Computing and Informatics, Vol. 42, 2023, 781–804, doi: 10.31577/cai_2023_4_781

PARAMETERIZED REACHABILITY GRAPH FOR SOFTWARE MODEL CHECKING BASED ON PDNET

Xiangyu Jia, Shuo Li*

Department of Computer Science and Technology
Tongji University
Shanghai 201804, China
e-mail: jiaxy1999@163.com, lishuo2017@tongji.edu.cn

Abstract. Model checking is a software automation verification technique. However, the complex execution process of concurrent software systems and the exhaustive search of state space make the model-checking technique limited by the state-explosion problem in real applications. Due to the uncertain input information (called system parameterization) in concurrent software systems, the state-explosion problem in model checking is exacerbated. To address the problem that reachability graphs of Petri net are difficult to construct and cannot be explored exhaustively due to system parameterization, this paper introduces parameterized variables into the program dependence net (a concurrent program model). Then, it proposes a parameterized reachability graph generation algorithm, including decision algorithms for verifying the properties. We implement LTL-$_\mathcal{X}$ verification based on parameterized reachability graphs and solve the problem of difficulty constructing reachability graphs caused by uncertain inputs.

Keywords: Model checking, PDNet, parameterized reachability graph

Mathematics Subject Classification 2010: 68-Q60

* Corresponding author

1 INTRODUCTION

With the rapid development of information technology, software systems have become increasingly large and complex, and the number of defects in software systems has increased dramatically. It has become challenging to verify software programs solely by manual inspection [1], and it is urgent to develop automated verification methods to solve this problem to help programmers quickly discover defects in software systems [2, 3, 4, 5]. Formal verification methods have received increasing attention in existing research.

Formal verification techniques include two main approaches, i.e., theorem proving and model checking [6, 7]. Theorem proving can represent the system and properties to be verified as logical formulas in a suitable logical system and then use a theorem prover to prove whether the properties are satisfied in the system [8, 9]. The advantage of theorem proving is that it can be applied to most systems, including infinite-state systems. Its disadvantage is that it is not highly automatic and requires much manual intervention while proving. However, theorem proving does not provide relevant diagnostic information if the formula is falsified. Model checking is one of the most promising automatic verification methods for concurrent software systems [10, 11], and it is an algorithmic approach to verify whether a given model satisfies a particular property expressed by a temporal logic formula using a state space search [7, 12]. For finite state systems, this problem is decidable, i.e., it can be determined automatically in finite time by using a computer program [13, 14, 15, 16]. It verifies the specification through an exhaustive state space enumeration, aiming to achieve higher reliability, correctness, and satisfiability. The advantage of model-checking techniques is that they are highly automated and do not require extensive logic knowledge. When the system does not satisfy a certain property, the model-checking tool returns a counterexample. The interpretation of the counterexample gives the reason why the property does not hold and provides important clues for the correction. There have been many powerful model checkers, such as SPIN [17] and NuSMV [18]. In addition, many reduction techniques have been developed to alleviate the state-explosion problem, such as symbolic model checking, partial order reduction, and symmetry reduction [19, 20].

Petri nets are an important formal model, and they are powerful in describing the internal execution and external interactions of concurrent systems. In contrast to other formal models such as automaton and communication sequential process (CSP), Petri nets can represent true-concurrency rather than interleaving semantics, and they can provide a compact and comprehensive description of control, synchronization, and data operations [21, 22, 23, 24, 25, 26, 27, 28, 29]. However, since the exponential growth of the state space with the increase of the actual software system size, in many cases, the reachability graph analysis method is not feasibly caused by the calculation complexity. On the other hand, since the reachability graph is calculated based on the initial marking, if the initial input parameters are uncertain, a completely different reachability graph may have to be calculated for each assignment of the input parameters. The uncertainty of the input may not generate a reachability graph, resulting in the inability to analyze the properties.

To solve the challenge caused by the uncertain input, this paper proposes a parameterized reachability graph for software model checking based on Program Dependence Net (PDNet) [30]. The main contributions of this paper are as follows:

1. Parameterized reachability graph based on PDNet is proposed by introducing the definition of parameterized variables in PDNet. We define the corresponding occurrence rules and make it possible to generate parameterized reachability graphs even for PDNet with uncertain inputs.
2. The generation algorithm for the parameterized reachability graphs is proposed. It classifies markings using parameterized marking and then uses these parameterized reachability graphs to perform a determination for model checking.
3. We implemented the parameterized reachability graph generation algorithm on DAMER, a concurrent program model checking tool based on PDNet, to enhance the ability of DAMER to handle uncertain input parameters.

Section 2 presents some related works. Section 3 introduces the definition of PDNet based on multisets and Color Petri Net (CPN). Section 4 proposes the definition with parameterized variables, including the corresponding algorithm for generating parameterized reachability graphs. Section 5 verifies the effectiveness of our algorithms through comparative experiments. Section 6 concludes the paper and gives some following works.

2 RELATED WORKS

Model checking is a technique used to automatically verify the correct behavioral properties of a computer system. The basic approach is to use a state transition graph to represent the model of the system under test and to describe the correct behavioral properties of the computer system using computation tree logic (CTL), and linear temporal logic (LTL). Correct behavioral properties of the computer system. The main bottleneck of model checking in practical applications is the state explosion problem. In 1987, McMillan adopted a symbolic approach to representing a state transition graph that allowed him to check large-scale systems [31]. This method is based on Bryant's ordered bifurcation decision diagram (OBDD) [32], which is more concise than the conjunction or disjunction normal form. His team also developed an efficient OBDD algorithm and a symbolic model checking system SMV [33]. Symbolic methods are suitable for hardware system verification with strong structured features and have achieved many successful cases. Still, software systems are less structured than hardware, and concurrent software is asynchronous, so software system verification poses some problems for model checking. Partial order reduction has made great progress in suppressing the state space explosion of software systems [34, 35, 36], and the technique is based on the independence between concurrent events to approximate the state space of a model by reducing the number of interleaved sequences. The partial order reduction technique treats all independent intertwined executions on the transition relations between states as a set. It selects its subsets to reduce its state space, with significant strategies such as Peled's ample sets [35], Valmari's stubborn sets [36], Godefroid's solid and sleeping sets [37], etc. Although symbolic methods and partial order reduction techniques greatly increase the size of verifiable systems, many practical applications are too large to handle the problem size caused by uncertain inputs; therefore, it becomes important to find new techniques to enhance verification in combination with symbolic methods. Petri nets not only have a rich theoretical foundation but also have graphical features, which are more intuitive and easier to understand than algebraic descriptions in textual form. Reachability graphs are the main analysis method for Petri Net models. Because the classical reachability graph cannot handle the model of the checked system that contains parameters or uncertain inputs, it makes the model properties of the checked system becomes very difficult. Usually, parameterized reachability graph (PRG) and symbolic reachability graph (SRG) is used to solve this problem.

The core idea of PRG is to simplify the reachability graph using state classification, and the representation of the state is parameterized. The state classification in the parameterized approach will depend on whether certain specific conditions hold. The literature [38] proposes a method for constructing parameterized reachability graphs based on Petri nets, which defines two kinds of partial order relations for parameterized state marking: $\supseteq$ and $>$. It parametrizes the marking can represent all reachable marking of the verified system and defines the execution of all instantiation procedures; [39] defines a transition implementation rule for PRG, which first calculates the parametrized marking of each place in the reachability graph based on the incoming arcs and outgoing arcs of that place, and splits the marking if the parametrized marking cannot represent the same transition; If the parameterized marking is larger than one of its ancestors, infinite branching should be avoided. The relevant properties of the system are verified based on the enabled and occurrence rules.

