Computing and Informatics, Vol. 43, 2024, 1161–1184, doi: 10.31577/cai_2024_5_1161

FORMAL MODELLING OF PROGRAM DEPENDENCE NET FOR SOFTWARE MODEL CHECKING

Shuo Li

School of Information Science and Technology
Taishan University Taian, 271000, China
e-mail: lishuo20062005@126.com

Zhijun Ding

Department of Computer Science
Tongji University
Shanghai, 201804, China
e-mail: dingzj@tongji.edu.cn

Meiqin Pan*

School of Business and Management
Shanghai International Studies University
Shanghai, 200083, China
e-mail: panmqin@sina.com

Abstract. Program dependence net (PDNet) is a kind of Petri nets which can represent concurrent systems and software to apply the automata-theoretic approach for software model checking on Linear Temporal Logic (LTL). This paper presents a formal modelling method to construct a PDNet which is consistent with the behavior of multi-threaded C programs (PThread programs) from a source code. For concurrent programs with a function call and POSIX threads, we propose the formal operational semantics by the labeled transition system (LTS). We formalize the statements by the basic PDNet structure based on LTS operational

* Corresponding author

semantics. Then, we propose the formal modelling method to build a basic flow to simulate the execution of PThread programs. Finally, we give a case study to illustrate the formal modelling method for verifying PThread programs on LTL properties.

Keywords: Formal modelling, operational semantics, PThread, PDNet, LTL

1 INTRODUCTION

The multi-threaded C programs (PThread programs) are becoming increasingly common in modern concurrent systems and software, which are universally utilized with the advent of multi-core processors and the concurrency support of popular programming languages. Thus, property verification of their safety, correctness, user-defined specification are critical and urgent. Over the past three decades, model checking [1, 2] has received much attention as a formal verification method, which could automatically explore the whole state space and effectively validate the crucial requirements of concurrent systems and software.

However, the automatic construction of a verifiable finite model is still an urgent problem. There exist some formal models, such as automata [3] and Petri nets [4]. Owning to the analysis ability of structure and behavior, Petri nets have been widely used for modelling concurrent systems and software. This formal model can explicitly describe concurrency and synchronization. Thus, it is a powerful model to represent concurrent programs. There are some works that represent concurrent programs [5, 6, 7, 8, 9]. They are able to construct Petri nets for a specific class of properties, which can be used to analyze and verify the satisfaction of such properties. But these modelling approaches either do not target a particular type of code specification, or do not propose formal modelling methods.

Therefore, constructing a verifiable model directly from the source code requires formal specifications. This is a challenging problem. Recent studies have been devoted to model multi-threaded programs using a control flow graph [10, 11]. But there is still a need for a formal modelling approach based on Petri nets. In our previous work, we proposed a program dependence net (PDNet) [12] based on Colored Petri Net (CPN) for concurrent systems and software. It also extends the types of token, guard functions and arc expression, which makes it capable to describe the operation of data. Then, it was used to verify the linear temporal properties for model checking.

In this paper, we propose a formal modelling method based on PDNet model. The overview of our PDNet-based formal modelling method is shown in Figure 1. The modelling result (a PDNet model) can be used for software model checking. The entire formal modelling process is divided into PDNet structure modelling for basic structures and basic flow modelling for control flows and data flows. Thus, the main contributions of this paper are summarized as follows:

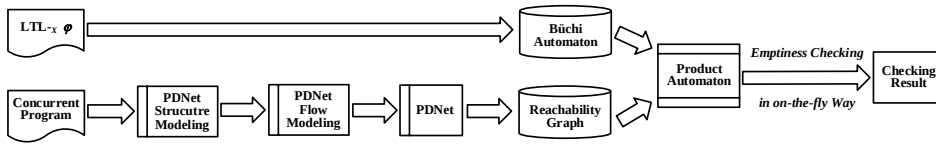


Figure 1. Overview of a formal modelling method

1. For the concurrent programs with a function call and POSIX threads, we define the formal operational semantics by the labeled transition system (LTS).
2. We propose the basic PDNet structure based on the formal operational semantics to formalize the statement.
3. We propose the formal modelling method to build a basic flow to simulate the execution of concurrent programs.

In Section 2, some definitions related to PDNet are introduced. The semantics of concurrent programs is proposed in Section 3. The modelling method based on the PDNet of structure and flow are proposed in Sections 4 and 5. Then, the model checking problem is defined in Section 6. The related works are reviewed in Section 7. Finally, we summarize the whole paper.

2 PRELIMINARIES

2.1 Introduction of PDNet

Program dependence net (PDNet) [12, 13] can combine the control-flow structure with the control-flow dependencies and provide the on-demand data-flow dependencies calculating ability for the concurrent programs [13]. We introduce the PDNet in this section.

In the following, $\mathbb{B}$ is the set of Boolean predicates with standard logic operations. $\mathbb{E}$ is a set of expressions. $Type[e]$ is the type of an expression $e \in \mathbb{E}$, i.e., the type of the values obtained when evaluating e. $Var(e)$ is the set of all variables in an expression e. $\mathbb{E}_V$ for a variable set V is the set of expressions $e \in \mathbb{E}$ such that $Var(e) \subseteq V$. $Type[v]$ is the type of a variable v.

Definition 1 (PDNet)**.** PDNet 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 of typed variables. $\forall v \in V : Type[v] \in \Sigma$.
3. P is a finite set of places. $P = P_c \cup P_v \cup P_f$. Concretely, P_c is a subset of control places, P_v is a subset of variable places, and P_f is a subset of execution 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. $F = F_c \cup F_{rw} \cup F_f$. Concretely, $F_c \subseteq (P_c \times T) \cup (T \times P_c)$ is a subset of control arcs, $F_{rw} \subseteq (P_v \times T) \cup (T \times P_v)$ is a subset of read-write arcs, and $F_f \subseteq (P_f \times T) \cup (T \times P_f)$ is a subset of execution arcs.
6. $C\colon P \to \Sigma$ is a color set function, that assigns a color set $C(p)$ belonging to the set of types Σ to each place p.
7. $G\colon T \to \mathbb{E}_V$ is a guard function, that assigns an expression $G(t)$ to each transition t. $\forall t \in T\colon Type[G(t)] \in \mathbb{B} \wedge Type[Var(G(t))] \subseteq \Sigma$.
8. $E\colon F \to \mathbb{E}_V$ is a function, that assigns an arc expression $E(f)$ to each arc f. $\forall f \in F\colon Type[E(f)] = C(p(f))_{MS} \wedge Type[Var(E(f))] \subseteq \Sigma$, where $p(f)$ is the place connected to arc f.
9. $I\colon P \to \mathbb{E}_\emptyset$ is an initialization function, that assigns an initialization expression $I(p)$ to each place p. $\forall p \in P\colon Type[I(p)] = C(p)_{MS} \wedge Var(I(p)) = \emptyset$.

Other related definitions (e.g., the enabled and occurrence rules) and more details of PDNet can be found in [12, 13]. To define the verification problem, the occurrence sequence of PDNet is defined based on its enabled and occurrence rules [12, 13].

Definition 2 (Occurrence Sequence of PDNet)**.** Let N be a PDNet, M_0 be the initial marking of N, (t, b) be a binding element of N, and $\widetilde{\mathbb{B}}(t)$ be a set of all binding elements of t. An occurrence sequence ω of N can be defined by the following inductive scheme:

1. $M_0[\varepsilon\rangle M_0$ (ε is an empty sequence),
2. $M_0[\omega\rangle M_1 \wedge M_1[(t, b)\rangle M_2\colon M_0[\omega(t, b)\rangle M_2$.

An occurrence sequence ω of N is maximal, iff

1. ω is of infinite length (e.g., (t_1, b_1), (t_2, b_2), $\cdots$, (t_n, b_n), $\cdots$), or
2. $M_0[\omega\rangle M_1 \wedge \forall t \in T$, $\nexists (t, b) \in \widetilde{\mathbb{B}}(t)\colon M_1[(t, b)\rangle$.

All maximal occurrence sequences constitute the language accepted by a PDNet N, denoted by $\mathcal{L}(N)$. The state-space of the PDNet consists of the marking set, where every marking is reached from the initial marking by an occurrence sequence. And a marking sequence $M[\omega]$ could be generated by occurring all binding elements in ω. Hence, the language $\mathcal{L}(N)$ accepted by the PDNet N represents the semantics of all marking sequences starting from the initial marking M_0 of N.

