Journal Pre-proof

Program dependence net and on-demand slicing for property verification of concurrent system and software

Zhijun Ding, Shuo Li, Cheng Chen, Cong He

PII: S0164-1212(24)00265-6
DOI: https://doi.org/10.1016/j.jss.2024.112221
Reference: JSS 112221

To appear in: *The Journal of Systems & Software*

Received date : 7 November 2023
Revised date : 1 July 2024
Accepted date : 16 September 2024

Please cite this article as: Z. Ding, S. Li, C. Chen et al., Program dependence net and on-demand slicing for property verification of concurrent system and software. *The Journal of Systems & Software* (2024), doi: https://doi.org/10.1016/j.jss.2024.112221.

This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2024 Published by Elsevier Inc.

Journal Pre-proof

Manuscript tex Click here to view linked References

Program Dependence Net and On-demand Slicing for Property Verification of Concurrent System and Software

Zhijun Ding[a], Shuo Li[a,*], Cheng Chen[a] and Cong He[a]

[a]Tongji University, Shanghai, 201804, China

ARTICLE INFO

Keywords:
Petri nets
Model Checking
Linear Temporal Logic
On-demand Slicing

ABSTRACT

When checking concurrent software using a finite-state model, we face a formidable state explosion problem. One solution to this problem is dependence-based program slicing, whose use can effectively reduce verification time. It is orthogonal to other model-checking reduction techniques. However, when slicing concurrent programs for model checking, there are conversions between multiple irreplaceable models, and dependencies need to be found for variables irrelevant to the verified property, which results in redundant computation. To resolve this issue, we propose a Program Dependence Net (PDNet) based on Petri net theory. It is a unified model that combines a control-flow structure with dependencies to avoid conversions. For reduction, we present a PDNet slicing method to capture the relevant variables' dependencies when needed. PDNet and its on-demand slicing in verifying linear temporal logic are used to significantly reduce computation cost. We implement a model-checking tool based on PDNet and its on-demand slicing and validate the advantages of our proposed methods.

1. Introduction

Verifying Linear Temporal Logic (abbreviated as LTL) properties that specify the correctness of concurrent systems and software is a challenging task. Finite-state model checking [9] is one of the most widely used methods for this purpose. However, the state explosion problem seriously hinders its practical application. To address this issue, researchers have developed reduction techniques like partial order [1] from a state-space perspective. This technique can reduce possible orderings of independent statements, but it cannot guarantee that all orderings irrelevant to the verified property are completely reduced.

The program slicing theory suggests that a slicing criterion is used to identify the essential parts needed to capture all relevant information for verified properties. Slicing methods with their focus on properties have been a widely used reduction technique in software verification [18, 19, 7]. The existing evaluations [13, 6, 27] have confirmed that they are effective in reducing the verification time and are orthogonal to other reduction techniques for model checking. Traditionally, program slicing builds a program dependence graph (PDG) [15, 22, 36] based on a control-flow graph (CFG). In PDG, nodes represent statements and edges capture the relationships among them. These relationships fall into two categories: dependencies on control flow and data flow. The former is determined by analyzing the conditions that dictate statement execution in CFG, while the latter is established by the definition-use relationship of variables on CFG's nodes. By applying transitive closure to PDG's edges starting from the nodes meeting a slicing criterion, one can identify the remaining nodes. These nodes correspond to the statements in the residual program, also known as a program slice. Dependence-based program slicing is commonly employed as a preprocessing technique for model checking [18, 20]. Control-flow automata (CFAs) are used to represent a slice, which are formal models that describe a control-flow structure, i.e., the order in which statements are executed, in many advanced tools [32]. CFA's nodes signify control locations, with the edges denoting program operations. The executed operation is labeled on the edges between two nodes when a control location transitions from a source to destination. The reachable tree of a CFA is utilized to determine state space for a model checking purpose.

The above-mentioned methods have the drawback of imposing significant computation cost. Firstly, the utilization of multiple models (such as the PDG for slicing and the CFA for model checking) at distinct stages necessitates conversions between them. The implementation of a unified model can obviate the necessity for such conversions and diminish computation cost from PDG to CFA. Secondly, to produce the PDG, all dependencies on data flow must be thoroughly captured in advance. It is important to note that some variables' dependencies on data flow may not be relevant to the verified property in PDG. Our proposal aims to tackle these two shortcomings by presenting a unified model that combines a control-flow structure with program dependencies. This model can determine the necessary dependencies on data flow as needed during slicing instead of capturing them beforehand. However, implementing this model in the CFA poses a challenge. CFA edges represent executed operations or statements, while PDG edges indicate program dependencies among operations or statements. Thus, both types of edges are crucial and irreplaceable, making it difficult to merge them directly and determine dependencies on data flow based solely on CFA.

As a concurrent system model, Petri nets (PNs) [46, 8] can represent the control-flow structure of a concurrent program. An automaton-theoretic [24] approach can be applied

*Corresponding author
dingzj@tongji.edu.cn (Z. Ding); lishuo20062002@126.com (S. Li)
ORCID(s): 0000-0003-2178-6201 (Z. Ding); 0000-0002-1984-6271 (S. Li)

Journal Pre-proof

to PNs for LTL model checking. In the context of CFA, an edge is limited to a single source. Differently, a PN's transition's input place is not constrained to a single source. It allows for the representation of a statement's execution order condition and domination condition via distinct input places of the corresponding transition. PNs provide an appropriate means to amalgamate a control-flow structure and program dependencies. Furthermore, the definition-use relationship of variables used to capture dependencies on data flow should be calculated by using variable operations and a control-flow structure. Colored Petri nets (CPNs), which are a form of high-level Petri net, can represent variable operations with colored places and expressions, thereby enabling PNs to effectively capture dependencies on data flow as needed.

While current verification methods employing PNs or CPNs exist [16, 11], they fail to fully combine a control-flow structure with dependencies on control flow. As a solution, we propose a new model, *Program Dependence Net* (PDNet), which leverages CPNs as a unified model to minimize the computation cost incurred in the above-mentioned conversions.

Some methods of PN slicing [26] are used to reduce concurrent models like workflows. However, they fail to take program dependencies into consideration. To address the mentioned issues, we propose on-demand slicing of PDNet to reduce computation cost associated with dependencies on data flow of irrelevant variables. We aim to make the following novel contributions to the field of model checking:

1. We propose PDNet as a unified model for concurrent programs. It combines a control-flow structure with dependencies on control flow, thus avoiding the computation cost of model conversions required by traditional PDG-based program slicing methods.

2. We propose a new method to reduce PDNet by using a slicing technique that extracts criterion from LTL formulae. The key to our approach is capturing dependencies on data flow in an on-demand way based on PDNet, thus avoiding unnecessary computation.

3. We implement a model checking tool named *DAMER*, standing for Dependence Analyser and Multi-threaded programs checkER. It automatically translates concurrent programs to PDNet without any manual intervention and reduces PDNet by on-demand slicing.

The next section gives a motivating example. Section 3 proposes PDNet and dependency modeling with PDNet. Section 4 proposes on-demand PDNet slicing. Section 5 discusses experimental results. Section 6 briefs the related works. Section 7 concludes this article.

2. Motivating Example

To explore the challenge of verifying LTL in concurrent programs that utilize POSIX threads [2], we present an example program in Figure 1(a), which contains an error on Line 9. The safety of the program is defined by the LTL-$\mathcal{X}$ formula $\mathcal{G}\,\neg error()$, which ensures that the function $error()$ is not executed in any state along any path. CFA in Figure 1(b) shows the program's control-flow structure. The nodes are represented as circles with integers indicating the control location (matching those in Figure 1(a)). The edges are represented by arrows indicating the statements. Statement executions cause the control location to shift. For instance, the node labeled by 9 corresponds to Location 9 of the program, and the edge from 9 to 10 corresponds to statement $error()$. PDG [18] is in Figure 1(c). For dependencies on control flow represented by dotted arrows, x=1, y=3 and z=y+2 all depend on the entry of $thr1$, $error()$ depends on the condition if(x<1), while if(x<1) depends on the entry of $thr2$. For dependencies on data flow represented by bold arrows, if(x<1) of $thr2$ depends on x=1 of $thr1$ because x<1 references x defined in x=1 and they belong to different concurrently executing threads. z=y+2 depends on y=3 of $thr1$ because z=y+2 references y defined in y=3 and z=y+2 is reachable from y=3. The criterion for $\mathcal{G}\,\neg error()$ is represented by bold nodes, and the remaining nodes are filled with light gray in Figure 1(c).

Figure 1(d) gives the traditional CPN model [23] for this program. For the sake of simplicity, the labels on the arcs are not shown. In this model, each transition signifies the execution of a statement when it occurs. The variables represented by the places are operated upon by the corresponding transition occurrences. For instance, after the firing of t_{10} (x=1), t_{11} (y=3) is enabled. After t_{10} occurs, x is assigned to 1. However, it is important to note that this model only represents the control-flow structure and does not show program dependencies. To combine the control-flow structure and dependencies on control flow, we establish specific places and arcs within the PDNet transition illustrated in Figure 1(e). Although the labels on the arcs have been excluded for clarity purposes, they are essential in distinguishing between a statement's execution order condition and domination condition. The former represents the actual execution syntax and semantics in the control-flow structure, while the latter represents the dependencies on control flow. For instance, f_{11}, (t_{10}, f_{11}) and (f_{11}, t_{11}) characterize the fact that y=3 executes after x=1. c_{11}, (t_{1b}, c_{11}) and (c_{11}, t_{11}) characterize the fact that y=3 depends on the entry of $thr1$.

The state space depicted in Figure 1(f) is represented by the reachability graph of the PDNet. The graph's nodes are labeled rectangles that correspond to place names, signifying markings. The edges are arrows that denote the fired transitions of the PDNet, with transition labels excluded for clarity. The presence of tokens in places represents the markings, such as the marking M_7 where tokens are in the places c_{1e}, c_{2e}, v_0, v_1, and v_2. Due to the firing of t'_{20}, (M_6, M_7) takes place. The superscripts assigned to places v_0, v_1, and v_2 indicate the corresponding variable values in this marking. For example, the symbol v_0^1 in M_7 represents that the value of x is 1.

Moreover, superfluous computation cost may arise from dependencies on data flow pertaining to irrelevant variables. As illustrated in Figure 1(c), the PDG captures dependencies on data flow associated with y, which are not encompassed in any slice. Conversely, dependencies on data flow can

Journal Pre-proof

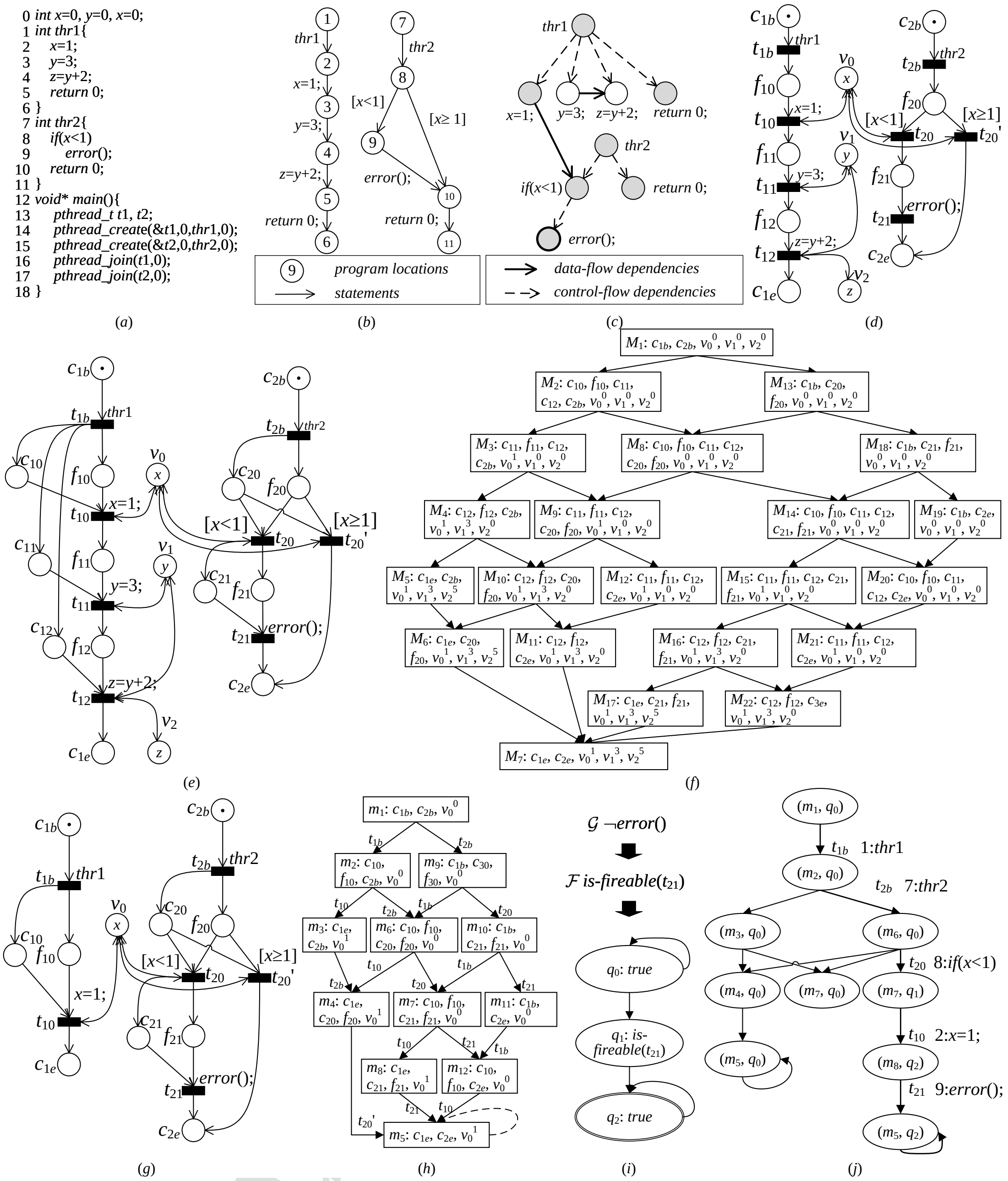


Figure 1: Motivating example (a) A concurrent program with an error location (b) The CFA of this program (c) The PDG of this program (d) A CPN converted from this program (e) A PDNet converted from this program (f) The state-space of the PDNet (g) The PDNet slice for $\mathcal{G}\ \neg error()$ (h) The state space of the PDNet slice (i) The Büchi automaton for the negation of $\mathcal{G}\ \neg error()$ (j) The explored product automaton of PDNet slice for $\mathcal{G}\ \neg error()$

