periodica polytechnica

Electrical Engineering
and Computer Science
58/1 (2014) 3–13
doi: 10.3311/PPee.2080

Creative Commons Attribution

RESEARCH ARTICLE

Advanced Saturation-based Model Checking of Well-formed Coloured Petri Nets

András Vörös / *Dániel* Darvas / *Attila* Jámbor / *Tamás* Bartha

Received 2013-04-02, revised 2014-01-10, accepted 2014-03-03

Abstract

The failure of safety-critical embedded systems may have catastrophic consequences, therefore their development process requires a strong verification procedure to obtain a high confidence of correctness in the specification and implementation. Formal modelling and model checking provides a rigorous, mathematically precise verification method. Practical embedded systems are typically complex, distributed and asynchronous, thus they need expressive and compact formal models, and efficient model checking approaches.

The saturation algorithm *has an efficient iteration strategy. Combined with symbolic data structures, it can be used for state space generation and model checking of asynchronous systems.* Coloured Petri nets *are a good choice for modelling distributed and asynchronous systems, however their integration with saturation has not been solved in the past. In this paper we describe a new approach for applying saturation-based state space generation and model checking to coloured Petri nets. We demonstrate the performance of our new algorithm on the verification of a safety function used in the Reactor Protection System of a nuclear power plant.*

Keywords

model checking · saturation · safety-critical system · nuclear power plant · verification · Coloured Petri Net

András Vörös
Department of Measurement and Information Systems, Budapest University of Technology and Economics, Magyar tudósok krt. 2., H-1117 Budapest, Hungary
e-mail: vori@mit.bme.hu

Dániel Darvas
Attila Jámbor
Tamás Bartha
Department of Measurement and Information Systems, Budapest University of Technology and Economics, Magyar tudósok krt. 2., H-1117 Budapest, Hungary

1 Introduction

The formal verification of complex, distributed, and asynchronous embedded systems is an important but difficult task, complicated by the state space explosion problem. Such systems can be modelled in a compact and well-structured manner with coloured Petri nets. However, coloured Petri nets lacked efficient analysis methods in the past, making their application for practical verification problems hard or even impossible.

Our work focuses on an efficient model checking algorithm for asynchronous systems, the so-called *saturation* algorithm. It is a symbolic state space generation and model checking algorithm that uses a special iteration strategy during the exploration. The problem with saturation is that the symbolic representation it builds for the next-state relation during the state exploration phase imposes a high overhead in the analysis of practical systems. We address this problem by introducing a new strategy to handle the complex logic and large local state spaces encoded in the next-state representation of well-formed coloured Petri nets.

The structure of this paper is the following. Section 2 gives an overview of the theoretical background. Section 3 introduces the saturation algorithm, and its application to coloured Petri nets in particular. Our contribution is a new saturation algorithm presented in Section 4. We examine an industrial case study for our algorithm: the verification of a safety function. The description of the verification process and our results are given in Section 5.

1.1 Related work

There is much work in the field of model checking of Petri net models. The different techniques could be sorted into two groups: symbolic and explicit model checking algorithms. A BDD-based symbolic algorithm was presented in [16, 17], which was one of the first attempts to combine decision diagram techniques with Petri nets. Saturation is an efficient symbolic state space exploration [6] and CTL model checking [7] algorithm. SAT-based symbolic approaches proved their efficiency in hardware verification, but there are also efficient techniques for the analysis of Petri nets. We refer the reader to [12, 15]. Explicit techniques are also present for the verification of Petri net models. They usually use some kind of reduction techniques like

symmetry reduction [18] or partial order reduction based on stubborn sets [2], persistent sets [4] or ample sets [10]. An other partial order based technique is the so called Petri net unfolding algorithm [11].

Our case study, the PRISE (primary-to-secondary leaking) safety function [14] was an important motivation for our research. It has a huge state space (of $> 10^{12}$ states) and many different behaviours, therefore most of the previous verification attempts failed to handle it entirely. The first successful verification of PRISE was reported in [14], where the authors used coloured Petri nets and the Design/CPN modelling tool. Design/CPN has a simple explicit state model checker without built-in reduction methods, therefore the authors had to use state space reduction techniques manually, then partition the state space and separately analyse the different subspaces.

Later, we have created a formal model of the PRISE safety function in the Uppaal tool. Uppaal has symbolic state space representation, built-in state space reduction methods, and a (partial) Computation Tree Logic (CTL) model checker. It has failed to handle the complete state space due to memory overflow, although we have at least succeeded proving some of the requirements by reducing the model. We have also tried the Symbolic Analysis Laboratory (SAL) model checker [24]. SAL uses a Binary Decision Diagram based efficient state space representation, nevertheless this verification attempt has failed as well due to insufficient memory.

We have experimented with using other advanced Petri net verification methods [22]. In [6] the authors introduced an efficient symbolic state space generation and model checking method for asynchronous systems, especially for Petri nets. We have implemented and run the algorithm with different settings. Even using this method both the state space representation and the next-state relation have exceeded our resources.

We have published the first successful attempt to explore and verify the full state space of the PRISE safety function with our new algorithm in [21]. This paper is an improvement on our previous solution.

2 Background

In this section we outline the theoretical background of our work. First, we present coloured Petri nets, the modelling formalism we used. Then, we introduce Multiple-valued Decision Diagrams. They form the underlying data structures of our algorithms that store the state space during model checking. Finally, we give an overview of the model checking background.

2.1 Petri Nets

Petri nets are graphical models for concurrent and asynchronous systems, making both structural and dynamic analysis possible. A (marked) discrete ordinary Petri net is defined by a 5-tuple $PN = (P, T, E, w, M_0)$, represented graphically by a directed bigraph. $P = \{p_1, \ldots, p_n\}$ is a finite set of *places*, $T = \{t_1, \ldots, t_m\}$ is a finite set of *transitions* ($P \cap T = \emptyset$), $E \subseteq (P \times T) \cup (T \times P)$ is a finite set of *edges*, and $w\colon E \to \mathbb{Z}^+$ is the *weight function* assigning weights $w(e)$ to each edge $e \in E$. $M\colon P \to \mathbb{N}$ is a *marking function*, where $M(p_i)$ represents the number of *tokens* in place p_i. M_0 is the *initial marking* function of the net. A transition t is enabled, if for every $e = (p_i, t)$ incoming arc of t : $M(p_i) \geq w(p_i, t)$. An *event* in the system is the firing of an enabled transition t_i, which decreases the number of tokens in all the input places p_j by $w(p_j, t_i)$ and increases the number of tokens in every p_k output places by $w(t_i, p_k)$. The firing of the transitions is nondeterministic [13].