Since this approach uses integers to represent the minimum number of tokens in a place, it results in its inability to fully express the information in a parallel program when faced with a parallel program. It requires the definition of new symbolic tokens for description.

The main idea of SRG [40] is to use the inherent symmetry of the system to obtain a compressed representation of the reachable states, which is also a simplified representation of the Well-Formed Colored Petri Net (WN) reachability graph. The SRG simplifies the state representation based on the symmetry of WN by introducing a color function syntax definition to reduce the state space. The SRG is constructed directly by using symbolic marking to represent the equivalence classes in the WN state space, and by introducing the canonical representation of symbolic marking and the enabled and occurrence rules, the SRG is constructed by the same algorithm as the regular reachability graph, except that the SRG uses canonical symbolic marking instead of initial marking and the ordinary enabled and occurrence rules.

Based on SRG and WN theory, [41] defines Stochastic Well-Formed Colored Nets (SWN), which introduce syntactic restriction rules in SWN to reduce the complexity of Markov performance evaluation using SRG. SWN allows to represent of any color function in a structured form so that any unrestricted high-level semantic net can be transformed into a canonical net; [42] defines Extend Symbolic Reachability Graph (ESRG), which simplifies the state space of the checked system by exploiting the partial symmetry in the WN net, and the model analysis and simulation algorithms automatically exploit the model symmetry to improve their efficiency.

It is worth pointing out that the reduction of the SRG approach for reachability graphs strongly relies on the symmetry of the model itself. the more equivalent behaviors between model objects, the more symbolic marking in the same equivalence class, and thus the higher the state compression rate of the original state reachability graph. SRG does not provide significant gains when asymmetric actions appear in the behavior description.

The above methods alleviate the problem of difficulty in constructing the reachability graphs of the Petri net model due to the system parameterization, which leads to the inability of space state exploration, and thus has some limitations in model checking. Based on the analysis of existing reachability graph methods, we propose a new reachability graph construction method using parameterized marking to solve the problem of difficult generation of reachability graph for Petri net caused by uncertain input.

3 PDNET WITH PARAMETERIZED VARIABLES

3.1 Introduction of PDNet

PDNet is our previously proposed model based on CPN, which combines the control-flow structure and dependencies. To define PDNet, we first introduce the definition of multiset and CPN.

Definition 1 (Multiset)**.** Let S be a non-empty set. A multiset $ms : S \to N$ on S is a function that maps each element to a non-negative integer. S_{MS} is the set of all multisets over S. We use $+$ and $-$ for the sum and difference of two multisets. $=$, $>$, $<$, $\geq$, $\leq$ are comparisons of multisets, which are defined in the standard way.

Also, we give some symbolic terms for the following definitions: $BOOL = \{false, true\}$ is the set of Boolean predicates with standard logical operations; $EXPR$ is the set of expressions; $Type[e]$ is the type of an expression $e \in EXPR$, i.e., the type of the value obtained when evaluating e; $Var(e)$ is the set of all variables in an expression e; $EXPR_V$ for a variable set V is the set of expressions $e \in EXPR$ such that $Var(e) \in V$. $Type[v]$ is the type of a variable v.

Definition 2 (Colored Petri Nets)**.** CPN is defined by a 9-tuple,

$$N ::= (\Sigma, V, P, T, F, C, G, E, I),$$

where:

1. Σ is a finite non-empty set of types called color sets;
2. V is a finite set with type variables, $\forall v \in V$, there is $Type[v] \in \Sigma$;
3. P is a finite set of places;
4. T is a finite set of transitions and $T \cap P = \emptyset$;
5. $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of directed arcs;
6. $C : P \to \Sigma$ is a color set function that assigns the color set $C(p)$ belonging to the type set Σ to each place p;
7. $G : T \to EXPR_V$ is a guard function, that assigns an expression $G(t)$ to each transition t, $\forall t \in T : (Type[G(t)] \in BOOL) \wedge (Type[Var(G(t))] \subseteq \Sigma)$;
8. $E : F \to EXPR_V$ is a function, that assigns an arc expression $E(f)$ to each arc f, $\forall f \in F : (Type[E(f)] = C(p(f))_{MS}) \wedge (Type[Var(E(f))] \subseteq \Sigma)$, where $p(f)$ is the place connected arc f;
9. $I : P \to EXPR_\emptyset$ is an initialization function, that assigns an initialization expression $I(p)$ to each place p, $\forall p \in P : (Type[I(p)] = C(p)_{MS}) \wedge (Var(I(p)) = \emptyset)$.

   PDNet is also a 9-tuple, which differs from CPN in P and F.

1. P is divided into three subsets, i.e., $P = P_c \cup P_v \cup P_f$. Concretely, P_c is a subset of control places, P_v is a subset of variable places, and P_f is the subset of execution places.
2. F is divided into three subsets, i.e., $F = F_c \cup F_{rw} \cup F_f$, Concretely, F_c is a subset of control arcs, F_{rw} is a subset of read-write arcs, and F_f is a subset of execution arcs.

Except for the two differences, the other definitions and constraints of PDNet are consistent with CPN, and in the following definitions, we give some basic concepts of PDNet.

Definition 3 (Basic concepts in PDNet)**.**

1. $M : P \to EXPR_\emptyset$ is a marking function that assigns an expression $M(p)$ to each place p, $\forall p \in P : Type[M(p)] = C(p)_{MS} \wedge (Var(M(p)) = \emptyset)$; for convenience, the marking of N is denoted by M or M with subscript, and in particular, M_0 represents the initial marking $\forall p \in P : M_0(p) = I(p)$;
2. $Var(t) \subseteq V$ is the variable set of a transition t. It consists of the variables appearing in the expression $G(t)$ and arc expressions of all arcs connected to the transition t;
3. $B : V \to CON$ is a binding function that assigns a constant value $B(v)$ to each variable v. $B[t]$ is the set of all bindings of a transition t, that maps $V \in Var(t)$ to a constant value, and $b \in B[t]$ is a binding of a transition t;
4. A binding element (t, b) is a pair where $t \in T$ and $b \in B[t]$, $B[t]$ is a set of all binding elements of a transition t.

3.2 Parameterized Variables

Formally, $e\langle b\rangle$ represents the evaluation result of expression e in binding b by assigning a constant to each variable $v \in Var(e)$ through binding b. Therefore, under the binding element (t, b), the evaluation result of $G(t)$ (or $E(f)$) is represented by $G(t)\langle b\rangle$ (or $E(f)\langle b\rangle$), where f is the arc connected to the transition t. Here, we specifically use v_s to denote parameterized variables and V_s to denote the set of parameterized variables, where $v_s \in V_S, V_S \subseteq V$.

Definition 4 (Parameterized variables for PDNet)**.**

1. e_s: assuming that the assignment operator to the parameterized variable v_s is $v_s := \omega$, e_s is an expression obtained by computing ω based on the current execution state and is any expression involving a unitary or binary operator with symbols and specific values;
2. $EXPR_s$: any finite set of expressions involving variables $v \in V$ and constants $o \in CON$ for unitary or binary operators, $e_s \in EXPR_s$;
3. σ: denotes the symbolic state, a mapping from a variable to a symbolic expression e_s, denoted $\sigma : v_s \mapsto e_s$, i.e. $\sigma(v_s) = e_s$;
4. $SS : V_s \mapsto EXPR_s$, the set of symbolic storage functions $\sigma \in SS$.