only be noted when the relevant variable is verified to be in the slice. To avoid such expense, we propose a novel PDNet slicing method for capturing dependencies on data flow solely when required. In Figure 1(g), the PDNet slice effectively captures the dependencies on data flow of x, leading to a reduction in several variables such as c_{11}, f_{11}, t_{11}, c_{12}, f_{12}, t_{12}, v_1, and v_2 as compared to traditional PN slicing methods [26]. Consequently, $y = 3$ and $z = y + 2$ are sliced away. The state space of the PDNet slice, which is represented by the marking graph, is depicted in Figure 1(h).

Journal Pre-proof

Transitions labeled on the edge signify the fired transition. For example, $m_7 \xrightarrow{t_{10}} m_8$ means t_{10} fired under m_7 and m_8 is generated. As compared to 22 states shown in Figure 1(f), there are 12 states are generated in Figure 1(h). Thus, the number of states is reduced by ten using our PDNet slicing method. Additionally, a dotted arrow is added to an arc of M_5 pointing to itself because the LTL-$_\mathcal{X}$ model checking is based on the infinite path [24].

In order to establish the safety property of the example program through the LTL-$_\mathcal{X}$ formula $\mathcal{G}\neg error()$, we begin by converting $error()$ into $is\text{-}fireable(t_{21})$ of the PDNet. When t_{21} is enabled in a marking, this proposition is considered to be true. For instance, in m_{10} of Figure 1(h), t_{21} is enabled and $is\text{-}fireable(t_{21})$ is true in m_{10}. Next, we convert the formula to its negation form $\mathcal{F}$ $is\text{-}fireable(t_{21})$, which is then translated to a Büchi automaton [12] depicted in Figure 1(i). The Büchi automaton features three states, where true pertains to nodes q_0 and q_2 that can be synchronized with any reachable markings. Given Figure 1(h), only the feasible markings that enable t_{21} can synchronize with q_1, identified as $is\text{-}fireable(t_{21})$. In the on-the-fly exploration [12], the first counterexample in Figure 1(j) assesses 10 product states only. One of these states, (m_7, q_1), synchronizes marking m_7 in Figure 1(h) and state q_1 in Figure 1(i) because t_{21} is enabled in m_7. Consequently, the example program violates the safety property $\mathcal{G}\neg error()$ in Figure 1(a). The execution of statements 1, 7, 8, 2 and 9, corresponding to the occurrence sequence t_{1b}, t_{2b}, t_{20}, t_{10} and t_{21} identified by the marking sequence $m_1, m_2, m_6, m_7, m_8, m_5, \cdots$ in Figure 1(j), serves as a counterexample.

3. Program Dependence Net (PDNet)

3.1. PDNet

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 $v\in V$, $\mathbb{O}$ is the set of constants, and $Type[o]$ is the type of constant $o\in\mathbb{O}$.

Definition 1 (PDNet)**.** *PDNet is defined as 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. $P = P_c \cup P_v \cup P_f$ is a finite set of places. 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. $C: P\rightarrow\Sigma$ is a color set function that assigns a color set $C(p)$ belonging to the set of types Σ to each place p.

7. $G: T\rightarrow\mathbb{E}_V$ 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. $E: F\rightarrow\mathbb{E}_V$ 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 to arc f.

9. $I: P\rightarrow\mathbb{E}_\emptyset$ 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 differs from CPNs in P and F. Control places P_c with their adjacent control arcs F_c are used to model dominant relationships of dependencies on control flow, variable places P_v with their adjacent read-write arcs F_{rw} are used to model variables and their read/write relationships, and execution places P_f with their adjacent execution arcs F_f are used to model execution relationships in control-flow structures in concurrent programs. Other definitions and constraints of PDNet are consistent with CPN, as shown in Appendix A.

As the example in Figure 1(g), P_v=$\{v_0\}$ where v_0 is a variable place corresponding to variable x, P_c=$\{c_{1b}, c_{2b}, c_{1e}, c_{2e}, c_{10}, c_{20}, c_{21}\}$, P_f=$\{c_{1b}, c_{2b}, c_{1e}, c_{2e}, f_{10}, f_{20}, f_{21}\}$, and t_{21} corresponds to the statement $error()$. For a node $x \in P\cup T$, its preset ${}^\bullet x = \{y|(y,x)\in F\}$ and its postset $x^\bullet = \{y|(x,y)\in F\}$ are two subsets of $P\cup T$. For instance, ${}^\bullet t_{21} = \{f_{21}, c_{21}\}$, $t_{21}{}^\bullet = \{c_{2e}\}$, and ${}^\bullet v_0 = v_0{}^\bullet$=$\{t_{10}, t_{20}, t'_{20}\}$ in Figure 1(e).

Definition 2. *Let N be a PDNet.*

1. $M: P \rightarrow \mathbb{E}_\emptyset$ is a marking function that assigns an expression $M(p)$ to each place p. $\forall p \in P: Type[M(p)] = C(p)_{MS} \wedge (Var(M(p)) = \emptyset)$. M_0 represents the initial marking, i.e., $\forall p \in P: M_0(p) = I(p)$.

2. $Var(t) \subseteq V$ is the variable set of transition t. It consists of the variables appearing in expression $G(t)$ and in arc expressions of all arcs connected to t.

3. $B: V \rightarrow \mathbb{O}$ is a binding function that assigns a constant value $B(v)$ to variable v. $B[t]$ presents the set of all bindings for transition t, that maps $v\in Var(t)$ to a constant value, and $b\in B[t]$ is a binding of t.

4. A binding element (t, b) is a pair where $t\in T$ and $b\in B[t]$. $\mathbb{T}(t)$ is a set of all binding elements of t.

For convenience, a marking of N is denoted by M or M with a subscript. Then, guard and arc expressions are evaluated as follows. Formally, $e\langle b\rangle$ represents the evaluation result of expression e in binding b by assigning a constant from b to variable $v\in Var(e)$. Thus, under a binding element $(t, b)\in\mathbb{T}(t)$, $G(t)\langle b\rangle$ (or $E(f)\langle b\rangle$) represents the evaluation result of $G(t)$ (or $E(f)$), where f is an arc connected to t. For instance, $[x{<}1]\langle\{b(x)=1\}\rangle$=false for guard expression $G(t_{20})$=$[x{<}1]$ in Figure 1(e).

Definition 3 (Enabling and Occurrence Rules of PDNet)**.** *Let N be a PDNet, (t, b) a binding element, and M a marking. A binding element (t, b) is enabled under M, denoted by $M[(t, b)\rangle$, if*

Journal Pre-proof

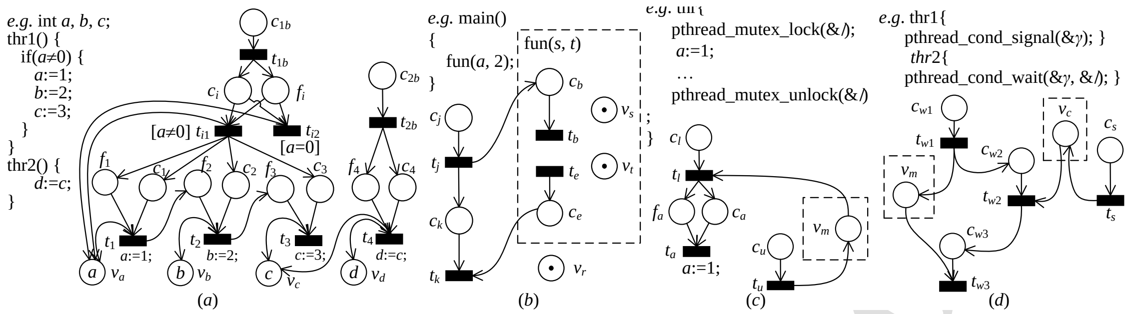


Figure 2: Example for Control-flow Structure (a) Example for actions asi, jum, ret, tcd and fcd (b) Example for actions $call$ and $rets$ (c) Example for acq and rel (d) Example for actions sig, wa_1, wa_2 and wa_3

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

When (t, b) is enabled under M, it can occur and lead to a new marking M_1 of N, denoted by $M[(t, b)\rangle M_1$, such that $\forall p \in P:\ M_1(p) = M(p) - E(p,t)\langle b\rangle + E(t,p)\langle b\rangle$.

For ease of expression, if binding $b{\in}B[t]$ enables binding element $(t, b){\in}\mathbb{T}(t)$ under a marking, we call t enabled and can occur or this marking can fire t. For instance, under marking m_7 in Figure 1(f), t_{21} is enabled because $G(t_{21})$=true, f_{21} and c_{21} belonging to ${}^{\bullet}t_{21}$ are marked in m_7, and $E(f_{21}, t_{21}){\leq}M(f_{21})$ and $E(c_{21}, t_{21}){\leq}M(c_{21})$ are satisfied. Given $b_{21}{\in}B[t_{21}]$, $m_7[(t_{21}, b_{21})\rangle m_{12}$ where $c_{2e}{\in}t_{21}{}^{\bullet}$ is marked in m_{12}. That is, m_7 can fire t_{21}.

Definition 4 (Occurrence Sequence of PDNet)**.** *Let N be a PDNet, M_0 be the initial marking, and (t, b) be a binding element. An occurrence sequence ω of N is defined as the following inductive scheme: 1) $M_0[\varepsilon\rangle M_0$($\varepsilon$ is an empty sequence), and 2) $M_0[\omega\rangle M_1{\wedge}M_1[(t, b)\rangle M_2:\ M_0[\omega(t, b)\rangle M_2$. An occurrence sequence ω of N is maximal, if 1) ω is of infinite length (e.g., (t_1, b_1), (t_2, b_2), $\cdots$, (t_n, b_n), $\cdots$), or 2) $M_0[\omega\rangle M_1{\wedge}\forall t{\in}T,\ \nexists(t, b){\in}\mathbb{T}(t):\ M_1[(t, b)\rangle$.*

For example, a finite marking sequence w.r.t. ω, denoted by $M[\omega]$ or M_1, M_2, $\cdots$, and M_n, is generated by occurrence of all binding elements in ω. For ease of expression, ω is represented by $\langle t_1, t_2, \cdots, t_n\rangle$. For instance, $\langle t_{1b}, t_{2b}, t_{20}, t_{10}, t_{21}\rangle$ is a maximal occurrence sequence of the PDNet in Figure 1(e). And $\langle m_1, m_2, m_6, m_7, m_8, m_5\rangle$ is a marking sequence by firing t_{1b}, t_{2b}, t_{20}, t_{10} and t_{21} one after another.

3.2. PDNet Transitions for Concurrent Program

We explain the syntax and semantics of concurrent programs and present their operations, as outlined in Appendix B. Based on the operational semantics, we define action asi for assignment operation, jum and ret for jump operation, tcd and fcd for branch conditional operation, $call$ for call site operation, $rets$ for return site operation, acq (rel) for lock (unlock) operation, sig for signal operation and wa_1, wa_2 and wa_3 for wait operation. Thus, each of 13 actions corresponds to a specific PDNet transition, as shown in Table 1, when modeling the control-flow structure of concurrent programs using PDNet.

