Computing and Informatics, Vol. 40, 2021, 195–215, doi: 10.31577/cai_2021_1_195

MORE EFFICIENT ON-THE-FLY VERIFICATION METHODS OF COLORED PETRI NETS

Cong He, Zhijun Ding*

Department of Computer Science and Technology
Tongji University
No. 4800, Caoan highway
Shanghai, China
e-mail: {1930766, dingzj}@tongji.edu.cn

Abstract. Colored Petri Nets (CP-nets or CPNs) are powerful modeling language for concurrent systems. As for CPNs' model checking, the mainstream method is unfolding that transforms a CPN into an equivalent P/T net. However the equivalent P/T net tends to be too enormous to be handled. As for checking CPN models without unfolding, we present three practical on-the-fly verification methods which are all focused on how to make state space generation more efficient. The first one is a basic one, based on a standard state space generation algorithm, but its efficiency is low. The second one is an overall improvement of the first. The third one sacrifices some applicability for higher efficiency. We implemented the three algorithms and validated great efficiency of latter two algorithms through experimental results.

Keywords: Model checking, CPN, on-the-fly, LTL, state space

Mathematics Subject Classification 2010: 93-A30

1 INTRODUCTION

CPNs are powerful graphical language for modeling concurrent systems introduced by Jensen in 1981 [9]. As a kind of high-level Petri nets, CPN is a Petri net that extends the type of place (token) to describe different data types. Moreover, arcs

* Corresponding author

in CPN are labelled with arc expression functions to describe data operations; transitions in CPN are labelled with guard functions to describe branch conditions. In this way, CPN combines the capabilities of Petri nets and a high-level programming language. Success stories of CPN can be found in many industrial domains, such as network protocols [14], systematic softwares [11, 15], embedded systems [3], e-commerce systems [20], etc.

Explicit-state on-the-fly verification [4, 8, 7] is an universal optimization approach for model checking. It integrates state space generation, product automaton construction and counterexample detection (in LTL (Linear Temporal Logic) model checking, a counterexample is an accepting cycle in product automaton). An advantage of this approach is that the algorithm can give an answer without generating full state space. Though success stories of on-the-fly in P/T nets clearly demonstrate its effectiveness and applicability, there are few works dedicated to directly applying on-the-fly in checking CPN models. As for checking CPN models, the mainstream approach is unfolding [16, 13, 12, 2], which transforms a CPN into an equivalent P/T net and implements model checking on the latter. With unfolding, one can directly apply all successful optimization techniques which are difficult to extend to CPNs on the equivalent net, like Data Decision Diagram (DDD) [5, 1], P-invariants [18]. However, a big disadvantage of unfolding is that the equivalent P/T nets transformed from a CPN tends to be too enormous to be handled, with much more places and transitions. Also, the transformed P/T nets cannot directly describe the system to be verified. If a counterexample is detected by verification process, it is difficult to be directly reflected into the system, which is not friendly for debugging.

Concerned with checking CPN models without unfolding, we present a basic on-the-fly method, named *full-info algorithm* (*FullInfo*). It is based on the standard state space generation algorithm [10]. Its core idea is that once a new reachable state m is generated, it calculates a set of all enabled binding elements (we call the set ENBE) in m and stores ENBE together with marking. ENBE serves two purposes. One is to calculate successors of m. Another one is to help check atomic propositions carried by a Büchi automaton state during the generation of product automaton states (or product states for short). For example, some atomic propositions may check enabling of transitions, and enabling of transitions can be reflected by ENBE (for a transition t, if there exists an enabled binding element of t in ENBE, t is enabled. Otherwise, t is not enabled). The algorithm is simple to implement but may generate much redundant information during on-the-fly. A great characteristic of on-the-fly is that it terminates exploration upon a counterexample is detected. Thus, in most cases, the set ENBE of every state m is not fully utilized, cause many successors of m have not been calculated before termination. Also, for some special LTL formulas whose atomic propositions are all related to numbers of tokens (this kind of LTL formulas are called LTLCardinality[1] formulas), ENBE can help

[1] This terminology originated from MCC (Model Checking Contest) which is an annual competition for model checking. `https://mcc.lip6.fr/`.

nothing, because checking these atomic propositions never refers to information in ENBE. This leads to waste of computing resources and low efficiency.

Besides *FullInfo*, we introduce two more efficient state space generation methods integrated into on-the-fly, namely, *minimum representative algorithm* (*MinRep*) and *dynamic exploration algorithm* (*DynExp*). *MinRep* is inspired by *canonical representative* in [17]. Its core idea is that for every enabled transition t in a newly calculated reachable state m, only a representative of enabled binding element of t is initially calculated. While in *DynExp*, none enabled binding element is initially calculated in every newly generated reachable state m. Every enabled binding element in m is calculated on demand when a new successor of m needs generating to start a new path. However, without complete enabled transitions, *DynExp* is hard to check atomic propositions related to enabling of transitions. Thus, it is limited to LTLCardinality formulas.

In short, the main contributions of this paper are summarized as follows:

1. Concerned with LTL model checking of CPN without unfolding, we present an efficient on-the-fly verification method, named *MinRep*. It is an overall improvement of *FullInfo*.
2. For LTLCardinality formulas, we present another more efficient on-the-fly verification method, named *DynExp*.
3. We implemented *FullInfo*, *MinRep* and *DynExp* and did a number of experiments to demonstrate high efficiency of the latter two algorithms.

The rest of this paper is organized as follows: In Section 2, we introduce the definition of Colored Petri Nets and Linear Temporal Logics. In Section 3, we briefly introduce standard state space generation and on-the-fly verification. Then in Section 4, we specify a binding elements calculation problem from the core part of state space generation and on-the-fly. In Sections 5, 6, 7, we elaborate on *FullInfo*, *MinRep* and *DynExp*. Their strengths and weaknesses are discussed as well. Implementation and experimental results are given in Section 8. Finally, in Section 9, we present our conclusion.

2 PRELIMINARIES

2.1 Colored Petri Nets

In this section, definitions of multi-set and non-hierarchical CPN are cited [10] and definitions of LTL are cited [6, 21]. As a matter of convenience, $Bool = \{false, true\}$ is the set of Boolean types, where *true* and *false* are two predicates respectively. $Type[v]$ is the data type of variable v. $Type[ex]$ is the type of expression ex. $EXPR_V$ is an expression constituted by elements from set V.

