Saturation Enhanced with Conditional Locality: Application to Petri Nets

Vince Molnár[1,2] and István Majzik[1]

[1] Fault Tolerant Systems Research Group, Department of Measurement and Information Systems, Budapest University of Technology and Economics
[2] MTA-BME Lendület Cyber-Physical Systems Research Group

Abstract. The saturation algorithm for symbolic state space generation has proved to be an efficient way to tackle the state space explosion problem in the verification of concurrent, asynchronous systems. Since its original publication in 2001, several variants and extensions have been introduced. The reason for altering the algorithm in these variants is often specific to how it handles transitions. Saturation heavily relies on the notion of locality: transitions tend to affect only some of the state variables. The saturation effect, however, can be achieved and even enhanced with a weaker notion of locality, which we call conditional locality. In this paper, we define a generalized version of the saturation algorithm (GSA) for multi-valued decision diagrams that works with conditional locality and show that it enables the direct usage of transition relations that previously required a specialized algorithm such as variants of constrained saturation. Focusing on Petri nets, we also empirically demonstrate on models of the Model Checking Contest that the GSA often outperforms the original saturation algorithm whenever conditional locality can be exploited and has virtually no overhead for other models.

Keywords: Generalized saturation · Symbolic model checking · Formal verification · Conditional locality

1 Introduction

Model checking is a formal verification technique that looks for specified behavioral patterns in a discrete-state system by exploring its state space. Even though we can sometimes avoid the full exploration of the state space, the huge number of reachable states in non-trivial systems tend to limit the applicability of model checking. Concurrent, asynchronous systems are especially problematic for approaches based on a total ordering of events, because the interleaving of the behavior of independent components can easily cause a combinatoric explosion.

This problem has been tackled in many ways, one of which is symbolic model checking with decision diagrams. Decision diagrams can efficiently encode large state spaces by exploiting the regularities between states. Furthermore, the symbolic encoding of states and transitions enables the efficient computation of next states by working with sets of states and relations. Even though this technique

was a great step forward [2], simpler exploration strategies like breadth-first search suffered from the large size of intermediate decision diagrams.

Decision diagrams have an interesting property: their size is not proportional to the number of encoded states. In fact, after some point, adding more states will *reduce* the size of the diagram because more and more regularities will be introduced. This is what the *saturation algorithm*, first introduced in [3], exploits.

To saturate means *to fill completely.* The main idea of the algorithm is to saturate smaller parts of a decision diagram before moving on to larger parts. Specifically, saturation processes decision diagrams in a bottom-up fashion, exploring the state space starting with transitions that do not require any component whose state variable is higher in a variable ordering than the processed level – transitions that are *local* on the currently processed variables. This is often possible because in concurrent systems, transitions usually affect only a small number of the components. With this strategy, saturation can keep all subdiagrams empirically small.

The saturation algorithm initially required the Kronecker condition of the transition relation, but this restriction was removed when [9] introduced *matrix-diagrams* and [5] did the same with decision diagrams with 2 levels per variable.

Efficient saturation-based model checking for computational-tree logic (CTL) has been introduced in [11]. The main novelty was the introduction of *constrained saturation*, which provides an efficient way to handle constraints on the state space. Constrained saturation takes a set of states – the constraint – and performs state space exploration without leaving the constraint. With the modified algorithm, it is possible to avoid intersecting the transition relations with the constraint, which preserves the locality of transitions and the beneficial properties of saturation.

Building on constrained saturation, [10] introduced further extensions to support the verification of linear temporal logic (LTL) and [7] proposed a new approach for the model checking of prioritized Petri nets. Both of them proposed ways to preserve locality for a transition relation that is composed of simple transitions and additional constraints (such as synchronization between the system and the property automaton or enabledness based on priorities).

In this paper, we propose a new algorithm for saturation that generalizes the attempts of preserving locality in the approaches above. We introduce *conditional locality* to relax the original notion of locality and automatically handle transition relations that previously required a form of constrained saturation to process efficiently (such as [7, 10]). In addition to generalizing a family of algorithms, using conditional locality can increase the saturation effect, which is intuitively associated with better performance. We investigate this effect in the context of Petri nets, where we empirically show that the *generalized saturation algorithm* (GSA) can be orders of magnitude faster than the original saturation algorithm (presented in detail in [4]) and is virtually never slower.

The main motivation of conditional locality is to compute fixed points even more locally. Saturation ignores variables that are independent of the processed events to avoid computing the fixed point for each of their valuations. With

conditional locality, we can ignore even those variables that are not written but read by an event (because they will not change), but compute the fixed point as many times as the value of those variables would cause a different result. The intuition is that the resulting nodes will be part of the final decision diagram more often than those created by the original saturation algorithm, leading to less intermediate nodes and therefore improved performance.

The most important related work is [9], where the authors propose a method to *split* a transition-relation such that the resulting relations are as local as possible. The key idea is to extract relations which do not depend on the variables higher in the variable ordering and therefore the method works well when the transition is a "sum" of such a relation and another one (i.e. $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$). Our approach also handles the cases when the transition is the result of "removing" certain cases from a transition that normally does not depend on a variable (i.e. $\mathcal{R} = \mathcal{R}_1 \setminus \mathcal{R}_2$). Another work that is similar in spirit is [8], where the dependencies of high-level transitions on state variables are more fine-grained than *dependent* and *independent*, which enables a more compact encoding and more efficient update of the transition relation. Our approach also refines this dependency relation to relax the notion of locality.

The key novelties introduced in this paper are the following: *1)* the introduction of *conditional locality* to relax the original notion of locality; *2)* the *generalization* of a family of saturation-based algorithms using conditional locality; and *3)* an *empirical demonstration* of the efficiency of the proposed approach on Petri nets. The paper is structured as follows. Section 2 presents the formalisms and notations used in the rest of the paper. Section 3 introduces conditional locality and the generalized saturation algorithm. The empirical evaluation on Petri nets is in Section 4. Finally, Section 5 concludes the paper.

2 Background

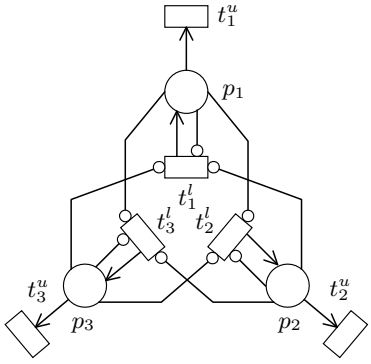
In this section we summarize the theoretical background of our work and introduce the necessary notations. First, we briefly present Petri nets in Section 2.1, then we introduce partitioned transition systems in Section 2.2. Building on the latter, we define locality in Section 2.3, then formalize multi-valued decision diagrams for encoding states (Section 2.4) and abstract next-state diagrams for encoding transition relations (Section 2.5). Finally, we present the saturation and constrained saturation algorithms in Section 2.6.

2.1 Perti Nets

Petri nets are a widely used formalism to model concurrent, asynchronous systems. The formal definition of a Petri net (including inhibitor arcs) is as follows (see Figure 1 for an illustration of the notations).