Table 1
PDNet Transition

Action	Transition	Operation
asi	$assign$ transition	$v := w$
jum ret	$jump$ transition $exit$ transition	$jump$
tcd fcd	$branch$ transition	if(w)then(τ_1 $*$)else(τ_2 $*$) while(w)do(τ $*$)
$call$ $rets$	$call$ transition $return$ transition	$calls$
acq	$lock$ transition	$\langle lock, \ell\rangle$
rel	$unlock$ transition	$\langle unlock, \ell\rangle$
sig	$signal$ transition	$\langle signal, \gamma\rangle$
wa_1 wa_2 wa_3	$wait$ Transition	$\langle wait, \gamma, \ell\rangle$

For instance, t_1 for a:=1, t_2 for b:=2, t_3 for c:=3 and t_4 for d:=c in Figure 2(a) are $assign$ transitions, and t_{i1} and t_{i2} are $branch$ transitions for if($a{\neq}0$). In Figure 2(b), t_j is $call$ transition, and t_k is $return$ transition. In Figure 2(c), t_l is $lock$ transition for pthread_mutex_lock(&ℓ), and t_u is $unlock$ transition for pthread_mutex_unlock(&ℓ). In Figure 2(d), t_s is $signal$ transition for pthread_cond_signal(&γ), t_{w1} for action wa_1, t_{w2} for action wa_2 and t_{w3} for action are $wait$ transitions from pthread_cond_wait(&γ, &ℓ).

Specially, t_b is called $enter$ transition of the function $fun(s, t)$, t_e is called $exit$ transition of $fun(s, t)$, (t_j, c_b) is called $enter$ arc, and (c_e, t_k) is called $exit$ arc in Figure 2(b).

3.3. Dependency Modeling Based on PDNet

Dependencies on control flow among statements were characterized as domination, as noted in Masud et al. [33]. However, the definition of dependencies on control flow differs among such graphs as those in Qi et al. [36]. In this paper, we classify dependencies on control flow into four types: control, call, lock, and prior-occurrence ones. For a

Journal Pre-proof

complete and cohesive representation of them, we utilize PDNet in a comprehensive way.

As mentioned above, $branch$ transition is modeled for actions tcd or fcd, $lock$ ($unlock$) transition is modeled for acq (rel), and $signal$ ($wait$) transition is modeled for sig (wa_1, wa_2, wa_3) in Table 1. $enter$ ($exit$) transition is constructed for a function like t_b (t_e) in Figure 2(b). Next, we define the execution path, control scope and critical region of PDNet.

Definition 5 (Execution Path of PDNet)**.** *Let N be a PDNet, and t_m and t_n be the two different transitions of N. A sequence π along the transitions and places of N is denoted by $(t_1, p_1, t_2, p_2, \ldots, t_{k-1}, p_{k-1}, t_k)$, where (t_i, p_i) or (p_i, t_{i+1}) ($1 \leq i < k$) is an arc of N. π is an execution path from t_m to t_n if t_1 is t_m, t_k is t_n, and $k \geq 1$.*

All execution paths from t_m to t_n constitute the execution path set, denoted by $\mathbb{P}(t_m, t_n)$. The transition t_n is connected from t_m, denoted by $\mathbb{R}(t_m, t_n)$, if $\mathbb{P}(t_m, t_n) \neq \emptyset$. Particularly, t_n is connected from t_m without requiring arc (t_i, p_i), denoted by $\mathbb{R}(t_m, t_n)_{\overline{t_i p_i}}$, if $\forall \pi \in \mathbb{P}(t_m, t_n)$, $(t_i, p_i) \notin \pi$.

Definition 6 (Control Scope of PDNet)**.** *Let N be a PDNet, t_m be a $branch$ or $enter$ transition of N, and t_n be a transition of N. t_n is in the control scope of t_m, denoted by $\mathbb{S}(t_m, t_n)$, if t_n is reachable from t_m, and there exists an execution path starting in t_m such that it does not contain t_n.*

Definition 7 (Critical Region of PDNet)**.** *Let N be a PDNet, t_m be a $lock$ transition of N that can acquire a mutex ℓ, and t_n be a transition of N. t_n is in the critical region of t_m, denoted by $\mathbb{C}(t_m, t_n)$, if t_n is reachable from t_m, and there exists no $unlock$ transition that releases the same mutex ℓ in any execution path in $\mathbb{P}(t_m, t_n)$.*

In Figure 2(a), $\langle t_{i1}, f_1, t_1, f_2, t_2, f_3, t_3 \rangle$) is an execution path. t_3 is reachable from t_{i1}, i.e., $\mathbb{R}(t_{i1}, t_3)$. t_1, t_2 and t_3 are in the control scope of t_{i1}, i.e., $\mathbb{S}(t_{i1}, t_1)$, $\mathbb{S}(t_{i1}, t_2)$ and $\mathbb{S}(t_{i1}, t_3)$. In Figure 2(c), t_a is in the critical region of t_l, i.e., $\mathbb{C}(t_l, t_a)$. Next, we define four kinds of dependencies on control flow.

Definition 8 (Dependencies on control flow of PDNet)**.** *For concurrent program $\mathcal{P}$, N is the PDNet of $\mathcal{P}$, t_m and t_n are two transitions of N:*

1. t_n is control-dependent on t_m, denoted by $t_m \xrightarrow{co} t_n$, if 1) t_m is a $branch$ transition or $enter$ transition, 2) $\mathbb{S}(t_m, t_n)$, and 3) there exists no other $branch$ transition t_o in the control scope of t_m such that $\mathbb{S}(t_m, t_o)$.

2. t_n is call-dependent on t_m, denoted by $t_m \xrightarrow{ca} t_n$, if t_n is an $enter$ transition of the called function, and t_m is the $call$ transition of the calling function, making the execution flow turn to t_n, or t_m is an exit transition of a called function, and t_n is the return transition of a calling function, making the execution flow turn back to t_n.

3. t_n is lock-dependent on t_m, denoted by $t_m \xrightarrow{lo} t_n$, if t_m is a $lock$ transition acquiring ℓ, and $\mathbb{C}(t_m, t_n)$, or t_m and t_n are all $lock$ transitions that acquire ℓ.

4. t_n is prior-occurrence-dependent on t_m, denoted by $t_m \xrightarrow{po} t_n$, if 1) t_n is a wait transition waiting for a condition variable γ, and 2) t_m is a $signal$ transition notifying on the same condition variable γ.

Intuitively, we define the control dependence for the nearest $branch$ or $enter$ transition of PDNet. A $branch$ or $enter$ transition can dominate the execution of the following transitions that are not in the control scope of other transitions. As mentioned above, the PDNet control-flow structure provides the control place interfaces (e.g., c_1, c_2 and c_3 in Figure 2(a)) to describe the control dependencies. The control arcs between a $branch$ or $enter$ transition and control places are constructed for the control dependencies.

In Figure 2(a), a, b and c are global variables initialized to 1 in this program. The PDNet structures of the branching operation if($a \neq 0$) and assignment operations $a := 1$, $b := 2$ and $c := 3$ are constructed by modeling these operations. Thus, the control arcs (t_{i1}, c_1), (t_{i1}, c_2) and (t_{i1}, c_3) are constructed to describe the control dependencies based on the Definition 8. That is, the control dependencies $t_{i1} \xrightarrow{co} t_1$, $t_{i1} \xrightarrow{co} t_2$ and $t_{i1} \xrightarrow{co} t_3$ are represented explicitly.

Other dependencies on control flow have been described by modeling function call operations and POSIX thread operations. For example, the call dependencies $t_j \xrightarrow{ca} t_b$ and $t_e \xrightarrow{ca} t_k$ are represented in Figure 2(b), the lock dependence $t_l \xrightarrow{lo} t_a$ is represented in Figure 2(c), and the prior-occurrence dependence $t_{w2} \xrightarrow{po} t_s$ is represented in Figure 2(d).

Dependencies on data flow describe the reachable definition-use relation of the variables. We classify these dependencies into data [15] and interference ones [36]. Next, we define a reference and definition set of PDNets.

Definition 9 (Reference Set and Definition Set of PDNet)**.** *Let N be a PDNet and t be a transition of N. The reference set of t $Ref(t) ::= \{p \mid \forall p \in {}^{\bullet}t \cap P_v : E(p,t) = E(t,p)\}$. The definition set of t $Def(t) ::= \{p \mid \forall p \in {}^{\bullet}t \cap P_v : E(p,t) \neq E(t,p)\}$.*

Definition 10 (Dependencies on data flow of PDNet)**.** *For concurrent program $\mathcal{P}$, N is the PDNet of $\mathcal{P}$, t_m and t_n are two transitions of N, if there is a variable place v, such that*

1. t_m is data-dependent on t_n, denoted by $t_n \xrightarrow{D} t_m$, if 1) $\mathbb{R}(t_n, t_m)$, and 2) $v \in Ref(t_m) \land v \in Def(t_n)$, 3). $\exists \pi \in \mathbb{P}(t_n, t_m)$, there exists no other transition $t_a \in \pi$ such that $v \in Def(t_a)$.

2. t_m is interference-dependent on t_n, denoted by $t_n \xrightarrow{I} t_m$, if 1) there exist execution places $f_m \in ({}^{\bullet}t_m \cap P_f)$ and $f_n \in ({}^{\bullet}t_n \cap P_f)$ such that $M(f_m) \neq M(f_n)$, and 2) $v \in Ref(t_m) \land v \in Def(t_n)$.

In Definition 10's 2.1), $M(f_m) \neq M(f_n)$ means that the operations corresponding to t_m and t_n belong to different concurrently executing threads. For instance, $t_{11} \xrightarrow{D} t_{12}$ is represented in Figure 1(e). $t_{10} \xrightarrow{I} t_{20}$ is represented in Figure 1(e), and $t_3 \xrightarrow{I} t_4$ in Figure 2(a).

Journal Pre-proof

Through the use of read-write arcs and control-flow structures in PDNet, we can capture these dependencies on data flow when needed without adding additional arcs. When slicing, the read-write arcs in PDNet are used to capture these dependencies on demand. It is worth noting that local variables exhibit only data dependencies between t_m and t_n, whereas t_m may be data or interference dependent on t_n if v represents a global variable. This approach can help reduce computation cost.

4. On-demand PDNet Slicing for Reduction

4.1. Linear Temporal Logic of PDNet

The LTL formalism, elucidated in Ding et al. [12], serves as a specification for delineating linear temporal properties, encompassing the safety and liveness properties of Petri nets. Nevertheless, if slicing methods are employed to condense the state space, the resultant model may not encompass the entire sequence of the original model. Consequently, our methods are not aligned with operator $\mathcal{X}$ and can corroborate LTL-$_\mathcal{X}$ formulae.

Definition 11 (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 an LTL-$_\mathcal{X}$ formula. The syntax of propositions is defined as: $a ::= true|false|is\text{-}fireable(t)$ $|token\text{-}value(p) \star c$. Here, $t{\in}T, p{\in}P_v, c{\in}C(p)_{MS}$ is a constant, $\star{\in}\{<, \leq, >, \geq, =\}$.*

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

$$is\text{-}fireable(t) = \begin{cases} true & if\ \exists b:\ M[(t,b)\rangle, \\ false & otherwise. \end{cases} \tag{1}$$

$$token\text{-}value(p) \star c = \begin{cases} true & if\ M(p) \star c, \\ false & otherwise. \end{cases} \tag{2}$$

The syntax of LTL-$_\mathcal{X}$ over $\mathbb{A}$ is defined as: $\psi ::= a$ $|\neg\psi|\psi_1 \wedge \psi_2|\psi_1 \vee \psi_2|\psi_1 \Rightarrow \psi_2|\mathcal{F}\psi|\mathcal{G}\psi|\psi_1\mathcal{U}\psi_2$. Here, $\neg$, $\wedge$, $\vee$ 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 condensed explanation of LTL-$_\mathcal{X}$ semantics under a marking sequence is similar to that of Petri nets as explained by Wolf [46]. For example, if $\mathcal{G}\ is\text{-}fireable(t) \Rightarrow \mathcal{F}$ $token\text{-}value(p)$=0, it means that whenever t is enabled, the token of p is zero in some subsequent states. Additionally, the Büchi automaton can be used to encode an LTL-$_\mathcal{X}$ formula for explicit-state model checking, as described in Ding et al. [12] and illustrated in Figure 1(i) [24].

For explicit-state model checking, the traditional approach has been automaton-theoretic. This involves exhaustively exploring all possible transition firings of a transition system (state space). For LTL, the problem is translated into an emptiness-checking problem. PDNet's analysis can also adopt the automaton-theoretic approach. Specifically, the marking of PDNet can synchronize with the states of Büchi automaton [12], which are known as Büchi states. To start, the initial product state is generated by the initial marking and the initial Büchi state. Then, an acceptable path starting from the initial product state is extended until reaching an acceptable product state [12]. Thus, $N{\vDash}\psi$ holds if no acceptable sequence is reachable from the initial product state. For instance, there exists an explored counterexample in Figure 1(j), meaning that $N{\vDash}\psi$ does not hold.

4.2. Slicing Criterion of PDNet

The concept behind PDNet slicing is to eliminate unnecessary parts that are not relevant to the verified property. This reduces both model size and reachable state space. The slicing criterion is determined by the program features specified in the verified LTL-$_\mathcal{X}$ property. As a result, we extract the relevant slicing criterion for an LTL-$_\mathcal{X}$ formula.