Definition 1 (Multi-set)**.** Let $S = \{s_1, s_2, s_3, \dots\}$ be a non-empty set. A multi-set m is a function over $S : S \to \mathbb{N}$ that maps each element $s \in S$ into a non-negative

integer $m(s) \in \mathbb{N}$ called the number of appearances (coefficient) of s in m. A multi-set m can also be written as a sum (the operator '++' is a natural addition '+' when two elements s_1, s_2 are the same data type, otherwise '++' is just a junction symbol without real meaning):

$$^{++}\sum_{s \in S} m(s)'s = m(s_1)'s_1 ++ m(s_2)'s_2 ++ m(s_3)'s_3 ++ \ldots$$

Operators: addition ($++$), scalar multiplication ($**$), comparison ($\ll=$), size ($|m|$) and subtraction ($--$) are defined as follows:

- addition: $\forall s \in S$, $(m_1 ++ m_2)(s) = m_1(s) + m_2(s)$,
- scalar multiplication: $\forall s \in S$, $(n ** m)(s) = n * m(s)$,
- comparison: $m_1 \ll= m_2 \Leftrightarrow \forall s \in S$, $m_1(s) \leq m_2(s)$,
- size: $|m| = \sum_{s \in S} m(s)$,
- when $m_1 \ll= m_2$, subtraction is defined as: $\forall s \in S$, $(m_2 -- m_1)(s) = m_2(s) - m_1(s)$.

Definition 2 (Non-hierarchical CPN)**.** A non-hierarchical CPN is a nine-tuple $N = (P, T, A, \Sigma, V, C, G, E, I)$, where P, T, A are finite sets of places, transitions and arcs such that $P \cap T = \emptyset$, $A \subseteq P \times T \cup T \times P$, Σ is finite set of non-empty color sets, V is a finite set of typed variables such that $Type[v] \in \Sigma$ for all variables $v \in V$, $C : P \to \Sigma$ is a color set function that assigns a color set to each place, $G : T \to EXPR_V$ is a guard function that assigns a guard to each transition t such that $Type[G(t)] = Bool$, $E : A \to EXPR_V$ is an arc expression function that assigns an arc expression to each arc a such that $Type[E(a)] = C(p)_{MS}$ where p is the place connected to the arc a, $I : P \to EXPR_\emptyset$ is an initialization function that assigns an initialization expression to each place p such that $Type[I(p)] = C(p)_{MS}$. The variables of a transition t are denoted $Var(t)$, $Var(t) \subseteq V$. $Var(t)$ includes all the variables appearing in t's guard $G(t)$ and arc expressions $E(a)$ for all $a \in A$, a is connected to t.

Definition 3 (Enabling and firing rules)**.** Let $N = (P, T, A, \Sigma, V, C, G, E, I)$ be a non-hierarchical CPN. A *marking* of N is a function M that maps each place $p \in P$ into a multi-set of tokens $M(p) \in C(p)_{MS}$. A *binding* of a transition t is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in Type[v]$. The set of all bindings for a transition t is denoted $B(t)$, called t's *binding space*. A *binding element* is a pair (t, b) such that $t \in T, b \in B(t)$. The set of all binding elements for a transition t is denoted $BE(t)$, called t's *binding element space*. $BE(t)$ is defined by $BE(t) = \{(t, b) \mid b \in B(t)\}$. The set of all binding elements in a CPN is denoted BE, called *binding element space*. A binding element $(t, b) \in BE$ is enabled in a marking M if and only if the following two properties are satisfied (denotation $G(t)\langle b \rangle$ expresses the evaluation of transition t's guard in the binding b and it is either *true* or *false*; denotation $E(p, t)\langle b \rangle$ expresses the evaluation of arc a's arc expression and it is a multi-set):

1. $G(t)\langle b\rangle = true$;
2. $\forall p \in P, E(p,t)\langle b\rangle \ll = M(p)$.

When (t,b) is enabled in M, it may occur and is leading to a marking M' (written $M \xrightarrow{(t,b)} M'$), such that $\forall p \in P, M'(p) = (M(p) - - E(p,t)\langle b\rangle) + + E(t,p)\langle b\rangle$. A transition t is enabled in a marking M if and only if $\exists (t,b) \in BE(t)$, (t,b) is enabled in M.

Definition 4 (State space)**.** For a marking M and a marking M', if there exists an enabled binding element (t,b) such that $M \xrightarrow{(t,b)} M'$, M' is said to be immediately reachable from M; if there exists an sequence of binding element $(t_1,b_1)(t_2,b_2)\dots(t_n,b_n)$ such that $M \xrightarrow{(t_1,b_1)} M_1 \xrightarrow{(t_2,b_2)} M_2 \dots \xrightarrow{(t_n,b_n)} M'$, M' is said to be reachable from M, written $M \xrightarrow{*} M'$. The state space of a CPN consists of the set $R(m_0) = \{m \mid m_0 \xrightarrow{*} m\}$ of states reachable from the initial state. Each state $m \in R(m_0)$ is called a *reachable state*[2].

2.2 Linear Temporal Logics

Linear Temporal Logic (abbreviated as LTL) is used to describe properties of a system execution. It consists of a non-empty finite set of atomic propositions AP, Boolean operators $\neg$ (negation), $\vee$ (disjunction) and $\wedge$ (conjunction), and temporal operators X (next), U (until), R (release), F (eventually) and G (always). In LTL model checking, the negation of a formula will be transformed into a Büchi automaton. There are many approaches to construct a Büchi automaton from the LTL formula [6, 19].

Definition 5 (Syntax of LTL)**.** The syntax of LTL is defined as follows:

$$\phi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid X\varphi \mid \varphi U \psi \mid \varphi R \psi \mid F\varphi \mid G\varphi$$

where p is an atomic proposition and ϕ, φ, ψ are well-formed LTL formulas. Referring to MCC and Wolf's [21] provisions for atomic propositions, we make the following provisions for an atomic proposition p:

$$\begin{aligned} p ::= & TRUE \mid FALSE \mid FIREABLE(t)\,(t \in T) \mid DEADLOCK \\ & \mid k_1 p_1 + \dots + k_n p_n \leq k\,(k_i, k \in \mathbb{Z}, p_i \in P) \end{aligned}$$

Let state m be the current state, $FIREABLE(t)$ holds if only if t is enabled in m, $DEADLOCK$ holds if and only if there are no transitions are enabled in m, $k_1 p_1 + \dots + k_n p_n \leq k$ holds if and only if $k_1 M(p_1) + \dots + k_n M(p_n) \leq k$ in m.

[2] State is a snapshot of a system, marking is a distribution of tokens. Though a state m can be uniquely identified by a marking M, they are different concepts. Throughout the paper, we use lower case m (subscripts or superscripts will be used if necessary) to represent a state, upper case M(subscripts or superscripts will be used if necessary) to represent the marking of m.

