, or for $\searrow$ if $D(N)$ is about to be built. For that matter, we like to point out that the new computation method of $D(N)$ does not fall back onto $\dot{\searrow}$ but, instead, processes every place into a set of possibly more than one homogenous pair. As the definition of these homogenous pairs is clear at this point, we do not go into the details of their computation in this section.
2. Iterating through the list of homogenous pairs, we progressively create an intermediate directed graph $H(N) = (X, Y, Z, \theta)$ on node sets X and Y, directed edges $Z \subseteq (X \times Y) \cup (Y \times X)$, and a partial function $\theta : T \to X$. This graph means to implicitly encode $G(N)$. Every node in $x \in X$ represents a set $T_x^0 = \{t \in T \mid \theta(t) = x\}$ of transitions with the same adjacency list. The adjacency between the different nodes of X is realized indirectly. More precisely, every $x \in X$ defines a compressed adjacency list by the homogenous pair $T_x^0 \to T_x^1$ with $T_x^1 = \bigcup_{(x,y) \in Z} \bigcup_{(y,x') \in Z} \{t' \mid \theta(t') = x'\}$. Altogether, every adjacency list of the target graph $G(N)$ is represented by one node of X. The efficient computation of $H(N)$ is detailed below.
3. In the final step, we take $H(N)$ and extract the compressed adjacency lists of $G(N)$ according to the definition above. After computing an inverse mapping of function θ, this is straightforward and does not need further explanation.

The computation of $H(N)$ works iteratively. We assume that we have computed the list of homogenous pairs $T_1^0 \to T_1^1, T_2^0 \to T_2^1, \ldots, T_k^0 \to T_k^1$ from the list of places. Then we start from an empty graph $H_0(N)$ and, step-by-step, integrate every pair $T_{j+1}^0 \to T_{j+1}^1$ into $H_j(N)$ to get $H_{j+1}(N)$ and, at the end, $H(N) = H_k(N)$. Accordingly, every partial solution $H_j(N)$ encodes a graph $G_j(N)$ with all the edges of $T_1^0 \to T_1^1, \ldots, T_j^0 \to T_j^1$.

Then, when integrating the pair $T^0_{j+1} \to T^1_{j+1}$, we firstly have to process all the nodes $x_1, \ldots, x_r, x'_1, \ldots, x'_s \in X$ where T^0_j properly intersects $T^0_{x_i}, i \in \{1, \ldots, r\}$ or T^1_j properly intersects $T^0_{x'_i}, i \in \{1, \ldots, s\}$. More precisely, we have to make copies $\hat{x}_1, \ldots, \hat{x}_r, \hat{x}'_1, \ldots, \hat{x}'_s$ having the same neighborhoods in Y as the original nodes. Moreover, we need to redefine $\theta(t) = \hat{x}_i$ for all $t \in T^0_j \cap T^0_{x_i}$ and $\theta(t') = \hat{x}'_i$ for all $t' \in T^1_j \cap T^0_{x'_i}$. This is necessary as, in contrast to the transitions $T^0_{x_i} \setminus T^0_j$, all elements of $T^0_{\hat{x}_i}$ will obtain an enhanced neighborhood in $G_{j+1}(N)$, namely T^1_{j+1}. Similarly, the transitions of $T^0_{\hat{x}'_i}$ have to be divided from $T^0_{x'_i} \setminus T^1_j$, as only they are neighbors of T^0_j.

Up to this point, however, $H_{j+1}(N)$ still represents $G_j(N)$ as no edge of $T^0_{j+1} \to T^1_{j+1}$ has been included. To this end, we add a new node $x \in X$ and define θ for $T^0_x = T^0_{j+1} \setminus (T^0_{x_1} \cup \cdots \cup T^0_{x_r})$. We also add a new node $x' \in X$ and define θ for $T^0_{x'} = T^1_{j+1} \setminus (T^0_{x'_1} \cup \cdots \cup T^0_{x'_s})$. Moreover, we add a new node $y \in Y$ for the connection $T^0_{j+1} \to T^1_{j+1}$. To implement this homogenous pair in $G_{j+1}(N)$, we, thus, include the edges (x, y) and $(\hat{x}_i, y), i \in \{1, \ldots, r\}$ into $H_{j+1}(N)$ as well as all the edges (y, x') and $(y, \hat{x}'_i), i \in \{1, \ldots, s\}$.