Definition 12 (Slicing Criterion)**.** *Let N be a PDNet, ψ a LTL-$_\mathcal{X}$ formula with propositions in Definition 11, $\mathbb{A}$ the proposition set from ψ, and $Crit$ the slicing criterion w.r.t. ψ. If a is in the form of $is\text{-}fireable(t)$, $Crit(a) ::= \{p|p{\in}{}^{\bullet}t \backslash P_f\}$. If a is in the form of $token\text{-}value(p_t)\ r\ c$, $Crit(a) ::= \{p|\forall t{\in}{}^{\bullet}p_t :\ E(t,p_t){\neq}E(p_t,t){\wedge}p{\in}{}^{\bullet}t \backslash P_f\}$. The slicing criterion w.r.t. ψ is $Crit ::= \{p|\forall a{\in}\mathbb{A} :\ p{\in}Crit(a)\}$.*

Intuitively, each proposition in the LTL-$_\mathcal{X}$ formula ψ should extract the places from PDNet N to constitute a slicing criterion w.r.t. ψ. For every proposition a in $\mathbb{A}$, its corresponding places are extracted based on Definition 12. If a is in the form of $is\text{-}fireable(t)$, the input places of t except execution places (i.e., P_f in Definition 1) are extracted. If a is in the form of $token\text{-}value(p)\ r\ c$ where p is a variable place, the transitions in ${}^{\bullet}p$ that can update the token of p are found by $E(t,p){\neq}E(p,t)$. Then, the places except execution places in ${}^{\bullet}({}^{\bullet}p)$ are extracted. For example, there is a proposition $is\text{-}fireable(t_{21})$ in $\mathcal{F}\ is\text{-}fireable(t_{21})$ of the PDNet in Figure 1(e). Due to ${}^{\bullet}t_{21}{=}\{f_{21}, c_{21}\}$, c_{21} is extracted to $Crit$.

4.3. On-demand PDNet Slicing Algorithm

As mentioned above, traditional program slicing methods should capture complete dependencies on data flow in advance when constructing PDG. Differently, we propose a PDNet slicing method to capture dependencies on data flow in Definition 10 in an on-demand way. Let N be the PDNet of concurrent program $\mathcal{P}$, and $Crit$ be a slicing criterion of N. We use $\xrightarrow{d}$ to represent the union of all dependencies of PDNet. We define the PDNet slice next.

Definition 13 (PDNet Slice)**.** *Let N be a PDNet, ψ be a LTL-$_\mathcal{X}$ formula, $Crit$ be the slicing criterion w.r.t. ψ, and N' be the PDNet slice of N w.r.t. $Crit$. $N' ::= \{x \in P{\cup}T|\forall p \in Crit :\ x \xrightarrow{d}{}^{*} p\}$.*

In this case, symbol $*$ stands for the potential transitive relationships of $\xrightarrow{d}$, while $\xrightarrow{d}{}^{*}$ denotes the transitive closure of the dependencies of PDNet. This means that regardless of the dependencies for adding a place, the transitive closure is calculated based on the dependencies of $\xrightarrow{d}$.

Journal Pre-proof

When control place p is in $Crit$, we add the control places of transitions that affect p through the constructed arcs that represent dependencies on control flow to N'.

As the example in Figure 1(e), $\{c_{21}\}$ is the slicing criterion for $\mathcal{F}\ is\text{-}fireable(t_{21})$. Due to $t_{2b}\xrightarrow{co}t_{20}$ and $t_{20}\xrightarrow{co}t_{21}$ according to Definition 8, $t_{2b}\xrightarrow{co}{}^{*}t_{21}$ (i.e., t_{2b} affects t_{21} indirectly), and the control place c_{2b} of t_{2b} should be added to P' of N'. Differently, when a variable place is added to P' of N', its dependencies on data flow are calculated through the control-flow structure and read-write arcs according to Definition 10. In Figure 1(e), when t_{20} is added to P', v_0 can be added to P'. Then, $t_{10}\xrightarrow{I}t_{20}$ concerning the variable place v_0 is captured, and the control place c_{10} of t_{10} is added to P' of N'.

Based on the above insights, our on-demand PDNet slicing is realized via Algorithm 1.

Algorithm 1 On-demand PDNet Slicing Algorithm

Input: A PDNet N and the slicing criterion $Crit$ w.r.t. ψ;
Output: The PDNet Slice N' with P' and T';

1: $P' := Crit$; /* P' is the place set of N' */
2: $T' := \emptyset$; /* T' is the transition set of N' */
3: $P_d := \emptyset$; /* P_d is a processed place set */
4: PROPA($enter$); PROPA($exit$);
5: **function** PROPA(Dir)
6: **for all** $p\in(P'\cap P_c\setminus P_d)$ **do**
7: **for all** $t\in({}^{\bullet}p\setminus T')\wedge(t,p)\in F_c$ **do**
8: **for all** $p'\in({}^{\bullet}t\setminus P')\wedge(p',t)\in F_c$ **do**
9: $P' := P'\cup\{p'\}$;
10: **for all** $t\in p^{\bullet}\wedge(p,t)\in F_c$ **do**
11: $T' := T'\cup\{t\}$;
12: $P' := P'\cup({}^{\bullet}t\cap P_f)$;
13: **for all** $p'\in({}^{\bullet}t\setminus P'\cap P_v)\wedge E(t,p')=E(p',t)$ **do**
14: **if** ISGLOBAL(p') **then**
15: $P' := P'\cup\{p'\}$;
16: **for all** $t'\in{}^{\bullet}p'\wedge E(t',p')\neq E(p',t')$ **do**
17: **if** PA(t',t,p')$\vee$CON(t',t,p') **then**
18: **for all** $p''\in({}^{\bullet}t'\setminus P')\cap P_c$ **do**
19: $P' := P'\cup\{p''\}$;
20: **if** ISLOCAL(p') **then**
21: $T_d :=$ FINDPRE(t,p',Dir);
22: **if** $T_d\neq\emptyset$ **then** $P' := P'\cup\{p'\}$;
23: **for all** $t'\in T_d$ **do** /* $t'\xrightarrow{D}t$ */
24: **for all** $p''\in({}^{\bullet}t'\setminus P')$ **do**
25: **if** $(p'',t')\in F_c$ **then**
26: $P' := P'\cup\{p''\}$;
27: $P_d := P_d\cup\{p\}$;

Firstly, P', T', and P_d are initialized (Lines 1-3). We refer to a place that does not belong to P_d as an unprocessed place. That is, $p\in P_d$ is called a processed place. There are two calls of PROPA(Dir) (Line 4) avoiding redundancy caused by multiple calls [22]. The difference between these two calls is the propagation direction from the call dependence in Definition 8's 2.1) and 2.2).

In function PROPA(Dir), if there exists an unprocessed control place p (Line 6), all nodes x meeting $x\xrightarrow{d}{}^{*}p$ should be captured. The key slicing steps to propagate potential transitive relationships are as follows.

The first step is to propagate dependencies on control flow from p (Lines 7-9). A transition $t\in{}^{\bullet}p$ is propagated by control arc (t,p) (Line 7). Here, the control arcs describe which statements dominate the current one corresponding to p. According to the dependencies on control flow in Definition 8, $t\xrightarrow{co}t'$ or $t\xrightarrow{ca}t'$ or $t\xrightarrow{lo}t'$ or $t\xrightarrow{po}t'$ where t' is the transition of the PDNet structure for the operation where p locates. Thus, the control place p' of t is added to P' (Lines 8-9).

The second step is to complete the PDNet structure where p is located. Its output transition $t\in p^{\bullet}$ is added to T' (Line 11) and the execution place in ${}^{\bullet}t$ is added to P' (Line 12).

The third step is to propagate dependencies on data flow from p (Lines 13-25). If there is a variable place $p'\in Ref(t)$ (Line 13), the dependencies on data flow relevant to p' are captured based on Definition 10. As the analysis mentioned above, the interference dependencies are captured only for a global variable to reduce the computation cost. Thus, if p' corresponds to a global variable (Lines 14-19), p' is added to P' (Line 15). If there exists transition t' such that $p'\in Def(t')$ (Line 16), function PA(t',t,p') judges whether $t'\xrightarrow{D}t$ according to Definition 10's 1.1) and 1.3), function CON(t',t,p') judges whether $t'\xrightarrow{I}t$ according to Definition 10's 2.1) (Line 17), and the control places of t' are added to P' (Lines 18-19). If p' corresponds to a local variable (Lines 20-25), function FINDPRE(t,p',Dir) finds the transition set $T_d ::= \{t'|p'\in Def(t')\wedge\mathbb{R}(t',t)_{\bar{Dir}}\wedge(\forall t''\in\mathbb{P}(t',t): p'\notin Def(t''))\}$ according to Definition 10's 1.1) and 1.3) (Line 21). Here, parameter Dir is $enter$ arc (e.g., (t_j,c_b) in Figure 2(b)) or $exit$ arc ((c_e,t_k) in Figure 2(b)). The control places of t' are added to P' (Lines 23-25).

The final step is that control place p is added to P_d marked as processed (Line 26).

Moreover, N' may be non-executable due to the incomplete execution orders concerning the control-flow structure. Hence, the slicing post-process algorithm is outlined in Appendix C. The complexity of Algorithm 1 is, in the best case, $O(n)$, where n is the statement count meaning the number of statements. Its worst-case complexity is $O(n^4)$ when the algorithm cannot slice any statement. The correctness is proved in Appendix D.

In Figure 3, we set $\mathcal{G}\,token\text{-}value(v_c)\leq 0$ as the example property. a:=1 and b:=2 are sliced for this property in Figure 3(a). The PDNet of the program is in Figure 3(b).

Firstly, we extract $Crit$ through Definition 12. There is only a proposition $token\text{-}value(v_c)\leq 0$, where variable place v_c corresponds to a variable c. Due to $E(t_3,v_c)\neq E(v_c,t_3)$, $Crit={}^{\bullet}t_3\setminus P_f=\{v_3,c_3\}$ is extracted and filled with dark gray.

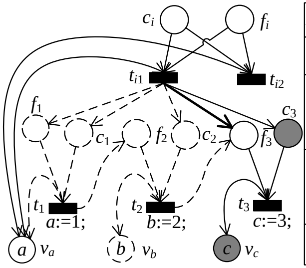
Then, we update P' and T' via Algorithm 1. In the table of Figure 3(c), P' and T' are updated in each iteration. Initially, $P'=\{v_c,c_3\}$, $T'=\emptyset$, and $P_d=\emptyset$ by Algorithm 1.

- In the first iteration, c_3 is selected as the control place by Algorithm 1. Control place c_i is added to

Journal Pre-proof

e.g. example program:
int a, b, c;
main() {
  if ($a \neq 0$) {
    ~~a:=1;~~ //sliced
    ~~b:=2;~~ //sliced
    c:=3;}
}
e.g. Original formula of example program: $\mathcal{G}\ c \leq 0$
Translated formula of PDNet: $\mathcal{G}\, \textit{token-value}(v_c) \leq 0$

(a)



(b)

Iteration		***P'***		***T'***		$\boldsymbol{P_d}$
0	/	*Crit*=$\{v_c, c_3\}$	Line 1	$\varnothing$	Line 2	$\varnothing$
1	$\boldsymbol{c_3}$	$\{v_c, c_3, \boldsymbol{c_i}\}$	Line 9	$\{\boldsymbol{t_3}\}$	Line 11	$\{c_3\}$
		$\{v_c, c_3, c_i, \boldsymbol{f_3}\}$	Line 12			
2	$\boldsymbol{c_i}$	$\{v_c, c_3, c_i, f_3, \boldsymbol{f_i}\}$	Line 12	$\{t_3, \boldsymbol{t_{i1}}, \boldsymbol{t_{i2}}\}$	Line 11	$\{c_3, c_i\}$
		$\{v_c, c_3, c_i, f_3, f_i, \boldsymbol{v_a}\}$	Line 15			
3	/	$\{v_c, c_3, c_i, f_3, f_i, v_a\}$	/	$\{t_3, t_{i1}, t_{i2}\}$	/	$\{c_3, c_i\}$

(c)

Figure 3: Example for slicing process (a) The property and the example program (b) The marked PDNet slice (c) The slicing process by Algorithm 1

P' according to control dependence in Definition 8. Transition t_3 is added to T', and f_3 is added to P' by Algorithm 1. c_3 is marked in P_d as a processed place by Algorithm 1.

- In the second iteration, c_i is selected in $P' \cap P_c \setminus P_d$. t_{i1} (t_{i2} with the same process) is added to T' and execution place f_i is added to P' by Algorithm 1. For dependencies on data flow in Definition 10, variable place v_a is selected due to $E(t_{i1}, v_a){=}E(v_a, t_{i1})$. a corresponding to v_a is a global variable, and v_a is added to P' by Algorithm 1. Then, transition t_a is selected due to $E(t_a, v_a){\neq}E(v_a, t_a)$. But there exists no execution path by function PA in Algorithm 1, and no control place is added to P'. c_i is marked in P_d as a processed place.
- Finally, no control place is selected via P' and P_d.

The dotted places and dotted arcs, as well as t_1 and t_2, are removed in Figure 3(b). However, there is no transition t in ${}^{\bullet}f_3$ such that (t, f_3) is an execution arc. Thus, a new execution arc (t_{i1}, f_3) expressed as a bold arrow is constructed based on the post-process.

5. Experimental Evaluation

5.1. PDNet v.s. its Peers

PDNet as a unified model and its on-demand slicing method are the core works in this paper. We compare the performance of our method, called *PDNet*, with slicing concurrent programs for model checking, called *PS*, and slicing Petri nets, called *TraPNSlice*.