Definition 6 (Semantics of LTL)**.** Let AP be a non-empty finite set of atomic propositions, $\xi = x_0x_1x_2\ldots$ be a sequence over alphabet 2^{AP}, ϕ, φ, ψ be LTL formulas. We write ξ_i for the suffix of ξ starting at x_i. The semantics $\xi \models \phi$ (ξ models ϕ) is defined as follows:

- $\xi \models p$, iff $p \in x_0$ for $p \in AP$,
- $\xi \models \neg\phi$, iff $\xi \not\models \phi$,
- $\xi \models \varphi \vee \psi$, iff $\xi \models \varphi$ or $\xi \models \psi$,
- $\xi \models X\phi$, iff $\xi_1 \models \phi$,
- $\xi \models \varphi U \psi$, iff $\exists i \geq 0$, $\xi_i \models \psi \wedge (\forall j < i, \xi_i \models \varphi)$.

Other operators ($\wedge$, R, F, G) can be derived from the above operators (X, U, $\neg$): $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$; $\varphi R \psi \equiv \neg(\neg\varphi U \neg\psi)$; $F\phi \equiv (TRUE)U\phi$; $G\phi \equiv \neg(F\neg\phi)$.

3 STANDARD STATE SPACE GENERATION AND ON-THE-FLY

3.1 Standard State Space Generation

The standard state space generation [10] works on three sets: Node, Unprocessed, Edges. Node stores reachable states. Unprocessed consists of states whose successors have not yet been calculated. Edges stores arcs. As illustrated in Algorithm 1, the algorithm firstly initializes Node, Unprocessed with initial state m_0 and Edges with empty set. Then it selects a reachable state m in Unprocessed and calculates all enabled binding elements in m. Each enabled binding element that occurs will lead to a reachable state m' and an arc from m to m'. If m' has not yet been encountered, it will be added into Node and Unprocessed. The algorithm terminates with full state space.

3.2 On-the-Fly

On-the-fly method was first proposed in [4]. The main idea is integrating state space generation, product automaton construction and detecting counterexamples (in LTL model checking, a counterexample is an accepting cycle in product automaton). In more detail, for a given product state $p :: (m, b)$ (a product state is composed by a reachable state m and an automaton state b), it calculates a successor m' of m, and a successor b' of b. if all atomic propositions carried by b' are satisfied in m', then a product state $p' :: (m', b')$ is generated. If some conditions are triggered, on-the-fly will implement counterexample detection, i.e., if on-the-fly finds the successor p' of p is an encountered product state where it may form a cycle, then on-the-fly will check that. This idea can be illustrated by Algorithm 2. Line 1 is state space generation, lines 3–4 are the product state generation and the line 6 is counterexample detection. As for counterexample detection, there are several ways to do that.

Algorithm 1 Standard state space generation

```
1: NODES ← {m_0}
2: UNPROCESSED ← {m_0}
3: EDGES ← ∅
4: while UNPROCESSED ≠ ∅ do
5:     Select a Marking m in UNPROCESSED
6:     UNPROCESSED ← UNPROCESSED − {m}
7:     for all binding elements (t, b) such that (t, b) is enabled in m do
8:         Calculate m' such that m --(t,b)--> m'
9:         EDGES ← EDGES ∪ {(m, (t, b), m')}
10:        if m' ∉ NODES then
11:            NODES ← NODES ∪ {m'}
12:            UNPROCESSED ← UNPROCESSED ∪ {m'}
13:        end if
14:    end for
15: end while
```

Like nested depth-first search algorithm [4], TCHECK[3] algorithm [7] and DCHECK algorithm [7].

Algorithm 2 on-the-fly

Input: $p :: (m, b)$: a product state
Output: *true* or *false*: checking result

```
1: for m' ← NEXTSUCCESSOR(m) ≠ 'no more' do
2:     for all b' ∈ SUCCESSOR(b) do
3:         if m' satisfies all atomic propositions carried by b' then
4:             Generate a produce state p' :: (m', b')
5:             if p' has been encountered then
6:                 Accepting cycle detection
7:                 if ∃ an accepting cycle then
8:                     Terminate with false
9:                 end if
10:            else
11:                on-the-fly(p')
12:            end if
13:        end if
14:    end for
15: end for
```

[3] The main procedure of TCHECK and DCHECK are non-recursive functions, and they work much more efficiently than nested depth-first search. *FullInfo*, *MinRep* and *DynExp* are integrated into TCHECK algorithm. More details can be referred to [7].

4 ENABLED BINDING ELEMENTS CALCULATION PROBLEM

Enabled binding elements are vitaly important during state space generation. Firstly, all successors of a reachable state are controlled by enabled binding elements (lines 7–8 in Algorithm 1 and line 1 in Algorithm 2). Secondly, enabled binding elements plays a part in product state generation (line 3 in Algorithm 2), because some atomic propositions may check enabling of some transitions, i.e., $FIREABLE(t)$ atomic propositions. The core problem that we encounter is: given a reachable state m, in which way to explore binding element space BE to find enabled binding elements in m to calculate successors of m and generate product states. Different solutions to this problem lead to huge different performances. Intuitively, we may come up with that upon a new reachable state m is generated, explore BE exhaustively at once to get *all* enabled binding elements ENBE in m and store ENBE in case to use. In this way, every time on-the-fly backtracks to m, the process can easily fetch a next enabled binding element from ENBE to calculate another successor of m. This is exactly how *FullInfo* works. We will detail it in the next section.

5 FULLINFO

The core idea of *FullInfo* is very simple: upon getting a new reachable state m, it calculates a set of all enabled binding elements in m called ENBE and stores ENBE together with marking M immediately. Then it uses ENBE to generate different successors of m and product states. The technical difficulties lie in how to get all enabled binding elements and how to manage them.