In particular, since the parameterized variables do not have a definite value, making it difficult to determine the relationship between their value intervals and the constraints, we also need to define the function $SAT()$, whose input is a string of first-order formulas without quantifiers, which uses the constraint solver [43] to solve for the existence of a solution to the input quantifier-free first-order formulas, with the output being $true$ or $false$; if a solution exists for $PC \wedge \sigma(G(t))$, then it is written as $SAT(PC \wedge \sigma(G(t))) = true$.

In the existing PDNet, P is divided into three subsets, $P = P_c \cup P_v \cup P_f$, where P_c is a subset of the control place, P_v is a subset of the variable place, and P_f is a subset of the execution place. We refer to the structure of the original variable place P_v to add a new class of parameterized variable place, denoted as P_s. That is, P is divided into four subsets $P = P_c \cup P_v \cup P_f \cup P_s$, where P_s is defined.

Definition 5 (Parameterized variable place in PDNet)**.** The parameterized variable place P_s is used to store the unassigned variables v_s. The parameterized variable place consists of a triple $\langle\sigma, PC, id\rangle$, where:

1. σ is a symbolic state representing the mapping from variables to parameterized expressions e_s;
2. PC is a quantifier-free first-order formula consisting of the expressions in $G(t)$ on the path and the truth-value selection of the expressions connected to describe the path constraints;
3. id is a unique marking of the P_s place.

where the initial value of the symbolic state σ is *null* and the initial value of the path constraint PC is *true*.

Definition 6 (Parameterized marking and parameterized binding). Parameterized marking $M_s\colon P \to EXPR_\emptyset$ is a parameterized marking function that specifies an expression $M_s(p)$ for each variable place p:

$$\forall p \in P : Type[M_s(p)] = C(p)_{MS} \wedge (Var(M_s(p)) = \emptyset).$$

For simplicity of representation, the parameterized marking of N is represented by M_s or M_s with subscript when $M_s(p)$ is present.

For a PDNet N whose variables are all non-parameterized, the marking function is $M : (P \setminus P_s) \to EXPR_\emptyset$, specifying an expression $M(p_v)$ for each non-parameterized variable banked by p_v:

$$\forall p_v \in (P \setminus P_s : Type[M(p_v)] = C(p_v)_{MS} \wedge (Var(M(p_v)) = \emptyset).$$

For convenience, the marking of N whose variables are all non-parameterized is denoted by M or M with subscripts. At the same time, we cannot determine a fixed binding element (t, b) for the transition t associated with the parameterized variable place by P_s; since the values of the parameterized variables represented by the parameterized variable place by P_s are jointly represented by σ and PC, there does not exist a specific value to take, and we can consider the range of values as a concatenation of one or more intervals; Since it costs more time and space to solve the value interval of each variable using the constraint solver, we do not directly calculate the value interval of the variables, but determine whether the transition can be enabled under the parameterized marking M_s by analyzing the relationship between σ and PC; define the parameterized binding element (t, σ, PC), where $t \in T, \sigma \in SS$; if the symbolic states and path constraints recorded in the parameterized binding element are covered by $M_s(p)$ after analysis, it is written as $E(p, t)\langle\sigma, PC\rangle \leq M_s(p)$.

Definition 7 (Parameterized enabled and occurrence rules). Let N be a PDNet, (t, b) be a binding element on N, M be a marking on N, and the binding element (t, b) is enabled under the marking M, denoted $M[(t, b)\rangle$, when and only when:

1. $G(t)\langle b\rangle = true$;
2. $\forall p \in {}^\bullet t : E(p, t)\langle b\rangle \leq M(p)$;

   When (t, b) is enabled under M, triggering the transition t leads to the generation of a new marking M_1, denoted as $M[(t, b)\rangle M_1$, such that:
3. $\forall p \in P : M_1(p) = M(p) - E(p, t)\langle b\rangle + E(t, p)\langle b\rangle$.

For parameterized variables, when (t, σ, PC) is enabled under M_s, it may lead to the generation of a new marking M_{s1}, denoted as $M_s[(t, \sigma, PC)\rangle M_{s1}$, when and only when:

1. $SAT(PC \wedge \sigma(G(t))) = true$;
2. $\forall p \in^{\bullet} t : E(p,t)\langle\sigma, PC\rangle \leq M_s(p)$;
3. $\forall p \in P : M_{s1}(p) = M_s(p) - E(p,t)\langle\sigma, PC\rangle + E(t,p)\langle\sigma, PC\rangle$.

The intuition of this rule is to update the path constraint and symbolic state stored in each token, $PC = PC \wedge \sigma(G(t))$, and not to update if $G(t)$ does not contain symbolic variables, see Algorithm 1 for the specific update algorithm. In particular, the two operation cases that we may encounter in the process of updating the symbolic state σ information in Algorithm 1 to define the variable v_s are the input operation and the assignment operation, where the input operation is an external input to the parameterized variable v_s in the form $v_s := sym\_input()$ and the assignment operation is an assignment of a value or expression to the parameterized variable v_s in the form $v_s := \omega$. The symbolic states and path constraints in the parameterized variable place are updated continuously as the parameterized binding elements are enabled and occur.

Algorithm 1 Parameterized variable place information update

Step 1. Determine whether σ, PC in the parameterized variables v_s satisfy the conditions in $G(t)$: $SAT(PC \wedge \sigma(G(t))) = true$;

Step 2. Update the value stored in the path constraint PC.
If $SAT(PC \wedge \neg\sigma(G(t))) = false$ or $G(t)$ does not contain constraints associated with the parameterized variables v_s **Then**
Not updating the contents stored in the PC;
Else
$PC' = PC \wedge \sigma(G(t))$;

Step 3. Update symbol status σ.
If Performing input operations on variables v_s **Then**
Update the mapping σ in v_s to: $v_s \rightarrow v_{si}$, where the initial value of i is 0 and the value of i takes increasing values with the update of the input mapping;
If Assign a value to the variable v_s in the form $a := \omega$ **Then**
Substitute the existing mapping σ in v_s into the formula ω to calculate the new mapping expression, and update σ with the new mapping expression.

The following example shows the update process of symbolic state σ and path constraint PC in the parameterized place, as detailed in Figure 1.

Definition 8 (Occurrence sequence of PDNet)**.** Let N be a PDNet, M_0 be the initial marking of N, and (t,b) be the binding elements of N. The sequence of occurrences in N can be defined by induction:

1. $M_0[\varepsilon\rangle M_0$,($\varepsilon$ is a null sequence);
2. $M_0[\omega\rangle M_1 \wedge M_1[(t,b)\rangle M_2 : M_0[\omega(t,b)\rangle M_2$.

The sequence of occurrences ω in N is maximal when and only when:

1. ω is infinite, e.g., (t_1, b_1), (t_2, b_2), $\ldots$ or
2. $M_0[\omega\rangle M_1 \wedge \forall t \in T, \nexists (t,b) \in BE(t) : M_1[(t,b)\rangle$.