The *state space* or *reachability graph* of a Petri net is the set of states reachable from the initial state(s) through transition firings. Let $\mathcal{N}$ be the next-state function which depicts the possible state changes. $\mathcal{N}(s)$ is a subset of possible states, containing the states reachable from s through one firing. The complete set of states reachable from s is $\{s\} \cup \mathcal{N}(s) \cup \mathcal{N}(\mathcal{N}(s)) \cup \ldots = \mathcal{N}^*(s)$, where $\mathcal{N}^*$ is the transitive closure of $\mathcal{N}$. The state space of a Petri net is $\mathcal{S} = \mathcal{N}^*(s_0)$, where s_0 is the initial state of the net (given by the initial marking M_0).

2.2 Coloured Petri Nets

The coloured Petri net (CPN) [1] formalism enriches the ordinary Petri nets with complex data structures, making CPN models more compact and clearer. There are many variants of CPNs in the literature, in this work we use *well-formed* coloured Petri nets.

Our modelling formalism has a $CPN = (P, T, E, \Sigma, C, G, A, M_0^c)$ structure. P, T and E have the same meaning as in ordinary Petri nets. $\Sigma = \{\sigma_1, \ldots, \sigma_\kappa\}$ is a set of *colour sets* (data types). In well-formed coloured Petri nets Σ is finite. $C\colon P \to \Sigma$ is the *colour function* assigning colour sets to each place. G is a function that assigns a *guard* to each transition. A is the *arc expression function* assigning an arc expression to each edge. M_0^c is the *initial marking* function assigning multisets of tokens to each place.

The firing semantic is different from ordinary Petri nets. Each $G(t)$ guard is a Boolean function containing variables, Boolean operators, and marking expressions. Every $A(e)$ arc expression is a function that evaluates to a multiset of tokens. The σ_i colour sets determine the allowed sets of tokens. A transition t is enabled, if $\bigwedge_{e=(p_i,t)\in E} A(e)$ expression is satisfied (e. g., there is a possible variable–token assignment (binding) for all variables in every arc expression on ingoing and outgoing edges of t and for the variables of the guard $G(t)$) and the value of the guard $G(t)$ is true. The firing of an enabled transition t takes $A(e)$ tokens from p_i for every incoming edge $e = (p_i, t) \in E$ and puts $A(f)$ tokens to p_o for every outgoing edge $f = (t, p_o) \in E$.

The different variants of CPNs have different constraints for colour sets, guards and arc expressions. In our formalism colour sets can be *simple* or *complex*. A *simple colour set* is a finite enumeration or a finite subset of integers. A *complex colour set* is a Cartesian product of simple colour sets. An arc expression can contain token constants and simple variables representing a

member of a simple colour set. The guard expressions can contain token constants, simple variables, Boolean operators, relation signs and the successor operator.

2.3 Multiple-valued Decision Diagrams

Decision diagrams are used in symbolic model checking for efficiently storing the state space and the possible state changes of the models [19]. A *Multiple-valued Decision Diagram* (MDD) is a directed acyclic graph, representing a function f consisting of K variables: $f\colon \{0,1,\ldots\}^K \to \{0,1\}$. An MDD has a node set containing two types of nodes: many *nonterminal nodes* and two *terminal nodes* (namely **0** and **1**). The nodes are ordered into $K+1$ levels. A nonterminal node is labelled by a variable index k ($1 \le k \le K$), referring to which level the node belongs (i. e., which variable it represents; denoted by $Level(p)$ for node p), and has n_k (domain size of the variable) arcs to nodes in level $k-1$. We write $p[i] = q$ if the ith edge of node p is pointing to node q. A terminal node is labelled by the variable index 0. Duplicate nodes are not allowed, so if two nodes have identical successors in the lower level, they are also identical [20]. These rules ensure that MDDs are canonical and compact equivalents of a given function or set. The evaluation of the function is a top-down traversal of the MDD along the variable assignments represented by the arcs between nodes.

2.4 Model Checking

Model checking is an automatic technique for verifying finite state systems. Given a model, model checking decides whether the model fulfils the specification. Formally: let M be a Kripke structure (i. e., state transition graph). Let f be a formula of temporal logic (i. e., the specification). The goal of model checking according to [10] is to find all states s of M such that $M, s \models f$. *Structural model checking* [10] computes the results by exploring first the reachable states and creating a symbolic transition relation representation, and based on these it we can perform the model checking procedure.

CTL (Computation Tree Logic) [10] is frequently used for temporal specification of systems. It has an expressive syntax, and there are efficient algorithms for its analysis. Operators occur in pairs in CTL: the path quantifier, either A (on all paths) or E (there exists a path), is followed by the tense operator, one of X (next), F (future or finally), G (globally), and U (until). However, we only need to implement 3 of the 8 possible pairings due to duality [10]: EX, EU, EG.

The semantics of the 3 required CTL operators are as follows (where p and q are predicates):

- **EX**: $s^0 \models \mathsf{EX}\ p$ iff $\exists s^1 \in \mathcal{N}(s^0)$ state such that $s^1 \models p$. This means that EX corresponds to the inverse $\mathcal{N}$ function, applying one step backward through the next-state relation.
- **EG**: $s^0 \models \mathsf{EG}\ p$ iff $\exists I = (s^0, s^1, s^2, \ldots)$ infinite path such that $\forall j \ge 0 : s^{j+1} \in \mathcal{N}(s^j)$ and $s^j \models p$, so there is a strongly connected component containing states satisfying p.
- **EU**: $s^0 \models \mathsf{E}(p\ \mathsf{U}\ q)$ iff $\exists n \ge 0, \exists I = (s^0, s^1, s^2, \ldots, s^n)$ path such that $\forall 1 \le j \le n : s^j \in \mathcal{N}(s^{j-1})$, $\forall 0 \le k < n : s^k \models p$ and $s^n \models q$.

3 Saturation

Saturation [9] is a symbolic state space generation and model checking algorithm that proved its efficiency in the verification of asynchronous systems [6]. In this section we introduce its main features and its application for the verification of CPN models.

3.1 Saturation-based State Space Exploration

Saturation stores the encoded state space of the model in an MDD. *Decomposition* serves as the prerequisite for the symbolic encoding in saturation: the algorithm maps the state variables of the high-level model into symbolic variables of the decision diagram. A global state $\mathbf{s}^j$ can be described as the composition of the local states of components: $\mathbf{s}^j = (s^j_1, \ldots, s^j_K)$, where K is the number of components, s^j_i is a *local state* of the ith component, and $\bigcup_j s^j_i = \mathcal{S}_i$ is the local state space. The global state space $\mathcal{S}$ is represented by an MDD with K variables (levels), where variable x_i corresponds to the state of the ith component. A global state $\mathbf{s}^j$ is encoded by a trace (path) of the MDD, where $x_1 = s^j_1, \ldots, x_K = s^j_K$. Decomposition helps the algorithm to efficiently exploit the inherent *locality* of asynchronous systems. Locality ensures that an event usually affects only a few components, or just certain parts of the submodels.