5.1 How to Get All Enabled Binding Elements

Traversing t's binding space $B(t)$ is essentially a combination problem that assigns a value $b(v) \in Type[v]$ to each variable $v \in Var(t)$. The binding space $B(t)$ can be depicted as a tree (we name it t's *binding space tree*). Assume that $t \in T$ is a transition, $|Var(t)| = k$ such that $Var(t) = \{v_1, v_2, \ldots, v_k\}$ and for each variable v_i, $|Type[v_i]| = n_i$ such that $Type[v_i] = \{c_{i1}, c_{i2}, \ldots, c_{in_i}\}$, then $B(t)$ can be depicted as a tree in Figure 1. The depth of this tree is equal to the number of variables in $Var(t)$. All direct successors of a node are overall mapping cases of next variable. For example, the direct successors of node '$v_1 = c_{11}$' list complete mapping cases of next variable v_2, which is from $v_2 = c_{21}$ to $v_2 = c_{2n_2}$ (we use horizontal ellipsis to represent all omitted nodes in its layer and vertical ellipsis to represent all omitted child nodes of one node). A path from *root* node to a leaf node is a specific binding of t, and all paths like this constitute t's binding space $B(t)$. We use a recursive function to traverse this tree to get all enabled binding elements. The function is presented in Algorithm 3. When the depth is lower than $|Var(t)|$, the function tries to assign a color to the variable which corresponds to the depth and then recurves down. If the depth is equal to or greater than $|Var(t)|$ which means the function

reaches a leaf node and a complete binding b is obtained, then it begins to check the enabling of (t, b). The specific checking procedure lies in lines 2–9. If (t, b) is enabled, it will be added into ENBE. After implementing this function on each transition $t \in T$, the complete ENBE will be obtained.

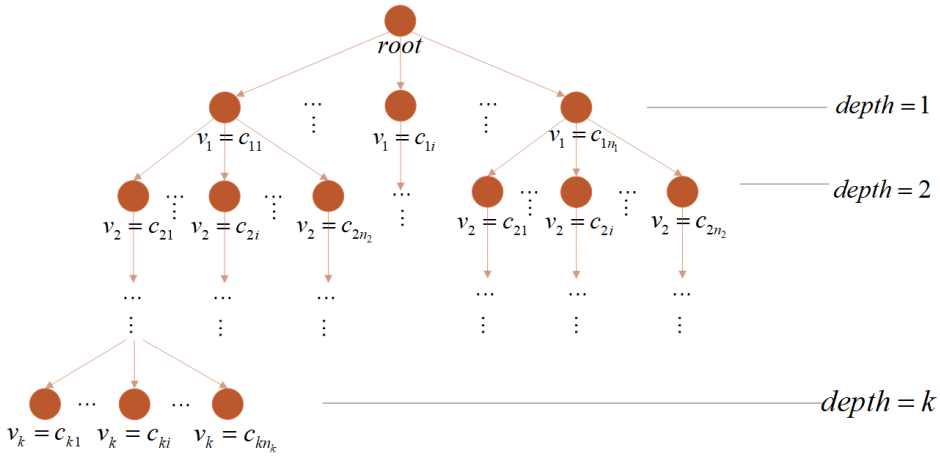


Figure 1. Binding space tree

Algorithm 3 getENBE$(m, t, b, depth)$

Input: m: reachable state, t: Transition, b: Binding, $depth$: int
Output: ENBE: a set stores enabled binding elements

```
1:  if depth ≥ |Var(t)| then
2:      if ¬G(t)⟨b⟩ then
3:          return
4:      end if
5:      for all p ∈ •t do
6:          if ¬(E(p,t)⟨b⟩ ≤ M(p)) then          ▷ M is m's marking
7:              return
8:          end if
9:      end for
10:     ENBE.ADD(b)
11: else
12:     for all c ∈ Type[v_depth] do
13:         b[depth] ← c
14:         getENBE(m, t, b, depth + 1)
15:     end for
16: end if
```

5.2 How to Manage All Enabled Binding Elements

In this subsection, we focus on how to take advantage of ENBE to serve for successor reachable states generation and product states generation. We use a two-level queue as data structure for ENBE. The first level queue stores enabled transitions, each enabled transition has a second level queue consisting of its bindings which render it enabled. Figure 2 is an example of

$$\text{ENBE} = \{(t_1, b_{11}), (t_1, b_{12}), (t_2, b_{21}), (t_2, b_{22}), (t_2, b_{23}), (t_3, b_{31})\}.$$

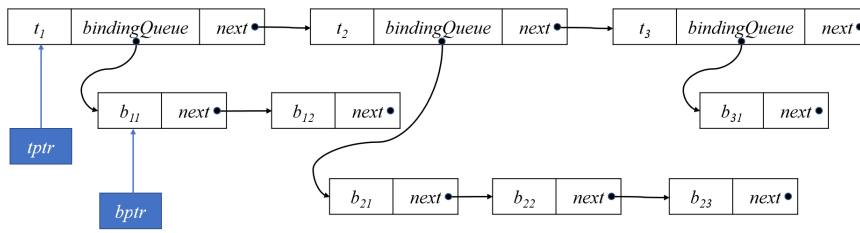


Figure 2. Data structure for ENBE

In the data structure of ENBE, there are two pointers, *tptr* and *bptr*, respectively pointing to an enabled transition and a binding of it. They are used to represent an enabled binding element, i.e., in Figure 2 they represent (t_1, b_{11}). Each time an enabled binding element occurs, *bptr* will move to next binding of the current queue. If next binding does not exist, i.e., it reaches the tail of the queue, *tptr* will move to next transition and *bptr* will point to the head of its *bindingQueue*. By this, the process can obtain different successors of a reachable state and this procedure is one possible way how line 7 in Algorithm 1 and line 1 in Algorithm 2 work.

Another crucial role of ENBE is to help generate a product state. For example, let $F\alpha$ be a LTL formula, where α is an atomic proposition $FIREABLE(t_2)$. During checking process, every state needs to check if t_2 is enabled in it. To do this, every state just needs to check its ENBE. If t_2 appears in the first level queue, it is enabled, otherwise it is not.

With *FullInfo*, we can basically solve the *enabled binding elements calculation problem*. The two core parts, successor reachable states generation and product states generation, can be done easily with the aid of ENBE. But a conspicuous disadvantage is that it may generate much redundant information. Or in other words, many states' ENBE may not be fully utilized. For example, if on-the-fly reports a checking result without generating the whole state space, that means there must exist some states where some enabled binding elements have not yet occurred and these enabled binding elements remain to be redundant. Another case is when none of atomic propositions of the LTL formula is form of $FIREABLE(t)$ or $DEADLOCK$, ENBE can do nothing to help in the product state generation. This disadvantage is

particularly obvious when a CPN's binding element space is huge or when on-the-fly detects a counterexample along a path with few backtrackings. Here, Figure 3 is an example to demonstrate the second case. Figure 3 is a partial state space of a CPN. We use solid cycles to represent reachable states, arrows marked by transitions to represent enabled transitions in a reachable state, solid squares to represent enabled binding elements which have occurred and hollow squares to represent enabled binding elements which have not yet occurred. If on-the-fly detected a counterexample $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_0$ after generating S_0, S_1, S_2 and then terminates, then the computing resources allocated for calculating the hollow squares are wasted because they had never been used during checking process. If on-the-fly went deeper along this path and detected a counterexample, the waste would be worse. Therefore, we need another algorithm to solve the *enabled binding elements calculation problem.*