PS first builds PDG based on control dependencies using post-dominance and data dependencies the transitive closure of use-def chains [17] to reduce the concurrent program itself. Then, the program slice is transformed to a traditional CPN model [23] as Definition A.2, and the properties are verified by the same model checking algorithm as *PDNet*.

TraPNSlice first builds a traditional CPN model as Definition A.2 for the concurrent program and then slices this model for reduction. Also, such a reduced model is verified by the same model checking algorithm as *PDNet*.

Thus, we evaluate the efficiency and effectiveness of our method by answering the following questions:

$Q1$ Are the unified model and on-demand slicing method based on PDNet more efficient than its peers?

$Q2$ Is our verification method based on our PDNet slice more effective than its peers?

PDNet model differs from the input form received by existing LTL checkers. And the state generated by the PDNet contains rich information about the distribution of places. The state of existing LTL checkers is difficult to fully cover this information from PDNet. At the same time, the LTL formulation form of PDNet is different from the form accepted by the existing LTL checkers. Therefore, we implemented our *PDNet* methods and its peers (*PS* and *TraPNSlice*) on the same tool, called *DAMER* (Dependence Analyser and Multi-threaded programs checkER) to evaluate our method.

5.2. Tool Implementation and Benchmarks

DAMER supports the features of C programs defined by Definition B.1 and offers support for various linear temporal properties. The input programs in *DAMER* are automatically translated into a PDNet, and our on-demand PDNet slicing method is used to reduce it. The resulting PDNet slice can then be checked by using explicit-state model checking [21]. It is also convenient for the subsequent design of new orthogonal reduction techniques. The input properties are expressed as LTL-$\mathcal{X}$ formulae, as defined in Definition 11. Once modeled, the relevant program variables or locations are automatically translated into PDNet propositions within *DAMER*.

To showcase the practical effectiveness of our methods in verifying LTL, we evaluated a set of multi-threaded C programs, randomly chosen from the Software Verification Competition [42] and benchmarks in Li et al. [28]. As outlined in Table 2, it involves five concurrent programs that do not use POSIX thread functions. *Fib* is a mathematical algorithm that divides tasks into multiple threads and combines the constraints of each computation. textitLamport [3], *Dekker* [4], *Szymanski* [17], and *Peterson* [39] four algorithms we use to solve mutual exclusion problems in concurrent systems. Additionally, it involves five concurrent

Journal Pre-proof

Table 2
Basic demographics of Bench. and For. Columns are: LoC: nr. of Lines; tC: nr. of threads; ψ: nr. of formulae; Property: concrete form; Res: Known results of each formula

Bench.			For.		
Program	LoC	tC	ψ	Property	Res
Fib	61	2	ψ_1	$\mathcal{G}\neg error()$	F
			ψ_2	$\mathcal{G}(k < 6)$	F
Lamport	81	2	ψ_1	$\mathcal{G}\neg error()$	T
			ψ_2	$\mathcal{G}(x = 0 \vee x = 1)$	F
Dekker	58	2	ψ_1	$\mathcal{G}\neg error()$	T
			ψ_2	$\mathcal{G}(flag1 = 0 \vee flag1 = 1)$	T
Szymanski	63	2	ψ_1	$\mathcal{G}\neg reach_error()$	T
			ψ_2	$\mathcal{G}(flag1 = 0 \vee flag1 = 1)$	F
Peterson	46	2	ψ_1	$\mathcal{G}\neg reach_error()$	T
			ψ_2	$\mathcal{G}(flag1 = 0 \vee flag1 = 1)$	T
Sync	70	2	ψ_1	$\mathcal{G}\neg error()$	T
			ψ_2	$\mathcal{G}(num = 0 \vee num = 1)$	T
Datarace	79	2	ψ_1	$\mathcal{G}\neg error()$	T
			ψ_2	$\mathcal{G}(flag1 = 0 \vee flag1 = 1)$	F
Rwlock	75	2	ψ_1	$\mathcal{G}\neg error()$	T
			ψ_2	$\mathcal{G}(w = 0 \vee w = 1)$	T
Varmutex	60	2	ψ_1	$\mathcal{G}\neg error()$	T
			ψ_2	$\mathcal{G}(block = 0 \vee block = 1)$	T
Lazy	55	3	ψ_1	$\mathcal{G}\neg error()$	F
			ψ_2	$\mathcal{G}(data = 0 \vee data = 1)$	F
Ccnf	37	9	ψ_1	$\mathcal{G}\neg error()$	F
			ψ_2	$\mathcal{G}(d = 0)$	F

programs that use POSIX thread functions. *Sync* [42] is an example program that implements thread synchronization through a condition variable. The remaining four programs, i.e., *Datarace*, *Varmutex*, *Rwlock*, and *Lazy* [42], all access shared memory protected by a single mutex lock. Specifically, *Ccnf* is an artificial example that needs to compute all variables' dependencies on data flow.

For each program, we set two formulae as ψ_1 and ψ_2 in Tables 2. ψ_1 specifies their safety properties expressed by a specific *error* location, and ψ_2 specifies constraint properties expressed by the constraints relevant to the key variables. The verification of these two categories of properties is of interest and of significance to the concurrent programs. Since there are currently no online available complex formulae for these concurrent programs, we construct some formulae to specify ψ_1 and ψ_2.

Our code and benchmarks are publically available [1]. The experiments are conducted with $16GB$ memory. For the used benchmarks, the variation in time among different runs is relatively small. It is sufficient to run them 10 times to use the average. The whole process of each verified property for each benchmark is called verification in the following.

5.3. Comparison

5.3.1. Slice Comparison

For question $Q1$, we report the concrete slicing time in Table 3. Their average values are on the last row. S_{com} (T_{com}) is the whole time of *PS* (*PDNet*) before model checking. Here, T_{mol} includes the time to calculate control-flow structure and dependencies on control flow, and T_{sli} includes the time to capture the dependencies on data flow as well as the transitive closure of PDNet.

As we can see from Table 3, S_{com}>T_{com} holds for each verification. The average of S_{com} is 84.760, and the average of T_{com} is 4.230. It implies that our proposed method is more efficient. We calculate S_{com}/T_{com} in Table 3, and its average is 17.306. Here, 10 verification reduces the whole time by more than 10 times, and 4 verification reduces the whole time by more than 30 times. Our method can reduce computation when converting among multiple models.

S_{dfd}+S_{clo} of *PS*, including the computation time of the dependencies on data flow and the transitive closure, corresponds to T_{sli} of *PDNet*. It is obvious that S_{dfd}+S_{clo}>T_{sli} holds for each verification of the first 10 concurrent programs. The reason is that a variable's dependencies on data flow are only calculated when it is added to the PDNet slice. This means that the use of PDNet can save on computation by avoiding the computation of irrelevant variables' dependencies. In particular, $Ccnf$ is an example containing dependencies on data flow between all statements. When the slices included all of the original PDNet, our on-demand slicing method needs to calculate dependencies on data flow for all variables. S_{dfd}+S_{clo} of $Ccnf$ is 2.182 on average, ans its T_{sli} is 2.288. Thus, the slicing time is slightly higher than that for computing dependencies on data flow in PDG using the traditional program slicing method.

Note that S_{mol}<T_{mol} holds for each verification. The average of S_{mol} is 1.855, and that of T_{mol} is 2.989. Their difference is 1.134 which is insignificant. Our approach to utilizing the unified model and on-demand slicing based on PDNet proves to be highly efficient, as it significantly reduces computation during model conversions and eliminates dependencies on data flow of irrelevant variables. Thus, the answer to question $Q1$ is positive. CPN is translated from a program slice; while PDNet is translated from the entire program. Thus, PDNet needs extra places and arcs to depict dependencies on control flow. Yet such extra ones lead to a small impact on the overall time. In summary, our approach is a much more cost-effective solution than its peers.

5.3.2. Verification Comparison

For question $Q2$, we report the verifying time and results (T represents true, while F represents false). As we can see from Table 4, S_{res}, C_{res}, and T_{res} are all correct. It implies that the three slicing methods are correct and effective in *DAMER*. We calculate S_{ver}/T_{ver} and C_{ver}/T_{ver} as shown in Table 4. Obviously, S_{ver}>T_{ver} and C_{ver}>T_{ver} holds for each verification, implying that our PDNet slicing in an on-demand way outperforms the program slicing and traditional Petri net slicing methods in terms of verification time. The average value of S_{ver}/T_{ver} is 1.249. As for the traditional program slicing method, it takes much time to calculate the domination relationship for the dependencies on control flow [36]. The average value of C_{ver}/T_{ver} is 2.151. It is evident that our PDNet slicing method can reduce the verifying

[1] https://github.com/shirleylee03/damer/

Journal Pre-proof

Table 3
Slice Comparison of *PS* and *PDNet* (in milliseconds). Columns are: S_{dfd}: the time to calculate the complete dependencies on data flow; S_{cfd}: the time to calculate the complete dependencies on control flow; S_{clo}: the time to capture the transitive closure of PDG; S_{ps}: S_{dfd}+S_{cfd}+S_{clo}; S_{mol}: the modeling time to construct a CPN model for the program slice; S_{com}: S_{ps}+S_{mol}; T_{mol}: the time to calculate the control-flow structure and dependencies on control flow; T_{sli}: the time to capture the dependencies on data flow as well as the transitive closure of PDNet; T_{com}: T_{mol}+T_{sli}

Bench.	For.	*PS*						*PDNet*			S_{com}/
		S_{dfd}	S_{cfd}	S_{clo}	S_{ps}	S_{mol}	S_{com}	T_{mol}	T_{sli}	T_{com}	T_{com}
Fib	ψ_1	1.334	18.183	0.301	19.818	1.578	21.396	2.240	0.744	2.984	7.170
	ψ_2	1.280	17.510	0.331	19.121	0.931	20.052	2.298	0.324	2.622	7.648
Lamport	ψ_1	2.006	165.664	0.578	168.248	2.552	170.800	4.201	1.610	5.811	29.393
	ψ_2	1.968	165.432	0.854	168.254	2.257	170.511	4.209	1.504	5.712	29.850
Dekker	ψ_1	1.428	64.792	0.430	66.650	1.949	68.599	3.213	1.286	4.499	15.249
	ψ_2	1.443	64.299	0.535	66.277	1.520	67.797	3.196	1.033	4.230	16.029
Szymanski	ψ_1	1.745	449.599	0.966	452.311	2.608	454.918	4.115	1.439	5.554	81.909
	ψ_2	1.750	448.491	1.174	451.415	1.982	453.397	4.149	1.363	5.512	82.256
Peterson	ψ_1	1.282	40.541	0.252	42.075	1.577	43.652	2.675	1.109	3.783	11.538
	ψ_2	1.317	40.171	0.340	41.828	1.200	43.028	2.585	0.836	3.421	12.576
Sync	ψ_1	1.757	14.505	0.342	16.605	1.265	17.870	1.838	0.743	2.581	6.923
	ψ_2	1.725	14.545	0.679	16.949	1.146	18.095	1.817	0.728	2.545	7.110
Datarace	ψ_1	1.869	51.096	0.415	53.380	2.056	55.436	3.114	0.962	4.076	13.599
	ψ_2	1.886	51.024	0.782	53.693	1.872	55.564	3.145	1.010	4.155	13.372
Rwlock	ψ_1	1.836	32.237	0.869	34.942	2.304	37.246	3.022	1.475	4.497	8.283
	ψ_2	1.847	31.838	0.934	34.619	1.895	36.514	3.051	1.393	4.444	8.217
Varmutex	ψ_1	1.730	29.709	0.675	32.114	3.464	35.578	4.598	1.648	6.245	5.697
	ψ_2	1.703	29.633	0.702	32.038	3.044	35.082	4.690	2.037	6.727	5.215
Lazy	ψ_1	1.620	11.914	0.232	13.766	1.356	15.122	1.784	0.754	2.538	5.959
	ψ_2	1.662	12.014	0.421	14.097	1.035	15.132	1.785	0.733	2.518	6.009
Ccnf	ψ_1	1.853	10.875	0.514	13.242	1.567	14.809	1.996	2.402	4.398	3.367
	ψ_2	1.652	10.478	0.345	12.475	1.651	14.126	2.028	2.174	3.902	3.362
Average		1.668	80.661	0.576	82.905	1.855	84.760	2.989	1.241	4.230	17.306

time more than traditional Petri net slicing. The reason is that it can reduce the number of places, transitions, and explored states of the original PDNet, which is not reduced by traditional Petri net slicing. Thus, the answer to question $Q2$ is positive.

Furthermore, the time required for modeling (T_{mol} in Table 3) and slicing (T_{sli} in Table 3) are significantly less than T_{ver} in Table 4. As a result, the expense of constructing a PDNet model is reasonable and the reduction achieved through PDNet slicing is outstanding. This confirms the effectiveness and practicality of PDNet slicing.