The reason that computing $H(N)$ is much faster than our old method, comes from the fact that finding the intersected nodes $x_1, \ldots, x_r, x'_1, \ldots, x'_s$ does not work by exhaustive search as before. Instead, we just once iterate through the elements t of T^0_{j+1} and T^1_{j+1}, respectively, and use $\theta(t)$ to get the intersected set. Moreover, having the intersected sets, we do not split entire transition arrays but copy only small amounts of edges between X and Y. But while the computation of $H(N)$ works in $\mathcal{O}(|P| \cdot |T|)$ time, the expansion of $H(N)$ into the compressed form of $G(N)$ can still take $\mathcal{O}(|P| \cdot |T|^2)$ time in the worst case. However, that the new approach is usually the better one, even with the overhead of the subsequent adjacency list extraction, is demonstrated experimentally in the next section.

6 Experimental validation and conclusion

Both methods discussed in the paper are implemented in our explicit model checker LoLA 2 [9]. For evaluating the methods, we used the benchmark provided by the model checking contest 2019 [2]. The benchmark consists of 94 Petri nets, also called models, which result in 1018 instances due to the scaling parameter of some models. We restrict the benchmark to P/T nets and for each net we only consider the largest available instance. If the model scales over more than one parameter, we choose for every parameter the largest instance. We ignored smaller instances since they have proportionally the same effect as their larger counterparts. For the "FamilyReunion" net we choose a smaller instance, since the largest instance runs out of memory on our test machine, while computing the former method. Overall our benchmark consists of 100 models.

Experiments were executed on a machine with 32 physical cores running at 2.7 GHz and 1 TB of RAM. All computations were done with no time and memory restrictions. The time was measured with the C++ chrono library using the high resolution clock. We only show experiments where one of the methods

Table 1. Time comparison in seconds between the former (here called old) and the new method to compute enabledness and conflicts.

	Model size			Decr.		Incr.		Difference		
Model	**|P|**	**|T|**	**|F|**	**Old**	**New**	**Old**	**New**	**Decr.**	**Incr.**	**Both**
AirplaneLD-PT-4000	28019	32008	122028	87.9	2.2	49.2	0.5	85.7	48.7	134.4
ASLink-PT-10b	4410	5405	16377	0.1	0.0	1.3	0.0	0.0	1.3	1.3
AutoFlight-PT-96b	7894	7868	18200	0.2	0.0	3.1	0.0	0.2	3.1	3.3
BART-PT-60	8130	12120	97200	7.8	0.1	3.9	0.1	7.7	3.8	11.5
BridgeAndVehicles-PT-V80P50N50	228	8588	67470	3.2	1.0	0.0	0.1	2.2	-0.1	2.1
CloudDeployment-PT-7b	2271	19752	389666	0.2	38.5	0.6	0.0	-38.3	0.6	-37.7
DatabaseWithMutex-PT-40	12920	12800	156800	0.3	0.1	9.7	0.1	0.2	9.6	9.8
Dekker-PT-200	1000	40400	320400	21.7	1.4	0.2	0.4	20.4	-0.2	20.2
DLCflexbar-PT-8a	3971	32571	129321	126.5	0.0	0.7	0.1	126.5	0.6	127.1
DLCflexbar-PT-8b	47560	76160	216499	9.8	0.4	335.2	0.2	9.4	335.1	344.5
DLCround-PT-13b	5343	8727	24849	0.1	0.0	2.2	0.0	0.1	2.2	2.3
DLCshifumi-PT-6a	3568	25936	101182	61.9	0.0	0.6	0.1	61.9	0.5	62.4
DLCshifumi-PT-6b	44243	66611	182532	8.3	0.3	260.2	0.1	8.0	260.1	268.1
DoubleExponent-PT-200	10604	9998	28194	0.2	0.0	5.5	0.0	0.2	5.5	5.7
DrinkVendingMachine-PT-10	120	111160	1026520	38.0	22.7	6.8	0.1	15.3	6.7	21.9
FamilyReunion-PT-L00200M0020C010P010G005	143908	134279	411469	36.7	0.5	2450.4	0.5	36.1	2450.0	2486.1
FlexibleBarrier-PT-22b	6478	7469	18797	0.1	0.0	1.8	0.0	0.1	1.7	1.8
GlobalResAllocation-PT-5	102	136662	1226388	1.9	0.7	1.0	0.2	1.2	0.8	2.0
HexagonalGrid-PT-816	3391	6174	24696	0.0	0.0	1.4	0.0	0.0	1.4	1.4
HypertorusGrid-PT-d5k3p2b10	7533	24300	97200	0.2	0.0	15.9	0.1	0.2	15.8	16.0
JoinFreeModules-PT-5000	25001	40001	115002	5.2	0.1	56.9	0.1	5.1	56.8	62.0
NeighborGrid-PT-d5n4m1t35	1024	196608	393216	0.6	0.1	1.9	2.1	0.5	-0.3	0.3
NeoElection-PT-8	10062	22266	129195	2.2	1.4	2.9	0.1	0.8	2.8	3.5
NoC3x3-PT-8B	9140	14577	30726	0.4	0.0	5.3	0.0	0.4	5.2	5.6
PhaseVariation-PT-D30CS100	2702	30977	216835	23.7	0.9	0.8	1.5	22.7	-0.7	22.0
Philosophers-PT-10000	50000	50000	160000	195.2	0.2	143.5	0.1	195.0	143.4	338.4
PhilosophersDyn-PT-20	540	17220	140780	3.8	0.8	1.5	0.4	3.0	1.1	4.2
Railroad-PT-100	1018	10506	62728	1.6	0.3	0.9	0.2	1.4	0.6	2.0
RERS17pb113-PT-9	639	31353	125418	5.6	0.5	1.4	0.3	5.1	1.1	6.2
RERS17pb114-PT-9	1446	151085	604252	107.5	6.2	27.1	4.5	101.2	22.6	123.9
RERS17pb115-PT-9	1399	144369	577414	90.8	7.4	30.4	4.5	83.3	25.9	109.3
RwMutex-PT-r2000w0010	6020	4020	52040	1.2	0.0	1.6	0.0	1.2	1.6	2.8
SafeBus-PT-20	1026	10461	77364 âĂŞ	1.0	0.5	0.2	0.3	0.6	-0.1	0.5
SharedMemory-PT-200	40601	80200	320000	2992.3	45.1	3821.9	38.6	2947.2	3783.4	6730.6
TokenRing-PT-50	2601	127551	510204	840.1	1.4	3.1	0.8	838.8	2.3	841.1