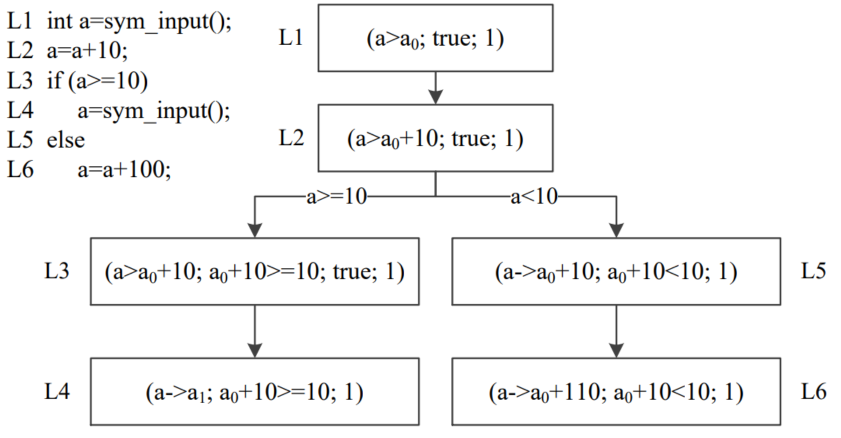


Figure 1. The update process of the parameterized variable place

4 MODEL CHECKING PDNET WITH PARAMETERIZED VARIABLES

4.1 Propositions and LTL of PDNet with Parameterized Variables

LTL describes linear temporal properties. Our approach can support the LTL-$\mathcal{X}$ formulae, so we formalize the following particular definition of propositions in PDNet with parameterized variables.

Definition 9 (Proposition of PDNet with LTL-$\mathcal{X}$ formula representation)**.** Let N be a PDNet containing parameterized variables, po a proposition, Po the set of propositions, and ψ an LTL-$\mathcal{X}$ formula, the syntax of a proposition containing parameterized variables can be defined:

$$po ::= \textit{is-fireable}(t)(t \in T) | \textit{token-value}(p_s) rop c (p_s \in P_s, c \in C(p)_{MS},$$
$$rop \in \{<, \leq, >, \geq\}).$$

Under a parameterized marking M_s, proposition semantics is defined:

$$\textit{is-fireable}(t) = \begin{cases} \textit{true}, & \text{if } \exists b : M_s[(t,b)\rangle, \\ \textit{false}, & \textit{otherwise}, \end{cases}$$

$$\textit{token-value}(p_s)\ \textit{rop}\ c = \begin{cases} \textit{true}, & \textit{if } M(p_s)\ \textit{rop}\ c\ \textit{holds}, \\ \textit{false}, & \textit{otherwise.} \end{cases}$$

LTL-$\mathcal{X}$ syntax on Po: $\psi ::= Po|\neg\psi|\psi_1 \wedge \psi_2|\psi_1 \vee \psi_2|\psi_1 \Rightarrow \psi_2|\mathcal{F}\psi|\mathcal{G}\psi|\psi_1\mathcal{U}\psi_2$ ($\neg, \wedge, \vee$ and $\Rightarrow$ are usual propositional, $\mathcal{F}$, $\mathcal{G}$, $\mathcal{U}$ are temporal operators.

For example, $\mathcal{G}$ *is-fireable*$(t) \Rightarrow \mathcal{F}$ *token-value*$(p) = 0$ implies that the number of tokens of p will be equal to 0 in some subsequent states regardless of when the transition is enabled.

4.2 Parameterized Reachability Graph for PDNet

The parameterized approach is attractive in solving the problem of parameterized variables in model checking. To enhance the expressive and analytical capabilities of PDNet, we propose a parameterized reachability graph with the following formal definitions of parameterized reachable marking and parameterized reachable marking set.

Definition 10 (Parameterized reachable marking). Let $N = (\Sigma, V, P, T, F, C, G, E, I)$ be a PDNet with parameterized variables if there exists a sequence of change occurrences σ_s such that the initial parameterized marking M_{s0} can get a new parameterized marking M_s after the occurrence of σ_s, then the parameterized marking M_s is said to be reachable from the initial parameterized marking M_{s0}, i.e., $M_{s0} \xrightarrow{\sigma_s} M_s$.

Definition 11 (Parameterized reachable marking set). The parameterized reachable marking set $R(M_{s0})$ of a PDNet system $N = (\Sigma, V, P, T, F, C, G, E, I)$ containing parameterized variables is a minimal set of marking satisfying the following conditions: $M(_{s0}) \in R(M_{s0})$; if $M_s \in R(M_{s0})$ and there exists $t \in T$, such that $M_s \xrightarrow{t} M'_s$, then there is $M'_s \in R(M_{s0})$.

Definition 12 (Parameterized reachability graph). Let N be a PDNet with parameterized variables. The parameterized reachability graph of N is a directed graph $PRG(N) = (V, E)$, where the set of nodes of the directed graph $V = R(M_{s0})$, defining ES as the execution sequence $\langle t, \sigma, PC\rangle$, and the set of edges of the directed graph $E = \{\langle M_s, t, M'_s\rangle \cup ES|M_s, M'_s \in R(M_{s0}) \wedge M_s \xrightarrow{t} M'_s\}$; i.e., a directed graph is a graph composed of nodes identified with arcs labeled by elements in the set of variables of N.

For the parameterized reachability graph in PDNet, the process of determining whether the parameterized reachable marking is old, updating the information stored in the parameterized reachable marking, and updating the path constraints are all different from the traditional methods of constructing reachability graphs because the parameterized reachable marking is defined. Determining whether the parameterized reachable marking is old or not by Algorithm 2. And the selection of upper bound k will be a difficult problem. Here, we use the cyclic dependency judgment algorithm [44, 45, 46] to give the upper bound k. The selection of upper bound k will significantly affect the processing efficiency of this algorithm in programs containing unbounded loops, loops, and boundary conditions of simple nested loops, which can alleviate the path explosion problem in loops to some extent. However, this loop-dependent judgment algorithm also has certain limitations: it cannot handle nonlinear loops and complex nested loops that contain dynamic boundary loops, branching conditions inside the loop, etc. The optimization of the algorithm for calculating the upper bound k will also be an important research direction for this topic in the future. The construction algorithm of the parameterized reachability graph *PRG* is proposed in Algorithm 2.

Algorithm 2 Construct $PRG(N)$

Use M_0 as the root node of $PRG(N)$ and label it as "*new*", with path constraint $PC = true$;

Step 1. **While** the Existence of nodes marked as "*new*" **Do**
    Choose any node labeled "*new*" and set it to M;

Step 2. **If** There is a node on the directed path from M_0 to M whose marking is equal to M, For parameterized reachable marking M, reach a maximum upper bound k or terminate when identical parameterized marking exists **Then**
    Change the label of M to "*old*" and return to **Step 1**;

Step 3. **If** $\forall t \in T : \neg M[t\rangle$ **Then**
    Change the label of M to "*endpoint*" and return to **Step 1**;

Step 4. **For** identifies each $t \in T$ in M that satisfies $M[\rangle$ **Do**
    If $L_v(t) \neq \emptyset$ **Then**;
        $PC = PC \cap L_v(t)$;
        4.1 According to Algorithm 1, calculate M' in $M[t\rangle M'$;
        4.2 Introduce a "*new*" node in $PRG(N)$, draw a directed
arc
        from M to M', and label this arc with t;

Step 5. Erase the "*new*" label of node M, reset the path constraint PC to $true$, and return to **Step 1**;