Saturation uses a *peculiar iteration strategy*: it iterates through the MDD nodes and generates the whole state space representation using a node-to-node transitive closure. Building the MDD representation of the state space starts by building the MDD representing the initial state. Then the algorithm saturates every node in a bottom-up manner, by applying saturation recursively when new states are discovered. The result is the state space representation encoded in MDD. This way saturation avoids during the iteration that the peak size of the MDD exceeds its final size, which is a critical problem in traditional approaches. We refer the reader for details and running example to [3].

3.2 Conjunctive and Disjunctive Partitioning

The *next-state* function $\mathcal{N}_e$ of event e describes the states reachable from a given state in one step (i. e., with a single firing of a transition). In [6] the authors used a Kronecker matrix-based representation of $\mathcal{N}_e$. In their solution the next-state function $\mathcal{N}_{(e,i)}$ of the event e (firing of the corresponding transition) in the ith submodel is encoded by a *Kronecker matrix*. The global next-state of event e is $\mathcal{N}_e = \mathcal{N}_{(e,1)} \times \ldots \times \mathcal{N}_{(e,K)}$. This encoding enables building the next-state functions locally, but it requires a Kronecker-consistent decomposition. Ordinary Petri nets are Kronecker-consistent for any partitioning of the places, but this is not guaranteed for more general models, like the well-formed CPNs [6].

In [8] the authors introduced a new next-state representation for saturation-based algorithms to be able to analyse a more general class of models. This solution uses MDDs with $2K$ levels to symbolically encode a next-state function $\mathcal{N}$ into the relation $\mathcal{R}$ of *from* and *to* variables: $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$. The variables $\mathbf{x} = (x_1, x_2, \ldots, x_K)$ in $\mathcal{R}$ refer to the current ('from') state, and the variables $\mathbf{x}' = (x'_1, x'_2, \ldots, x'_K)$ to the next ('to') states. $\mathcal{R}$ encodes the next-state function so that from state $\mathbf{x}$ we can go to states $\mathbf{x}'$ in one step.

The algorithm avoids creating a large, monolithic next-state relation, it divides the global next-state function into smaller parts instead. The first step is the *disjunctive decomposition* according to the set $\mathcal{E}$ of e events in the high-level model: $\mathcal{R} = \bigvee_{e \in \mathcal{E}} \mathcal{R}_e$. In many cases the computation of these local $\mathcal{R}_e$ relations is still expensive. So, in the next step the algorithm partitions the $\mathcal{R}_e$ disjuncts *conjunctively* according to the *enabling* and *updating* relations [8]: $\mathcal{R}_e = \bigwedge_k \mathcal{R}^{enable}_{e,k} \wedge \bigwedge_k \mathcal{R}^{update}_{e,k}$, where $e \in \mathcal{E}$, $1 \leq k \leq K$, and K is the number of components. The *enabling* relation is responsible for deciding if the given event is enabled in a certain state while the *update* relation decides to which next states the exploration can go.

The *enabling* relation consists of variables necessary for deciding the enabling of the transition related to a certain event. It contains only 'from' variables (in $\mathbf{x}$), and does not change the value of any 'to' variables (in $\mathbf{x}'$). The *updating* relation represents the local state changes, i. e., the local next-state functions, therefore it contains variables both from $\mathbf{x}$ and $\mathbf{x}'$.

This fine-grained decomposition approach makes it possible to handle arbitrary finite next-state functions, which is the key to handle complex events efficiently.

3.3 Saturation-based Analysis of Coloured Petri Nets

Well-formed coloured Petri nets can model complex systems in a compact form by utilizing the data content of tokens instead of pure structural constructs. However, this compactness takes its price during state traversal: local state spaces and transition relations of the submodels in a decomposed CPN are typically much larger and more complex than in simple Petri nets. Previous research [8] proved that the smaller the partitions are, the more efficient the saturation becomes, since the creation and maintenance of the smaller parts requires significantly less resources. In this section we present a new approach to analysing well-formed coloured Petri nets, using the framework to handle a general class of models introduced in [8].

The efficient construction of the relation $\mathcal{R}_e$ is a challenging task. The firing of a transition changes the state of both the input and output places. As a consequence, if decomposing the transition relation according to [8], then update relations will always contain the enable relations. Therefore, they cannot be divided into disjunct relations, so the manipulation of the enabled relations only leads to a computational and storage overhead. Another problem is that complex guard expressions prevent the algorithm from reaching fine-grained partitioning, since all explored possible variable assignments of a guard expression have to be stored in an MDD as well.

Our aim is to decompose the transition relation into small disjunct relations, where the simple conjuncts depend only on as few state variables as possible. In order to be able to build a disjunct enable relation we introduced [21] new variables (levels) $\mathbf{v} = \{v_1, \ldots, v_n\}$ in the next-state representation, corresponding to the CPN variables used in the guards and arc expressions (where n is the number of independent variables in the guard and arc expressions). The enabling constraint has the form $\mathcal{R}^{enable}_e(\mathbf{v})$ and it is expressed with the new $\mathbf{v}$ variables in a semantically equivalent way to the $\mathcal{R}^{enable}_e$ relation of the algorithm in [8]. An additional advantage of this encoding is that $\mathcal{R}^{enable}_e(\mathbf{v})$ is computable off-line, before running the algorithm. The update relation has the form $\mathcal{R}^{update}_{e,i}(\mathbf{v}, \mathbf{x}, \mathbf{x}')$. Using the new variables a fine-grained partitioning can be constructed, where each stateful element (i. e., place) has its own next-state relation. The final transition relation of event e is the following: $\mathcal{R}_e = \{(\mathbf{x}, \mathbf{x}') | \exists \mathbf{v}\ \mathcal{R}^{enable}_e(\mathbf{v}) \wedge \bigwedge_k \mathcal{R}^{update}_{e,k}(\mathbf{v}, \mathbf{x}, \mathbf{x}')\}$.

3.4 Performance Issues

The coloured saturation algorithm using the disjunctive-conjunctive partitioning introduced in this section is designed for a general class of CPNs, without restrictions. As a consequence, the algorithm does not have a priori knowledge about the possible states, local states, next-states and local next-states. Therefore, it builds the local state spaces and transition relations on-the-fly, without having additional information that could be used to optimize the traversal and the construction of the next-state relations.

Thus, when a new local state is discovered, both the local state space and the next-state relations need to be updated with regard to the new information. Since these updates are frequent (as all local states and next-state relations must be explored), they impose a big overhead on the algorithm. Moreover, incidental to the greedy transition relation building nature of symbolic methods, the algorithm builds many transition relations that will never be fired due to the restrictions by the state space.

In the next chapter we address these problems by adapting the saturation algorithm to those kinds of models, where the local state spaces of the components are not known before the model checking.

4 Lazy Saturation

In this section we introduce our new saturation algorithm, which uses a more resource-efficient strategy to compose the next-state relations during the state space traversal. The aim is to be able to filter out the unnecessary state changes by delaying the construction of the next state relation. We named this new approach as lazy coloured saturation, or *lazy saturation* for short.