Definition 1 (Petri net). *A* Petri net *is a tuple* $PN = (P, T, W, M_0)$ *where:*

- P *is the set of* places *(defining state variables);*



$P = \{p_1, p_2, p_3\} = V$; $K = 3$; ordering: $x_k = p_k$; $M_0 = (p_1 \leftarrow 0, p_2 \leftarrow 0, p_3 \leftarrow 0)$; $D(p_i) = \{0, 1\}$; $S^0 = \{(0,0,0)\}$; $T = \{t_1^l, t_1^u, t_2^l, t_2^u, t_3^l, t_3^u\} = \mathcal{E}$; $W^-(t_i^u, p_i) = W^+(t_i^l, p_i) = W^\circ(t_i^l, p_j) = 1$ (other weights are 0); $\mathcal{N}_{t_1^l} = \{\big((0,0,0),(1,0,0)\big)\}$;

Examples: t_1^l is locally read-only and t_1^u is locally invariant on p_2 and p_3; $Supp(t_1^u) = \{p_1\}$, $Top(t_1^u) = p_1$, $Supp(t_1^l) = \{p_1, p_2, p_3\}$, $Top(t_1^l) = p_3$.

Examples: t_1^l is conditionally local over $\{p_1\}$ with respect to $(0,0)_{(\{p_2,p_2\})}$; $Supp_c(t_1^l) = \{p_1\}$, $Guard(t_1^l) = \{p_2, p_3\}$, $Top_c(t_1^l) = p_1$.

Fig. 1: Petri net model of 3 concurrent processes locking (t_i^l) and unlocking (t_i^f) a mutually exclusive resource. Examples for the interpretations of the various notations introduced in Sections 2.1–2.3 and 3.1 are given on the right.

- T *is the set of* transitions *(defining behavior) such that* $P \cap T = \emptyset$*;*
- $W = W^- \cup W^+ \cup W^\circ$ *is a multiset of three types of* arcs *(the* weight function*), where* $W^-, W^\circ : P \times T \to \mathbb{N}$ *and* $W^+ : T \times P \to \mathbb{N}$ *are the set of input arcs, inhibitor arcs and output arcs, respectively;*
- $M_0 : P \to \mathbb{N}$ *is the* initial marking*, i.e. the number of* tokens *on each place.*

The three types of weight functions describe the structure of the Petri net: there is an input or output arc between a place p and a transition t iff $W^-(p,t) > 0$ or $W^+(t,p) > 0$, respectively, and there is an inhibitor arc iff $W^\circ(p,t) < \infty$.

The state of a Petri net is defined by the current marking $M : P \to \mathbb{N}$. The dynamic behavior of a Petri net is described as follows. A transition t is *enabled* iff $\forall p \in P : M(p) \in \big[W^-(p,t), W^\circ(p,t)\big)$. Any enabled transition t may fire non-deterministically, creating the new marking M' of the Petri as follows: $\forall p \in P : M'(p) = M(p) - W^-(p,t) + W^+(t,p)$. We denote the firing of transition t in marking M resulting in M' with $M \xrightarrow{t} M'$. A marking M_i is *reachable* from the initial marking if there exists a sequence of markings such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_i} M_i$. The set of reachable markings (i.e. the *state space* of the Petri net) is denoted by $\mathcal{S}_r$. This work assumes $\mathcal{S}_r$ to be finite.

2.2 Partitioned Transition Systems

A generic model for saturation is usually called a *partitioned transition system* (PTS), where high-level events (causing transitions) and their dependencies on state variables are preserved to partition the low-level next-state relations and localize the effect of transitions [4]. In decision diagram-based model checking, such models usually come with a user-specified variable ordering.

Definition 2 (Variable ordering). *A variable ordering over variables V ($|V| = K$) is a total ordering of elements of V that defines a sequence. The kth variable in the sequence is denoted by x_k. We will say that x_1 is the* lowest *and x_K is the* highest *in the ordering. We will use the notations $V_{\leq k} = \{x_1, \ldots, x_k\}$ and $V_{>k} = \{x_{k+1}, \ldots, x_K\}$ for sets of variables constituting a prefix or suffix (respectively) of the sequence.*

With a specified variable ordering, the formal definition of a PTS is as follows (again see Figure 1 for an illustration on the example model).

Definition 3 (Partitioned Transition System). *A partitioned transition system is a tuple $M = (V, D, S^0, \mathcal{E}, \mathcal{N})$ where:*

- *$V = \{x_1, \ldots, x_K\}$ is the finite set of variables with an arbitrary but well-defined variable ordering;*
- *D is the domain function such that $D(x_k) \subseteq \mathbb{N}$ for all $x_k \in V$;*
- *$S^0 \subseteq \hat{S}$ is the set of initial states, where $\hat{S} = \bigtimes_{x_k \in V} D(x_k)$ is the potential state space (the shape of which is unaffected by the chosen variable ordering);*
- *$\mathcal{E}$ is the set of high-level events, specifying groups of individual transitions;*
- *$\mathcal{N} \subseteq \hat{S} \times \hat{S}$ is the transition relation partitioned by $\mathcal{E}$ such that $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$. We often use $\mathcal{N}$ as a function returning the "next states": $\mathcal{N}(\mathbf{s}) = \{\mathbf{s}' | (\mathbf{s}, \mathbf{s}') \in \mathcal{N}\}$ and $\mathcal{N}(S) = \bigcup_{\mathbf{s} \in S} \mathcal{N}(\mathbf{s})$.*

A (concrete) state of the system is a vector $\mathbf{s} \in \hat{S}$, where each variable x_k has a value from the corresponding domain: $\mathbf{s}[k] \in D(x_k)$. A partial state over variables X is a vector assigning a specific value to variables in X and $\top$ (undefined) to those in $V \setminus X$. Sets of partial states are denoted by $S_{(X)}$ and when significant, a single partial state is denoted by $\mathbf{s}_{(X)}$. A partial state $\mathbf{s}_{(X)}$ matches *a concrete state $\mathbf{s}$ if $\mathbf{s}[k] = \mathbf{s}_{(X)}[k]$ for every $x_k \in X$, denoted by $\mathbf{s} \in \mathcal{M}(\mathbf{s}_{(X)})$.*

2.3 Locality

Exploiting the information preserved in a PTS, we can define different relationships between an event and a variable (illustrated in Figure 1).

Definition 4 (Locally read-only). *An event α is* locally read-only *on variable x_k if for any $(\mathbf{s}, \mathbf{s}') \in \mathcal{N}_\alpha$ we have that $\mathbf{s}[k] = \mathbf{s}'[k]$. Informally, the value of x is never modified by the transitions of event α.*

While the locally read-only property guarantees that the value of the variable will not change, the event can still depend on the information stored in the variable. The following property forbids this as well.

Definition 5 (Locally invariant). *An event α is* locally invariant *on variable x_k if it is locally read-only and for any $(\mathbf{s}, \mathbf{s}') \in \mathcal{N}_\alpha$ and $v \in D(x_k)$ we also have $(\mathbf{s}_{[x_k \leftarrow v]}, \mathbf{s}'_{[x_k \leftarrow v]}) \in \mathcal{N}_\alpha$, where $\mathbf{s}_{[x_k \leftarrow v]}$ is a state where the value of variable x_k is v, but all other variables have the same value as in $\mathbf{s}$. Informally, the value of x does not affect the outcome of event α.*

With the help of local invariance, we can now define locality, the central notion of the saturation algorithm.

Definition 6 (Locality). *An event $\alpha \in \mathcal{E}$ is said to be* local *over variables $X \subseteq V$ if it is locally invariant on variables in $V \setminus X$. If X is minimal (i.e. the event is dependent on variables in X) then we say that X is the set of* supporting variables *of α: $Supp(\alpha) = X$. The variable with the highest index among the supporting variables (according to a variable order) is the* top *variable ($Top(\alpha)$) of α. We use $\mathcal{E}_k = \{\alpha \mid Top(\alpha) = x_k\}$ and $\mathcal{N}_k = \bigcup_{\alpha \in \mathcal{E}_k} \mathcal{N}_\alpha$ to denote events and their next-state relations whose top variable is the x_k.*

The next-state relation of an event α local on variables $Supp(\alpha) = X$ can be defined over partial states $S_{(X)}$, because no other information is required to compute its image. This enables a compact representation and clever iteration strategies like saturation.