4.3 Product Automaton for Parameterized Reachability Graph

For the parameterized reachability graph *PRG*, in the process of synthesizing the product automata, since the parameterized reachability graph nodes contain parameterized propositional states, it is not possible to solve directly whether they can be synthesized as in the traditional product automata judgment algorithm, so here the $SAT()$ function is used to determine whether there is a feasible solution to make the parameterized propositional states synthesizable, and the constraints contained in the propositions are also added to the parameterized The constraints contained in the proposition are also added to the path constraints of the parameterized propositional state. The product automata synthesis judgment algorithm for parameterized graphs is shown in Algorithm 3, where $Label(v)$ denotes the propositional state in the parameterized reachability graph node and $L(s)$ is the set of propositions on state s in the labeled Büchi automata:

Algorithm 3 The product automaton generation algorithm for parameterized reachability graph

1: **For** Each proposition $l(s)$ in $L(s)$ **Do**
2: **For** $Label(v)$ for each node v in the set of nodes **Do**
3: **If** $l(s)$ contains the parameterized variable a **Then**
4: Find σ and PC stored in the parameterized variable a in $Label(v)$;
5: **If** $SAT(a.\sigma \wedge a.PC \wedge l(s) \neq false$ **Then**
6: $a.PC = a.PC \wedge l(s)$;
7: Synthetic product-state;
8: **Else** Non-synthetic;
9: **If** Proposition $l(s)$ does not contain parameterized variables **Then**
10: **If** $lable(v) \wedge l(s) \neq false$ **Then**
11: Synthetic cross-state;

To show more concretely the differences between Algorithm 2, Algorithm 3, and the traditional algorithms, we give an example of an LTL verification problem with parameterized variables in Section 4.4, which shows in detail the example graphs of the parameterized reachability graphs constructed in that case with a product automaton.

4.4 Verification Problems Based on PDNet with Parameterized Variables

Traditionally, the automata-theoretic approach for explicit model checking exhaustively explores all possible executions of the state space. The model-checking problem of LTL-$_{\mathcal{X}}$ is converted into an emptiness-checking problem [30] with the following steps:

Step 1. First model the system with parameterized variables using PDNet and construct the parameterized reachability graph $PRG(N)$ with parameterized variables;

Step 2. Describe the characteristics of the system subject to model checking using the linear temporal logic formula φ;

Step 3. Constructing Büchi automata that recognize linear temporal logic formulas φ that contain all sequences of states that violate the semantics of p;

Step 4. Constructing the parameterized reachability graph $PRG(N)$ and the product automata SP describing the Büchi automata of $\neg\varphi$, which accepts all infinite sequences of the system that are also acceptable to both the parameterized reachability graph and the Büchi automata;

Step 5. Testing whether the product automaton SP is empty, i.e., testing whether it does not accept any sequence. If SP is empty, it is proved that all runs of the system satisfy the specification p; otherwise, the system does not satisfy the specification p. Among them, Steps 4 and 5 can be handled dynamically, i.e., checking the emptiness while yielding the product automaton.

PDNet can apply an automata-theoretic approach [30], for which the marking of PDNet with parameterized variables can be generated from the initial parameterized marking and the initial state of the Büchi automaton. The acceptable paths from the initial product are extended until a product state is reached (e.g., a combined state with Büchi states). To yield the product automaton, the judgment of product automaton needs to be performed especially using Algorithm 3. Finally, all paths constitute the language accepted by the product automaton.

This example focuses on the LTL verification problems for a program containing parameterized variables. In the example program in Figure 2 a), the error location is at line 6. *ERROR*() is an error location for safety property. Figure 2 b) represents the path branch of the example program. Here, the values of x and y are input variables by the user from the outside, and the value of z is taken concerning y. Therefore, the three variables x, y, and z are parameterized variables. The execution path of the program is shown in Figure 2 b), which is divided into three main branches, among which, if the path conditions of $x == z$ and $x > y + 10$ are satisfied at the same time, it will reach *ERROR*(). In contrast, the other two branch paths are correctly executed.

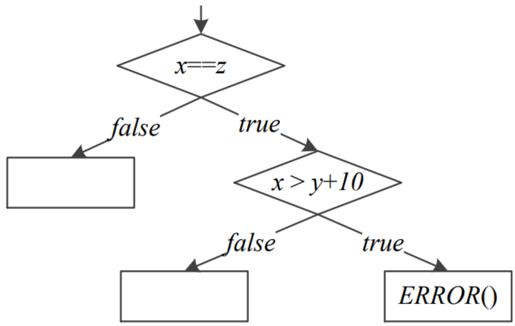
The PDNet of the example program is shown in Figure 3, with all labels on the arcs omitted for simplicity. Each transition can simulate the execution of a statement by its occurrence, and the corresponding transition occurrence can manipulate the variables represented by the place.

The state space of this PDNet is the reachability graph in Figure 4. The labeled nodes are represented by rectangles with the name of the place, and the names of the arrows on the state-labeled reachability graph correspond to the names of the transition corresponding to the occurrence of transition in the PDNet. The labels on all arcs are also omitted here for simplicity. In addition, since LTL-$_\mathcal{X}$ model checking is based on infinite paths, arcs pointing to themselves are added as dashed arrows for M_3, M_5, and M_7 in Figure 4.

The LTL-$_\mathcal{X}$ formula $\mathcal{G}\neg error()$ to specify the safety properties of the example program, $\mathcal{G}\neg error()$ is first converted to $is - fireable(t_3)$ in Figure 5. The node marked as $is - fireable(t_3)$ can only synchronize with the reachable marking enabled by the enabled transition t_3. The final product automaton is shown in Figure 6, and

```
......
1: x=get_input();
2: y=get_input();
3: z=2y;
4: if (x==z){
5:     if (x>y+10){
6:         ERROR();
7:     }
8: }
......
```

a) Example program in C language



b) Path branch of the example program

Figure 2. Example program with parameterized variables

it can be concluded that the example program violates the security property. The occurrence sequence t_b, t'_1, t'_2, t_3 is a counterexample path in this example.