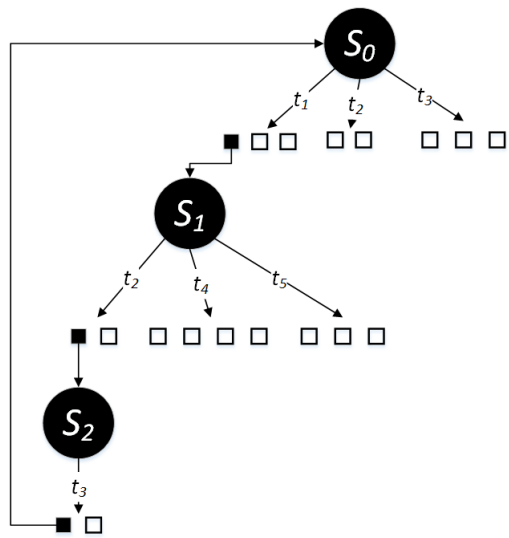


Figure 3. Partial state space

6 MINREP

In this section, we develop another solution to the *enabled binding elements calculation problem.* According to the definition of enabling of transitions (in Definition 3), if there exists one enabled binding element of a transition t, t is proven to be enabled. Thus, as for checking atomic propositions during product state generation, it is unnecessary to calculate complete ENBE in each reachable state. The core idea of *MinRep* is to specify an order over $B(t)$ such that $(B(t), \prec)$ for each transition t. And for t's binding element space, this algorithm only initially calculates one en-

abled representative which is the smallest enabled one in $BE(t)$. Certainly, if t is not enabled, there will not be such a representative. This idea is inspired by *canonical representative* [17].

Before presenting the order $(B(t), \prec)$, we firstly specify an order $(C, \prec)$ over each color set $C \in \Sigma$. Here are the orders:

1. $(C, \prec)$: $\forall c_i, c_j \in C$, $c_i \prec c_j$ iff $i < j$. Here the index i, j can be arbitrarily defined. Typecially we use the index in data structure storing color set C, i.e., sequence table.
2. $(B(t), \prec)$: $Var(t) = \{v_1, v_2, \ldots, v_n\}$, $\forall b_i, b_j \in B(t)$, $b_i = \langle c_{i1}, c_{i2}, \ldots, c_{in}\rangle$, $b_j = \langle c_{j1}, c_{j2}, \ldots, c_{jn}\rangle$, $b_i \prec b_j$ iff $\exists k, 1 \leq k \leq n, c_{ik} \prec c_{jk}$ and $\forall m, 1 \leq m < k, c_{im} = c_{jm}$. $(B(t), \prec)$ can be regarded as a lexicographical order induced by the vector of binding.

Calculating representative is similar to Algorithm 3. The minor difference is that for *MinRep*, upon getting an enabled binding element, it terminates. More specifically, we just need to insert a terminate clause after line 10. All representatives are organized in a set, and we name it Ent. Here we use a queue to organize Ent. Figure 4 is an example of

$$\text{Ent} = \{(t_1, b_{11}), (t_2, b_{21}), (t_3, b_{31})\}.$$

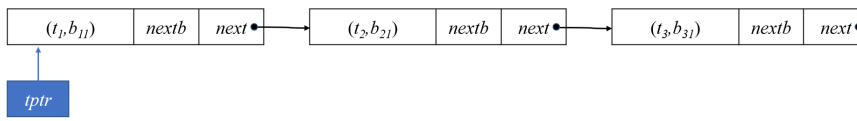


Figure 4. Data structure for Ent

In this data structure, *tptr* is a pointer pointing to a transition occurring last time and *nextb* is a binding that is prepared to calculate a successor reachable state next time (initially it equals to the binding part of the related representative). When on-the-fly backtracks to a reachable state m, the program uses *nextb* to calculate a successor m' of m and tries to check bindings behind $(tptr- > nextb)$ to update *nextb*. If there are no more bindings related to *tptr*, *tptr* will move to next representative, which indicates the binding element space of prior transition has been explored exhaustively. The successor generation procedure is presented in Algorithm 4 where $(m.tptr- > t)$ is the transition of corresponding representative which pointed by *tptr*. Function UPDATE(m) is to update $(tptr- > nextb)$. It regards the binding vector as a n-digit number, where n equals to $|Var(m.tptr- > t)|$, and tries to implement increment '++' on this number. During the increment operation, it may trigger carries (lines 7–8) like numbers. Function NEXTSUCCESSOR(m) is to get another successor of m according to binding element $(m.tptr- > t, m.tptr- > nextb)$.

As for checking atomic propositions, ENT together with marking is right enough. As for $FIREABLE(t)$ propositions, *MinRep* checks ENT of current reachable state m. If there is a related representative of t, t is enabled, otherwise, it is not. As for *DEADLOCK* propositions, *MinRep* also checks ENT of current reachable state m. If ENT is empty, the propositions are satisfied, otherwise, it is not. As for $k_1p_1 + \cdots + k_np_n \leq k$ propositions, *MinRep* checks them by the marking M of current reachable m. if $k_1M(p_1) + \cdots + k_nM(p_n) \leq k$ holds, the propositions are satisfied, otherwise, it is not.

Algorithm 4 MinRep

Input: m: current reachable state
Output: m': successor of m or 'no more'

```
1: function UPDATE(m)
2:     while m.tptr ≠ NULL do
3:         (t', b') ← (m.tptr− > t, m.tptr− > nextb)
4:         n : int ← |Var(t')|                          ▷ Var(t') = {v_1, v_2, ..., v_n}
5:         for i from n to 1 do
6:             c: color ← NEXTCOLOR(b[i])
7:             if c = 'no more' then
8:                 b'[i] ← first color of Type[v_i]
9:                 continue
10:            else
11:                b'[i] ← c
12:                if (t', b') is enabled in m then
13:                    m.tptr− > nextb ← b'
14:                    return
15:                end if
16:            end if
17:        end for
18:        tptr ← tptr− > next
19:    end while
20:    return
21: end function
22:
23: function NEXTSUCCESSOR(m)
24:    if tptr = NULL then
25:        return 'no more'
26:    else
27:        Calculate m' such that m --(m.tptr−>t, m.tptr−>nextb)--> m'
28:        UPDATE(m)
29:        return m'
30:    end if
31: end function
```

For each reachable state m, *MinRep* only calculates partial information from m's binding element space which is just enough to handle all kinds of atomic propositions. Compared with *FullInfo*, redundant information is much less and efficiency would be higher. But it has the same disadvantage that if all atomic propositions of the LTL formula are neither form of *DEADLOCK* nor *FIREABLE*(t), i.e., LTLCardinality formulas, ENT can help nothing and remain to be redundant. So we need another more efficient algorithm to handle LTLCardinality formulas.