2.2 LTL Properties of PDNet

For model checking, linear temporal logic (LTL) [14] is an adequate formalism to specify the linear temporal properties of the concurrent programs. As we have defined in [12, 13], we remove the $\mathcal{X}$ operator in this paper. We introduce LTL properties based on the proposition and LTL-$_\mathcal{X}$ formula of PDNet.

Definition 3 (Proposition and LTL-$\mathcal{X}$ Formula of PDNet)**.** Let N be a PDNet, a be a proposition, $\mathbb{A}$ be a set of propositions, and ψ be a LTL-$\mathcal{X}$ formula. The syntax of propositions is defined:

$$a ::= \mathit{true} \mid \mathit{false} \mid \mathit{is\text{-}fireable}(t) \mid \mathit{token\text{-}value}(p) \star c. \tag{1}$$

The proposition semantics is defined w.r.t. a marking M:

$$\mathit{is\text{-}fireable}(t) = \begin{cases} \mathit{true}, & \text{if } \exists b\colon M[(t,b)\rangle, \\ \mathit{false}, & \text{otherwise}, \end{cases} \tag{2}$$

$$\mathit{token\text{-}value}(p) \star c = \begin{cases} \mathit{true}, & \text{if } M(p) \star c \text{ holds}, \\ \mathit{false}, & \text{otherwise}. \end{cases} \tag{3}$$

The syntax of LTL-$\mathcal{X}$ over $\mathbb{A}$ is defined:

$$\psi ::= a \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \psi_1 \Rightarrow \psi_2 \mid \mathcal{F}\psi \mid \mathcal{G}\psi \mid \psi_1 \mathcal{U} \psi_2, \tag{4}$$

where $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ are usual propositional connectives, $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{U}$ are temporal operators, ψ, ψ_1 and ψ_2 are LTL-$\mathcal{X}$ formulae.

The semantics of LTL-$\mathcal{X}$ is the same as [13] of Petri nets. For example, $\mathcal{G}$ *is-fireable*$(t) \Rightarrow \mathcal{F}$ *token-value*$(p) = 0$ means that whenever the transition t is enabled, the token of p will be equal to 0 in some subsequent states. The Büchi automaton can encode an LTL-$\mathcal{X}$ formula as [14, 15] for the explicit model checking. That is, the negation of the verified property ψ is translated into a Büchi automaton $A_{\neg\psi}$. We define the model checking problem in Section 6.

3 OPERATIONAL SEMANTICS OF CONCURRENT PROGRAM

C programs using POSIX threads [16] refers to the concurrent programs in this paper. POSIX threads extension specifies primitives to provide mutual exclusion, as well as synchronized waiting. For simplicity, we consider the assignment statements to be atomic.

Take inspiration from existing researches on the function call [10] and concurrency primitive [17], we describe the complete semantics for the function calls and the concurrency primitives. The syntax of concurrent programs is defined in [12]. To express the operational semantics of a concurrent program for our PDNet modelling, we define the labeled transition system (LTS) semantics of the concurrent programs.

Definition 4 (LTS Semantics of Concurrent Programs)**.** Let $\mathcal{P}$ be a concurrent program, $\mathcal{N}_{\mathcal{P}} ::= \langle \mathcal{S}, \mathcal{A}, \rightarrow, \mathcal{S}_0 \rangle$ be the labeled transition system of $\mathcal{P}$, where:

1. $\mathcal{S} \subseteq \mathcal{H} \times \mathcal{M} \times (\mathcal{L} \rightarrow \mathcal{I}) \times (\mathcal{C} \rightarrow \mathcal{I}_{MS})$ is a set of the program configurations.

2. $\mathcal{A} \subseteq \mathcal{T} \times \mathcal{B}$ is a set of actions, where $\mathcal{T}$ comes from $\mathcal{P}$.
3. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is a set of transition relations on the concurrent program configurations $\mathcal{S}$.
4. $\mathcal{S}_0 \in \mathcal{S}$ is the initial configuration.

Formally, $s ::= \langle h, m, r, u\rangle$ is a configuration of $\mathcal{S}$, where $h \in \mathcal{H}$ is a function that indicates the current program location of every thread, $m \in \mathcal{M}$ is the current memory state, r is a function which maps every mutex to a thread identifier, and u is also a function that maps every condition variable to a multiset of thread identifiers of the threads which are currently waiting on the condition variable. $\mathcal{S}_0 ::= \langle h_0, m_0, r_0, u_0\rangle$ where $h_0 \in \mathcal{H}$ and $m_0 \in \mathcal{M}$ come from $\mathcal{P}$, $r_0 : \mathcal{L} \rightarrow \{0\}$ represents every mutex, and it is not initially held by any threads, and $u_0 : \mathcal{C} \rightarrow \emptyset$ represents every condition variable that do not initially block any threads. Hence, we characterize the states of $\mathcal{P}$ by the configurations $\mathcal{S}$ of $\mathcal{N}_{\mathcal{P}}$. $\alpha ::= \langle \tau, \beta\rangle$ is an action of $\mathcal{A}$, where $\tau \in \mathcal{T}$ is a statement of $\mathcal{P}$ and $\beta \in \mathcal{B}$ is an effect for the operation q from the statement τ. The transition relation $\rightarrow$ on the configurations is represented by $s \xrightarrow{\langle \tau, \beta\rangle} s'$. The interleaved execution of τ could update the configuration s to a new configuration s' based on the effect β corresponding to the operation of τ. In fact, the effect of an action $\alpha \in \mathcal{A}$ characterizes the nature of the transition relations with this action on configurations of $\mathcal{N}_{\mathcal{P}}$. The effect is defined by $\mathcal{B} := (\{asi, jum, ret, tcd, fcd, call, rets\} \times K) \cup (\{acq, rel\} \times \mathcal{L}) \cup (\{sig\} \times \mathcal{C}) \cup (\{wa_1, wa_2, wa_3\} \times \mathcal{C} \times \mathcal{L}))$.

To formalize our PDNet modelling methods, the semantics of a concurrent program is expressed by the transition relations $\rightarrow$ on the program configurations under a current configuration $s = \langle h, m, r, u\rangle$ of $\mathcal{P}$ in Table 1. The intuition behind the semantics is how s updates based on the transition relations with the actions of $\mathcal{A}$. Thus, the execution of a statement gives rise to a transition relation in correspondence with the operation of the statement. For convenience, the action referenced later is denoted by an abbreviation at the end of each row in Table 1. For instance, *asi* represents the action $\langle \tau, \langle asi, l'\rangle\rangle$ where $\langle asi, l'\rangle$ is the effect corresponding to the operation of τ, updating the program location to l'. Here, suppose an assignment operation is $\nu := w$ in statement τ. $[\![w]\!]m$ denotes that the value evaluating by the expression w under the memory state m. And this value is assigned to the variable ν. Thus, $m' = m[\nu \mapsto [\![w]\!]m]$ denotes the new memory state where $m'(\nu) = [\![w]\!]m$ and $m'(y) = m(y)$ $(\forall y \in \mathcal{V} : y \neq \nu)$.