2.4 State Space Encoded in Multi-Valued Decision Diagrams

Saturation works with different types of decision diagrams. This paper addresses the version that uses multi-valued decision diagrams to encode the state space.[3]

Definition 7 (Multi-valued decision diagram). *An ordered quasi-reduced multi-valued decision diagram (MDD) over a set of variables V ($|V| = K$), a variable ordering domains D is a tuple $(\mathcal{V}, lvl, children)$ where:*

- *$\mathcal{V} = \bigsqcup_{i=0}^{K} \mathcal{V}_i$ is the set of* nodes*, where items of $\mathcal{V}_0$ are the* terminal *nodes* **1** *and* **0***, the rest ($\mathcal{V}_{>0} = \mathcal{V} \setminus \mathcal{V}_0$) are* internal *nodes;*
- *$lvl : \mathcal{V} \to \{0, 1, \ldots, K\}$ assigns non-negative* level numbers *to each node, associating them with variables according to the variable ordering (nodes in $\mathcal{V}_k = \{n \in \mathcal{V} \mid lvl(n) = k\}$ belong to variable x_k for $1 \leq k \leq K$ and are terminal nodes for $k = 0$);*
- *$children : \mathcal{V}_{>0} \times \mathbb{N} \to \mathcal{V}$ defines edges between nodes labeled with elements of $\mathbb{N}$ (denoted by $n[i] = children(n, i)$, $n[i]$ is left-associative), such that for each node $n \in \mathcal{V}_{>0}$ and value $i \in \mathbb{N} : lvl(n[i]) = lvl(n) - 1$ or $n[i] = \mathbf{0}$;*
- *for every pair of nodes $n, m \in \mathcal{V}_{>0}$, if for all $i \in \mathbb{N} : n[i] = m[i]$, then $n = m$.*

An MDD node $n \in \mathcal{V}_k$ encodes a set of partial states $S(n) = S_{(V_{\leq k})}$ over variables $V_{\leq k}$ such that for each $\mathbf{s} \in S_{(V_{\leq k})}$ the value of $n[\mathbf{s}[k]] \cdots [\mathbf{s}[k]]$ (recursively indexing n with components of $\mathbf{s}$) is $\mathbf{1}$ and for all $\mathbf{s} \notin S_{(V_{\leq k})}$ it is $\mathbf{0}$.

There are efficient recursive algorithms that compute the result of set operations directly on MDDs (e.g. union is described in [4]).

An interesting property of MDDs is that the number of nodes does not grow proportionally with the size of the encoded set. In fact, the size of an MDD can decrease when adding new states because of the exploited regularities. This phenomenon can be observed on Figure 4, where each MDD from left to right encodes one more state, but has either 3 internal nodes or 5. Also note that the right-most MDD encodes the sate space of the Petri net from Figure 1.

[3] See [6] for saturation with hierarchical set decision diagrams.

2.5 Next-State Representations

We have introduced a generalization of next-state representations compatible with saturation in [7] – we will build on this notion heavily in the generalization of saturation variants.

Definition 8 (Abstract next-state diagram). *An* abstract next-state diagram *over a set of variables V ($|V| = K$) and corresponding domains D is a tuple $(\mathcal{D}, lvl, next)$*

- *$\mathcal{D} = \sqcup_{i=0}^{K} \mathcal{D}_i$ is the set of* next-state descriptors *(NS descriptor or descriptor for short), where items of $\mathcal{D}_0$ are the terminal identity $\mathbf{1}$ and the terminal empty $\mathbf{0}$ descriptors, the rest ($\mathcal{D}_{>0} = \mathcal{D} \setminus \mathcal{D}_0$) are* non-terminal *descriptors;*
- *$lvl : \mathcal{D} \to \{0, 1, \ldots, K\}$ assigns non-negative* level numbers *to each descriptor, associating them with variables (descriptors in $\mathcal{D}_k = \{d \in \mathcal{D} \mid lvl(d) = k\}$ belong to variable x_k for $1 \leq k \leq K$ and are terminal nodes for $k = 0$);*
- *$next : \mathcal{D} \times \mathbb{N} \times \mathbb{N} \to \mathcal{D}$ is the indexing function that given a descriptor d and a pair of "before" and "after" variable values returns another descriptor d' such that $lvl(d') = lvl(d) - 1$ or $d' = \mathbf{0}$. Also denoted by $d[v, v'] = d' \Leftrightarrow (d, v, v', d') \in next$ (with $d, d' \in \mathcal{D}$, $v, v' \in \mathbb{N}$, $d[v, v']$ is left-associative) and $d[\mathbf{s}, \mathbf{s}'] = d[v_K, v'_K] \cdots [v_1, v'_1]$. We require for any $v, v', v'' \in \mathbb{N}$ and $v \neq v'$ that $\mathbf{1}[v, v] = \mathbf{1}$, $\mathbf{1}[v, v'] = \mathbf{0}$, and $\mathbf{0}[v, v''] = \mathbf{0}$.*

The abstract NS descriptor $d \in \mathcal{D}_k$ encodes the relation $\mathcal{N}(d) \subseteq \mathbb{N}^K \times \mathbb{N}^K$ iff for all $\mathbf{s}, \mathbf{s}' \in \mathbb{N}^K$ the following holds:

$$\big((\mathbf{s}, \mathbf{s}') \in \mathcal{N}(d) \Leftrightarrow d[\mathbf{s}, \mathbf{s}'] = \mathbf{1}\big) \wedge \big((\mathbf{s}, \mathbf{s}') \notin \mathcal{N}(d) \Leftrightarrow d[\mathbf{s}, \mathbf{s}'] = \mathbf{0}\big)$$

Decision diagram-based representations such as MDDs with $2K$ levels or matrix decision diagrams naturally implement abstract next-state diagrams – descriptors are nodes of the diagram, the identity descriptor is the terminal one node ($\mathbf{1}$), the empty descriptor is the terminal zero node ($\mathbf{0}$) and the indexing is the same (in case of MDDs with $2K$ levels $d[x, x']$ is implemented by $d[x][x']$). The main difference between these representations and abstract next-state diagrams is that the latter are *abstract* – they can have any representation as long as it can be mapped to the definition and they can be compared for equality.

In case of Petri nets, the simplest representation is the weight function of the net. Given a Petri net with $K = |P|$ places each constituting a separate state variable (p_k denoting the kth variable in the ordering encoding the number of tokens on place $p \in P$), a mapping to an abstract next-state diagram for every transition $t \in T$ is as follows.