5.4. Threats to Validity

To carry out a qualitative assessment and visualize the distributions in Table 3 and 4, we plotted the slicing time and verification time of the three methods as two box plots in Figure 4. As we can see from the first set of boxes in Figure 4(a), the median of *PS* is larger than the slicing time of the on-demand slicing method, due to *PS* needs to calculate the complete dependencies on data flow in advance. Although the median of slicing time of *TraPNSlice* is the smallest, the median of verification time is the largest in Figure 4(b). Because the slicing process of *TraPNSlice* does not need to calculate dependencies, and the time complexity is minimal. However, the resulting slices of *TraPNSlice* are usually not reduced. The median of PDNet slicing time is middle, but the median of its verification time is minimal compared to *PS* and *TraPNSlice* in Figure 4(b). That is, PDNet slice produces the smallest state space, which results in the shortest verification time.

To assess the statistical significance and analyze the effect size of results in Table 3 and 4, we calculate p-value using the Mann-Whitney U test [14] and Cliff's δ [40]. *PDNet* is tested in comparison with *PS* and *TraPNSlice* on both slicing and verification time. At a significance level of 0.05, the p-value on slicing time in comparison with *PS* (*TraPNSlice*) is 0.000161 (0.000000394), meaning that the difference in the median is statistically highly significant. And the p-value on verification time in comparison with *PS* (*TraPNSlice*) is 0.34 (0.01). It represents that the difference in the median between *TraPNSlice* and *PDNet* is statistically significant. However, there is no statistically

Journal Pre-proof

Table 4
Verification Comparison of *PS*, *TraPNSlice* and *PDNet* (in milliseconds). Columns are: S_{ver}: the time for verifying the CPN model of the program slice; S_{res}: the verifying result of *PS*; C_{mol}: the modeling time to construct a CPN model for the whole program; C_{sli}: the time of traditional Petri net slicing; C_{ver} is the time for verifying traditional Petri net slice; C_{res}: the verifying result of *TraPNSlice*; T_{ver}: the time for verifying PDNet slice; T_{res}: the verifying result of *PDNet*.

Bench.	For.	*PS*		*TraPNSlice*				*PDNet*		S_{ver}/ T_{ver}	C_{ver}/ T_{ver}
		S_{ver}	S_{res}	C_{mol}	C_{sli}	C_{ver}	C_{res}	T_{ver}	T_{res}		
Fib	ψ_1	14931.152	F	3.505	0.201	28951.807	F	10544.388	F	1.416	2.746
	ψ_2	9.686	F	3.501	0.152	18.267	F	7.511	F	1.290	2.432
Lamport	ψ_1	38.590	T	5.917	0.617	77.759	T	35.822	T	1.077	2.171
	ψ_2	8.075	F	5.937	0.631	13.454	F	7.339	F	1.100	1.833
Dekker	ψ_1	13.782	T	4.658	0.348	22.719	T	11.251	T	1.225	2.019
	ψ_2	12.810	T	4.637	0.369	23.224	T	11.489	T	1.115	2.021
Szymanski	ψ_1	27.037	T	5.778	0.534	44.302	T	20.703	T	1.306	2.140
	ψ_2	9.968	F	5.851	0.534	17.827	F	9.666	F	1.031	1.844
Peterson	ψ_1	12.695	T	3.900	0.282	25.197	T	11.344	T	1.119	2.221
	ψ_2	11.689	T	3.889	0.290	26.034	T	11.634	T	1.005	2.238
Sync	ψ_1	8.420	T	2.726	0.248	17.253	T	8.415	T	1.001	2.050
	ψ_2	8.657	T	2.735	0.230	17.584	T	8.346	T	1.037	2.107
Datarace	ψ_1	8009.776	T	4.744	0.361	15999.411	T	6947.670	T	1.153	2.303
	ψ_2	17.369	F	4.768	0.350	31.782	F	15.225	F	1.141	2.088
Rwlock	ψ_1	40.028	T	5.183	0.466	70.768	T	33.947	T	1.179	2.085
	ψ_2	52.972	T	5.143	0.466	72.940	T	34.491	T	1.536	2.115
Varmutex	ψ_1	45.990	T	8.657	0.546	79.711	T	27.537	T	1.670	2.895
	ψ_2	82.718	T	8.485	0.534	81.173	T	30.926	T	2.675	2.625
Lazy	ψ_1	11.292	F	2.798	0.193	18.521	F	8.857	F	1.275	2.091
	ψ_2	8.074	F	2.801	0.181	13.325	F	7.465	F	1.082	1.785
Ccnf	ψ_1	23.801	F	8.271	0.218	32.418	F	23.801	F	1.001	1.362
	ψ_2	23.483	F	9.221	0.320	48.917	F	22.654	F	1.037	2.159
Average		1168.003	/	5.141	0.367	2077.472	/	810.931	/	1.249	2.151

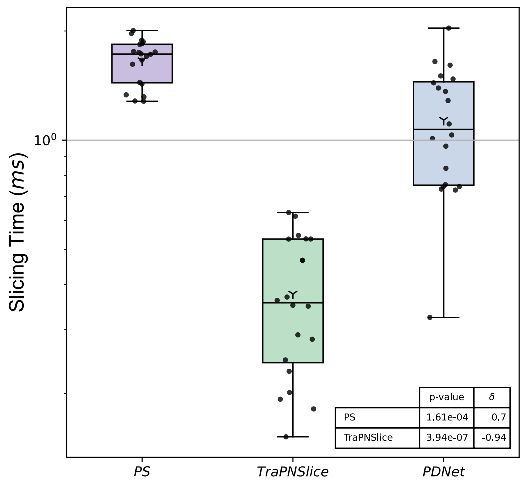
significant difference in verification time between *PS* and *PDNet* according to p-value$>$ 0.05. The reason is that *PS* for model checking does not fully consider lock-dependence and prior-occurrence-dependence as proposed by Hatcliff [20]. If the two dependencies do not exist in concurrent programs, or all dependencies on data flow need to be computed during slicing, the verification time of *PS* and *PDNet* should be comparable.

As a useful complementary analysis, Cliff's δ quantifies the amount of difference in comparison with *PS* (*TraPNSlice*). As we can see from Figure 4, δ on slicing time in comparison with *PS* (*TraPNSlice*) is 0.7 (-0.94). Due to $\delta < 0$ (even close to -1) between *TraPNSlice* and *PDNet*, all slice time of *PDNet* is larger than *TraPNslice*. This difference is also consistent with the intuitive distribution of Figure 4(a). $\delta = 0.7$ between *PS* and *PDNet* means that the on-demand slicing time has a large difference in strengths in comparison with the time of dependencies on data flow of *PS*. δ on verification time in comparison with *PS* (*TraPNSlice*) is 0.18 (0.485). This implies that the difference in strengths of *PDNet* on verification time in comparison with *TraPNSlice* is larger than the one with *PS*.

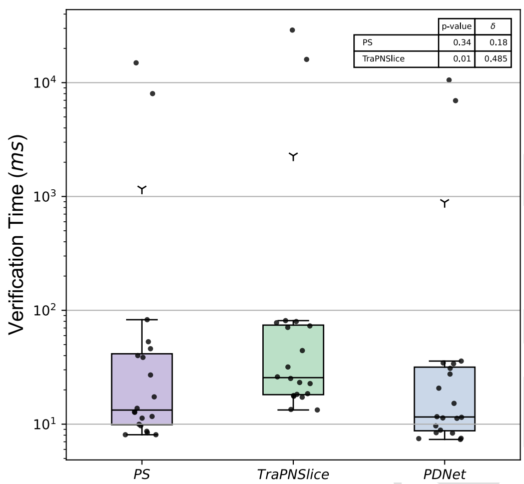
6. Related Work

Rakow [37] suggested two static slicing algorithms to get a simplified Petri net with preserving CTL*-$_\chi$ properties. Khan [25] improved Rakow's algorithms and suggested a dynamic slicing algorithm for Algebraic Petri nets (APN). Lorens et al. [30] and Yu et al. [47] proposed two dynamic slicing algorithms that took into account the initial marking of Petri nets, further reducing the scale of the Petri net slice. Then, two algorithms are improved by Llorens et al. [31] that encounters the maximal slice and the minimal one from the initial net slice. Roci et al. [38] proposed a restraining algorithm to slice from the Petri Nets model the nodes that participate in some particular executions. Wang et al. [44] proposed an improved Dynamic-Slicing-based Vulnerability Detection method to locate vulnerabilities in Workflow Nets. They [45] then proposed a Dynamic-Data-Slicing-based Vulnerability Detection method for data inconsistency detection of E-Commerce Systems. These Petri net slicing algorithms [26] are primarily suitable for workflow. The dependencies outlined in this paper are more complex and pose a challenge to the existing methods [26, 38, 44, 45, 31].

Journal Pre-proof



(a) Box plot for slicing time from Table 3. Light purple box: the time to compute dependencies on data flow of *PS*; Light green box: the time of traditional Petri nets slicing method; Light blue box: the time of on-demand slicing method



(b) Box plot for verification time from Table 4. Light purple box: the verification time of *PS*; Light green box: the verification time of *TraPNSlice*; Light blue box: the verification time of *PDNet*

Figure 4: Box plots with p-value and Cliff's δ

Danicic et al. [10] defined non-termination insensitive (weak) slices and non-termination sensitive (strong) slices for non-deterministic programs. Chalupa et al. [5] proposed new algorithms for computing non-termination sensitive control dependence (NTSCD) and decisive order dependence (DOD) for fast Computation. Masud et al. [33] contributed a general proof of correctness for dependence-based slicing methods for interprocedural, possibly nonterminating programs. Although these works provide more precise definitions, semantics and proof of dependencies for nonterminating programs, they are used for sequential programs, instead of concurrent programs.

Ryder et al. [41] present incremental update algorithms by forward and backward data-flow analysis, which can efficiently handle changes in evolving software systems. Lity et al. [29] propose an approach for incremental testing of software product lines (SPLs) based on incremental model slicing and change impact analysis. Pietsch et al. [35] present a new generic incremental slicer that can slice models of arbitrary type, create editable slices, configure itself automatically, and adapt slices to changes Taentzer et al. [43] propose a formal framework for defining model slicers that support incremental slice updates based on a general concept of model modifications. They are based on the idea of incrementally data flow analysis and are used in slicing methods. However, these methods are mainly used when the system or software changes, and incrementally slice according to the change. Different, the PDNet slicing method in this paper is aimed at the on-demand analysis of dependencies on relevant variables in current concurrent programs rather than changes in a new version. We proposed incremental modeling and checking methods based on PDNet for version change [28].

7. Conclusion

This paper introduces PDNet as a unified model for representing control-flow structures with dependencies. This saves computation by eliminating the need to convert among multiple models. Then, an on-demand PDNet slicing method is proposed that reduces the scope of model checking by capturing dependencies on data flow related to variables from verified LTL-$_\mathcal{X}$. Our methodology can save cost from model conversions and complete calculation of dependencies on data flow, which is never seen in the existing work. We also implement an automatic concurrent program model checking tool called *DAMER* based on PDNet for LTL-$_\mathcal{X}$ formulae. Our experiments have produced promising results.

Based on the current work, we continue to optimize *DAMER* and extend the concurrent program syntax of this paper to model some statements containing undefined functions or instructions, which improves the scalability. We will also add the heuristic information from the dependencies information to help find counterexample paths for LTL-$_\mathcal{X}$ formulae as quickly as possible during exploration.

Acknowledgments

Thanks to the editor and two reviewers for insightful and valuable comments on the improvement of this paper.

Funding This work was supported by National Key Research and Development Program of China under Grant 2022YFB4501704.

CRediT authorship contribution statement

Zhijun Ding: Conceptualization, Writing - Review & Editing. **Shuo Li:** Methodology, Formal analysis, Writing - Original Draft. **Cheng Chen:** Software. **Cong He:** Software.

Journal Pre-proof

References

[1] Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A., 2018. Constrained dynamic partial order reduction, in: Computer Aided Verification, Springer. pp. 392–410. doi:10.1007/978-3-319-96142-2_24.

[2] Barney, B., . Posix threads programming. https://hpc-tutorials.llnl.gov/posix/.

[3] Ben-Zvi, I., Moses, Y., 2014. Beyond Lamport's *Happened-before*: On time bounds and the ordering of events in distributed systems. J. ACM 61. doi:10.1145/2542181.

[4] Burnim, J., Sen, K., Stergiou, C., 2011. Testing concurrent programs on relaxed memory models, in: Proceedings of the 2011 Int. Symp. on Software Testing and Analysis, New York, NY, USA. p. 122–132. doi:10.1145/2001420.2001436.

[5] Chalupa, M., Klaška, D., Strejček, J., Tomovič, L., 2021a. Fast computation of strong control dependencies, in: Computer Aided Verification, pp. 887–910. doi:10.1007/978-3-030-81688-9_41.

[6] Chalupa, M., Strejček, J., 2019. Evaluation of program slicing in software verification, in: Integrated Formal Methods, Springer. pp. 101–119. doi:10.1007/978-3-030-34968-4_6.

[7] Chalupa, M., Vitovská, M., Jašek, T., Šimáček, M., Strejček, J., 2021b. Symbiotic 6: generating test cases by slicing and symbolic execution. Int. J. Softw. Tools Technol. Transf. 23, 875–877. doi:10.1007/s10009-020-00573-0.

[8] Cheng, L., Liu, C., Zeng, Q., 2023. Optimal alignments between large event logs and process models over distributed systems: An approach based on Petri nets. Information Sciences 619, 406–420. doi:10.1016/j.ins.2022.11.052.