4.1 Overview of the Approach

Symbolic algorithms encode all of the possible state changes in the transition relation. The disjunctive-conjunctive partitioning algorithm decomposes this relation, and saturation benefits from the efficient manipulation of the smaller parts. During the iteration these subrelations are updated according to the new information found about the substates: every time a new local state s_i^j is discovered, all possible local state transitions from s_i^j are computed and added to every next-state relation. However, there can be state transitions that are possible locally, but the algorithm will never reach a state where they become enabled on the global, Petri net level. Since these infeasible local state transitions have been added to the local next-state relations, the decomposed symbolic representation becomes bigger than necessary.

The aim of our new algorithm is to filter out as many infeasible transition relations as possible. We introduce a new $\mathcal{ER}$ relation that only stores the states from which state transitions are possible. In other words, this relation contains only 'from' states ($\mathbf{x}$), contrary to the next-state relation the 'to' states ($\mathbf{x}'$) are not stored. This lets the building of the next-state relations be delayed until the algorithm can exactly decide which relation should be updated with the new information. First, we build only the $\mathcal{ER}$ relation, and we include a state transition in the next-state relation only when it becomes globally enabled. In this way the next-state relation will contain only a few globally infeasible state transitions, and its size will be significantly more compact. The motivation of our work is based on the observation that the size of the $\mathcal{ER}$ relation is always smaller than the size of the $\mathcal{R}$ relations: using this smaller $\mathcal{ER}$ relation to postpone or to skip the building of the $\mathcal{R}$ relations is a good pay-off regarding the performance of the algorithm.

We include the pseudocode of the lazy saturation algorithm in the rest of this section. The cache manipulation and decision diagram specific operations are omitted for brevity, but the interested reader can find them in [6]. The changes compared to former approaches are marked with "stars" ($*$), the rest of the algorithms are from [6].

The entry point of saturation is the `GenerateStateSpace` function (Algorithm 1). This function creates a new MDD node for every submodel to represent the initial state, and immediately saturates each one of them in a bottom-up manner. The saturation of nodes are done by the `Saturate` (Algorithm 2) and `SatFire` (Algorithm 3) functions. These functions build the MDD of the state space by firing all enabled events in a recursive, exhaustive manner. If a new state is discovered, all states reachable from it are explored by calling the `Confirm` (Algorithm 4) function. This function is explained in detail in Section 4.2. The $\mathcal{R}$ relation is updated by the `UpdateRelation` function at this point of saturation. Its operation is described in Section 4.3.

Algorithm 1 GenerateStateSpace

```
Require: initial state
Ensure: set of reachable states
 1: last ← 1
 2: for k ← 1 to K do
 3:     Confirm(k, 0)
 4:     r ← NewNode(k)
 5:     r[0] ← last
 6:     Saturate(r)
 7:     last ← r
 8: end for
 9: return last
```

Algorithm 2 Saturate

```
Require: p : node
 1: k ← Level(p)
 2: chng ← true
 3: while chng do
 4:     chng ← false
 5:     for all e : Top(e) = k do
 6:         UpdateRelation(e, p, ER_e, R_e) {*}
 7:         for all i ∈ S_k, j ∈ L_k : p[i] ≠ 0 ∧ R_e[i][j] ≠ 0 do
 8:             f ← SatFire(e, p[i], R_e[i][j])
 9:             if f ≠ 0 then
10:                 u ← Union(f, p[j])
11:                 if u ≠ p[j] then
12:                     p[j] ← u
13:                     chng ← true
14:                     if j ∉ S_k then
15:                         Confirm(k, j)
16:                     end if
17:                 end if
18:             end if
19:         end for
20:     end for
21: end while
```

Algorithm 5 UpdateEventEnable

```
Require: e : event
 1: ER_e ← 1
 2: for k = 1 → K do
 3:     ER_e ← ER_e ∧ ER_{e,k}
 4: end for
```

Algorithm 6 UpdateEvent

```
Require: e : event
 1: R_e ← R_e^{enable}
 2: for k = 1 → K do
 3:     R_e ← R_e ∧ R_{e,k}^{update}
 4: end for
```

Tab. 1. Brief description of the used functions

Function	Description
Confirm(k, i)	Registers state i on level k to be globally reachable, refreshes enabled relations.
GenerateStateSpace	Entry point of the algorithm, generates the symbolic representation of the state space.
NewNode(k)	Creates a new MDD node on level k.
SatFire(e, p, $\mathcal{R}$)	Exhaustively fires event e using the next-state relation $\mathcal{R}$ on the states represented by the subgraph of p.
Saturate(p)	Exhaustively fires all enabled events for the states represented by the subgraph of p.
Top(e), Bot(e)	Returns the number of highest (lowest) level affected by event e.
UpdateConjunct($\mathcal{R}$, i)	Updates the conjunct represented by $\mathcal{R}$ when new parts of the state space are discovered.
UpdateEvent(e)	Builds (or rebuilds) the $\mathcal{R}_e$ relation from the $\mathcal{R}_{e,k}^{update}$ conjuncts.
UpdateEventEnable(e)	Builds (or rebuilds) the $\mathcal{ER}_e$ relation from the $\mathcal{ER}_{e,k}$ conjuncts.
UpdateRelation(e, p, $\mathcal{ER}$, $\mathcal{R}$)	Decides if the next-state relation has to be updated according to the newly explored states.

Algorithm 3 SatFire

Require: e : event, p : node, $\mathcal{R}$: relation
Ensure: node

```
1: k ← Level(p)
2: if k < Bot(e) then
3:     return p
4: end if
5: s ← NewNode(k)
6: chng ← false
7: for all i ∈ S_k, j ∈ L_k : p[i] ≠ 0 ∧ R[i][j] ≠ 0 do
8:     f = SatFire(e, p[i], R[i][j])
9:     if f ≠ 0 then
10:        u ← Union(f, p[j])
11:        if u ≠ p[j] then
12:            p[j] ← u
13:            chng ← true
14:            if j ∉ S_k then
15:                Confirm(k, j)
16:            end if
17:        end if
18:    end if
19: end for
20: if chng then
21:     Saturate(s)
22: end if
23: return s
```

Algorithm 4 Confirm

Require: k, i : int

```
1: for all e : Bot(e) ≤ k ≤ Top(e) do
2:     if N_{e,k}(i) ≠ ∅ then
3:         UpdateConjunct(ER_{e,k}, i) {*}
4:         UpdateEventEnable(e) {*  }
5:     end if
6: end for
7: S_k ← S_k ∪ {i}
```