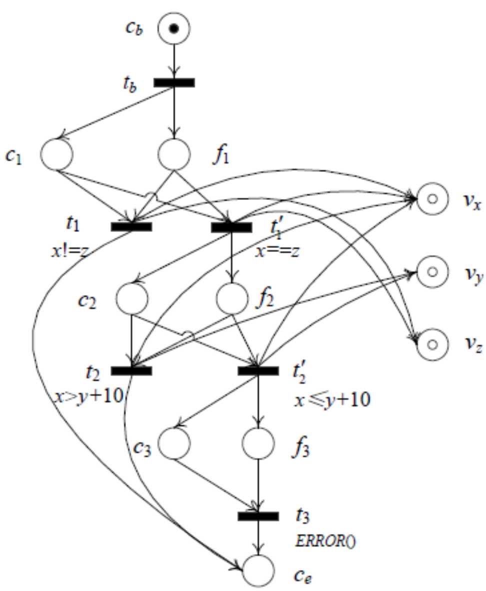


Figure 3. PDNet for the example program

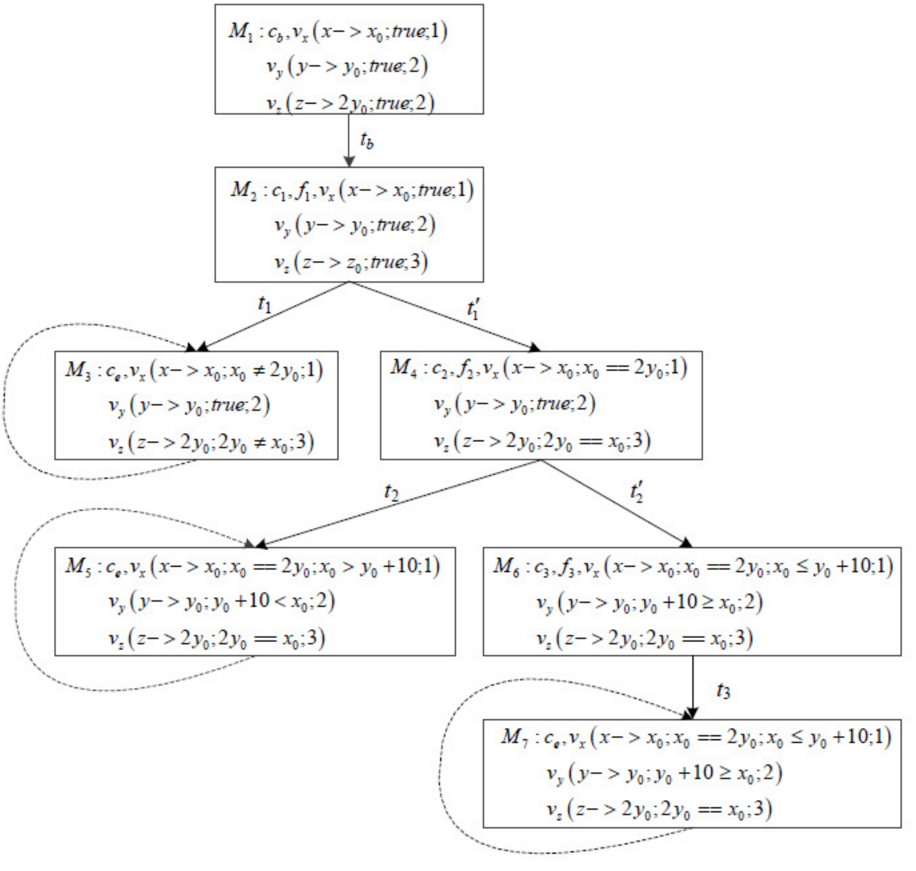


Figure 4. Parameterized state reachability graph

5 EXPERIMENTAL VERIFICATION

5.1 Experimental Benchmarks

To verify the validity of the definitions and algorithms in this paper, we construct eight typical benchmarks to evaluate the analysis capability of the system. The source code of these benchmarks includes multiple branching condition judgments on parameterized variables, repeated input judgments on parameterized variables, simple and complex computation judgments on parameterized variables, loops related to the values of parameterized variables, etc. The basic conditions are shown in Table 1 for this experiment. The benchmark is mainly judged by two aspects the tool running time and output results. In Table 1, Lines, Variables, Branches, Loops, Transitions, and Places denote the number of lines of code, variables, branches, loops, transitions, and places, respectively.

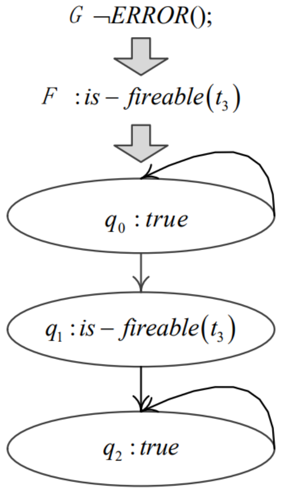


Figure 5. Büchi automaton

No.	Test program	Lines	Variables	Branches	Loops	Transitions	Places
1	Sym_Basictype	17	1	1	0	20	37
2	Sym_Branch	22	3	2	0	25	47
3	Sym_Year	21	1	1	0	23	43
4	Sym_Sum	21	2	1	0	23	44
5	Sym_Reinput	19	2	1	0	22	42
6	Sym_Loop_1	22	1	–	–	26	48
7	Sym_Loop_2	24	1	80	80	29	55
8	Sym_Loop_3	23	1	200	200	29	55

Table 1. Parameters of test program

5.2 Experimental Comparison

For each benchmark given in Table 1, the average value is taken as the experimental result after 10 runs of each benchmark algorithm because of the relatively small variation in time consumption between different runs of the same algorithm during the test. The experimental results are shown in Table 2.

Among them, the three methods used to perform comparative testing are the methods that outputs a series of test cases using the symbolic execution tool CREST and brings the benchmarks into DAMER separately for model checking, which is denoted as SymbolicExec in Table 2, the symbolic reachability graph SRG (Symbolic Reachability Graph) based model checking tool GreatSPN [47], and model checking tool CPN-AMI [48] based on Parameterized Reachability Graph PRG (Parameterized Reachability Graph).

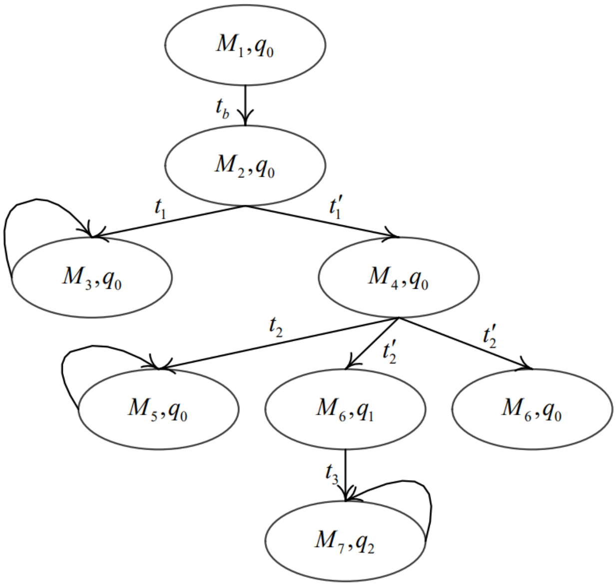


Figure 6. Product automation

t and V in the following table denote time and output results, respectively. Concretely, T and F in Table 2 denote the output result $True$ and $False$, respectively.

Test case	SymbolicExec		GreatSPN		CPN-AMI		Our method		Truth
	t	V	t	V	t	V	t	V	
Sym_Basictype	127.321	F	86.352	F	57.020	F	20.496	F	F
Sym_Branch	168.257	F	101.367	F	61.265	F	24.923	F	F
Sym_Year	126.395	F	88.215	F	58.895	F	20.586	F	F
Sym_Sum	136.257	F	93.012	F	57.958	F	21.505	F	F
Sym_Reinput	118.354	T	84.210	F	56.364	F	19.880	F	F
Sym_Loop_1	–	–	–	–	–	–	–		
Sym_Loop_2	764.258	F	397.352	T	251.035	T	104.084	T	T
Sym_Loop_3	–	–	742.362	T	422.238	F	241.715	F	F

Table 2. Comparison of experimental results

From the test results listed in Table 2, it can be seen that the SymbolicExec method misjudged or failed to judge in four test cases, including Sym_Reinput, Sym_Loop_1, Sym_Loop_2, and Sym_Loop_3; the GreatSPN method misjudged or failed to judge in two test cases, including Sym_Loop_2 and Sym_Loop_3; The CPN-AMI method does not have any misjudgment, but it also fails to judge Sym_Loop_1; the method in this paper makes correct judgments for the test cases and takes the least time, but it also fails to judge Sym_Loop_1, which is mainly caused by the fact that the loop-dependent algorithm used in this experiment fails to judge the symbolic boundary. This is mainly caused by the fact that the loop-dependent algorithm used in this experiment cannot determine the symbolic boundary loop. It can be seen that this paper can detect programs with parameterized variables and output correct test results, which has obvious advantages in terms of test time consumption. Moreover, it can deal with branching conditions, operations, repeated input, and bounded loops of parameterized variables in programs containing parameterized variables.