In the same way, *jum* represents the action $\langle \tau, \langle jum, l'\rangle\rangle$ where the jump operation of τ is *break* or *continue*, updating the program location to l'. But *jum* does not update the memory state. *ret* represents the action $\langle \tau, \langle ret, l'\rangle\rangle$ where the jump operation of τ is *return*, updating the program location to l'. For a branching operation, *tcd* represents the action $\langle \tau, \langle tcd, l'\rangle\rangle$, where the Boolean condition w is evaluated by *true* (i.e., $[\![w]\!]m = true$), and *fcd* represents the action $\langle \tau, \langle tcd, l'\rangle\rangle$, where the boolean condition w is evaluated by *false* (i.e., $[\![w]\!]m = false$). Neither *tcd* or *fcd* updates the memory state. And they update the program locations to different pro-

Action	Semantics
asi	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := \nu := w\, h(i) = l}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle ass, l' \rangle \rangle} \langle h[i \mapsto l'], m', r, u \rangle}$
jum	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\, q := \textit{break or continue}\ h(i) = l}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle jum, l' \rangle \rangle} \langle h[i \mapsto l'], m, r, u \rangle}$
ret	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := \textit{return}\ h(i) = l}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle ret, l' \rangle \rangle} \langle h[i \mapsto l'], m', r, u \rangle}$
tcd	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := if(w)\ \textit{or}\ while(w)\, [\![w]\!]m = true\, h(i) = l}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle tcd, l' \rangle \rangle} \langle h[i \mapsto l'], m, r, u \rangle}$
fcd	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := if(w)\ \textit{or}\ while(w)\, [\![w]\!]m = false\, h(i) = l}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle fcd, l' \rangle \rangle} \langle h[i \mapsto l'], m, r, u \rangle}$
call	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := call\, h(i) = l}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle call, l' \rangle \rangle} \langle h[i \mapsto l'], m', r, u \rangle}$
rets	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := rets\, h(i) = l}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle rets, l' \rangle \rangle} \langle h[i \mapsto l'], m', r, u \rangle}$
acq	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := \langle lock, \ell \rangle\, h(i) = l\, r(\ell) = 0}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle acq, \ell \rangle \rangle} \langle h[i \mapsto l'], m, r[\ell \mapsto i], u \rangle}$
rel	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := \langle unlock, \ell \rangle\, h(i) = l\, r(\ell) = i}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle rel, \ell \rangle \rangle} \langle h[i \mapsto l'], m, r[\ell \mapsto 0], u \rangle}$
sig	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := \langle singal, \gamma \rangle\, h(i) = l\, \{j\} \in u(\gamma)}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle sig, \gamma \rangle \rangle} \langle h[i \mapsto l'], m, r, u[\gamma \mapsto u(\gamma) \setminus \{j\} \cup \{-j\}] \rangle}$
wa_1	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := \langle wait, \gamma, \ell \rangle\, h(i) = l\, r(\ell) = i\, \{i\} \notin u(\gamma)}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle wa_1, \gamma, \ell \rangle \rangle} \langle h, m, r[\ell \mapsto 0], u[\gamma \mapsto u(\gamma) \cup \{i\}] \rangle}$
wa_2	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := \langle wait, \gamma, \ell \rangle\, h(i) = l\, r(\ell) = 0\, \{-i\} \in u(\gamma)}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle wa_2, \gamma, \ell \rangle \rangle} \langle h, m, r, u[\gamma \mapsto u(\gamma) \setminus \{-i\}] \rangle}$
wa_3	$\dfrac{\tau := \langle i, q, l, l', m, m' \rangle \in \mathcal{T}\ q := \langle wait, \gamma, \ell \rangle\, h(i) = l\, r(\ell) = 0}{\langle h, m, r, u \rangle \xrightarrow{\langle \tau, \langle wa_3, \gamma, \ell \rangle \rangle} \langle h[i \mapsto l'], m, r[\ell \mapsto i], u \rangle}$

Table 1. Semantics of concurrent programs

gram locations l'. For a function call, *call* represents the action $\langle \tau, \langle call, l' \rangle \rangle$, where l' is the entry of the called function, and *rets* represents the action $\langle \tau, \langle rets, l' \rangle \rangle$, where l' is the return site of the calling function. Similarly, suppose *cassign* of *call* is $\nu_1 := w_1$ and *rassign* of *rets* is $\nu_2 := w_2$. $m' = m[\nu_1 \mapsto [\![w_1]\!]m]$ denotes the new memory state for *call* and $m' = m[\nu_2 \mapsto [\![w_2]\!]m]$ denotes the new memory state for *rets*. In addition, *asi*, *jum*, *tcd*, *fcd*, *call* and *rets* do not update r and u of s.

Moreover, *acq* represents the action $\langle \tau, \langle acq, \ell \rangle \rangle$ corresponding to the operation $\langle lock, \ell \rangle$. If ℓ is not held by any thread ($r(\ell) = 0$), *acq* represents thread i obtains this mutex ℓ ($r[\ell \mapsto i]$) and updates the program location to l'. However, if ℓ is held by other thread, thread i could be blocked by ℓ and current configuration s cannot be updated by *acq*. And *rel* represents the action $\langle \tau, \langle rel, \ell \rangle \rangle$ corresponding to the operation $\langle unlock, \ell \rangle$. Here, $r(\ell) = i$ means the mutex ℓ is held by thread i. If $r(\ell) = i$, thread i could release this mutex ℓ ($r[\ell \mapsto 0]$) and updates the program location to l'. Then, *sig* represents the action $\langle \tau, \langle sig, \gamma \rangle \rangle$ corresponding to the operation $\langle signal, \gamma \rangle$. Thread i could notify a thread j belonging to $u(\gamma)$ ($\{j\} \in u(\gamma)$). Thus, thread j could be notified by thread i ($u[\gamma \mapsto u(\gamma) \setminus \{j\} \cup \{j, i\}]$). And it updates the program location to l'. Particularly, the operation $\langle wait, \gamma, \ell \rangle$ corresponds to three actions wa_1, wa_2 and wa_3, where only wa_3 updates the program location to l'. If the mutex ℓ is held by thread i ($r(\ell) = i$) and thread i is not waiting for γ currently ($\{i\} \notin u(\gamma)$), wa_1 ($\langle wa_1, \gamma, \ell \rangle$) represents the action releases the mutex ℓ ($r[\ell \mapsto 0]$), and thread i is added to the current thread multiset waiting on condition variable γ ($u[\gamma \mapsto u(\gamma) \cup \{i\}]$). Then, wa_2 ($\langle wa_2, \gamma, \ell \rangle$) represents the thread i is blocked until a thread j ($\{i, j\} \in u(\gamma)$) notifies by condition variable γ. Thus, thread i no long waits for a notification on γ ($u[\gamma \mapsto u(\gamma) \setminus \{i, j\}]$). Finally, if the thread i has been woken up by the other thread and ℓ is not held ($r(\ell) = 0$), wa_3 ($\langle wa_3, \gamma, \ell \rangle$) represents the action acquires the mutex ℓ again ($r[\ell \mapsto i]$) and updates the program location to i. In addition, *acq*, *rel*, *sig*, wa_1, wa_2 and wa_3 do not update m of the current configuration s.

4 PDNET STRUCTURE MODELLING

The intuition of our formal modelling is to construct specific PDNet structures for the concurrent program semantics, as shown in Table 1. Declarations of variables and functions are constructed firstly. The PDNet structures for the operations from the syntax in [12] are constructed based on the semantics in Table 1. Then, the basic flows are constructed to complete the control-flow structure. In Figure 2, those that have been built before the corresponding modelling are in the dotted boxes.

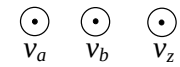
4.1 Declaration Modelling

There are three kinds of declarations we support as follows.

Variable category	***Variable declaration form***	**Σ *for global variable***	**Σ *for location variable***
Basic variable	υ ::= *char* | *unsigned char* | *short* | *unsigned short* | *int* | *unsigned int* | *long* | *unsigned long* | *float* | *double* | *long int* | *long long* | *long double*	$D \times int$	$D \times int \times string$
Pointer variable	$*\upsilon$	$int \times int$	$int \times int \times string$
Array variable	$\upsilon[n]$	$D \times int \times int$	$D \times int \times int \times string$
Structure variable	$\upsilon_1 \mid \ldots \mid \upsilon_n$	$D_1 \times \ldots \times D_n \times int$	$D_1 \times \ldots \times D_n \times int \times string$

a)

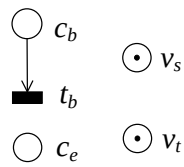
e.g. main() {
  int a=2, *b*=1;
  int *z=&*a*;
}



Type	P_v	$I(P_v)$
Basic variable	v_a	(2, *aid*)
Basic variable	v_b	(1, *bid*)
Pointer variable	v_z	(*aid*, *zid*)

b)

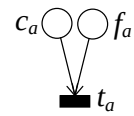
e.g.
int fun(*int s*, *int t*);