needed more than one second. Table 1 lists the results of our experiments. It shows that the new method is in almost all cases at least one order of magnitude faster. Although both methods are asymptotically of the same complexity, it seems that they have different worst case inputs. Our experiments suggest that the ones that slow down the new approach are less frequent in practice than those making the former one slow.

In the remainder of this section, we want to take a closer look at the outcome of our experiments. As Table 1 shows for the decreasing computation, both methods needed less than 1 ms for 29 of all models. In 7 models the former method was faster, whereas in 6 of them the difference amounted to only a couple of milliseconds. In the remaining 64 models the new approach was faster. All in all the new approach needed only 2.86 % of the time compared to the former method. Even if we leave out the biggest outliers the new approach needs less than 10 % of the time.

The picture for the increasing computation is even better. There are 19 models where both methods needed less than 1 ms. In 8 models the former method was faster, but in all of these cases the difference was also only a couple of milliseconds. In the remaining 73 models the new method was faster. All in all the new approach needed only 0.79 % of the time compared to the former method. And if we again leave out the biggest outliers the new approach still needs less than 10 % of the time.

There are only three models, where the new method is slower. In two of them, the difference is only a couple of milliseconds. But for the CloudDeployment-PT-7b model the decrease computation needed more than 38 seconds, while the former approach needed not even one. We thoroughly investigated this case, but we could not find any useful hints, why this model performs completely different than the rest.

Starting from the 2020 edition of the MCC, LoLA 2 will use the new method and with this the model checking performance will increase, since more time is available for the actual verification. Especially in large Petri nets there will be a lot more time available to validate a given specification. The new approach addresses mainly the runtime performance. In the future we also plan to use $DI(N)$ for the reduction of memory used to save the decreasing and increasing transitions for each transition.

References

1. José Luis Briz and José Manuel Colom. Implementation of weighted place/transition nets based on linear enabling functions. In *Proc. PETRI NETS, LNCS 815*, pages 99–118, 1994.
2. Fabrice Kordon et al. Presentation of the 9th edition of the model checking contest. In *Proc. TACAS, LNCS 11429*, pages 50–68, 2019.
3. P. Godefroid and P. Wolper. A partial approach to model checking. *Inf. Comput.*, 110(2):305–326, 1994.
4. Marko Mäkelä. Optimising enabling tests and unfoldings of algebraic system nets. In *Proc. PETRI NETS, LNCS 2075*, pages 283–302, 2001.

5. D. A. Peled. All from one, one for all: on model checking using representatives. In *Proc. CAV, LNCS 697*, pages 409–423, 1993.
6. Karsten Schmidt. Lola: A low level analyser. In *Proc. PETRI NETS, LNCS 1825*, pages 465–474, 2000.
7. Dirk Taubner. On the implementation of petri nets. In *Advances in Petri Nets, LNCS 340*, volume 340, pages 418–434, 1987.
8. A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets, LNCS 483*, pages 491–515, 1989.
9. K. Wolf. Petri net model checking with LoLA 2. In *Proc. PETRI NETS, LNCS 10877*, pages 351–362, 2018.