Algorithm 7 UpdateRelation

Require: e : event, p : node, $\mathcal{ER}, \mathcal{R}$: relation
Ensure: bool

```
1: if p = 0 or ER = 0 then
2:     return false
3: end if
4: if p = 1 then
5:     return true
6: end if
7: k ← Level(p)
8: a ← false
9: for all i ∈ S_k : p[i] ≠ 0 ∧ ER[i] ≠ 0 do
10:    for all j ∈ N_{e,k}(i) do
11:        if UpdateRelation(e, p[i], ER[i], R[i][j]) then
12:            a ← true
13:            if R[i][j] = 0 then
14:                UpdateConjunct(R^{update}_{e,k}, i, j)
15:                L_k ← L_k ∪ {j}
16:                UpdateEvent(e)
17:            end if
18:        end if
19:    end for
20: end for
21: return a
```

4.2 Building the $\mathcal{ER}$ Relation

We apply the conjunctive-disjunctive decomposition also to the $\mathcal{ER}$ relation. The algorithm creates a separate $\mathcal{ER}_e$ relation for each event e. For efficient manipulation, the algorithm partitions each $\mathcal{ER}_e$ relation into K smaller parts, and stores them separately: $\mathcal{ER} = \bigvee_{e \in \mathcal{E}} \mathcal{ER}_e$ and $\mathcal{ER}_e = \bigwedge_{1 \le k \le K} \mathcal{ER}_{e,k}$. This way we can exploit event-locality and the other advantages of the saturation algorithm.

Contrary to the next-state representation of the traditional algorithm, our algorithm builds primarily the $\mathcal{ER}$ relation during the iteration. The algorithm discovers the new states from which we can fire an event. The states we get after firing the event are

ignored in this phase of the iteration. Formally, our $\mathcal{ER}_e$ relation is the 'simplified' version of the $\mathcal{R}_e$ relation of the traditional algorithm: $\forall e \in \mathcal{E} : \mathbf{x} \in \mathcal{ER}_e$, iff $\exists \mathbf{x}' : (\mathbf{x}, \mathbf{x}') \in \mathcal{R}_e$.

The pseudocode of the `Confirm` function that updates the next-state information is shown in Algorithm 4. The parameters are the following: i denotes the local state at the kth level we are in before firing. The found possible state change from this state is updated in the $\mathcal{ER}_{e,k}$ relation by the `UpdateConjunct` function (its pseudocode is omitted, since it only does simple decision diagram manipulations). After a conjunct of $\mathcal{ER}_e$ was updated, we must update the whole $\mathcal{ER}_e$ relation by computing symbolically: $\mathcal{ER}_e = \bigwedge_{1\leq k\leq K} \mathcal{ER}_{e,k}$. This is carried out by the `UpdateEventEnable` function.

4.3 Updating the Next-state Relation

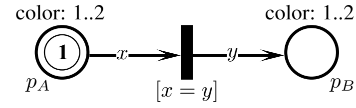
The next-state relations are updated by the `UpdateRelation` function shown in Algorithm 7. This function recursively computes if an event is enabled, and updates the next-state relation if needed. It traverses recursively all event firings from the $\mathcal{ER}$ and $\mathcal{R}$ relations, and the MDD denoted by p that encodes the state space. During this traversal the algorithm decides whether a state transition is enabled or not. If the algorithm finds an enabled state transition, i. e., the `UpdateRelation` called in a deeper level has returned with a **true** value, and this state change has not appeared in $\mathcal{R}$ yet (i. e., $\mathcal{R}[i][j] = \mathbf{0}$, this means that the path in the MDD containing the 'from' value i and 'to' value j leads to the terminal zero), then it must be put into $\mathcal{R}$. After every update we must recalculate the $\mathcal{R}$ relation by calling the `UpdateEvent` function (Algorithm 6). This function updates the $\mathcal{R}_e$ relation by calculating: $\mathcal{R}_e = \mathcal{R}_e^{enable} \wedge (\bigwedge_{1\leq k\leq K} \mathcal{R}_{e,k}^{update})$.

4.4 Operation of Lazy Saturation

We illustrate the operation of the lazy saturation algorithm with an example. The chosen model shown in Figure 1 a. is a simple coloured Petri net consisting of two places and a transition. Both places have the same colour set with two values: 1 and 2. Initially the place p_A is marked with a token valued 1, and the place p_B is empty. During the decomposition we create two submodels, one for place p_A and one for place p_B. The encoding of the local states is shown in Figure 1 b. Based on the table, the initial (local) state of place p_A is 1 and the initial state of place p_B is 0. The decomposition of the $\mathcal{R}$ relation (and of the $\mathcal{ER}$ relation) conforms to the decomposition of the state space, i. e., there are two update conjuncts, $\mathcal{R}_A^{update}$ and $\mathcal{R}_B^{update}$.

The MDDs created during the event handling of saturation are shown in Table 2. The content of the first row belongs to the former coloured saturation algorithm, while the second row belongs to our new lazy algorithm. The decision diagram levels (variables) corresponding to the variables of the guard and arc expressions are omitted from the MDDs for brevity.

The execution steps of the former coloured saturation algorithm and the new lazy algorithm for the example are the following:



a. Example CPN model

Local state	Meaning
0	The place is empty.
1	The place is marked with a token valued 1.
2	The place is marked with a token valued 2.
11	The place is marked with two tokens, both valued 1.
12	The place is marked with two tokens, the value of the first is 1, the value of the second is 2.
etc.	

b. Encoding of the local states

Fig. 1. Example to illustrate the operation of lazy saturation

1 `GenerateStateSpace` is called. It creates the $\mathcal{R}^{enable}$ relation, and calculates its content off-line. (This relation is not shown in Table 2.)
2 `Confirm`$(1, 0)$ is called. Coloured saturation explores and collects all possible state changes into the $\mathcal{R}_B^{update}$ relation. Locally there are two new reachable states depending on the assignment of the y variable. Lazy saturation examines only whether the transition is fireable from state 0, and the `Confirm` function collects this enabled state into the $\mathcal{ER}_B$ relation.
3 The `Saturate` function cannot make any steps, as the $\mathcal{R}$ and $\mathcal{ER}$ relations are still empty, since the $\mathcal{R}_A^{update}$ and $\mathcal{ER}_A$ conjuncts are still empty.
4 The `GenerateStateSpace` function calls `Confirm`$(2, 1)$. Coloured saturation creates the $\mathcal{R}_A^{update}$ conjunct, lazy saturation creates the $\mathcal{ER}_A$ relation.
5 The `UpdateEvent` function results the $\mathcal{R}$ relation. In this step lazy saturation calculates only the $\mathcal{ER}$ relation by calling the `UpdateEventEnable` function. The (not represented) $\mathcal{R}^{enable}$ conjunct prevents the next-state relation from storing the $(A, A', B, B') = (1, 0, 0, 2)$ global state change (i. e., the $(1, 0) \rightarrow (0, 2)$ state transition, which is evidently impossible).
6 The `Saturate` function is called. Coloured saturation fires the $(1, 0, 0, 1)$ global state change (the $(1, 0) \rightarrow (0, 1)$ state transition) while it builds the state space MDD. This is the point where the lazy saturation algorithm realizes that it should update the $\mathcal{R}$ relation, because it is still empty. The algorithm first calls `UpdateRelation`. After updating the next-state relation, it makes the same steps as coloured saturation.
7 The newly reached local states must be confirmed. Calling `Confirm`$(2, 0)$ (i. e., confirming the local state 0 at the level of place p_A) does nothing, because the transition cannot fire when place p_A is empty. However, the transition is enabled locally, if place p_B contains a token. So we need to update the $\mathcal{R}_B^{update}$ of coloured saturation, and the $\mathcal{ER}_B$ of lazy saturation.
8 The algorithm updates the $\mathcal{R}$ and $\mathcal{ER}$ relations, respectively.