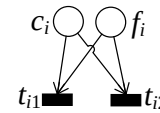
P		***I*(*P*)**	***F***	***E*(*F*)**
P_c P_f	c_b	/	(c_b, t_b)	*fun*
	c_e	/	***T***	***G*(*T*)**
P_v	v_s	(0, *sid*, *fun*)	t_b	*true*
	v_t	(0, *tid*, *fun*)		

c)

e.g.
main() {
  a=*b*+2;
}

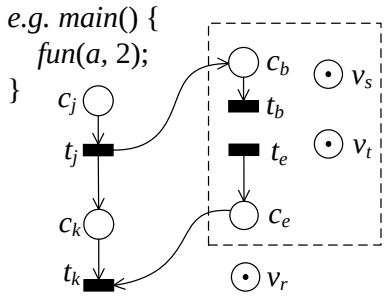


e.g.
fun() {
  if(*a*>0) { }
  else { }
}



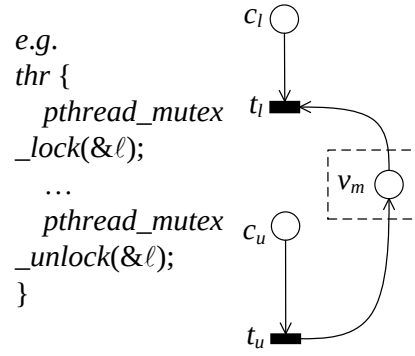
P		***M*(*P*)**	***T***	***G*(*T*)**	***F***		***E*(*F*)**
P_c	c_a	*main*	t_a	*true*	F_c	(c_a, t_a)	*main*
	c_i	*fun*	t_{i1}	[*a*>0]		(c_i, t_{i1})	*fun*
P_f	f_a	*main*				(c_i, t_{i2})	*fun*
	f_i	*fun*	t_{i2}	[*a*≤0]	F_f	(f_a, t_a)	*main*
						(f_i, t_{i1})	*fun*
						(f_i, t_{i2})	*fun*

d)



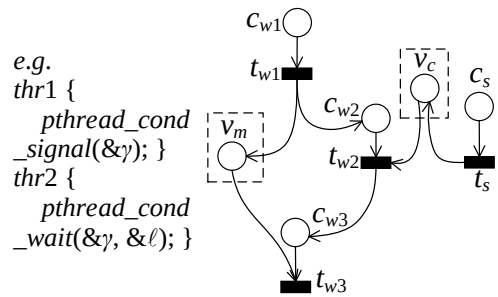
P		I(P)	T	G(T)	F	E(F)
P_c	c_j	/	t_j	true	(c_j, t_j)	main
P_f	c_k	/			(c_k, t_k)	main
P_v	v_r	(0, rid, fun)	t_k	true	(t_j, c_b)	NULL
					(c_e, t_k)	NULL

e)



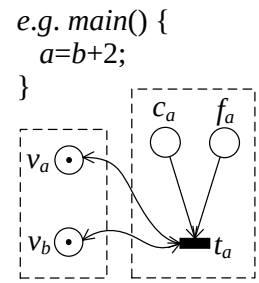
P	M(P)	T	G(T)	F	E(F)	F	E(F)
c_l	thr	t_l	true	(c_l, t_l)	thr	(v_m, t_l)	thr
c_u	thr	t_u	true	(c_u, t_u)	thr	(t_u, v_m)	thr

f)



P	M(P)	T	G(T)	F	E(F)	F	E(F)
c_{w1}	thr1	t_{w1}	true	(c_{w1}, t_{w1})	thr1	(t_{w1}, c_{w2})	thr1
c_{w2}	thr1	t_{w2}	true	(c_{w2}, t_{w2})	thr1	(t_{w2}, c_{w3})	thr1
c_{w3}	thr1	t_{w3}	true	(c_{w3}, t_{w3})	thr1	(t_{w1}, v_m)	thr1
c_s	thr2	t_s	true	(c_s, t_s)	thr2	(v_m, t_{w3})	thr1
				(v_c, t_{w2})	NULL	(t_s, v_c)	NULL

g)

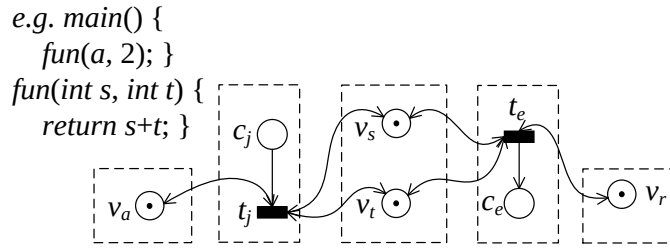


F_{rw}	$E(F_{rw})$	F_{rw}	$E(F_{rw})$
(v_a, t_a)	(b+2, aid)	(v_b, t_a)	(b, bid)
(t_a, v_a)	(a, aid)	(t_b, v_a)	(b, bid)

h)

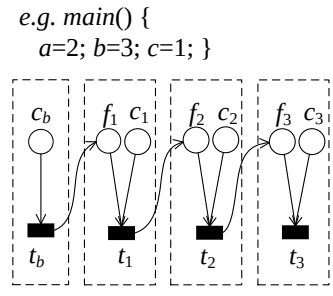
4.1.1 Variable Declarations Modelling

For variable declarations, we support four kinds of variable categories in this paper. In Figure 2 a), the variable declaration forms of basic variable – pointer variable, array variable and structure variable are listed in the second column. The color sets Σ for global variables and local variables are all products listed in the third column and the fourth column, respectively. Here, ν is a variable identifier and D represents the corresponding variable type. For Σ of a global variable, the first component of



F_{rw}	$E(F_{rw})$	F_{rw}	$E(F_{rw})$	F_{rw}	$E(F_{rw})$	F_{rw}	$E(F_{rw})$	F_{rw}	$E(F_{rw})$	F_{rw}	$E(F_{rw})$
(v_a, t_j)	(a, aid)	(v_s, t_j)	(s, sid, fun)	(v_t, t_j)	(t, tid, fun)	(v_s, t_e)	(s, sid, fun)	(v_t, t_e)	(t, tid, fun)	(v_r, t_e)	(t, rid, fun)
(t_j, v_a)	(a, aid)	(t_j, v_s)	(a, sid, fun)	(t_j, v_t)	$(2, tid, fun)$	(t_e, v_s)	(s, sid, fun)	(t_e, v_t)	(t, tid, fun)	(t_e, v_t)	$(s+t, rid, fun)$

i)



F_f	$E(F_f)$	F_f	$E(F_f)$	F_f	$E(F_f)$
(t_b, f_1)	*main*	(t_1, f_2)	*main*	(t_2, f_3)	*main*

j)

Figure 2. Modelling a) Color sets for four kinds of variable declaration forms; b) Variable declaration c) Function declaration d) PDNet structures for local operations e) PDNet structures for function call operations f) PDNet structures for operations $\langle wait, \gamma, \ell \rangle$ and $\langle signal, \gamma \rangle$ g) PDNet structures for operations $\langle lock, \ell \rangle$ and $\langle unlock, \ell \rangle$ h) PDNet arcs for data-flow of variable read/write i) PDNet arcs for data-flow of parameter passing j) PDNet arcs for traditional control-flow structure

the basic variable and array variable is the data type, and the first component of the pointer variable is *int* representing the variable identifier it currently points to. The second component of the basic variable, pointer variable and array variable is the variable identifier expressed by an integer. The third component of the array variable is the array index. Particularly, if there are n different elements in a structure variable, D_i $(i \leq n)$ represents the type of the i^{th} element. Moreover, for Σ of a local variable, the components are all the same like the global variable of the corresponding type, except that the last component is the extra thread identifier. Then, we construct specific variable places with diverse color sets. The initial markings of variable places represent the initial values. For instance, $\nu[10]$ represents a 10 dimensional local array variable of thread *thr* whose type is *double*. A variable place whose Σ is the product $double \times int \times int \times string$ is constructed for $\nu[10]$. If $\nu[5] = 1.5$, $(1.5, \nu id, 5, thr)$ represents a token corresponding to $\nu[5]$ in this variable place, where νid is the variable identifier. As the examples in Figure 2 b), variables a and b are declared as two global basic variables with the type of *int*, and z is declared as a global pointer variable. The variable places v_a, v_b and v_z are constructed when they are declared, whose identifiers are denoted by *aid*, *bid* and *zid*, respectively. Here, the first component of the initial marking $I(v_a)$ is 2 meaning the value of a is 2, and the first component of $I(v_z)$ is *aid* meaning the pointer variable z currently points to the variable a.

For two special declarations of mutex and condition variable, a place with color sets $int \times string$ is constructed for the following POSIX thread operations. And the place is not only a control place but also an execution place.