7 DYNEXP

In this section, we develop another algorithm dedicated to handling LTLCardinality formulas. As for checking atomic propositions of this formula type, it is unnecessary to calculate any enabled binding elements. The sole function of enabled binding elements here is to calculate successors. In order to be more efficient, *DynExp* will not initially calculate any enabled binding element in each newly calculated reachable state m. Instead of calculating all enabled binding elements at once, enabled binding elements are obtained dynamically. Once an enabled binding element is obtained, the algorithm will let it occur immediately and calculate a successor m' of m, then continue on-the-fly on m'. To obtain different successors of a given reachable state m when on-the-fly backtracks to m, *DynExp* extends the orders defined in Section 6 to the whole binding element space BE such that $(BE, \prec)$, and each reachable state would record the binding element that occurred last time, called *lastbe*. In this way, when on-the-fly backtracks to m, it can check binding elements behind *lastbe* until an enabled one is detected or there are no more in BE.

Before presenting the order $(BE, \prec)$, we firstly specify an order $(T, \prec)$ over transition set T. They are defined as follows:

1. $(T, \prec)$: $\forall t_i, t_j \in T$, $t_i \prec t_j$ iff $i < j$. Here the index value i, j can be arbitrarily defined. Typically we use its index value in a specific data structure that stores the transition set.
2. $(BE, \prec)$: $\forall (t_i, b_{ik}), (t_j, b_{jm}) \in BE$, $(t_i, b_{ik}) \prec (t_j, b_{jm})$ iff $t_i \prec t_j$ or $i = j$, $b_{ik} \prec b_{jm}$.

With the aid of order $(BE, \prec)$ and *lastbe*, the algorithm can iterate over BE exhaustively to get different enabled binding elements in each reachable state. We specify three functions to implement the idea, namely, NEXTBINDING$((t, b))$, NEXTTRANSITION$((t, b))$ and NEXTSUCCESSOR(m). They are illustrated in Algorithm 5. Function NEXTBINDING$((t, b))$ is similar to function UPDATE(m) illustrated in Algorithm 4. It is to fetch (t, b)'s next binding of transition t according to order $(BE, \prec)$. Where function NEXTCOLOR(c) is to get color c's next color in c's color set C by order $(C, \prec)$. Function NEXTTRANSITION$((t, b))$ is very simple. Its job is to fetch next transition of t according to order $(T, \prec)$, and initiate the binding (lines 23–25). If there are no more transitions, 'NULL' will be returned.

Algorithm 5 DynExp

Input: m: current reachable state
Output: m': successor of m

```
1: function NEXTBINDING((t, b))
2:     (t', b') ← (t, b)
3:     n : int ← |Var(t)|
4:     for i from n to 1 do
5:         c: color ← NEXTCOLOR(b[i])
6:         if c = 'no more' then
7:             b'[i] ← first color of Type[v_i]
8:             continue
9:         else
10:            b'[i] ← c
11:            return (t', b')
12:        end if
13:    end for
14:    return 'no more'
15: end function
16:
17: function NEXTTRANSITION((t, b))
18:    (t', b') : binding element
19:    t' ← t.index++
20:    if t' = 'no more' then
21:        return 'no more'
22:    else
23:        for i from 1 to |Var(t')| do
24:            b'[i] ← first color of Type[v_i]
25:        end for
26:    end if
27:    return 'no more'
28: end function
29:
30: function NEXTSUCCESSOR(m)
31:    repeat
32:        repeat
33:            (t, b) ← NEXTBINDING(m.lastbe)
34:            if (t, b) is enabled in m then
35:                Calculate m' such that m --(t,b)--> m'
36:                return m'
37:            end if
38:        until (t, b) = 'no more'
39:        (t, b) ← NEXTTRANSITION(m.lastbe)
40:    until (t, b) = 'no more'
41:    return 'no more'
42: end function
```

Function NEXTSUCCESSOR(m) keeps calling NEXTBINDING($m.lastbe$) to iterate over $B(t)$, trying to find an enabled one. If there are no more or do not exist at all, it will move to next transition by calling NEXTTRANSITION($m.lastbe$). Upon obtaining an enabled binding element (t, b), (t, b) will occur immediately leading to a successor m' of m and terminates this function. Or if there are no more enabled binding element, NEXTSUCCESSOR(m) will return 'no more'.

As for generating product states, we have nothing to worry about, because checking LTLCardinality formulas just need information of markings and markings are never absent.

Because upon getting a successor, the algorithm terminates exploring binding element space and continues on-the-fly, any enabled binding element and corresponding successor are calculated on demand during the checking process. Hence, no redundant information is generated and *DynExp* would be more efficient than *FullInfo* and *MinRep*. However, it is limited to LTLCardinality formulas. It sacrifices applicability for greater efficiency.

8 EXPERIMENT

We implemented all three algorithms in C++, and they are all integrated into the non-recursive on-the-fly TCHECK. The source code is available from:

- *FullInfo*: `https://github.com/Tj-Cong/EnPAC_CPN`,
- *MinRep*: `https://github.com/Tj-Cong/EnPAC_CPN_F`,
- *DynExp*: `https://github.com/Tj-Cong/EnPAC_CPN_C`.

We get testing data from MCC. There are two kinds of models provided by MCC:

- Academic models: these were designed in universities by researcher, to benchmark some tools, to illustrate a typical situation or within the context of academic projects and cooperations.
- Industrial models: these where designed within the context of industrial projects.

Both kinds of models have practical meanings and each model is provided with a file describing it which can be found from `https://mcc.lip6.fr/models.php`. Each model can result in several instances due to the scaling parameter (the parameters are often indicated at the end of its instance name). There are two kinds of LTL formulas. One is called LTLCardinality formulas whose atomic propositions are all form of $k_1p_1 + \cdots + k_np_n \leq k$. Another kind is called LTLFireability formulas whose atomic propositions are all form of $FIREABLE(t)$. Based on this testing data, we have done two sets of experiments. One is designed to measure the performance on LTLcardinality formulas. Another one is designed to measure the performance on LTLfireability formulas. They are both implemented on a Linux PC with Intel(R) Core(TM) i7-7700HQ CPU @ 2.80 GHz and 16 GB RAM. Operating system is Ubuntu 18.04 LTS.