- The set of descriptors is $\mathcal{D}_i = \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathcal{D}_{i-1}$ for $1 \leq i \leq K$, i.e. tuples of the input weight, inhibitor weight and output weight for p_i and the descriptor of for the next place if the transition is enabled with respect to p_i ($\mathcal{D}_0 = \{\mathbf{1}, \mathbf{0}\}$).
- For the *next* function, if $d = (v^-, v^\circ, v^+, d')$, the result of indexing is $d[i, j]$ is d' if $v^- \leq i < v^\circ$ and $j = i - v^- + v^+$ and $\mathbf{0}$ otherwise.

```
Input: MDD node n, c
Output: saturated MDD node n'
1  if n = 0 or n = 1 or c = 0 then
     return n
2  if ¬SatCacheGet(n, c, n') then
3    n' ← newMddNode(lvl(n))
4    for each i where n[i] ≠ 0 do
5      Confirm(lvl(n), i)
6      n'[i] ← ConsSaturate(n[i], c[i])
7    n' ← CheckIn(n')
8    D ← descriptors for E_lvl(n)
9    repeat
10     changed ← false
11     for each d ∈ D do
12       n'' ← ConsFire(n', c, d)
13       if n' ≠ n'' then
14         n' ← n'', changed ← true
15   until ¬changed
16   SatCachePut(n, d, n')
17 return n'
```

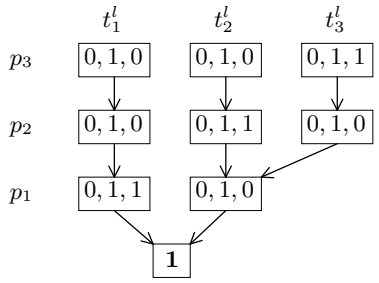
(a) Procedure ConsSaturate.

```
Input: MDD node n, c, NS descriptor d
Output: states in c in or reachable from
        n through d + node saturated
1  if n = 0 or d = 0 then return 0
2  if d = 1 then return n
3  n' ← newMddNode(lvl(n))
4  for each i, j where n[i] ≠ 0 and
     c[j] ≠ 0 and d[i, j] ≠ 0 do
5    s ← ConsRecFire(n[i], c[j], d[i, j])
6    if s ≠ 0 then Confirm(lvl(n), j)
7    n'[j] ← n'[j] ∪ s
8  n' ← CheckIn(n')
9  return n' ∪ n
```

(b) Procedure ConsFire.

```
Input: MDD node n, c, NS descriptor d
Output: states in c reachable from n
        through d + node saturated
1  if n = 0 or d = 0 then return 0
2  if d = 1 then return n ∩ c
3  if ¬RecFireCacheGet(n, c, d, n'')
     then
4    n' ← newMddNode(lvl(n))
5    for each i, j where n[i] ≠ 0 and
       c[j] ≠ 0 and d[i, j] ≠ 0 do
6      s ← ConsRecFire(n[i], c[j], d[i, j])
7      if s ≠ 0 then Confirm(lvl(n), j)
8      n'[j] ← n'[j] ∪ s
9    n' ← CheckIn(n')
10   n'' ← ConsSaturate(n', d)
11   RecFireCachePut(n, c, d, n'')
12 return n''
```

(c) Procedure ConsRecFire.



(d) NS descriptors of t_i^l.

Fig. 2: Pseudocode of constrained saturation and NS descriptors.

– For each transition t the corresponding descriptor is defined recursively: $d_0 = \mathbf{1}$ and $d_i = \big(W^-(p_i, t), W^\circ(p_i, t), W^+(p_i, t), d_{i-1}\big)$.

Figure 2d illustrates the NS descriptors of transitions t_i^l of the example Petri net. A descriptor $d = (v^-, v^\circ, v^+, d')$ is denoted by a node with (v^-, v°, v^+) and d' is denoted by an arrow from d. descriptors can be shared between transitions.

2.6 Saturation

Saturation is a symbolic state space generation algorithm working on decision diagrams [4]. Formally, given a PTS M, its goal is to compute the set of states

S that are reachable from the initial states S^0 through transitions in $\mathcal{N}$: $S = S^0 \cup \mathcal{N}(S^0) \cup \mathcal{N}\big(\mathcal{N}(S^0)\big) \cdots = \mathcal{N}^*(S^0)$, where $\mathcal{N}^*$ is the reflexive and transitive closure of $\mathcal{N}$. This is equivalent to computing the least fixed point $S = S \cup \mathcal{N}(S)$ that contains S^0. The main strength of saturation is a recursive computation of this fixed point, which is based on the following definitions.

Definition 9 (Saturated state space). *Given a partitioned transition system M, a set of (partial) states $S_{(X)}$ over variables X is* saturated *iff $S_{(X)} = S_{(X)} \cup \mathcal{N}_X(S_{(X)})$, where $\mathcal{N}_X = \bigcup_{\alpha | Supp(\alpha) \subseteq X} \mathcal{N}_\alpha$.*

A saturated state space is a fixed point of $\mathcal{N}_X$. In model checking, the goal of state-space exploration is to find a least fixed point $S = S \cup \mathcal{N}(S)$ that contains the initial states S^0. Saturation computes this fixed point recursively based on the structure of a decision diagram.

Definition 10 (Saturated node). *Given a partitioned transition system M, an MDD node n on level $lvl(n) = k$ is* saturated *iff it encodes a set of (partial) states $S(n)$ that is saturated. Equivalently, the node is saturated iff all of its children $n[i]$ are saturated and $S(n) = S(n) \cup \mathcal{N}_k(S(n))$, where $\mathcal{N}_k = \bigcup_{\alpha | \, Top(\alpha) = x_k} \mathcal{N}_\alpha$ for $1 \leq k \leq K$ and $\mathcal{N}_0 = \emptyset$.*

As suggested by the definition, locality is mainly used to compute a *Top* value for each event, which is the lowest level on which fixed point computation involving the event can happen. By definition, the terminal nodes **1** and **0** are saturated because they do not have child nodes and $\mathcal{N}_0$ is empty. The saturation algorithm is then easily defined as a recursive algorithm that given a node n computes the *least* fixed point $S(n_s) = S(n_s) \cup \mathcal{N}_k(S(n_s))$ that contains $S(n)$, making sure that child nodes are always saturated by recursion. When applied on a node encoding the set of initial states, the result will be a node encoding the states reachable through transitions in $\mathcal{N}$.

The motivation of this decision diagram-driven strategy comes from the observation that larger sets may often be encoded in smaller MDDs. By exploring as many variations in the lower variables as possible, intermediate diagrams may be much smaller than in traditional BFS and chaining BFS strategies (also described in [4]). Another intuition is that in an MDD encoding the set of reachable states, all nodes are by definition saturated – therefore it is impractical to create nodes which have unsaturated child nodes. In other words, a saturated node has a chance of being in the final MDD, while an unsaturated one has not.

The Constrained Saturation Algorithm The *constrained saturation algorithm* has been introduced in [11] to limit the exploration inside the boundaries of a predefined set of states (the constraint). Even though this is possible with the original algorithm by removing transitions in $\mathcal{N}$ that end in states not inside the constraint, it would damage the locality of events by making them dependent on additional variables (the event has to decide whether it is leaving the constraint or not). Constrained saturation avoids this by traversing an MDD

representation of the constraint along with the MDD of the state space, and deciding the enabledness of events when firing them.

Formally, given a constraint set C, the goal of constrained saturation is to compute the least fixed point $S = S \cup \big(\mathcal{N}(S) \cap C\big)$ that contains the initial states inside the constraint $S^0 \cap C$. Definitions 9 and 10 are modified as follows.

Definition 11 (Saturated state space with constraint). *Given a partitioned transition system M and a constraint C, a set of (partial) states $S_{(X)}$ over variables X is* saturated *iff $S_{(X)} = S_{(X)} \cup \big(\mathcal{N}_X(S_{(X)}) \cap C\big)$, where $\mathcal{N}_X = \bigcup_{\alpha | Supp(\alpha) \subseteq X} \mathcal{N}_\alpha$.*

Definition 12 (Saturated node with constraint). *Given a partitioned transition system M and a constraint node n_c ($S(n_c) = C$), an MDD node n on level $lvl(n) = k$ is* saturated *iff it encodes a set of (partial) states $S(n)$ that is saturated with respect to constraint C. Equivalently, the node is saturated iff all of its children $n[i]$ are saturated with respect to constraint node $n_c[i]$, and $S(n) = S(n) \cup \big(\mathcal{N}_k(S(n)) \cap C\big)$, where $\mathcal{N}_k = \bigcup_{\alpha | Top(\alpha) = x_k} \mathcal{N}_\alpha$ for $1 \leq k \leq K$ and $\mathcal{N}_0 = \emptyset$.*

The recursive computation of $\mathcal{N}_k(S(n)) \cap C$ is done by simultaneously traversing n with the source states, the descriptor d of $\mathcal{N}_k$ with source and target states, and n_c with target states. Note that n_c does not encode the partial state determined by the path through which recursion reached the current node, but "remembers" just enough to decide if the transition is allowed based only on the rest of the state.