4.1.2 Function Declarations Modelling

For function declarations, we construct specific places and transitions to describe a function. As an example in Figure 2 c), v_s and v_t are constructed for the parameter list (*int s*, *int t*). A place c_b, a transition t_b (named *enter* transition) and an arc (c_b, t_b) model the *entry* of this function. If c_b gets tokens, it means the function is called and could execute the subsequent statements. Then, a place c_e models the *export* of this function. If c_e gets tokens, it means the function returns. And there could be one or more transitions in ${}^\bullet c_e$ (these transitions for action *ret* are constructed by the following operations modelling). Particularly, c_b and c_e belong to P_c and P_f, and (c_b, t_b) belongs to F_c and F_f. $E(c_b, t_b)$ and the last component of $I(v_s)$ and $I(v_t)$ are all *fun* to identify the function they belong to.

4.2 Operations Modelling

There are three kinds of statements $\tau = local \mid calls \mid syns$. We propose specific structures for the operations in *local*, *calls* and *syns* [12].

4.2.1 Local Operations

There are five actions *asi*, *jum*, *ret*, *tcd* and *fcd* of local operations. We propose two kinds of PDNet structures for five actions of local operations as follows.

1. **The PDNet structure for the assignment operation and jump operation.** From the previous explanation, *asi* of the assignment operation and *jum*, *ret* of the jump operation could update the current location to a successor location. Thus, we construct a transition to represent the action. Here, the transition for *asi* is named *assign* transition, the transition for *jum* is named *jump* transition. Moreover, there may be one or more statements corresponding to the operations *ret* of a function, and the transitions for *ret* are named *exit* transitions. As the example in Figure 2 d), a transition t_a models *asi* of the assignment operation $a := b + 2$ of $a = b + 2$. And $G(t_a) = true$ can be ignored when checking whether t_a is enabled. Then, the occurrence of t_a in a binding shall express the action *asi* for $a = b + 2$.

2. **The PDNet structure for the branching operation.** Branching operations $if(w)$ or $while(w)$ could produce two possible subsequent executions. And actions *tcd* and *fcd* could update the current location to two successor locations determined by the evaluated result $[\![w]\!]m$ under a memory state m. Thus, we construct two *branch* transitions to represent *tcd* and *fcd* for a branching operation. As the example in Figure 2 d), t_{i1} models *tcd* and t_{i2} models *fcd* of $if(a > 0)$. Concretely, $G(t_{i1}) = [a > 0]$ represents the condition $[\![a > 0]\!]m = true$ of *tcd*, and $G(t_{i2}) = [a \leq 0]$ represents the condition $[\![a \leq 0]\!]m = false$ of *fcd* under m. Then, the occurrence of t_{i1} shall express action *tcd*, and the occurrence of t_{i2} shall express action *fcd*. Here, we call t_{i1} and t_{i2} as *branch* transitions.

There is a control place and a execution place constructed for a transition. The reason behind these structures is that whether the statement gets the domination condition from the control-flow dependencies and the execution order condition from the control-flow structure are represented by different input places of the transition corresponding to this statement. That is, the control place represents the control-flow dependencies condition, and the execution place represents the execution order condition of *assign*, *jump*, *exit* or *branch* transitions. This is the key difference of our PDNet structures from the traditional CPN modelling [8]. For instance, a control place c_a and an execution place f_a are connected to t_a by a control arc (c_a, t_a) and an execution arc (f_a, t_a). Whether c_a obtains a token means the control-flow dependencies execution condition of t_a, and whether f_a obtains a token means the execution order condition of t_a in Figure 2 d). $M(c_a)$ and $M(f_a)$ are *main* representing the token that the place can obtain. $E(c_a, t_a)$ and $E(f_a, t_a)$ are *main* meaning the function they belong to.

4.2.2 Function Call Operations

As shown in Table 1, there are two actions *call* and *rets* of the operations in *calls*. We propose two kinds of PDNet structures for two actions of function call operations as follows.

1. **The PDNet structure for the call site operation.** *call* represents the action of the call site operation updating the current location to a successor location. Thus, we construct a *call* transition to represent this action. As the example in Figure 2 e), a transition t_j models the *call* transition of the function *fun*. A place c_j with an arc (c_j, t_j) models the execution condition of t_j. An arc (t_j, c_b) (named *enter* arc) models the execution from function *main* to the called function *fun*. Here, c_b constructed in Figure 2 c) is the *entry* place of function *fun*.
2. **The PDNet structure for the return site operation.** *rets* represents the action of the return site operation updating the current location to a successor location. Thus, we construct a *return* transition to represent this action. As the example in Figure 2 e), a transition t_k models the *return* transition of *fun*. A place c_k with an arc (c_k, t_k) models the execution condition of transition t_k. An arc (c_e, t_k) (named *exit* arc) models the execution from the called function *fun* to *main*. Here, c_e constructed in Figure 2 c) is the *export* place of function *fun*. Moreover, a variable place v_r models the return value of *fun*. The last component of $I(v_r)$ is *fun* to identify the function it belongs to.

Absolutely necessary, an arc (t_j, c_k) in Figure 2 e) should be constructed to identify the return side operation after the called function returns. Actually, it is unnecessary to construct control places (or arcs) and execution places (or arcs) for *call* and *return* transitions. That is, the places connected to *call* and *return* transitions belonging to control place set and execution place set represent the control-flow dependencies condition as well as the execution order condition. For example, $c_j \in P_c \wedge c_j \in P_f$ represents the domination condition and the execution order condition of t_j in Figure 2 e). And $(t_j, c_b) \in F_c \wedge (t_j, c_b) \in F_f$ represents the call condition according to control-flow dependencies and the execution order condition according to the control-flow structure in Figure 2 e). In the same way, c_k belonging to P_c and P_f, and (c_j, t_j), (c_k, t_k) and (c_e, t_k) belonging to F_c and F_f in Figure 2 e) are designed for these PDNet structures.

4.2.3 POSIX Thread Operations

As shown in Table 1, there are six actions *acq*, *rel*, *sig*, wa_1, wa_2 and wa_3 for the operations in *syncs*. We propose four kinds of PDNet structures for six actions of POSIX thread operations as follows. A place v_m represents mutex ℓ and a place v_c represents a condition variable γ in Figures 2 f) and 2 g).

1. **The PDNet structures for $\langle lock, \ell \rangle$.** From the previous explanation, *acq* of the operation $\langle lock, \ell \rangle$ could acquire ℓ and update the current location to a successor

location. Thus, we construct a *lock* transition to represent the action *acq*. If ℓ is held by other thread, the current thread could be blocked. Thus, we construct an arc connected the place of ℓ to the *lock* transition to model this blocked state. As the example in Figure 2 f), a transition t_l models the *lock* transition. An arc (v_m, t_l) models the blocked condition if ℓ is held by other thread. And a place c_l with an arc (c_l, t_l) models the execution condition of t_l. $M(c_l)$ are *thr* representing the token that the place can obtain. $E(c_l, t_l)$ and $E(v_m, t_l)$ are *thr* meaning the function they belong to.

2. **The PDNet structures for $\langle unlock, \ell \rangle$.** From the previous explanation, *rel* of the operation $\langle unlock, \ell \rangle$ could release ℓ and update the location to a successor location if ℓ is held by the current thread. Thus, we construct a *unlock* transition to represent *rel*. As the example in Figure 2 f), a transition t_u models the *unlock* transition. An arc (t_u, v_m) models ℓ released by the current thread. And a place c_u with an arc (c_u, t_u) models the execution condition of t_u. $M(c_u)$ are *thr* representing the token that the place can obtain. $E(c_u, t_u)$ and $E(t_u, v_m)$ are *thr* meaning the function they belong to.

3. **The PDNet structures for $\langle signal, \gamma \rangle$.** From the previous explanation, *sig* of $\langle signal, \gamma \rangle$ could notify a thread waiting for the notification on γ and update the current location to a successor location. Thus, we construct a *signal* transition to represent the action *sig*. As the example in Figure 2 g), a transition t_s models the *signal* transition. An arc (t_s, v_c) models the notification on γ by the current thread. And a place c_s with an arc (c_s, t_s) models the execution condition of t_s. $M(c_s)$ are $thr2$ representing the token that the place can obtain. $E(c_s, t_s)$ and are $thr2$ meaning the function they belong to. $E(t_s, v_c)$ is *NULL* meaning it does not belong to any threads.