[9] Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R., 2018. Handbook of model checking, in: Cambridge International Law Journal. doi:10.1007/978-3-319-10575-8.

[10] Danicic, S., Laurence, M.R., 2018. Static backward slicing of non-deterministic programs and systems. ACM Trans. Program. Lang. Syst. 40. doi:10.1145/2886098.

[11] Dietsch, D., Heizmann, Matthiasand Klumpp, D., Naouar, M., Podelski, A., Schätzle, C., 2021. Verification of concurrent programs using Petri net unfoldings, in: Verification, Model Checking, and Abstract Interpretation, Springer, Cham. pp. 174–195. doi:10.1007/978-3-030-67067-2_9.

[12] Ding, Z., He, C., Li, S., 2023. EnPAC: Petri net model checking for linear temporal logi, in: 2023 IEEE Int. Conf. on Networking, Sensing and Control, pp. 1–6. doi:10.1109/ICNSC58704.2023.10318998.

[13] Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V., Robby, Wallentine, T., 2006. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg. pp. 73–89. doi:10.1007/11691372_5.

[14] Fay, M.P., Proschan, M.A., 2010. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. Statistics Surveys 4, 1 – 39. doi:10.1214/09-SS051.

[15] Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9, 319–349. doi:10.1145/24039.24041.

[16] Gan, M., Wang, S., Ding, Z., Zhou, M., Wu, W., 2018. An improved mixed-integer programming method to compute emptiable minimal siphons in S^3PR nets. IEEE Trans. Control Syst. Technol. 26, 2135–2140. doi:10.1109/TCST.2017.2754982.

[17] Guo, S., Kusano, M., Wang, C., 2016. Conc-ise: Incremental symbolic execution of concurrent software, in: 31st IEEE/ACM Int. Conf. on Automated Software Engineering, p. 531–542. doi:10.1145/2970276.2970332.

[18] Hatcliff, J., Corbett, J., Dwyer, M., Sokolowski, S., Zheng, H., 1999. A formal study of slicing for multi-threaded programs with JVM concurrency primitives, in: Static Analysis, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 1–18. doi:10.1007/3-540-48294-6_1.

[19] Hatcliff, J., Dwyer, M., 2001. Using the bandera tool set to model-check properties of concurrent Java software, in: CONCUR 2001 - Concurrency Theory, Springer. pp. 39–58. doi:10.1007/3-540-44685-0_5.

[20] Hatcliff, J., Dwyer, M.B., Zheng, H., 2000. Slicing software for model construction. Higher Order Symbol. Comput. 13, 315–353. doi:10.1023/A:1026599015809.

[21] He, C., Ding, Z., 2021. More efficient on-the-fly verification methods of colored petri nets. COMPUTING AND INFORMATICS 40, 195–215. doi:10.31577/cai_2021_1_195.

[22] Horwitz, S., Reps, T., Binkley, D., 1990. Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12, 26–60. doi:10.1145/989393.989419.

[23] Jensen, K., Kristensen, L.M., 2009. Formal definition of non-hierarchical coloured Petri nets, in: Coloured Petri Nets, Springer. pp. 79–94. doi:10.1007/b95112_4.

[24] Kahlon, V., Gupta, A., 2006. An automata-theoretic approach for model checking threads for LTL property, in: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, USA. p. 101–110. doi:10.1109/LICS.2006.11.

[25] Khan, Y.I., 2013. Optimizing verification of structurally evolving algebraic Petri nets, in: Int. Work. on Software Engineering for Resilient Systems, Springer. pp. 64–78.

[26] Khan, Y.I., Konios, A., Guelfi, N., 2018. A survey of Petri nets slicing. ACM Computer Survey (CSUR) 51. doi:10.1145/3241736.

[27] Kumar, N., Neema, S., Das, M., Mohan, B.R., 2021. Program slicing analysis with KLEE, DIVINE and Frama-C, in: 26th Int. Conf. on Automation and Computing, pp. 1–5. doi:10.23919/ICAC50006.2021.9594142.

[28] Li, S., Chen, C., Huang, Z., Ding, Z., 2024. Change-aware model checking for evolving concurrent programs based on Program Dependence Net. Journal of Software: Evolution and Process 36. doi:10.1002/smr.2626.

[29] Lity, S., Morbach, T., Thüm, T., Schaefer, I., 2016. Applying incremental model slicing to product-line regression testing, in: Kapitsaki, G.M., Santana de Almeida, E. (Eds.), Software Reuse: Bridging with Social-Awareness, Springer International Publishing, Cham. pp. 3–19. doi:10.1007/978-3-319-35122-3_1.

[30] Llorens, M., Oliver, J., Silva, J., Tamarit, S., 2016. Dynamic slicing of concurrent specification languages. Parallel Computing 53, 1–22.

[31] Llorens, M., Oliver, J., Silva, J., Tamarit, S., 2023. Maximal and minimal dynamic Petri net slicing. Fundam. Inf. 188, 239–267. doi:10.3233/FI-222148.

[32] Löwe, S., Mandrykin, M., Wendler, P., 2014. CPAchecker with sequential combination of explicit-value analyses and predicate analyses, in: Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Springer. pp. 392–394. doi:10.1007/978-3-642-54862-8_27.

[33] Masud, A.N., Lisper, B., 2021. Semantic correctness of dependence-based slicing for interprocedural, possibly nonterminating programs. ACM Trans. Program. Lang. Syst. 42. doi:10.1145/3434489.

[34] Peled, D., Wilke, T., 1997. Stutter-invariant temporal properties are expressible without the next-time operator. Information Processing Letters 63, 243–246. doi:10.1016/S0020-0190(97)00133-6.

[35] Pietsch, C., Ohrndorf, M., Kelter, U., Kehrer, T., 2017. Incrementally slicing editable submodels, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 913–918. doi:10.1109/ASE.2017.8115704.

[36] Qi, X., Jiang, Z., 2017. Precise slicing of interprocedural concurrent programs. Frontiers of Computer Science 11, 971–986. doi:10.1007/s11704-017-6189-3.

[37] Rakow, A., 2012. Safety slicing Petri nets, in: Int. Conf. on Application and Theory of Petri Nets and Concurrency, Springer. pp. 268–287.

[38] Roci, A., Davidrajuh, R., 2020. A new dynamic algorithm for Petri nets slicing, in: IEEE 14th Int. Conf. on Application of Information and Communication Technologies, pp. 1–7. doi:10.1109/AICT50176.2020.9368613.

[39] Rodríguez, C., Sousa, M., Sharma, S., Kroening, D., 2015. Unfolding-based partial order reduction, in: 26th Int. Conf. on Concurrency Theory (CONCUR 2015), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. pp. 456–469. doi:10.4230/LIPIcs.

Journal Pre-proof

CONCUR.2015.456.

[40] Romano, J., Kromrey, J., Coraggio, J., Skowronek, J., 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?, in: annual meeting of the Florida Association of Institutional Research, pp. 1–3.

[41] Ryder, B.G., Paull, M.C., 1988. Incremental data-flow analysis algorithms. ACM Trans. Program. Lang. Syst. 10, 1–50. doi:10.1145/42192.42193.

[42] SV-COMP, 2023. software verification competition. https://sv-comp.sosy-lab.org/2023/.

[43] Taentzer, G., Kehrer, T., Pietsch, C., Kelter, U., 2018. A formal framework for incremental model slicing, in: Russo, A., Schürr, A. (Eds.), Fundamental Approaches to Software Engineering, Springer International Publishing, Cham. pp. 3–20. doi:10.1007/978-3-319-89363-1_1.

[44] Wang, M., Ding, Z., Liu, G., Jiang, C., Zhou, M., 2020a. Measurement and computation of profile similarity of workflow nets based on behavioral relation matrix. IEEE Trans. Syst. Man Cybern. -Syst. 50, 3628–3645. doi:10.1109/TSMC.2018.2852652.

[45] Wang, M., Ding, Z., Zhao, P., Yu, W., Jiang, C., 2020b. A dynamic data slice approach to the vulnerability analysis of E-commerce systems. IEEE Trans. Syst. Man Cybern. -Syst. 50, 3598–3612. doi:10.1109/TSMC.2018.2862387.

[46] Wolf, K., 2019. How Petri net theory serves Petri net model checking: A survey, in: Transactions on Petri Nets and Other Models of Concurrency XIV, Springer. pp. 36–63. doi:10.1007/978-3-662-60651-3_2.

[47] Yu, W., Ding, Z., Fang, X., 2015. Dynamic slicing of Petri nets based on structural dependency graph and its application in system analysis. Asian Journal of Control 17, 1403–1414.

Appendix

A. Colored Petri Nets

To define PDNet (Program Dependence Net) based on CPN, we introduce the definitions of multiset and CPNs [23].

Definition A.1 (Multiset)**.** *Let S be a non-empty set. A multiset ms over S is a function $ms : S \to \mathbb{N}$ that maps each element into 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. And $=$, $<$, $>$, $\leq$, $\geq$ are comparisons of multisets, which are defined in the standard way.*

Definition A.2 (Colored Petri Net)**.** *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 of the typed variables. $\forall v \in V : Type[v] \in \Sigma$.

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 a color set $C(p)$ belonging to the set of types Σ to each place p.

7. $G : T \to \mathbb{E}_V$ 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. $E : F \to \mathbb{E}_V$ 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 to arc f.

9. $I : P \to \mathbb{E}_\emptyset$ is an initialization function, that assigns an initialization expression $I(p)$ to each place p. $\forall p \in P : (Type[I(p)] = C(p)_{MS}) \wedge (Var(I(p)) = \emptyset)$.

The difference between PDNet and CPNs as the following aspects: 1. P is divided into three subsets in PDNet, 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 a subset of execution places. 2. F is divided into three subsets in PDNet, i.e., $F = F_c \cup F_{rw} \cup F_f$. 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.

B. Concurrent Program Semantics

C programs using POSIX threads [2] refer to the concurrent programs in this paper. For simplicity, we consider the assignment statements to be atomic. Take inspiration from the existing research on the function call [22, 33] and concurrency primitive [2], we introduce a simple concurrent program definition. Suppose that $\mathcal{V}$ is a set of basic program variables (e.g., the types of int and $double$), val is a set of all values that a variable v in $\mathcal{V}$ can take. Suppose that $\mathcal{I}$ is a set of thread identifiers (i.e., *pthread_t*), W is a set of program expressions, Q is a set of operations that characterize the nature of the action performed by the statement.

Definition B.1 (Concurrent Program)**.** *$\mathcal{P} ::= \langle K, \mathcal{M}, \mathcal{L}, \mathcal{C}, \mathcal{T}, \mathcal{H}, \mathcal{R}, m_0, h_0 \rangle$ is a concurrent program, where:*

1. K is a finite set of all program locations.

2. $\mathcal{M}$ is a set of all memory states.

3. $\mathcal{L}$ is a finite set of POSIX mutex variables (i.e., pthread_mutex_t). $\ell \in \mathcal{L}$ is a mutex.

4. C is a finite set of condition variables (i.e., pthread_cond_t). $\gamma \in C$ is a condition variable.

5. $\mathcal{T} \subseteq \mathcal{I} \times Q \times K \times K \times \mathcal{M} \times \mathcal{M}$ is a finite set of statements.

6. $\mathcal{H} : \mathcal{I} \to K$ is a function that assigns its current location to each thread identifier.

7. $\mathcal{R} : \mathcal{M} \to val_{MS}$ is a function that assigns the current values of variables to each memory state.

8. $h_0 \in \mathcal{H}$ is the initial location function that assigns the initial location to each thread identifier.

9. $m_0 \in \mathcal{M}$ is an initial memory state.

A statement $\tau ::= \langle i, q, l, l', m, m' \rangle$ intuitively represents that a thread $i \in \mathcal{I}$ can execute an operation $q \in Q$, updating the location from $l \in K$ to $l' \in K$ and the memory state from $m \in \mathcal{M}$ to $m' \in \mathcal{M}$.

The corresponding syntax of concurrent program $\mathcal{P}$ in this paper is described in Table B.1, where ϵ, val, v, γ, ℓ, uop, rop, $break$, $continue$, $return$, $entry$, $export$, i, if, then, else, while, do, $call$, $rets$, $lock$, $unlock$, $wait$, and $signal$ are the terminal symbols of a syntax. Here, ϵ means the default value, uop is a set of unary operators, rop is a set of binary operators. $\mathcal{P}$ contains a series of variable declarations v^* and function declarations fun^+. A function fun is uniquely identified by i, and contains an $entry$ and an $exit$, and v^*

Journal Pre-proof

represents a parameter list that is defaulted. τ^* is a set of statements within this function.

The behavior of a statement $\tau \in \mathcal{T}$ is represented by its operation q, characterizing the nature of the action performed by this statement. Then, we distinguish the following operation sets $\{local\}$, $\{calls\}$ and $\{syncs\}$ for $\tau ::= local|calls|syncs$ in Table B.1. Here, operations assignment, jump, and branching belong to $\{local\}$, operations call site and return site belong to $\{calls\}$, and $\langle lock, \ell \rangle$, $\langle unlock, \ell \rangle$, $\langle signal, \gamma \rangle$ and $\langle wait, \gamma, \ell \rangle$ belong to $\{syncs\}$.