Tab. 2. Data structures (MDDs) of coloured saturation and lazy saturation

	2nd step	4th step	5th step	6th step	7th step	8th step	9th step
Coloured Saturation	$\mathcal{R}_B^{update}$: B –0→ B′ –1, 2→ **1**	$\mathcal{R}_A^{update}$: A –1→ A′ –0→ **1**	$\mathcal{R}$: A –1→ A′ –0→ B –0→ B′ –1→ **1**		$\mathcal{R}_B^{update}$: B –0→ B′ –1, 2→ **1**; B –1→ B′ –11, 12→ **1**	$\mathcal{R}$: A –1→ A′ –0→ B; B –0→ B′ –1→ **1**; B –1→ B′ –11→ **1**	
Lazy Saturation	$\mathcal{ER}_B$: EB –0→ **1**	$\mathcal{ER}_A$: EA –1→ **1**	$\mathcal{ER}$: EA –1→ EB –0→ **1**	$\mathcal{R}_B^{update}$: B –0→ B′ –1→ **1**; $\mathcal{R}_A^{update}$: A –1→ A′ –0→ **1**; $\mathcal{R}$: A –1→ A′ –0→ B –0→ B′ –1→ **1**	$\mathcal{ER}_B$: EB –0, 1→ **1**	$\mathcal{ER}$: EA –1→ EB –0, 1→ **1**	$\mathcal{R}_B^{update}$: B –0→ B′ –1→ **1**; $\mathcal{R}$: A –1→ A′ –0→ B –0→ B′ –1→ **1**

9 Similarly to the 5th step, we need to update the $\mathcal{R}$ with the enabled state changes, before the next step of the iteration progresses. However, there is no newly enabled state change, so lazy saturation does not extend the relation with the change represented by $(A, A', B, B') = (1, 0, 1, 11)$, because $(1, 1) \rightarrow (0, 11)$ is not possible with the given initial marking.

10 There is no newly enabled relation for neither the lazy nor the coloured saturation algorithm, so the procedure is finished. The next-state relation of the lazy saturation algorithm contains less next-states.

5 Analysis

In this section we show how our new algorithm performs on an industrial case study. We have implemented our lazy saturation-based state space generation algorithm using on-the-fly state updates. We have created a coloured Petri net model of a safety function of a real industrial embedded system, and used the implemented lazy saturation algorithm for state space generation and model checking. We could successfully prove the correctness of the safety function by exploring its entire state space. In the following, we present the case study and our measurement results.

5.1 The Modelled Industrial System

Our industrial case study is a safety function included within the Reactor Protection System of a nuclear power plant [14]. This safety function initiates an emergency operation when a predefined chain of events happens. The detection of the specific event chain requires a complex logic, the design of which is error prone. This also puts emphasis on the necessity of using formal verification to ensure correctness.

The safety function receives inputs from 9 different sensors, and computes the values of 2 outputs, one of which initiates the emergency protection action. The values of the outputs depend on the recent and past values of the inputs, and some internal timers. The design of the controller was specified by a Functional Block Diagram (FBD). The FBD representation contains simple combinatorial (OR gates, AND gates, and inverters), and sequential (SR flip-flops, delay and pulse modules) logic gates. The proper combination of these logic elements needs to guarantee that the emergency protection action will be initiated only in the case of a specific dangerous event happened.

We have created a coloured Petri net model of the safety logic. The structure of the CPN model preserves the data flow characteristics of the FBD description. Therefore, the high-level view of the CPN model (shown in Figure 2) is isomorphic to the FBD description. The subnets of the CPN model are the models for the functional modules of the FBD. After the subnets have been derived and verified separately, they only had to be connected together properly.

An example CPN subnet (modelling the operation of a functional block, namely the *Delay module* or TON module) is shown in Figure 3a. The functionality of the Delay module is given by a time diagram in Figure 3b. The purpose of the module (as its name implies) is to delay a rising edge pulse for a predefined D number of cycles. When the module detects a rising edge, it starts a counter. If the pulse is active (the input remains 1) for at least D number of cycles, the Delay module will “let the pulse pass”, that is it sets its output to 1 (the **true** Boolean value). The output will remain 1 as long as the input is active. When a falling edge is detected, the module resets itself to its