4. **The PDNet structures for $\langle wait, \gamma, \ell \rangle$.** Differently, $\langle wait, \gamma, \ell \rangle$ corresponds to three actions wa_1, wa_2 and wa_3. wa_1 could release ℓ if ℓ is held by the current thread. Then wa_2 could be blocked until the current thread is notified on γ. Finally wa_3 could acquire ℓ if ℓ is not held by other thread and update the current location to a successor location. Thus, we construct three *wait* transitions to represent wa_1, wa_2 and wa_3. As the example in Figure 2 g), t_{w1}, t_{w2} and t_{w3} model wa_1, wa_2 and wa_3, respectively. At the same time, places c_{w1}, c_{w2} and c_{w3} with arcs (c_{w1}, t_{w1}), (c_{w2}, t_{w2}) and (c_{w3}, t_{w3}) model the execution condition of t_{w1}, t_{w2} and t_{w3}, respectively. An arc (v_c, t_{w2}) models the current thread notified on γ (action wa_2). Unless v_c is obtained the token, t_{w2} is blocked and waiting for the notification on γ. An arc (t_{w1}, v_m) models the current thread releasing ℓ (action wa_1), and an arc (v_m, t_{w3}) models the current thread acquiring ℓ (action wa_3). (t_{w1}, c_{w2}) and (t_{w2}, c_{w3}) model the execution orders between wa_1, wa_2 and wa_3. Moreover, $M(c_{w1})$, $M(c_{w2})$ and $M(c_{w3})$ are *thr* representing the token that the places can obtain. $E(c_{w1}, t_{w1})$, $E(c_{w2}, t_{w2})$, $E(c_{w3}, t_{w3})$, $E(t_{w1}, v_m)$, $E(v_m, t_{w3})$, $E(t_{w1}, c_{w2})$ and $E(t_{w2}, c_{w3})$ are $thr1$ meaning the function they belong to. $E(v_c, t_{w2})$ is *NULL* meaning it does not belong to any threads.

Similarly, it is also unnecessary to construct control places (or arcs) and execution places (or arcs) for *lock*, *unlock*, *signal* and *wait* transitions. For example, $c_l \in P_c \wedge c_l \in P_f$ represents the domination condition and the execution order condition of t_l, and $(v_m, t_l) \in F_c \wedge (v_m, t_l) \in F_f$ represents the lock condition according to control-flow dependencies and the execution order condition according to the control-flow structure of t_l in Figure 2 f). Thus, c_u, c_s, c_{w1}, c_{w2}, c_{w3} and c_{w1} belonging to P_c and P_f, (c_l, t_l), (c_u, t_u), (t_u, v_m), (c_s, t_s), (t_s, v_c), (v_c, t_{w2}), (c_{w1}, t_{w1}), (c_{w2}, t_{w2}), (c_{w3}, t_{w3}), (t_{w1}, c_{w2}), (t_{w2}, c_{w3}), (t_{w1}, v_m) and (v_m, t_{w3}) belonging to F_c and F_f are designed for these PDNet structures in Figures 2 f) and 2 g).

5 BASIC FLOWS MODELLING

After constructing the PDNet structures for the operations, we construct specific arcs for basic flows to complete the control-flow structure.

5.1 Data-Flow of Variable Read and Write

Reading and writing variables could change the memory state in the configuration. Due to updating the memory state in the configuration, there are variable read or write in the action *asi*. And there are variable read in the actions *tcd* and *fcd*, when evaluating the expression under the current configuration. Thus, PDNet arcs for data-flow of variable read/write are constructed for *assign* and *branch* transitions. As the example in Figure 2 h), variable a and b have been declared in Figure 2 b), and the PDNet structure for $a = b + 2$ has been constructed in Figure 2 d). The read-write arcs (v_b, t_a) and (t_a, v_b) model reading the current values of variable b. And read-write arcs (v_a, t_a) and (t_a, v_a) model writing the evaluating result of $b + 2$ to variable a. Note that a pair of read-write arcs is bidirectional arc and their arc expressions are listed at the bottom of Figure 2 h). Here, $E(v_b, t_a) = E(t_a, v_b)$ implies t_a could reference the token of v_b corresponding to reading the value of variable b, and $E(v_a, t_a) \neq E(t_a, v_a)$ implies t_a could update the token of v_a corresponding to writing the value of variable a.

5.2 Data-Flow of Parameter Passing

As shown in Table 1, there are the assignments *cassign* to formal input parameters of *call* and the assignments *rassign* to actual return parameter of *rets* for parameter passing. As an example in Figure 2 i), *fun* has been declared in Figure 2 c), the PDNet structure for $fun(a, 2)$ has been constructed in Figure 2 e), and the *exit* transition t_e has been constructed for action *ret* of *return* $s + t$ according to local operations modelling. Thus, the read-write arcs (v_a, t_j) and (t_j, v_a) model reading the current values of variable a. The read-write arcs (v_s, t_j) and (t_j, v_s) ((v_t, t_j) and (t_j, v_t)) model the assignment to the formal parameter s (t), and the read-write arcs

(v_r, t_e) and (t_e, v_r) model the assignment to the return variable r. In the same way, the residual read-write arcs are constructed to model the parameter passing listed in the bottom of Figure 2 i).

5.3 Control-Flow Structure

After constructing the operations, the PDNet structures are separated. Thus, we construct execution arcs to model the execution orders of the control-flow structure. As an example in Figure 2 j), the PDNet structures for three operations $a := 1$, $b := 3$ and $c := 1$ have been constructed similarly to $a := b + 2$ of Figure 2 d). The execution arcs (t_b, f_1), (t_1, f_2) and (t_2, f_3) model the actual execution order. The occurrence order is consistent with the actual execution order. $E(t_b, f_1)$, $E(t_1, f_2)$ and $E(t_2, f_3)$ are *main* meaning the function they belong to. Therefore, our current PDNet is a complete control-flow structure and provides the control place interfaces to describe the control-flow dependencies.

6 MODEL CHECKING BASED ON PDNET

6.1 Model Checking Problem for LTL

Traditionally, the automata-theoretic approach [18] to an explicit model checking explores all possible executions of the transition systems exhaustively. The model-checking problem for LTL properties is translated to an emptiness checking problem as the following steps:

1. translate the negation of the verified property into a Büchi automaton [14],
2. compute a product automaton of the concurrent program by intersecting the transition system of each thread,
3. synchronize the product automaton and the Büchi automaton in an adequate way to yield a composed automaton, and
4. check the emptiness of the language of the product automaton.

Among them, step 3. and 4. can be processed with an on-the-fly way, i.e., checking the emptiness while synchronizing, which is called on-the-fly exploration [19]. PDNet can apply this automata-theoretic approach [18]. For the product automaton of 3., the marking of PDNet could synchronize with the states of Büchi automaton (i.e., Büchi states) [14] according to Definition 3. That is, the initial product state is generated by the initial marking of the PDNet and the initial Büchi state, and an acceptable path starting from the initial product state is extended until reaching an acceptable product state (i.e., a product state with an acceptable Büchi state) [14]. Thus, all acceptable paths constitute the language accepted by the product automaton. In the following, the property formula without $\mathcal{X}$ operator is denoted by LTL-$_{\mathcal{X}}$.

Definition 5 (Semantics of Product Automaton)**.** Let N be a PDNet, ψ be an LTL-$\mathcal{X}$ formula. Büchi automaton corresponding to the negation of ψ is denoted by $A_{\neg\psi}$. The product automaton synchronizing N and $A_{\neg\psi}$ is denoted by $N \cap A_{\neg\psi}$. The acceptable path of the product automaton is denoted by ϖ. The semantics of the product automaton:

$$[\![N \cap A_{\neg\psi}]\!] ::= \bigcup_{\varpi \in L(N \cap A_{\neg\psi})} \varpi. \tag{5}$$

The semantics is identified by the language of $N \cap A_{\neg\psi}$. Then, we formalize our LTL verification task as an emptiness checking problem based on the semantics.