As for the first experiment, all three algorithms were tested on four different instances with different size of binding element space. Each instance is checked by two formulas. Testing time for each formula is limited to 300 seconds, and if one algorithm does not finish checking one formula within 300 seconds, the corresponding table entry will be marked as "?". Memory for each formula is limited to 16 GB and if one algorithm cannot finish checking one formula within 16 GB, the corresponding table entry will be marked as "Overflow". Of course, the three algorithms are set to traverse paths in the same order. The result is presented in Table 1. |States| represents the number of states explored by on-the-fly before termination. The unit of time is seconds; the unit of memory is MB; the size of binding element space $|BE|$ is calculated by:

$$|BE| = \sum_{t \in T} \left(\prod_{v \in Var(t)} (|Type[v]|) \right).$$

From Table 1, we can find that *DynExp* always consumes the least time and memory. Thus, we can conclude that *DynExp* is the most efficient algorithm for LTLcardinality formulas, no matter with respect to memory consumption or time used. *MinRep* ranks the second and *FullInfo* is the least efficient. Beginning from Formula 2 of DWM-COL-40, memory for *FullInfo* overflows. And beginning from Formula 1 of GRA-COL-11, time for *MinRep* runs out of 300 seconds. When $|BE|$ goes larger, the advantage of dynamic exploration becomes more salient. By checking Formula 1 and Formula 2 of every instance, we can also find out that the more states on-the-fly explores, more obvious the advantage of *DynExp* is.

Models	ALD-COL-10[1]		DWM-COL-40[2]		GRA-COL-11[3]		DVM-COL-16[4]	
|Places|	20		11		5		6	
|Transitions|	15		8		7		7	
|BE|	132		12 800		2 705 087		4 433 952	
formulas	1	2	1	2	1	2	1	2
|States|	38 115	43 109	20 936	1 251 201	286 755	545 605	178 433	457 369
Time (FullInfo)	2.510	3.193	36.417	?	?	?	?	?
Time (DynExp)	0.899	1.644	8.089	28.381	27.462	80.120	10.377	57.256
Time (MinRep)	0.905	2.751	14.4249	151.37	> 300	> 300	> 300	> 300
Memory (FullInfo)	506.367	529.363	2 766.367	Overflow	Overflow	Overflow	Overflow	Overflow
Memory (DynExp)	433.348	449.348	504.344	8 080.348	871.244	1387.344	1181.348	2,518.352
Memory (MinRep)	433.348	449.348	508.348	8 252.348	?	?	?	?

[1]The full name is AirplaneLD-COL-0010.
[2]The full name is DatabaseWithMutex-COL-40.
[3]The full name is GlobalResAllocation-COL-11.
[4]The full name is DrinkVendingMachine-COL-16.

Table 1. Comparison on LTLCardinality formulas

As for the second experiment, *FullInfo* and *MinRep* were implemented on four different instances with different size of binding element space. Same as the first experiment, each instance is checked by two formulas and each formula is limited to 300 seconds and 16 GB. The result is presented in Table 2. Obviously, *MinRep* works more efficiently than *FullInfo*. Comparing Formula 2 of FR-COL-G005 with

Formula 1 of GRA-COL-9, we can find that for *MinRep*, the memory consumption is much lower in GRA-COL-09, while for *FullInfo*, the memory consumption is much higher in GRA-COL-9, because the size of binding element space is much bigger. We can also conclude that when $|BE|$ goes larger, the advantage of *MinRep* becomes more obvious. Also, the more states on-the-fly explores, more salient the advantage of *MinRep* is.

Models	ALD-COL-50[1]		DWM-COL-40[2]		FR-COL-G005[3]		GRA-COL-9[4]	
|Places|	20		11		104		5	
|Transitions|	15		8		66		7	
|BE|	612		12 800		134 480		1 003 437	
formulas	1	2	1	2	1	2	1	2
|States|	7	209	2 077	9 596	115 121	31 812	36424	?
Time (FullInfo)	0.104	0.121	3.054	13.600	22.739	57.739	?	?
Time (MinRep)	0.091	0.105	1.628	5.092	3.609	7.097	113.148	>300
Memory (FullInfo)	327.359	334.357	630.359	2 222.359	1 791.363	3 894.363	Overflow	Overflow
Memory (MinRep)	327.351	327.351	347.355	405.355	712.351	1 553.348	400.348	?

[1]The full name is AirplaneLD-COL-0050.
[2]The full name is DatabaseWithMutex-COL-40.
[3]The full name is FamilyReunion-COL-L00200M0020C010P010G005.
[4]The full name is GlobalResAllocation-COL-09.

Table 2. Comparison on LTLFireability formulas

From the two experiments, we can conclude that no matter what kind of LTL formulas is, *MinRep* is always more efficient that *FullInfo*. While for LTLCardinality formulas, *DynExp* works best.

9 CONCLUSIONS

We have presented a basic state exploration method and two more efficient ones under the framework of on-the-fly. The basic one, *FullInfo*, simply calculates all enabled binding elements for every newly generated reachable state. It is easy to implement but efficiency is low. *MinRep* is 'semi-dynamic'. It calculates all enabled transitions for every newly generated reachable state, but for each enabled transition, only a minimum representative of enabled binding elements is initially calculated, others are calculated dynamically on demand. While *DynExp* is 'fully-dynamic'. Every enabled binding element is calculated on demand by on-the-fly. As for applicability, *FullInfo*=*MinRep*>*DynExp*. As for efficiency, *DynExp*>*MinRep*>*FullInfo*.

Acknowledgement

This work is partially supported by the National Key Research and Development Program of China under the Grant No. 2018YFB2100800 and the National Natural Science Foundation of China under Grant No. 61672381, and in part by the Fundamental Research Funds for the Central Universities under Grant No. 22120180508.

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. In: Koutny, M., Kristensen, L., Penczek, W. (Eds.): Transactions on Petri Nets and Other Models of Concurrency XIII. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 11090, 2018, pp. 73–92, doi: 10.1007/978-3-662-58381-4_4.

[2] Bergenthum, R.—Lorenz, R.—Mauser, S.: Faster Unfolding of General Petri Nets Based on Token Flows. In: van Hee, K. M., Valk, R. (Eds.).: Applications and Theory of Petri Nets (PETRI NETS 2008). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5062, 2008, pp. 13–32, doi: 10.1007/978-3-540-68746-7_6.