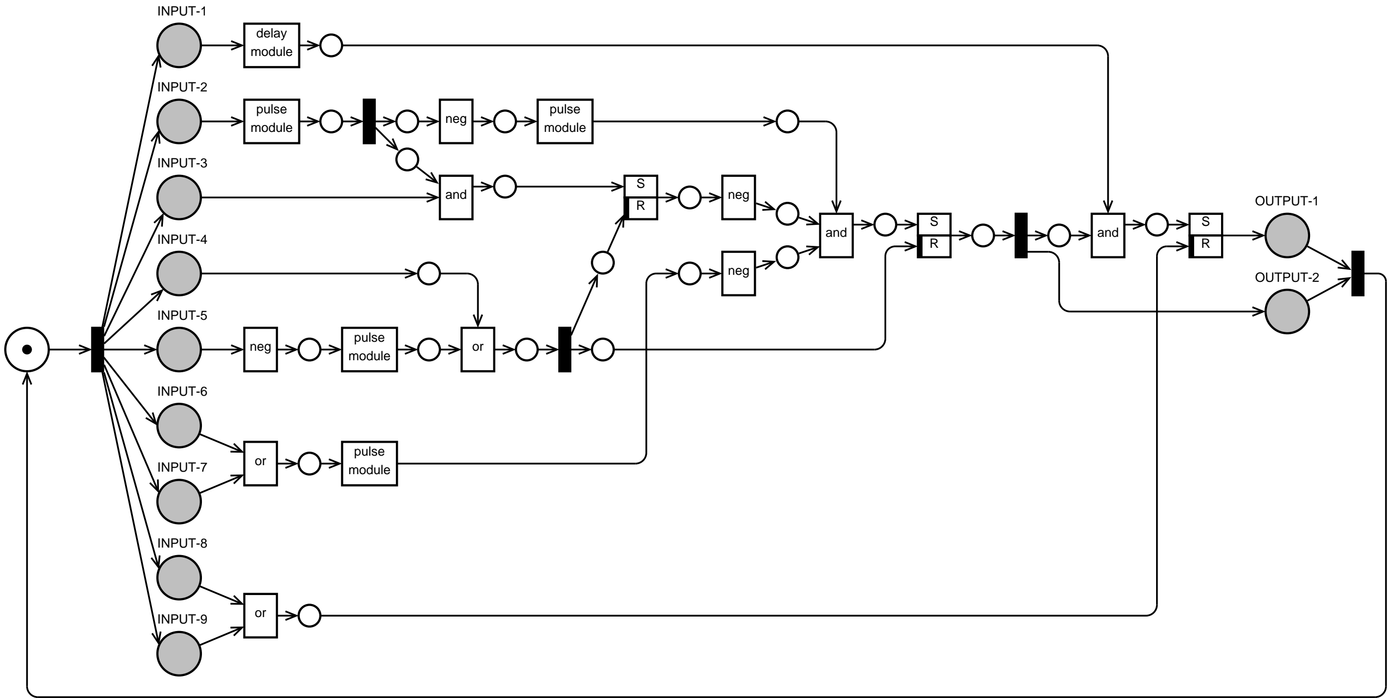


Fig. 2. The coloured Petri net model of the safety function

default inactive state.

In the next step we have formalized the required operation of the safety function. We could translate the functional requirements into the following verification goals:

- *Liveness requirement:* the emergency protection action is always initiated when the specific type of accident has occurred (no actuation masking).
- *Safety requirement:* the emergency protection action is never initiated if no, or another type of accident has occurred (no unintended actuation).
- *Deadlock freeness:* No deadlock situation can arise for any combination and sequence of input signals.

These requirements were formalized with CTL temporal logic in the following way:

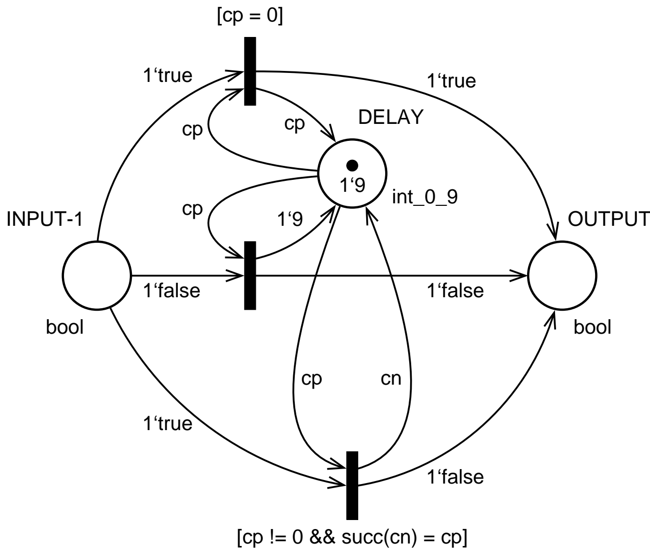
- First, we checked the deadlock freeness of the system. Informally this means that in every state there exists at least one reachable successor state. The equivalent CTL temporal logic expression is: $\mathsf{AG}(\mathsf{EX}(\mathbf{true}))$.
- We also checked if the model is *reversible*, that is from every state we can reach the initial state. We expressed it with the following CTL expression: $\mathsf{AG}(\mathsf{EF}([init]))$. This property ensures that the safety function can be made ready to fulfil its purpose in all circumstances.
- We used indirect proof to prove the safety requirement. We transformed the inverse requirement into: $\mathsf{E}(\neg\,[accident\ event]\ \mathrm{U}\ [actuation])$. This formula is satisfied only if the emergency protection action is initiated without a proper *accident event*.
- The liveness requirement was also easier to prove by indirect proof. We formalised the inverse requirement as: $\mathsf{EF}([accident\ event] \wedge \mathsf{EG}(\neg\,[actuation] \wedge \neg\,[reset\ event]))$. Informally, we are searching for strongly connected components in the state space that contain no *actuation* and *reset event*, but contain an *unsafe event*.

5.2 Results

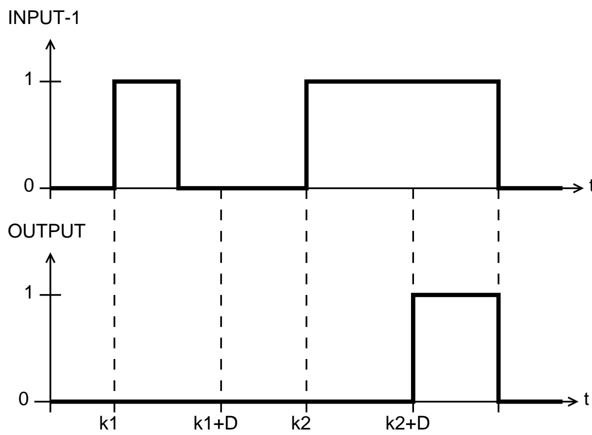
The next step of the verification was to explore and store the state space of the CPN model of the safety function, using our lazy saturation algorithm and state space storage data structures described in Section 4. After obtaining the complete state space we could evaluate the four CTL expressions introduced in the previous section. For state space traversal and temporal logic-based model checking we developed our own experimental implementation of our algorithms written in the C# programming language. We used the following configuration for our measurements: Intel L5420 2.5 GHz processor, 8 GB memory, Windows Server 2008 R2 (x64) operation system, .NET 4.0 runtime. The measurement results are listed in Table 3.

Tab. 3. Characteristics of the state space traversal

Parameter	Coloured saturation	Lazy saturation
Run time	367 s	242 s
Number of global states	$2.701 \cdot 10^{12}$	
State space representation (nodes)	1 587	
Number of local state changes	10 084 401	1 864
Sum of nodes in next-state relations	164 711	66 741
Sum of nodes in $\mathcal{ER}$ relations	0	2 419
Total number of nodes	$2.131 \cdot 10^7$	$1.338 \cdot 10^7$



a. The CPN subnet of the Delay (TON) module



b. Time diagram of the operation

Fig. 3. Delay module: model and operation

Run time represents the time needed to explore the state space. The state space generation required 367 s for the CPN model of the safety function using our former coloured saturation algorithm, and only 242 s with the new lazy saturation algorithm. This is a 35% improvement considering the runtime. Note, that former, non saturation-based approaches [14] could not discover the full state space of the model. The evaluation of the temporal expressions took considerably less time: deadlock freedom and reversibility checking temporal expressions took 6 s each to evaluate on the existing state space representation. The liveness and safety requirements were evaluated in 2 s and 3 s, respectively.

Beside the run time, the memory requirement is also the subject of interest. Measuring the memory consumption of programs executed in managed environment is problematic, because the garbage collector does not free up all the unused memory necessarily [23]. However, as most of the memory is used by the nodes and edges of the decision diagrams, the number of these elements can be used as a representative of the memory consumption.