Definition 6 (LTL-$\mathcal{X}$ Verification Problem)**.** A PDNet $N \models \psi$ iff the semantic of the PDNet N synchronized with the Büchi automaton transformed from ψ is empty:

$$[\![N \cap A_{\neg\psi}]\!] = \emptyset. \tag{6}$$

It holds if and only if there exists no acceptable sequence reachable from initial product state in $N \cap A_{\neg\psi}$. That is, there exists a counter-example such that the semantics of this product automaton is not empty ($N \not\models \psi$).

6.2 Case Study

We address the LTL-$\mathcal{X}$ verification problem of concurrent programs using POSIX threads [16]. Based on Definition 6, we present a case study to illustrate the LTL model detection method in this paper.

Here, a program with an *error* location in Line 7 is an example program in Figure 3 a). An LTL-$\mathcal{X}$ formula $\mathcal{G}\ \neg error()$ expresses the safety property of this program, which means $error()$ does not execute in any case, based on Definition 3.

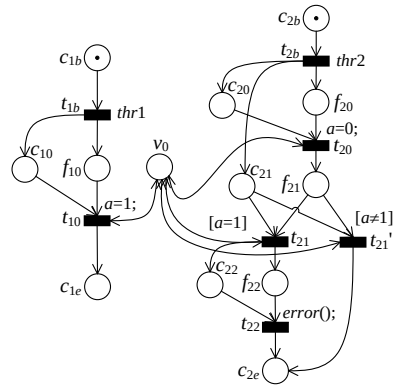
We use specific places and arcs for the transition of PDNet in Figure 3 b) (the labels on all arcs are omitted for simplicity) to distinguish the execution order condition and domination condition of a statement (modelling detail in [12]). The execution order condition reflects the actual execution syntax and semantics in the control-flow structure. For instance, f_{20}, (t_{20}, f_{21}) and (f_{21}, t_{21}) characterize the fact that $a = 0$ executes after $if(a = 1)$.

For the LTL-$\mathcal{X}$ formula $\mathcal{G}\ \neg error()$, the atomic proposition $error()$ is firstly translated to *is-fireable*(t_{22}) of PDNet in Figure 3 b). If a marking enabling t_{22}, such an atomic proposition is *true*. For example, t_{22} is enabled under M_{10} of Figure 3 d), and *is-fireable*(t_{22}) is *true* in M_{10}. Then, this formula is translated to its negation form $\mathcal{F}$ *-fireable*(t_{22}), which is further translated to a Büchi automaton [14] in Figure 3 c). There are three states in the Büchi automaton, where q_1 labeled by *is-fireable*(t_{22}) can only be synchronizing with the reachable markings enabling t_{22}.

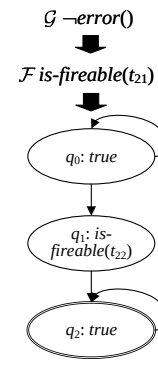
```
1 int a=0;
2 int thr1{
3    a:=1;
  }
4 int thr2{
5    a:=0;
6    if (a=1)
7       error();
  }
8 void* main(){
9    pthread_t t1, t2;
10   pthread_create(&t1,0,thr1,0);
11   pthread_create(&t2,0,thr2,0);
12   pthread_join(t1,0);
13   pthread_join(t2,0);
  }
```

LTL: $\mathcal{G}\neg error()$

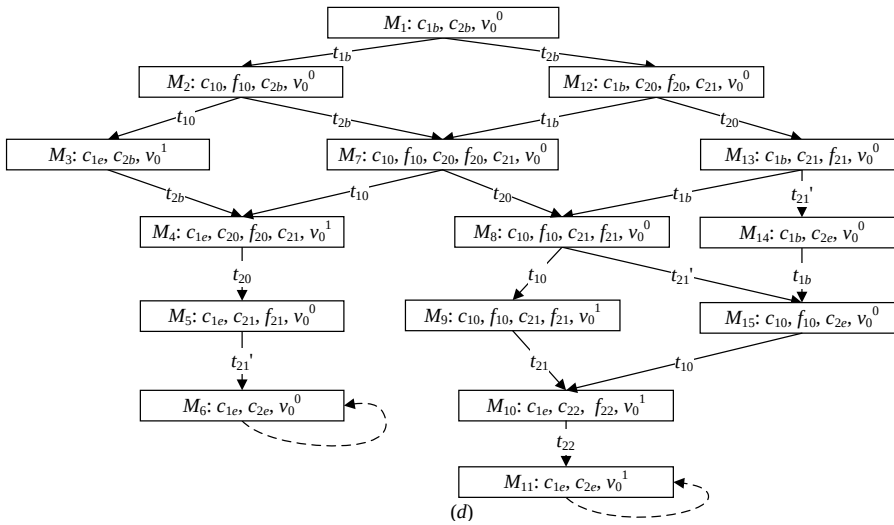
a) Example of a concurrent program



b) PDNet converted from this program



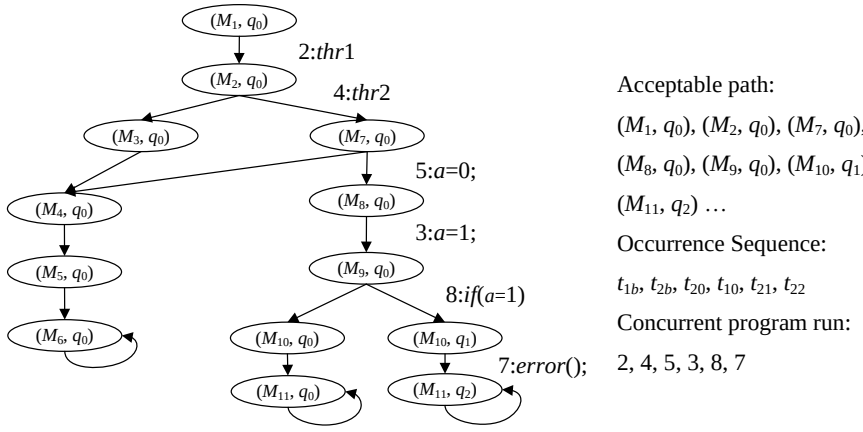
c) Büchi automaton



d) The reachability graph of PDNet

Here, *true* means that q_0 and q_2 can be synchronizing with any reachable markings in Figure 3 d).

Then, the reachability graph of PDNet in Figure 3 b) represents the state space in Figure 3 d). The nodes identify markings expressed as rectangles with place names, and the edges identify the marking transitions corresponding to the transition occurrence of PDNet expressed by arrows. That is, a transition labeled to the edge indicates this transition occurrence to give rise to the marking transition. The markings are represented by the places marked by tokens currently. For instance,



e) The explored product automaton of PDNet

Figure 3. Case study

M_6 is a marking, where places c_{1e}, c_{2e} and v_0 are marked by tokens. $M_1 \xrightarrow{t_{1b}} M_2$ is a marking transition from M_1 to M_2 by firing t_{1b}. Particularly, the superscripts of places v_0 indicate the variable value in markings. For instance, symbol v_0^0 in M_6 represents a value is 0. Moreover, arcs of M_6 and M_{11} pointing to itself are added as dotted arrows because LTL model checking problem is based on infinite paths.

Finally, the product states are explored in Figure 3 e) for the first counterexample with an on-the-fly way [14]. For instance, (M_{10}, q_1) is a product state synchronizing a marking M_{10} in Figure 3 d) and a state q_1 in Figure 3 e) because t_{22} is enabled in M_{10}. The execution 2, 4, 5, 3, 8, and 7 corresponding to the occurrence sequence t_{1b}, t_{2b}, t_{20}, t_{10}, t_{21} and t_{22}. It can be identified by the marking sequence (M_1, q_0), (M_2, q_0), (M_7, q_0), (M_8, q_0), (M_9, q_0), (M_10, q_1), and $(M_11, q_2) \cdots$ of Figure 3 e) is an acceptable path in this example concurrent program in Figure 3 a). As a result, the example program in Figure 3 a) violates the safety property $\mathcal{G}\ \neg error()$.

Thus, we can apply our formal modelling methods to construct a formal model as a verifiable finite model to verify an LTL property.

7 RELATED WORK

At present, the existing tools for software model checking usually adopt a formal model based on automata, i.e., control flow automata (CFA). At present, CFA-based model only describes execution order of sequence programs or concurrent programs. In addition, most software model checking tools combine predicate abstraction [20] to convert program into a Boolean program, which has the problem

of ignoring data flow. The automata model constructs state transition relationship of programs through state and arc, while Petri nets model can describe basic structure and flow relationship of variable resources through transition, place and arc. The compression of each transition may reduce multiple states, and compression of each place can reduce all states, which can achieve better effect than automata.