For Sym_Loop_1, Sym_Loop_2, and Sym_Loop_3, all three test cases have a more serious path explosion problem, mainly caused by the loop structure present in the test cases. In Algorithm 2, the choice of the upper bound k of the loop can greatly affect the processing efficiency of this algorithm in programs containing loops. In this comparison experiment, the loop test cases are divided into the following two types according to the boundary conditions:

1. Symbolic boundary: the boundary condition expression of the loop contains parameterized variables, and the number of executions is indeterminate;
2. Constant boundary: the boundary condition expression of the loop does not contain parameterized variables, and the number of executions is constant.

Although constant-bounded loops do not execute an indeterminate number of times as symbolic-bounded loops, they also generate redundant paths leading to multiple loop expansions. In the test case, a loop dependency is implied between the parameterized variable x and the variable a such that in each loop, there are $\{x_n = x - n\}$, $\{a_n = n\}$, where, n is the number of loops. At present, we have only used a simple cyclic dependency judgment algorithm to give the cyclic upper bound k. The optimization of this algorithm will also be an important research direction for this topic in the future.

6 CONCLUSION AND FUTURE WORK

This paper improves PDNet to support parameterized variables of concurrent programs. To address the problem that it is difficult to construct the reachability graph caused by the system parameterization, we propose a new method for constructing a fully parameterized reachability graph of PDNet. We define parameterized variables on PDNet and improve the corresponding rules. The corresponding parameterized reachability graph generation algorithm is given. A PDNet-based model-checking tool that supports parameterized variables is implemented based on DAMER. The experimental results show the effectiveness of our method.

Due to parameterized variables with path information to avoid problems such as repeated execution, the amount of information in a single node of the generated reachability graph can be large. If the reachability graph is fully generated and combined with Büchi automata, the state-explosion problem is aggravated. Future research will consider using cyclic recursive processing methods to solve this problem.

REFERENCES

[1] Mei, H.—Wang, Q. X.—Zhang, L.—Wang, J.: Software Analysis: A Road Map. Chinese Journal of Computers, Vol. 32, 2009, No. 9, pp. 1697–1710 (in Chinese).

[2] Clarke, L. A.: A System to Generate Test Data and Symbolically Execute Programs. IEEE Transactions on Software Engineering, Vol. SE-2, 1976, No. 3, pp. 215–222, doi: 10.1109/TSE.1976.233817.

[3] Weber, S.—Karger, P. A.—Paradkar, A.: A Software Flaw Taxonomy: Aiming Tools at Security. ACM SIGSOFT Software Engineering Notes, Vol. 30, 2005, No. 4, pp. 1–7, doi: 10.1145/1082983.1083209.

[4] Binkley, D.: Source Code Analysis: A Road Map. Future of Software Engineering (FOSE '07), IEEE, 2007, pp. 104–119, doi: 10.1109/FOSE.2007.27.

[5] Sekar, R.—Bendre, M.—Dhurjati, D.—Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. Proceeding 2001 IEEE Symposium on Security and Privacy (S & P 2001), 2001, pp. 144–155, doi: 10.1109/SECPRI.2001.924295.

[6] Schumann, J. M.: Automated Theorem Proving in Software Engineering. Springer, 2001, doi: 10.1007/978-3-662-22646-9.

[7] Clarke, E. M.—Emerson, E. A.—Sifakis, J.: Model Checking: Algorithmic Verification and Debugging. Communications of the ACM, Vol. 52, 2009, No. 11, pp. 74–84, doi: 10.1145/1592761.1592781.

[8] Boulton, R. J.: Efficiency in a Fully-Expansive Theorem Prover. Technical Report. University of Cambridge, Computer Laboratory, 1994, doi: 10.48456/tr-337.

[9] Rajan, S.—Shankar, N.—Srivas, M. K.: An Integration of Model Checking with Automated Proof Checking. In: Wolper, P. (Ed.): Computer Aided Verification (CAV '95). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 939, 1995, pp. 84–97, doi: 10.1007/3-540-60045-0_42.

[10] Clarke, E. M.: Model Checking. In: Ramesh, S., Sivakumar, G. (Eds.): Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1997). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1346, 1997, pp. 54–56, doi: 10.1007/BFb0058022.

[11] Atlee, J. M.—Gannon, J.: State-Based Model Checking of Event-Driven System Requirements. IEEE Transaction on Software Engineering, Vol. 19, 1993, No. 1, pp. 24–40, doi: 10.1109/32.210305.

[12] Clarke, E. M.—Emerson, E. A.: Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In: Kozen, D. (Ed.): Logic of Programs (Logic of Programs 1981). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 131, 1982, pp. 52–71, doi: 10.1007/BFb0025774.

[13] Jensen, K.—Kristensen, L. M.—Wells, L.: Coloured Petri Nets and CPN Tools for Modeling and Validation of Concurrent Systems. International Journal on

Software Tools for Technology Transfer, Vol. 9, 2007, No. 3-4, pp. 213–254, doi: 10.1007/s10009-007-0038-x.

[14] Yang, R.—Ding, Z.—Guo, T.—Pan, M.—Jiang, C.: Model Checking of Variable Petri Nets by Using the Kripke Structure. IEEE Transactions on Systems, Man, and Cybernetics: Systems, Vol. 52, 2022, No. 12, pp. 7774–7786, doi: 10.1109/TSMC.2022.3163741.

[15] Jensen, K.—Kristensen, L. M.: Colored Petri Nets: A Graphical Language for Formal Modeling and Validation of Concurrent Systems. Communications of the ACM, Vol. 58, 2015, No. 6, pp. 61–70, doi: 10.1145/2663340.

[16] Kheldoun, A.—Barkaoui, K.—Ioualalen, M.: Formal Verification of Complex Business Processes Based on High-Level Petri Nets. Information Sciences, Vol. 385-386, 2017, pp. 39–54, doi: 10.1016/j.ins.2016.12.044.

[17] Holzmann, G. J.: The Model Checker SPIN. IEEE Transactions on Software Engineering, Vol. 23, 1997, No. 5, pp. 279–295, doi: 10.1109/32.588521.

[18] Cimatti, A.—Clarke, E.—Giunchiglia, E.—Giunchiglia, F.—Pistore, M.—Roveri, M.—Sebastiani, R.—Tacchella, A.: NuSMV 2: An Open Source Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K. G. (Eds.): Computer Aided Verification (CAV 2002). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2404, 2002, pp. 359–364, doi: 10.1007/3-540-45657-0_29.

[19] Bolton, M. L.—Bass, E. J.—Siminiceanu, R. I.: Using Formal Verification to Evaluate Human-Automation Interaction: A Review. IEEE Transactions on Systems, Man, and Cybernetics: Systems, Vol. 43, 2013, No. 3, pp. 488–503, doi: 10.1109/TSMCA.2012.2210406.

[20] Bolton, M. L.—Bass, E. J.: Generating Erroneous Human Behavior from Strategic Knowledge in Task Models and Evaluating Its Impact on System Safety with Model Checking. IEEE Transactions on Systems, Man, and Cybernetics: Systems, Vol. 43, 2013, No. 6, pp. 1314–1327, doi: 10.1109/TSMC.2013.2256129.

[21] Katsaros, P.: A Roadmap to Electronic Payment Transaction Guarantees and a Colored Petri Net Model Checking Approach. Information and Software Technology, Vol. 51, 2009, No. 2, pp. 235–257, doi: 10.1016/j.infsof.2008.01.005.