Figures 2a–2c present the pseudocode of the constrained saturation algorithm. To retrieve the pseudocode of the original saturation algorithm, one should assume that at any point $c \neq \mathbf{0}$ and $c[i] \neq \mathbf{0}$ for any i. The pseudocode also contains a stub for the Confirm procedure that serves for the on-thy-fly update of the transition relations whenever new states are found (as described in [4] and enhanced in [8]).

The ConsSaturate procedure starts by checking the terminal cases. Line 2 checks if the same problem has already been solved. Caching – as in all operations on decision diagrams – is crucial to have optimal performance. If there is no matching entry in the cache, the algorithm recursively saturates the children of the input node n, calling Confirm for every encountered local state. The resulting node is checked for uniqueness in line 7 and is replaced by an already existing node if necessary (to preserve MDD canonicity). In line 8, we get the NS descriptors for each event belonging to the current level, then iteratively apply them again and again in lines 9–14 until no more states are discovered – a fixed point is reached. This version of the iteration is called *chaining* and is discussed in [4].

The result of firing an event on a set of states is computed by ConsFire and ConsRecFire. The only differences between them are that ConsRecFire also saturates the resulting node before returning it and also caches it – ConsFire is called as part of a saturation process so this is not necessary. The common

parts (3–7 in ConsFire and 4–8 in ConsRecFire) compute the resulting node by recursively processing their child nodes. It is important to note that the arguments of the recursive call are $n[i]$, $c[j]$ and $d[i, j]$, that is, n is traversed along the source state and c is traversed along the target state. The recursive saturation of the result node in ConsRecFire in line 10 ensures that child nodes of the currently saturated node always stay saturated during the fixed point computation in accordance with Definitions 10 and 12.

3 The Generalized Saturation Algorithm

As we could see, the motivation of the constrained saturation algorithm (and all of its variants like those in [7, 10]) is to handle a modified transition relation without losing locality. This paper generalizes these attempts by introducing the notion of conditional locality, a concept that expresses the most important consideration of all kinds of saturation: computing fixed points as locally (i.e. low in the decision diagram) as possible. This intuition has been discussed in Section 2.6, and the conclusion – that saturated nodes have a chance of being in the final MDD – can be used to improve the definitions to enhance this effect even further, which we do in the *generalized saturation algorithm* (GSA).

3.1 Conditional Locality

The concept of locality enables the saturation algorithm to ignore the value of variables outside the support of the currently processed event because it does not depend on them in any way. The result is that a fixed point can be calculated over partial states, which has to be computed *only once* regardless of the number of matching concrete states. The main motivation of conditional locality is to ignore even those variables that are not written but read by an event and compute the fixed point over even shorter partial states, but as many times as the value of those variables would cause a different result. The intuition is that the resulting nodes will be part of the final MDD more often than those created by the original saturation algorithm, leading to less intermediate nodes and therefore improved performance. Concepts are again illustrated in Figure 1.

Definition 13 (Conditional locality). *An event $\alpha \in \mathcal{E}$ is said to be* conditionally local *over variables X and with respect to* condition variables *Y ($X \cap Y = \emptyset$) iff it is* local *over $X \cup Y$ and locally read-only on variables in Y. If Y is maximal and $X \cup Y = Supp(\alpha)$, then we call $Y = Guard(\alpha)$ the* guard variables *and $X = Supp_c(\alpha)$ the* conditional support *of α. The variable with the highest index among the conditionally supporting variables (according to a variable order) is the* conditional top *variable ($Top_c(\alpha)$) of α.*

The (full) next-state relation of a PTS can be automatically repartitioned based on conditional locality. The resulting partitions (events) will either be locally read-only on a variable or will always change its value (behaviors like "test-and-set" may combine these and be read-only sometimes but change the

value other times – in this case, we can split the next-state relation). A special case of this repartitioning is built into the GSA as described in Section 3.2.

The following definition of conditionally saturated state spaces and MDD nodes can be considered as relaxations of Definitions 9 and 10 based on conditional locality.

Definition 14 (Conditionally saturated state space). *Given a partitioned transition system M, a set of (partial) states $S_{(X)}$ over variables X is conditionally saturated with respect to the partial state $\mathbf{s}_{(Y)}$ ($Y \subseteq V \setminus X$) iff $S' = S' \cup \mathcal{N}_X(S')$, where $S' = \{\mathbf{s}_{(Y)}\} \times S_{(X)}$ and $\mathcal{N}_X = \bigcup_{\alpha \mid Supp_c(\alpha) \subseteq X, Guard(\alpha) \subseteq X \cup Y} \mathcal{N}_\alpha$.*

Note that a set of (partial) states $S_{(X)}$ over variables X that is conditionally saturated with respect to a zero-length state $\mathbf{s}_{(\emptyset)}$ is also saturated over X, therefore the goal is the same as before: generate a minimal, conditionally saturated set of states S with respect to $\mathbf{s}_{(\emptyset)}$ that contains the initial states S^0.

Definition 15 (Conditionally saturated node). *Given a partitioned transition system M, an MDD node n on level $lvl(n) = k$ is* conditionally saturated *with respect to the partial state $\mathbf{s}_{(V_{>k})}$ iff it encodes a set of (partial) states $S(n)$ that is conditionally saturated with respect to $\mathbf{s}_{(V_{>k})}$.*

The equivalent definition in terms of child nodes is now phrased as a theorem.

Theorem 1 (Conditionally saturated node – recursive definition). *Given a partitioned transition system M, an MDD node n on level $lvl(n) = k$ is conditionally saturated with respect to the partial state $\mathbf{s}_{(V_{>k})}$ iff 1) all of its children $n[i]$ are conditionally saturated with respect to $\mathbf{s}_{(V_{>k-1})}$, $\mathbf{s}_{(V_{>k-1})} \in \mathcal{M}(\mathbf{s}_{(V_{>k})})$ and $\mathbf{s}_{(V_{>k-1})}[k] = i$; and 2) $S' = S' \cup \mathcal{N}_k(S')$, where $S' = \{\mathbf{s}_{(V_{>k})}\} \times S(n)$ and $\mathcal{N}_k = \bigcup_{\alpha \mid Top_c(\alpha) = x_k} \mathcal{N}_\alpha$ for $1 \leq k \leq K$ and $\mathcal{N}_0 = \emptyset$.*

Proof. We prove only that a node described in the theorem encodes a conditionally saturated set of states. To prove the fixed point, we have to show that for any state $\mathbf{s} \in \{\mathbf{s}_{(V_{>k})}\} \times S(n)$ we have $\mathcal{N}_{V_{\leq k}}(\mathbf{s}) \subseteq \{\mathbf{s}_{(V_{>k})}\} \times S(n)$. Note that $\mathcal{N}_{V_{\leq k}} = \bigcup_{i=0}^{k} \mathcal{N}_k$ because if $Supp_c(\alpha) \subseteq V_{\leq k}$ then $Top_c(\alpha) \leq k$ and $Guard(\alpha) \subseteq V_{\leq k} \cup V_{>k} = V$ always holds. Assume there is a state $\mathbf{s}' \in \mathcal{N}_{V_{\leq k}}(\mathbf{s})$ that is not in $\{\mathbf{s}_{(V_{>k})}\} \times S(n)$. We know that $(\mathbf{s}, \mathbf{s}') \in \mathcal{N}_l$ for some $l \leq k$. If $l = k$ then we have a direct contradiction with the second requirement of the theorem. If $l < k$, then $\mathbf{s}'[k] = \mathbf{s}[k] = i$, because the transition cannot change the value of x_k. Because the first requirement of the theorem says that $n[i]$ is conditionally saturated with respect to $\mathbf{s}_{(V_{>k-1})}$ as defined above, and $\mathcal{N}_l \subseteq \mathcal{N}_{V_{\leq k-1}}$, it follows that $\mathbf{s}'$ must be in $\{\mathbf{s}_{(V_{>k-1})}\} \times S(n[i]) \subseteq \{\mathbf{s}_{(V_{>k})}\} \times S(n)$.