[3] Christensen, S.—Jørgensen, J. B.: Analysing Bang & Olufsen's Beolink® Audio/Video System Using Coloured Petri Nets. In: Azéma, P., Balbo, G. (Eds.): Application and Theory of Petri Nets 1997 (ICATPN 1997). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1248, 1997, pp. 387–406, doi: 10.1007/3-540-63139-9_47.

[4] Courcoubetis, C.—Vardi, M. Y.—Wolper, P.—Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In: Clarke, E. M., Kurshan, R. P. (Eds.): Computer-Aided Verification (CAV 1990). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 531, 1991, pp. 233–242, doi: 10.1007/BFb0023737.

[5] Couvreur, J.-M.—Encrenaz, E.—Paviot-Adet, E.—Poitrenaud, D.—Wacrenier, P.-A.: Data Decision Diagrams for Petri Net Analysis. In: Esparza, J., Lakos, C. (Eds.): Applications and Theory of Petri Nets 2002 (ICATPN 2002). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2360, 2002, pp. 101–120, doi: 10.1007/3-540-48068-4_8.

[6] Gastin, P.—Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (Eds.): Computer Aided Verification (CAV 2001). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2102, 2001, pp. 53–65, doi: 10.1007/3-540-44585-4_6.

[7] Geldenhuys, J.—Valmari, A.: More Efficient On-the-Fly LTL Verification with Tarjan's Algorithm. Theoretical Computer Science, Vol. 345, 2005, No. 1, pp. 60–82, doi: 10.1016/j.tcs.2005.07.004.

[8] Gerth, R.—Peled, D. A.—Vardi, M. Y.—Wolper, P.: Simple On-the-Fly Automatic Verification of Linear Temporal Logic. In: Dembinski, P., Sredniawa, M. (Eds.): Protocol Specification, Testing and Verification XV (PSTV 1995). Springer, Boston, MA, IFIP Advances in Information and Communication Technology, 1996, pp. 3–18, doi: 10.1007/978-0-387-34892-6_1.

[9] Jensen, K.: Coloured Petri Nets and the Invariant-Method. Theoretical Computer Science, Vol. 14, 1981, No. 3, pp. 317–336, doi: 10.1016/0304-3975(81)90049-9.

[10] Jensen, K.—Kristensen, L. M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, Berlin, Heidelberg, 2009, doi: 10.1007/b95112.

[11] Jørgensen, J. B.—Bossen, C.: Requirements Engineering for a Pervasive Health Care System. Proceedings of the 11th IEEE International Conference on Requirements

Engineering (RE 2003), Monterey Bay, CA, USA, September 2003, pp. 55–64, doi: 10.1109/ICRE.2003.1232737.

[12] Kordon, F.—Linard, A.—Paviot-Adet, E.: Optimized Colored Nets Unfolding. In: Najm, E., Pradat-Peyre, J. F., Donzeau-Gouge, V. V. (Eds.): Formal Techniques for Networked and Distributed Systems – FORTE 2006. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4229, 2006, pp. 339–355, doi: 10.1007/11888116_25.

[13] Kozura, V. E.: Unfoldings of Coloured Petri Nets. In: Bjørner, D., Broy, M., Zamulin, A. V. (Eds.): Perspectives of System Informatics (PSI 2001). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2244, 2001, pp. 268–278, doi: 10.1007/3-540-45575-2_27.

[14] Kristensen, L. M.—Jensen, K.: Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networks. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (Eds.): Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 3147, 2004, pp. 248–269, doi: 10.1007/978-3-540-27863-4_15.

[15] Kristensen, L. M.—Jørgensen, J. B.—Jensen, K.: Application of Coloured Petri Nets in System Development. In: Desel, J., Reisig, W., Rozenberg, G. (Eds.): Lectures on Concurrency and Petri Nets (ACPN 2003). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 3098, 2004, pp. 626–685, doi: 10.1007/978-3-540-27755-2_18.

[16] McMillan, K. L.: Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In: von Bochmann, G., Probst, D. K. (Eds.): Computer Aided Verification (CAV 1992). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 663, 1993, pp. 164–177, doi: 10.1007/3-540-56496-9_14.

[17] Schmidt, K.: Integrating Low Level Symmetries into Reachability Analysis. In: Graf, S., Schwartzbach, M. I. (Eds.): Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1785, 2000, pp. 315–330, doi: 10.1007/3-540-46419-0_22.

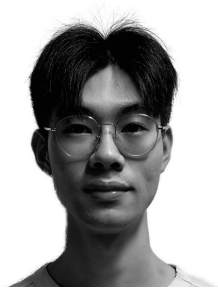
[18] Schmidt, K.: Using Petri Net Invariants in State Space Construction. In: Garavel, H., Hatcliff, J. (Eds.): Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2619, 2003, pp. 473–488, doi: 10.1007/3-540-36577-x_35.

[19] Tian, C.—Song, J.—Duan, Z.—Duan, Z.: LtlNfBa: Making LTL Translation More Practical. In: Liu, S., Duan, Z. (Eds.): Structured Object-Oriented Formal Language and Method (SOFL + MSVL 2015). Springer, Cham, Lecture Notes in Computer Science, Vol. 9559, 2016, pp. 179–194, doi: 10.1007/978-3-319-31220-0_13.

[20] Wang, Z.—Luan, W.—Du, Y.—Qi, L.: Composition and Application of Extended Colored Logic Petri Nets to E-Commerce Systems. IEEE Access, Vol. 8, 2020, pp. 36386–36397, doi: 10.1109/access.2020.2974883.

[21] Wolf, K.: How Petri Net Theory Serves Petri Net Model Checking: A Survey. In: Koutny, M., Pomello, L., Kristensen, L. (Eds.): Transactions on Petri Nets and

Other Models of Concurrency XIV. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 11790, 2019, pp. 36–63, doi: 10.1007/978-3-662-60651-3_2.



Cong He received his B.Sc. degree in computing science and technology from the Tongji University, Shanghai, China, in 2019. He is currently pursuing his M.Sc. degree in the Department of Computer Science and Technology, Tongji University, Shanghai, China. His current research interests include Petri nets and model checking.

Zhijun Ding received his M.Sc. degree from the Shandong University of Science and Technology, Tai'an, China, in 2001, and his Ph.D. degree from the Tongji University, Shanghai, China, in 2007. He is currently Professor with the Department of Computer Science and Technology, Tongji University. He has published over 100 papers in domestic and international academic journals and conference proceedings. His research interests are in formal engineering, Petri nets, services computing, and mobile internet.