[22] Ding, Z.—Qiu, H.—Yang, R.—Jiang, C.—Zhou, M.: Interactive-Control-Model for Human-Computer Interactive System Based on Petri Nets. IEEE Transactions on Automation Science and Engineering, Vol. 16, 2019, No. 4, pp. 1800–1813, doi: 10.1109/TASE.2019.2895507.

[23] Yin, X.—Lafortune, S.: On the Decidability and Complexity of Diagnosability for Labeled Petri Nets. IEEE Transactions on Automatic Control, Vol. 62, 2017, No. 11, pp. 5931–5938, doi: 10.1109/TAC.2017.2699278.

[24] Yang, R.—Ding, Z.—Pan, M.—Jiang, C.—Zhou, M.: Liveness Analysis of ω-Independent Petri Nets Based on New Modified Reachability Trees. IEEE Transactions on Systems, Man, and Cybernetics: Systems, Vol. 47, 2017, No. 9, pp. 2601–2612, doi: 10.1109/TSMC.2016.2524062.

[25] Ding, Z.—Yang, R.: Modeling and Analysis for Mobile Computing Systems Based on Petri Nets: A Survey. IEEE Access, Vol. 6, 2018, pp. 63038–68056, doi:

10.1109/ACCESS.2018.2878807.

[26] Jensen, K.—Kristensen, L. M.: Colored Petri Nets: A Graphical Language for Formal Modeling and Validation of Concurrent Systems. Communications of the ACM, Vol. 58, 2015, No. 6, pp. 61–70, doi: 10.1145/2663340.

[27] He, C.—Ding, Z.: More Efficient On-the-Fly Verification Methods of Colored Petri Nets. Computing and Informatics, Vol. 40, 2021, No. 1, pp. 195–215, doi: 10.31577/cai_2021_1_195.

[28] Ding, Z.—Yang, R.—Cui, P.—Zhou, M.—Jiang, C.: Variable Petri Nets for Mobility. IEEE Transactions on Systems, Man, and Cybernetics: Systems, Vol. 52, 2022, No. 8, pp. 4784–4797, doi: 10.1109/TSMC.2021.3103072.

[29] Drakaki, M.—Tzionas, P.: A Colored Petri Net-Based Modeling Method for Supply Chain Inventory Management. Simulation, Vol. 98, 2022, No. 3, pp. 257–271, doi: 10.1177/00375497211038755.

[30] Ding, Z.—Li, S.—Chen, C.—He, C.: Program Dependence Net and Its Slice for Verifying Linear Temporal Properties. CoRR, 2023, doi: 10.48550/arXiv.2301.11723.

[31] Burch, J. R.—Clarke, E. M.—McMillan, K. L.—Dill, D. L.—Hwang, L. J.: Symbolic Model Checking: 1020 States and Beyond. Information and Computation, Vol. 98, 1992, No. 2, pp. 142–170, doi: 10.1016/0890-5401(92)90017-A.

[32] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, Vol. C-35, 1986, No. 8, pp. 677–691, doi: 10.1109/TC.1986.1676819.

[33] McMillan, K. L.: Symbolic Model Checking: An Approach to the State Explosion Problem. Ph.D. Thesis. Carnegie Mellon University, Pittsburgh, 1992.

[34] Godefroid, P.—Pirottin, D.: Refining Dependencies Improves Partial-Order Verification Methods. In: Courcoubetis, C. (Ed.): Computer Aided Verification (CAV 1993). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 697, 1993, pp. 438–449, doi: 10.1007/3-540-56922-7_36.

[35] Peled, D.: Combining Partial Order Reductions with On-the-Fly Model-Checking. Formal Methods in System Design, Vol. 8, 1996, No. 1, pp. 39–64, doi: 10.1007/BF00121262.

[36] Valmari, A.: A Stubborn Attack on State Explosion. Formal Methods in System Design, Vol. 1, 1992, No. 4, pp. 297–322, doi: 10.1007/BF00709154.

[37] Godefroid, P.: Using Partial Orders to Improve Automatic Verification Methods. 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. 176–185, doi: 10.1007/BFb0023731.

[38] Ma, Z.—Zhu, G.—Li, Z.: Marking Estimation in Petri Nets Using Hierarchical Basis Reachability Graphs. IEEE Transactions on Automatic Control, Vol. 66, 2021, No. 2, pp. 810–817, doi: 10.1109/TAC.2020.2983088.

[39] Cousot, P.—Cousot, R.: Refining Model Checking by Abstract Interpretation. Automated Software Engineering, Vol. 6, 1999, No. 1, pp. 69–95, doi: 10.1023/A:1008649901864.

[40] Abid, C. A.—Zouari, B.: Synthesis of Controllers for Symmetric Systems. International Journal of Control, Vol. 83, 2010, No. 11, pp. 2354–2367, doi:

10.1080/00207179.2010.520415.

[41] Chiola, G.—Dutheillet, C.—Franceschinis, G.—Haddad, S.: Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications. IEEE Transactions on Computers, Vol. 42, 1993, No. 11, pp. 1343–1360, doi: 10.1109/12.247838.

[42] Chiola, G.—Franceschinis, G.—Gaeta, R.: Modeling Symmetric Computer Architectures by SWNs. In: Valette, R. (Ed.): Application and Theory of Petri Nets 1994 (ICATPN 1994). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 815, 1994, pp. 139–158, doi: 10.1007/3-540-58152-9_9.

[43] Lahiri, S.—Qadeer, S.: Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. ACM SIGPLAN Notices, Vol. 43, 2008, No. 1, pp. 171–182, doi: 10.1145/1328897.1328461.

[44] Tsitovich, A.—Sharygina, N.—Wintersteiger, C. M.—Kroening, D.: Loop Summarization and Termination Analysis. Vol. 6605, 2011, pp. 81–95, doi: 10.1007/978-3-642-19835-9_9.

[45] Godefroid, P.—Luchaup, D.: Automatic Partial Loop Summarization in Dynamic Test Generation. Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA '11), 2011, pp. 23–33, doi: 10.1145/2001420.2001424.

[46] Brumley, D.—Wang, H.—Jha, S.—Song, D.: Creating Vulnerability Signatures Using Weakest Pre-Conditions. Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF '07), 2007, pp. 311–325, doi: 10.1109/CSF.2007.17.

[47] Vernier, I.: Symbolic Executions of Symmetrical Parallel Programs. Proceedings of 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96), IEEE, 1996, pp. 327–334, doi: 10.1109/EMPDP.1996.500604.

[48] Hamez, A.—Hillah, L.—Kordon, F.—Linard, A.—Paviot-Adet, E.—Renault, X.—Thierry-Mieg, Y.: New Features in CPN-AMI 3: Focusing on the Analysis of Complex Distributed Systems. Sixth International Conference on Application of Concurrency to System Design (ACSD '06), IEEE, 2006, pp. 273–275, doi: 10.1109/ACSD.2006.15.

Xiangyu Jia received her B.Sc. in computer science and technology from the Shandong University of Science and Technology, Qingdao, China, in 2021. She is currently pursuing her M.Sc. degree with the Department of Computer Science and Technology, Tongji University, Shanghai, China. Her current research interests include model checking and machine learning.

Shuo Li received her B.Sc. in software engineering from the Shandong University of Science and Technology, Qingdao, China, in 2017. She is currently pursuing her Ph.D. degree with the Department of Computer Science and Technology, Tongji University, Shanghai, China. Her current research interests include model checking, Petri nets, and formal methods.