Based on Theorem 1 and the observation after Definition 14, the set of reachable states is encoded as a conditionally saturated MDD node on level K.

The key difference compared to Definitions 9 and 10 is the inclusion of a partial state with respect to which we can define a fixed point. Because we consider the repartitioned events that are now conditionally local, the partial

state can be used to bind their guard variables, which will specify their effect on the variables in their conditional support. Since the guard variables are not changed when executing the transitions, we can compute a fixed point on only those variables that are in the conditional support of the event.

Even though the definition uses a partial state to define the fixed point, it is generally enough to traverse the NS descriptors just like the constraint in constrained saturation: whenever we navigate to $n[i]$, we should also navigate through $d[i, i]$. The resulting descriptor will characterize all the partial states that cause the same behavior in the rest of the transitions.

3.2 Detailed Description of the GSA

The pseudocode for the GSA is presented in Figure 3. The inputs are an MDD node n encoding the initial states S^0 of a PTS, and a NS descriptor d representing the whole next-state relation $\mathcal{N}$. Since the algorithm will automatically partition the next-state relation based on conditional locality, d can be an union of all d_α (descriptors for events).

Sometimes, computing the full next-state relation is not practical, either because of its cost (e.g. we have to change representation) or because we want to use chaining in the fixed point computation. An advantage of abstract next-state diagrams is the ability to represent operations in a lazy manner. For example, the union of two descriptors may be represented by extending the set of descriptors $\mathcal{D}$ with elements of $\mathcal{D} \times \mathcal{D} \times \{union\}$ ($lvl((d_1, d_2, union)) = lvl(d_1) = lvl(d_2)$) and extending *next* such that $(d_1, d_2, union)[i, j]$ is: $\mathbf{1}$ if d_1 or d_2 is $\mathbf{1}$; d_1 if d_2 is $\mathbf{0}$; d_2 if d_1 is $\mathbf{0}$; and $(d_1[i, j], d_2[i, j], union)$ otherwise. The *lazy descriptor* $(d_1, d_2, union)$ will not be equivalent to any non-lazy descriptor (even if they encode the same relation), but will be equivalent to $(d_1, d_2, union)$ or $(d_2, d_1, union)$, which is not optimal cache-wise but is often better than pre-computing the union. This approach can be generalized to more than two operands.

Compared to (constrained) saturation in Figures 2a–2c, the main differences and points of interest are listed below. In SATURATE:

- Next-state descriptors are not retrieved for each level, but are a parameter.
- Recursive saturation of child nodes in line 7 passes $d[i, i]$ as the NS descriptor to use on the lower level $k-1$, which encodes a set of transitions that do not modify the variable associated to this level (and any above), therefore they are conditionally local over $V_{\leq k-1}$ with respect to the partial state specified by the SATURATE procedures currently on the call stack.
- Cache lookup in line 3 considers d instead of the partial state specified by the call stack because every partial state leading to d would produce the same result.
- In the fixed-point iteration in line 10 the SPLIT procedure is used to retrieve the operands of a lazy union descriptor to support chaining. It may be implemented in any other way as long as the returned set of descriptors cover the relation encoded by the descriptor passed as argument.
- In lines 6 and 9, the UPDATE procedure supports on-the-fly next-state relation building by providing a hook for replacing parts of d.

In SatFire:

- There are two descriptors: d_s for recursive saturation and d_f to fire.
- In the loop computing local successors in line 4 we omit locally read-only transitions ($i \neq j$), because they will be processed by recursive saturation.
- In the recursive firing in line 5, d_s is indexed by $[y, y]$ because (like in constrained saturation) the resulting node will be $n'[y]$ (and therefore $d_s[y, y]$ describes the conditionally local transitions), while d_f is indexed as usual.

In SatRecFire:

- Cache lookup in line 3 considers both next-state descriptors.
- In the loop computing local successors in line 5 we now *consider* every transition even if they are read-only, (on some level above they changed a variable).
- Recursive saturation in line 9 will use d_s (which is still conditionally local).

3.3 Constrained Saturation as an Instance of the GSA

With the automatic partitioning offered by the GSA, next-state relations that motivated the introduction of constrained saturation and its variants can now be directly encoded into the transition relation without any cost. This is because a constraint is a *guard*, therefore it can cause en event only to become read-only on a variable instead of independent, but will still never write it. Adding a constraint will never raise the conditional top variable of events, but it can raise their unconditional top variable in many cases, which is associated with degraded performance.

Indeed, the handling of d_s in the GSA is very similar to the handling of the constraint node – we could say that our algorithm uses the next-state relation itself as a constraint. Combining this with the flexibility of abstract NS descriptors (lazy descriptors in particular), we get the properties of constrained saturation enhanced with every difference between the original saturation algorithm and the GSA (see Section 3.4).

We illustrate the usage of abstract NS descriptors for variants of constrained saturation with the kind of constraint used in the original constrained saturation algorithm [11].

Definition 16 (Constrained next-state descriptor). *Given a NS descriptor d and a constraint node c, the* constrained next-state descriptor d_c *describing $\mathcal{N}(d_c) = \mathcal{N}(d) \setminus \left(\mathbb{N}^K \times S(c)\right)$ is a tuple $d_c = (d, c)$ with $lvl(d_c) = lvl(d) = lvl(c)$, and $d_c[i, j]$ is: $\mathbf{0}$ if $d[i, j] = \mathbf{0}$ or $c[j] = \mathbf{0}$; and $(d[i, j], c[j])$ otherwise.*

3.4 Discussion

To estimate the efficiency of the algorithm, we will consider the advantages and disadvantages of the different modifications. First and foremost it is important to note that if $Top_c(\alpha) = Top(\alpha)$ for every event α, then the GSA degrades to

```
Input: MDD node n,
       NS descriptor d
Output: saturated MDD node n'
if n ∈ {0,1} or d ∈ {0,1} then
    return n
if ¬SatCacheGet(n, d, n') then
    n' ← new MddNode(lvl(n))
    for each i where n[i] ≠ 0 do
        Confirm(lvl(n), i), Update(d)
        n'[i] ← Saturate(n[i], d[i,i])
    repeat
        changed ← false, Update(d)
        for each d_f ∈ Split(d) do
            n'' ← SatFire(n', d, d_f)
            if n' ≠ n'' then
                n' ← n'', changed ← true
    until ¬changed
    SatCachePut(n, d, n')
return n'
```