Therefore, Petri nets model has more advantages in model checking. Multithread net [21] is a model to describe shared read and synchronous lock, which is used to detect concurrent errors (such as deadlock, data race). CNet [22] is an extended Petri net model, which introduces variable places and read/write arcs and can describe the interaction between implicit relationship between data, operations and resources of programs. But it lacks the corresponding firing rules. At present, these modelling methods based on Petri nets do not target a particular type of the code specification nor propose any formal modelling methods.

8 CONCLUSION

The existing modelling approaches either do not target a particular type of code specification or do not propose formal modelling methods. But the formal model construction of a verifiable finite model is still an urgent problem and a challenging task. In this paper, we define the formal operational semantics by a labeled transition system for concurrent programs with a function call and POSIX threads. Then, we propose the basic PDNet structure based on the formal operational semantics to formalize the executed statement, and a corresponding formal modelling method to build a basic flow to simulate the execution of concurrent programs. Finally, we give the model checking problem based on PDNet to illustrate the practical application of our formal modelling methods.

In the future, we will combine the advantage of *LLVM* compilers to improve the scalability of our formal modelling approach.

Acknowledgement

This work is supported by the Shanghai International Studies University Development Program under Grant 2019114009.

REFERENCES

[1] Clarke, E. M.—Emerson, E. A.—Sistla, A. P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 8, 1986, No. 2, pp. 244–263, doi: 10.1145/5397.5399.

[2] Holzmann, G. J.: Explicit-State Model Checking. In: Clarke, E. M., Henzinger, T. A., Veith, H., Bloem, R. (Eds.): Handbook of Model Checking. Springer, Cham, 2018, pp. 153–171, doi: 10.1007/978-3-319-10575-8_5.

[3] Kupferman, O.: Automata Theory and Model Checking. In: Clarke, E. M., Henzinger, T. A., Veith, H., Bloem, R. (Eds.): Handbook of Model Checking. Springer, Cham, 2018, pp. 107–151, doi: 10.1007/978-3-319-10575-8_4.

[4] Ding, Z.—Jiang, C.—Zhou, M.: Design, Analysis and Verification of Real-Time Systems Based on Time Petri Net Refinement. ACM Transactions on Embedded Computing Systems (TECS), Vol. 12, 2013, No. 1, Art. No. 4, doi: 10.1145/2406336.2406340.

[5] Ding, Z.—Jiang, C.: Verification of Concurrent Programs by Temporal Petri Nets. Journal of Computer Science, Vol. 25, 2002, No. 5, pp. 467–475 (in Chinese).

[6] Kavi, K. M.—Moshtaghi, A.—Chen, D.: Modeling Multithreaded Applications Using Petri Nets. International Journal of Parallel Programming, Vol. 30, 2002, No. 5, pp. 353–371, doi: 10.1023/A:1019917329895.

[7] Wang, S.—Dong, Y.: A Verifiable Low-Level Concurrent Programming Model Based on Colored Petri Nets. Science China Information Sciences, Vol. 54, 2011, No. 10, pp. 2013–2027, doi: 10.1007/s11432-011-4300-1.

[8] Westergaard, M.: Verifying Parallel Algorithms and Programs Using Coloured Petri Nets. In: Jensen, K., van der Aalst, W. M., Ajmone Marsan, M., Franceschinis, G., Kleijn, J., Kristensen, L. M. (Eds.): Transactions on Petri Nets and Other Models of Concurrency VI (TOPNOC). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7400, 2012, pp. 146–168, doi: 10.1007/978-3-642-35179-2_7.

[9] Gan, M.—Wang, S.—Ding, Z.—Zhou, M.—Wu, W.: An Improved Mixed-Integer Programming Method to Compute Emptiable Minimal Siphons in S3PR Nets. IEEE Transactions on Control Systems Technology, Vol. 26, 2018, No. 6, pp. 2135–2140, doi: 10.1109/TCST.2017.2754982.

[10] Masud, A. N.—Lisper, B.: Semantic Correctness of Dependence-Based Slicing for Interprocedural, Possibly Nonterminating Programs. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 42, 2021, No. 4, Art. No. 19, doi: 10.1145/3434489.

[11] Masud, A. N.: Efficient Computation of Minimal Weak and Strong Control Closure. Journal of Systems and Software, Vol. 184, 2022, Art. No. 111140, doi: 10.1016/j.jss.2021.111140.

[12] Ding, Z.—Li, S.—Chen, C.—He, C.: Program Dependence Net and On-Demand Slicing for Property Verification of Concurrent System and Software. Journal of Systems and Software, Vol. 219, 2025, Art. No. 112221, doi: 10.1016/j.jss.2024.112221.

[13] Li, S.—Chen, C.—Huang, Z.—Ding, Z.: Change-Aware Model Checking for Evolving Concurrent Programs Based on Program Dependence Net. Journal of Software: Evolution and Process, Vol. 36, 2024, No. 6, Art. No. e2626, doi: 10.1002/smr.2626.

[14] 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.

[15] Chen, C.—Li, S.—Ding, Z.: Automatic Modeling Method for PThread Programs Based on Program Dependence Net. Proceedings of the 2021 3rd International Conference on Software Engineering and Development (ICSED '21), ACM, 2021, pp. 9–18, doi: 10.1145/3507473.3507475.

[16] Barney, B.: POSIX Threads Programming. 2019, `https://computing.llnl.gov/tutorials/pthreads/`.

[17] Hatcliff, J.—Corbett, J.—Dwyer, M.—Sokolowski, S.—Zheng, H.: A Formal Study of Slicing for Multi-Threaded Programs with JVM Concurrency Primitives. In: Cortesi, A., Filé, G. (Eds.): Static Analysis (SAS 1999). Springer Berlin Heidelberg, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1694, 1999, pp. 1–18, doi: 10.1007/3-540-48294-6_1.

[18] Kahlon, V.—Gupta, A.: An Automata-Theoretic Approach for Model Checking Threads for LTL Propert. 21st Annual IEEE Symposium on Logic in Computer Science (LICS '06), 2006, pp. 101–110, doi: 10.1109/LICS.2006.11.

[19] Ding, Z.—He, C.—Li, S.: EnPAC: Petri Net Model Checking for Linear Temporal Logic. 2023 International Conference on Networking, Sensing and Control (ICNSC), IEEE, Vol. 1, 2023, pp. 1–6, doi: 10.1109/ICNSC58704.2023.10318998.

[20] Kroening, D.—Tautschnig, M.: CBMC – C Bounded Model Checker. In: Ábrahám, E., Havelund, K. (Eds.): Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 8413, 2014, pp. 389–391, doi: 10.1007/978-3-642-54862-8_26.

[21] Sun, J.—Liu, G.—Xiang, D.—Jiang, C.: A Petri-Net-Based Method for Detecting Bugs in Multiple Threads. 2019 IEEE 16th International Conference on Networking, Sensing and Control (ICNSC), 2019, pp. 150–156, doi: 10.1109/ICNSC.2019.8743177.

[22] Zhou, G. F.—Du, Z. M.: Petri Nets Model of Implicit Data and Control in Program Code. Journal of Software, Vol. 22, 2011, No. 12, pp. 2905–2918 (in Chinese).

Shuo Li received her Ph.D. degree from the Tongji University, Shanghai, China, in 2024. Currently, she is a Lecturer with the School of Information Science and Technology, Taishan University, Taian, Shandong, China. Her current research interests include formal methods, model checking, and Petri nets. She has published 8 papers in domestic and international academic journals and conference proceedings.

Zhijun Ding received his Ph.D. degree from the Tongji University, Shanghai, China, in 2007. Currently he is Professor with the Department of Computer Science and Technology, Tongji University, Shanghai, China. His research interests include formal method, Petri nets, services computing, and workflow. He has published more than 100 papers in domestic and international academic journals and conference proceedings.

Meiqin Pan received her Ph.D. degree from the Shandong University of Science and Technology, Qingdao, China, in 2008. Now she is Associate Professor at the School of Business and Management, Shanghai International Studies University, Shanghai, China. Her research interests include information systems, data mining and technology, optimization methods. She has published more than 20 papers in home and international academic journals and conference proceedings.