6 Conclusion

In this paper we have presented a new saturation algorithm for coloured Petri nets, called *lazy saturation*. It introduces a new next-state relation building strategy that partitions the transition relations in a temporal manner, and updates only the appropriate relations with relevant information, while filtering out infeasible transitions. This makes on-the-fly local state and transition relation construction more suitable for CPNs. Another benefit is that the MDDs storing the transition relations and guards get smaller, so their manipulation becomes more efficient.

In order to test lazy saturation, we have created a CPN model of a real industrial system: a safety function in a nuclear power plant. We successfully explored the state space and verified the correctness of this model with our tool. We could achieve this result only with our saturation algorithms, since the former attempts using other well-known tools have failed due to insufficient memory. We have also compared the performance of lazy saturation to our previous coloured saturation algorithm, and found a significant improvement in the run-time.

Acknowledgement

This work was partially supported by the ARTEMIS JU and the Hungarian National Development Agency (NFÜ) in the frame of the R3-COP project. Attila Jámbor and Dániel Darvas were partially supported by the MFB Hungarian Development Bank Plc.

References

1 **Jensen K, Kristensen LM, Wells L**, *Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems*, International Journal on Software Tools for Technology Transfer, **9**(3–4), (2007), 213-254 (English), DOI 10.1007/s10009-007-0038-x.

2 **Valmari A**, *Stubborn sets for reduced state space generation*, In: **Rozenberg G** (ed.), Advances in Petri Nets 1990, Lecture Notes in Computer Science, Vol. 483, Springer Berlin Heidelberg, 1991, pp. 491-515, DOI 10.1007/3-540-53863-1_36.

3 **Ciardo G, Luettgen G, Siminiceanu R**, *Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation*, In: **Margaria TaY Wang** (ed.), Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 2031, Springer Berlin Heidelberg, 2001, pp. 328-342 (English), DOI 10.1007/3-540-45319-9_23.

4 **Godefroid P, Hartmanis J, Goos GLeeuwen Jv** (ed.), *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Springer Berlin Heidelberg, 1996, ISBN 3540607617.

5 **Ciardo G**, *Data Representation and Efficient Solution: A Decision Diagram Approach*, 2007.

6 **Ciardo G, Marmorstein R, Siminiceanu R**, *Saturation Unbound*, In: Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer Berlin Heidelberg, 2003, pp. 379–393.

7 **Ciardo G, Siminiceanu R**, *Structural symbolic CTL model checking of asynchronous systems*, In: Computer Aided Verification (CAV'03), LNCS 2725, Springer Berlin Heidelberg, 2003, pp. 40–53.

8 **Ciardo G, Yu AJ**, *Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning*, Correct Hardware Design and Verification Methods, **3725**, (2005), 146–161.

9 **Ciardo G, Zhao Y, Jin X**, *Ten Years of Saturation: A Petri Net Perspective*, In: **Jensen KaD Susanna and Kleijn** (ed.), Transactions on Petri Nets and Other Models of Concurrency V, Lecture Notes in Computer Science, Vol. 6900, Springer Berlin Heidelberg, 2012, pp. 51–95.

10 **Clarke E, Grumberg O, Peled DA**, *Model Checking*, The MIT Press, 1999.

11 **Javier Esparza and Keijo Heljanko**, *Unfoldings – A Partial-Order Approach to Model Checking*, EATCS Monographs in Theoretical Computer Science, Springer-Verlag, 2008, March.

12 **Heljanko K**, *Bounded Reachability Checking with Process Semantics*, In: Proceedings of the 12th International Conference on Concurrency Theory, CONCUR '01, Springer Berlin Heidelberg; London, UK, 2001, pp. 218–232, `http://dl.acm.org/citation.cfm?id=646736.701769`.

13 **Murata T**, *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE, **77**(4), (1989, April), 541–580, DOI 10.1109/5.24143.

14 **Németh E, Bartha T**, *Formal Verification of Safety Functions by Reinterpretation of Functional Block Based Specifications*, In: **Cofer DaF Alessandro** (ed.), Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, Vol. 5596, Springer Berlin Heidelberg, 2009, pp. 199–214.

15 **Ogata S, Tsuchiya T, Kikuno T**, *SAT-Based Verification of Safe Petri Nets*, In: **Wang F** (ed.), Automated Technology for Verification and Analysis, Lecture Notes in Computer Science, Vol. 3299, Springer Berlin Heidelberg, 2004, pp. 79–92, DOI 10.1007/978-3-540-30476-0_11.

16 **Pastor E, Cortadella J, Roig O**, *Symbolic analysis of bounded Petri nets*, Computers, IEEE Transactions on, **50**(5), (2001), 432-448, DOI 10.1109/12.926158.

17 **Pastor E, Roig O, Cortadella J, Badia RM**, *Petri Net Analysis Using Boolean Manipulation*, In: Proceedings of the 15th International Conference on Application and Theory of Petri Nets, Springer Berlin Heidelberg; London, UK, UK, 1994, pp. 416–435, `http://dl.acm.org/citation.cfm?id=647741.735973`.

18 **Jensen K**, *Condensed state spaces for symmetrical coloured Petri nets*, Formal Methods in System Design, **9**(1), (1996), 7–40.

19 **Ciardo G**, *Data Representation and Efficient Solution: A Decision Diagram Approach*, In: **Bernardo MaH Jane** (ed.), Formal Methods for Performance Evaluation, Lecture Notes in Computer Science, Vol. 4486, Springer Berlin Heidelberg, 2007, pp. 371–394, DOI 10.1007/978-3-540-72522-0_9.

20 **Ciardo G, Zhao Y, Jin X**, *Ten Years of Saturation: A Petri Net Perspective*, In: **Jensen KaD Susanna and Kleijn** (ed.), Transactions on Petri Nets and Other Models of Concurrency V, Lecture Notes in Computer Science, Vol. 6900, Springer Berlin Heidelberg, 2012, pp. 51–95, DOI 10.1007/978-3-642-29072-5_3.

21 **Bartha T, Vörös A, Jámbor A, Darvas D**, *Verification of an Industrial Safety Function Using Coloured Petri Nets and Model Checking*, In: Proceedings of the 14th International Conference on Modern Information Technology in the Innovation Processes of the Industrial Entreprises (MITIP 2012), Hungarian Academy of Sciences, Computer and Automation Research Institute; Budapest, Hungary, 2012, pp. 472–485.

22 **Vörös A, Darvas D, Bartha T**, *Bounded saturation-based CTL model checking*, Proceedings of the Estonian Academy of Sciences, **62**(1), (2013, March), 59–70, DOI 10.3176/proc.2013.1.07.

23 **Goldshtein S, Zurbalev D, Flatow I**, *Pro .NET Performance*, Apress, 2012, ISBN 978-1-4302-4458-5, DOI 10.1007/978-1-4302-4459-2.

24 *Webpage of the SAL tool. (accessed: 1 August 2013) http://sal.csl.sri.com/*, `http://sal.csl.sri.com/`.