(a) Procedure Saturate.

```
Input: MDD node n,
       NS descriptor for saturate d_s,
       NS descriptor for fire d_f
Output: the result of firing d from the
        states n with the children
        saturated
if n = 0 or d = 0 then return 0
if d = 1 then return n
n' ← new MddNode(lvl(n))
for each x, y where x ≠ y and
  d[x,y] ≠ 0 and n[x] ≠ 0 do
    m ← SatRecFire(d[y,y], d[x,y], n[x])
    if m ≠ 0 then Confirm(lvl(n), y)
    n'[y] ← n'[y] ∪ m
CheckIn(n')
return n'
```

(b) Procedure SatFire.

```
Input: NS descriptor for saturate d_s, NS descriptor for fire d_f, MDD node n
Output: saturated MDD node n'' (after firing d_f from n saturated with d_s)
if n = 0 or d = 0 then return 0
if d = 1 then return n
if ¬RecFireCacheGet(d_s, d_f, n, n'') then
    n' ← new MddNode(lvl(n))
    for each x, y where d[x,y] ≠ 0 and n[x] ≠ 0 do
        m ← SatRecFire(d_s[y,y], d_f[x,y], n[x])
        if m ≠ 0 then Confirm(lvl(n), y)
        n'[y] ← n'[y] ∪ m
    CheckIn(n'), n'' ← Saturate(n', d_s), RecFireCachePut(d_s, d_f, n, n'')
return n''
```

(c) Procedure SatRecFire.

Fig. 3: Pseudocode of the GSA.

the original saturation algorithm from [4] or the corresponding constrained saturation algorithm from [7,10,11] with no difference in the iteration strategy and the virtually zero overhead of handling the next-state relation as a parameter. In every other case, there may be a complex interplay between the advantages and disadvantages discussed below.

An advantage of using conditional locality is that $Top_c(\alpha) \leq Top(\alpha)$, i.e. we can potentially use event α when saturating a node on a lower level, which is intuitively better because it raises the chance that the resulting node will be part

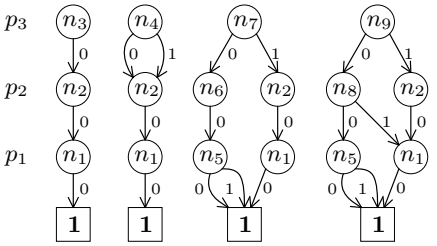


Fig. 4: A (degraded) run of saturation on the example model: n_3 encodes S^0, $n_4 = n_3 \cup \mathcal{N}_{t_3^l}(n_3)$, $n_7 = n_4 \cup \mathcal{N}_{t_1^l}(n_4)$, the state space is $n_9 = n_7 \cup \mathcal{N}_{t_2^l}(n_7)$. Note that t_i^u does not reach new states.

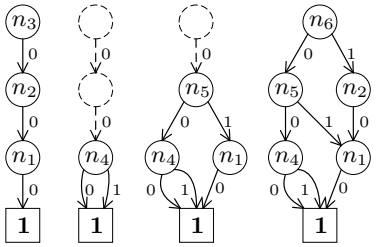


Fig. 5: A run of the GSA on the example model: n_3 encodes S^0, n_4 is the saturated n_1 (after firing t_1^l), n_5 is the saturated n_2 (after firing t_2^l) and n_6 is the state space.

of the final diagram. Figure 4 and 5 illustrate the MDDs that are created while exploring the state space of the example Petri net model from Figure 1 with saturation and the GSA. Saturation is degraded to a chaining version of BFS because every transition that can yield a new state is dependent on all variables. In the unfortunate case of firing t_3^l, t_1^l and t_2^l in this order, the number of created nodes will be 9 compared to the 6 nodes created by the GSA, which can still exploit the read-only dependencies.

A direct price of this is the diversification of cache entries. By repartitioning the events, we may introduce a lot more next-state relations to process, and it is not evident if their smaller size and the enhanced saturation effect can compensate this. Furthermore, by keeping track of d_s (the descriptor to saturate with), we spoil the cache of saturation due to the following.

Whenever we navigate through $d[i, i]$, we remember something from i in the context of the next-state relation, yielding a potentially large number of different descriptors to saturate with. The original saturation algorithm saturates each MDD node only once, because it uses the same next-state relation every time. In the GSA, we saturate every pair of different MDD node and NS descriptor, so the diversity of descriptors can be a crucial factor. In the extreme case, when at least one event remembers every value along the path (for example because it copies them to other variables below), caching can degrade to the point where everything will need to be computed from scratch.

The other extreme is when each event remembers only one thing from the values bound above: whether it is enabled or not (e.g. when variables are compared to constants in guard expressions). Fortunately, this is the case with Petri nets: each transition will check variables locally and decide whether it is still enabled or not. This means that given a descriptor d representing transitions in $|T|$, the number of possible successors for $d[i, i]$ will be $O(|T|)$ (n values can partition $\mathbb{N}$ into $n + 1$ partitions, each transition may contribute 2 values – one for an input arc and one for an inhibitor arc), but this number will also be limited by the number of non-zero child nodes of the saturated MDD node.

Given the facts that transitions of Petri nets are inherently conditionally local without repartitioning, and many nets are bounded (often safe), model checking of Petri net models with the GSA can be expected to yield favorable results. In fact, the experiment presented in Section 4 shows that the GSA is superior to the original saturation algorithm on every model that we analyzed.

For other types of models, we have yet to investigate the efficiency of the algorithm and the balance of benefits and overhead. It might be the case that we have to refine the read-only dependency into "local" and "global" evaluation (depending on whether we have to remember the value of the variable or can evaluate it immediately) and use conditional locality only with the "local" case. We also have to note that the efficient update of the next-state descriptors is not trivial and subject to future work.

4 Evaluation

In this section, we present the results of our experiments performed on a large set of Petri net models.

4.1 Research Questions

We have two main research questions about the GSA, both comparing it to the original saturation algorithm (SA) from [4]. Both questions will be answered by measuring the relevant metrics for each algorithm and comparing the results for each benchmark model.

We expect that *1)* the GSA will be identical to the SA when conditional locality cannot be exploited; as well as in other cases *2)* the GSA will create less MDD nodes than the SA and *3)* in these cases it will be faster than the SA.

4.2 The Benchmark

We have implemented both the original saturation algorithm and the GSA in Java. Both variants used the same libraries for MDDs and next-state descriptors, and their source code differs only in the points discussed in Section 3.2.

We used the latest set of 743 available models from the Model Checking Contest[4], excluding only the Glycolytic and Pentose Phosphate Pathways (G-PPP) model with a parameter of 10–1 000 000 000 (because the initial marking cannot be represented on 32-bit signed integers). We generated a variable ordering for each model using the *sloan* algorithm recommended by [1], and another one where we omitted read-only dependencies when building the dependency graph (motivated by the notion of conditional locality). We ran state space exploration on each model with each ordering 3-3 times, measuring several metrics of the algorithms. We will report the median of the running time of the algorithms (excluding the time of loading the model) and the total number of MDD nodes

[4] https://mcc.lip6.fr/

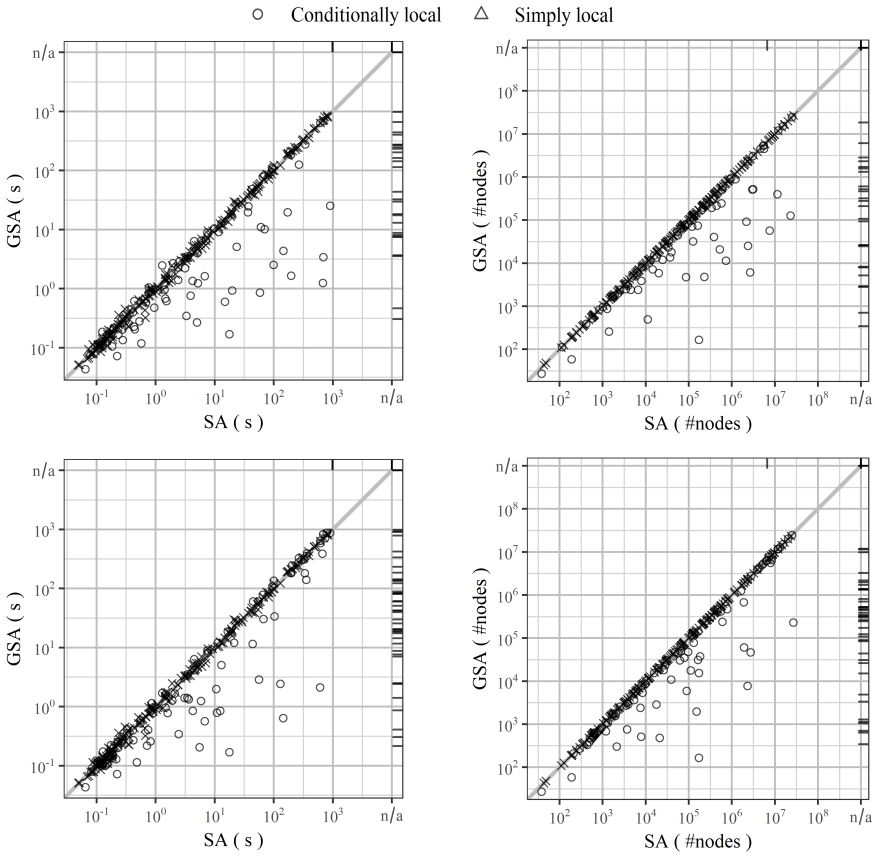


Fig. 6: Main results of the experiment. Top row: running times and total number of created nodes with sloan ordering. Bottom row: running time and total number of created nodes with modified sloan ordering.

created during each run, as well as the size of the state space and the final MDD for each model and each ordering.

Measurements were conducted on a bare-metal server machine rented from the Oracle Cloud (BM.Standard.E2.64), with 64 cores and 512 GB of RAM, running Ubuntu 18.04 and Java 11. Three processes were run simultaneously, each with a maximum Java heap size of 100 GB and stack size of 512 MB. No process has run out of memory and the combined CPU utilization never exceeded 70%. Timeout was 20 minutes (including loading the model and writing results).

4.3 Results

The main results of the experiments can be seen in Figure 6. Every point represents a model (dashes on the side means a timeout), classified into two groups:

"simply local" if none of the events had a read-only top variable and "conditionally local" otherwise. In the "simply local" group we expected no difference because the GSA should degrade into the original saturation algorithm, which was supported by the results. In the other group we were optimistic about the balance of advantages and disadvantages as discussed in Section 3.4, but the results were even better than what we expected. As the plots show, a significant part of the "conditionally local" models are below the reference diagonal, meaning that the GSA were often orders of magnitudes faster.

With the sloan ordering, 274 models were in the "conditionally local" group and the GSA was at least twice as fast as the SA in 53 cases. With the modified sloan ordering, these numbers are 69 out of 298. In one case (*SmallOperatingSystem-MT0256DC0128*), the SA managed to finish just in time while the GSA exceeded the timeout (scaling was similar for smaller instances). Models where the GSA finished successfully but the original saturation exceeded the timeout with the sloan ordering include *CloudDeployment*, *DiscoveryGPU*, *DLCround*, *DLCshifumi*, *EGFr*, *Eratosthenes*, *MAPKbis*, *Peterson* and *Raft* models and with the modified ordering also instances of *AirplaneLD*, *BART*, *Dekker*, *FlexibleBarrier*, *NeoElection*, *ParamProductionCell*, *Philosopher*, *Ring* and *SharedMemory*. Analyzing these models in detail may provide insights about when the GSA is especially efficient.

Looking at the plots about the number of created MDD nodes (i.e. the size of the unique table) reveals that our expectations about less intermediate diagrams were correct and this probably has direct influence on the execution time. Even though not visible in Figure 6, interactive data analysis revealed that the model instances are more-or-less located at the same point on the execution time and node count plots. The collected data also suggests a linear relationship between the number of created nodes and the execution time, but this is rather a lower bound than a general prediction.

As an auxiliary result and without any illustration, we also report that out of the 117 cases when the sloan ordering and the modified ordering were different and we have data about the final MDD size, the modified sloan ordering produced smaller final MDDs 69 times and larger MDDs 39 times. This motivates further work on variable orderings like in [1]. We have also compared the SA with sloan ordering and the GSA with the modified sloan ordering to find that the GSA with the modified sloan ordering was better in 78 cases and worse in 16 cases (considering only at least a factor of 2 in both cases).

5 Conclusions

In this paper, we have formally introduced the *generalized saturation algorithm* (GSA), a new saturation algorithm enhanced with the notion of *conditional locality*. We have shown that the GSA generalizes a family of constrained saturation variants and discussed the effects of using conditional locality. We have empirically evaluated our approach on Petri nets from the Model Checking Contest to find that the GSA has virtually no overhead compared to the original satura-

tion algorithm, but can outperform it by orders of magnitude when conditional locality can be exploited.

We have made theoretical considerations and prepared the algorithm to be compatible with a wide range of next-state representations as well as the on-the-fly update approach described in [4]. The GSA seems to be superior to the original saturation algorithm on Petri net models, but its efficiency over more general classes of models is yet to be explored.

Acknowledgements This work has been partially supported by Nemzeti Tehetség Program, Nemzet Fiatal Tehetségeiért Ösztönij 2018 (NTP-NFTÖ-18).

References

1. Amparore, E.G., Donatelli, S., Beccuti, M., Garbi, G., Miner, A.S.: Decision diagrams for Petri nets: A comparison of variable ordering algorithms. T. Petri Nets and Other Models of Concurrency **13**, 73–92 (2018)
2. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98**(2), 142–170 (1992)
3. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state-space generation. In: Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. pp. 328–342 (2001)
4. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. International Journal on Software Tools for Technology Transfer **8**(1), 4–25 (2006)
5. Ciardo, G., Yu, A.J.: Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In: Proc. of the 13th Int. Conf. Correct Hardware Design and Verification Methods. pp. 146–161 (2005)
6. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical set decision diagrams and automatic saturation. In: Hee, K.M.v., Valk, R. (eds.) Applications and Theory of Petri Nets, pp. 211–230. No. 5062 in Lecture Notes in Computer Science, Springer (2008)
7. Marussy, K., Molnár, V., Vörös, A., Majzik, I.: Getting the priorities right: Saturation for prioritised Petri nets. In: van der Aalst, W., Best, E. (eds.) Application and Theory of Petri Nets and Concurrency. pp. 223–242. Springer International Publishing, Cham (2017)
8. Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, write and copy dependencies for symbolic model checking. In: Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings. pp. 204–219 (2014)
9. Miner, A.S.: Implicit GSPN reachability set generation using decision diagrams. Perform. Eval. **56**(1-4), 145–165 (2004)
10. Molnár, V., Vörös, A., Darvas, D., Bartha, T., Majzik, I.: Component-wise incremental LTL model checking. Formal Aspects of Computing **28**(3), 345–379 (2016)
11. Zhao, Y., Ciardo, G.: Symbolic CTL model checking of asynchronous systems using constrained saturation. In: Proc. of the 7th Int. Conf. Automated Technology for Verification and Analysis. pp. 368–381 (2009)