(1) The local operations in $local$ are used to model statements within a local thread. $v:=w$ represents a simple assignment operation where $v \in \mathcal{V}$ is a program variable and $w \in W$ is an expression over the program variables. $jump$ represents a simple jump operation with a particular symbol, e.g., $break$, $continue$ and $return$. if(w) then (τ_1^*) else (τ_2^*) (abbreviated as if(w)) is a branch conditional structure with a boolean condition denoted by an expression w, and while(w) do (τ^*) (abbreviated as while(w)) is a loop structure with a boolean condition denoted by an expression w. The two structures are the branching operations that produce different possible subsequent executions.

(2) $calls$ represents possible many function calls. As the syntax in Table B.1, $call$ represents call site operation making the control-flow turn to the called function, and $rets$ represents return site one, which makes the control-flow turn back from the called function. Moreover, $cassign$ represents the assignments to all formal input parameters of $call$, and $rassign$ represents the assignments to the actual return parameters of $rets$.

(3) The POSIX thread operations in $syncs$ could model the synchronization statements in different threads. $syncs = (\{lock, unlock\} \times \mathcal{L}) \cup (\{signal\} \times \mathcal{C}) \cup (\{wait\} \times \mathcal{C} \times \mathcal{L})$. Operation $\langle lock, \ell \rangle$ represents a request operation to acquire $\ell \in \mathcal{L}$, i.e., pthread_mutex_lock(&ℓ), while $\langle unlock, \ell \rangle$ represents a request operation to release $\ell \in \mathcal{L}$, i.e., pthread_mutex_unlock(&ℓ). $\langle signal, \gamma \rangle$ represents a request operation to signal other thread on $\gamma \in \mathcal{C}$, i.e., pthread_cond_signal(&γ). $\langle wait, \gamma, \ell \rangle$ represents a request operation to wait for a notification on $\gamma \in \mathcal{C}$ with $\ell \in \mathcal{L}$, i.e., pthread_cond_wait(&γ, &ℓ). Concrete arguments of POSIX thread functions mentioned above are found from [2].

To express the operational semantics for PDNet modeling, we define labeled transition system (LTS) semantics based on Definition B.1 and Table B.1.

Definition B.2 (LTS Semantics of Concurrent Programs)**.** *Given concurrent program $\mathcal{P}$, $\mathcal{N}_\mathcal{P} ::= \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ is 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 $\mathcal{T}$ is from $\mathcal{P}$.

3. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is a set of transition relations on the program configurations $\mathcal{S}$.

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 that maps every mutex to a thread identifier, and u is also a function maps every condition variable to a multiset of thread identifiers of those threads that currently wait on that 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 $\mathcal{P}$, $r_0 \colon \mathcal{L} \rightarrow \{0\}$ represents that every mutex is not initially held by any threads, and $u_0 \colon \mathcal{C} \rightarrow \emptyset$ represents that every condition variable does 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 operation q from 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 configuration s to new one 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 modeling 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 B.2. 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 B.2. For instance, asi represents the action $\langle \tau, \langle ass, l' \rangle \rangle$ where $\langle ass, l' \rangle$ is the effect corresponding to the operation of τ, updating the program location to l'. Here, suppose that an assignment operation is $v:=w$ in statement τ. $[\![w]\!]m$ denotes that the value evaluating by the expression w under the memory state m. This value is assigned to variable v. Thus, $m' = m[v \mapsto [\![w]\!]m]$ denotes a new memory state where $m'(v) = [\![w]\!]m$ and $m'(y) = m(y)$ $(\forall y \in \mathcal{V} \colon y \neq v)$.

In the same way, jum represents 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 a memory state. ret represents action $\langle \tau, \langle ret, l' \rangle \rangle$ where

Table B.1
Simplified Syntax of Concurrent Programs

$\mathcal{P} ::= (v^*) fun^+$
$fun ::= entry\ i\ (v^*)\ (\tau^*)\ exit$
$\tau ::= local \mid calls \mid syncs$
$local ::= v:=w \mid jump \mid \text{if}(w)\text{then}(\tau_1^*)\text{else}(\tau_2^*) \mid \text{while}(w)\text{do}(\tau^*)$
$jump ::= break \mid continue \mid return$
$w ::= val \mid v \mid uop\ w \mid w\ rop\ w$
$calls ::= acall \mid cassign\ acall\ rassign$
$acall ::= call\ cassign\ acall\ rassign\ rets \mid acall\ acall\ \mid \epsilon$
$cassign ::= (v:=w)^*$
$rassign ::= (v:=w)^*$
$syncs ::= \langle lock, \ell \rangle \mid \langle unlock, \ell \rangle \mid \langle signal, \gamma \rangle \mid \langle wait, \gamma, \ell \rangle$

Journal Pre-proof

Table B.2
Semantics of Concurrent Programs

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=v:=w\ \ h(i)=l}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle ass,l'\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m',r,u\rangle}\ (asi)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ q:=break\ or\ continue\ \ h(i)=l}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle jum,l'\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m,r,u\rangle}\ (jum)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=return\ \ h(i)=l}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle ret,l'\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m',r,u\rangle}\ (ret)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=if(w)orwhile(w)\ \ [\![w]\!]m=true\ \ h(i)=l}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle tcd,l'\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m,r,u\rangle}\ (tcd)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=if(w)orwhile(w)\ \ [\![w]\!]m=F\ \ h(i)=l}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle fcd,l'\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m,r,u\rangle}\ (fcd)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=call\ \ h(i)=l}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle call,l'\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m',r,u\rangle}\ (call)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=rets\ \ h(i)=l}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle rets,l'\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m',r,u\rangle}\ (rets)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=\langle lock,\ell\rangle\ \ h(i)=l\ \ r(\ell)=0}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle acq,\ell\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m,r[\ell\mapsto i],u\rangle}\ (acq)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=\langle unlock,\ell\rangle\ \ h(i)=l\ \ r(\ell)=i}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle rel,\ell\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m,r[\ell\mapsto 0],u\rangle}\ (rel)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=\langle signal,\gamma\rangle\ \ h(i)=l\ \ \{j\}\in u(\gamma)}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle sig,\gamma\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m,r,u[\gamma\mapsto u(\gamma)\backslash\{j\}\cup\{-j\}]\rangle}\ (sig)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=\langle wait,\gamma,\ell\rangle\ \ h(i)=l\ \ r(\ell)=i\ \ \{i\}\notin u(\gamma)}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle wa_1,\gamma,\ell\rangle\rangle}{\longrightarrow}\langle h,m,r[\ell\mapsto 0],u[\gamma\mapsto u(\gamma)\cup\{i\}]\rangle}\ (wa_1)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=\langle wait,\gamma,\ell\rangle\ \ h(i)=l\ \ r(\ell)=0\ \ \{-i\}\in u(\gamma)}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle wa_2,\gamma,\ell\rangle\rangle}{\longrightarrow}\langle h,m,r,u[\gamma\mapsto u(\gamma)\backslash\{-i\}]\rangle}\ (wa_2)$$

$$\frac{\tau:=\langle i,q,l,l',m,m'\rangle\in\mathcal{T}\ \ q:=\langle wait,\gamma,\ell\rangle\ \ h(i)=l\ \ r(\ell)=0}{\langle h,m,r,u\rangle\overset{\langle\tau,\langle wa_3,\gamma,\ell\rangle\rangle}{\longrightarrow}\langle h[i\mapsto l'],m,r[\ell\mapsto i],u\rangle}\ (wa_3)$$

the jump operation of τ is *return*, updating the program location to l'. For a branching operation, tcd represents action $\langle\tau,\langle tcd,l'\rangle\rangle$, where $[\![w]\!]m$=true, and fcd represents action $\langle\tau,\langle tcd,l'\rangle\rangle$, where $[\![w]\!]m$=false. Neither tcd nor fcd updates the memory state. They update the program location to a different one l'. For a function call, $call$ represents action $\langle\tau,\langle call,l'\rangle\rangle$, where l' is the entry of the called function, and $rets$ represents action $\langle\tau,\langle rets,l'\rangle\rangle$, where l' is the return site of the calling function. Similarly, suppose that $cassign$ of $call$ is $v_1:=w_1$ and $rassign$ of $rets$ is $v_2:=w_2$. $m'=m[v_1\mapsto[\![w_1]\!]m]$ denotes the new memory state for $call$ and $m'=m[v_2\mapsto[\![w_2]\!]m]$ denotes the new one for $rets$. In addition, asi, jum, tcd, fcd, $call$ and $rets$ do not update r and u of s.

Moreover, acq represents action $\langle\tau,\langle acq,\ell\rangle$ corresponding to operation $\langle lock,\ell\rangle$. If ℓ is not held by any thread ($r(\ell)$=0), acq represents that thread i obtains this mutex ℓ ($r[\ell\mapsto i]$) and updates the program location to l'. However, if ℓ is held by another thread, thread i could be blocked by ℓ, and current configuration s cannot be updated by acq. rel represents action $\langle\tau,\langle rel,\ell\rangle$ corresponding to operation $\langle unlock,\ell\rangle$. Here, $r(\ell)$=i means that 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 action $\langle\tau,\langle sig,\gamma\rangle\rangle$ corresponding to operation $\langle signal,\gamma\rangle$. Thread i could notify thread j belonging to $u(\gamma)$ ($\{j\}\in u(\gamma)$). Thus, thread j could be notified by thread i ($u[\gamma\mapsto u(\gamma)\backslash\{j\}\cup\{-j\}]$). It updates the program location to l'. Particularly, 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 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 that the action releases 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 that the thread i is blocked until a thread j ($\{-i\}\in u(\gamma)$) is notified by condition variable γ . Thus, thread i no long waits for a notification on γ ($u[\gamma\mapsto u(\gamma)\backslash\{-i\}]$). Finally, if ℓ is not held ($r(\ell)=0$), wa_3 ($\langle wa_3,\gamma,\ell\rangle$) represents that the action acquires 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.

C. Post-process Algorithm

Algorithm C.1 Slicing Post-process Algorithm

Input: A PDNet N and its slice N' w.r.t. $Crit$;
Output: The final PDNet slice N'';
1: **for all** $p\in(P'\cap P_f)\wedge(\forall t\in{}^\bullet p:\ (t,p)\notin F'_f)$ **do** /∗ F'_f is the execution arc set∗/
2: │ T_f := FINDEXE(t,T); /∗ T is the transition set of N ∗/
3: │ **for all** $t'\in T_f$ **do**
4: │ │ ADDARC(t',p); /∗ (t',p) is a new execution arc∗/

If there exists an execution place p of N' that lacks execution arcs connected to any transition in ${}^\bullet p$, the execution order of p is not complete (Line 1). Function FINDEXE(t,T) can find the transitions performed before the transition in ${}^\bullet p$ based on T (Line 2). For each transition t' in T_f, a new execution arc (t',p) is constructed by ADDARC(t',p) (Line 4) to complete a control-flow structure. Finally, the final PDNet slice N'' is tractable for the model checking.

D. Correctness Proof

We prove the correctness of PDNet slice N' expressed by $N\vDash\psi\Leftrightarrow N'\vDash\psi$. Firstly, we give some definitions for the correctness proof. Note that an occurrence sequence is defined in Definition 4, and an LTL-$_\chi$ formula is defined in Definition 11.

Definition D.1 (Marking Projection)**.** *Let N be a PDNet, M a marking of N, ψ an LTL-$_\chi$ formula of N, and $Crit$ a place subset extracted from ψ. A projection function $\downarrow[\psi]M\ :\ Crit\to\mathbb{E}_\emptyset$ is a local marking function w.r.t. $Crit$, that assigns an expression $M(p)$ to each place p. $\forall p\in Crit$: $Type[M(p)]=C(p)_{MS}\wedge(Var(M(p))=\emptyset)$.*

Definition D.2 (ψ-equivalent Marking)**.** *Let N be a PDNet, ψ an LTL-$_\chi$ formula of N, and M and M' two markings of N. M and M' are ψ-equivalent, denoted by $M\overset{\psi}{\rightsquigarrow}M'$, if $\downarrow[\psi]M=\downarrow[\psi]M'$.*

Definition D.3 (ψ-stuttering Equivalent Transition)**.** *Let N be a PDNet, ψ an LTL-$_\chi$ formula of N, ω the occurrence sequence (t_1,b_1), (t_2,b_2), $\cdots$, (t_i,b_i), (t_{i+1},b_{i+1}), $\cdots$, (t_n,b_n)*

Journal Pre-proof

of N defined in Definition 4, and $M(\omega)$ the marking sequence M_0, M_1, $\cdots$, M_{i-1}, M_i, M_{i+1}, $\cdots$, M_n generated by occurring every binding element of ω in turn. t_i ($1 \le i \le n$) is a ψ-stuttering equivalent transition if $M_{i-1} \overset{\psi}{\rightsquigarrow} M_i$.

Definition D.4 (ψ-stuttering Equivalent Marking Sequences)**.** *Let N be a PDNet, ψ an LTL-$_\chi$ formula of N, ω an occurrence sequence of N, $M(\omega)$ the marking sequence of ω starting from M_0, and ω' be generated by eliminating some binding elements from ω, and $M(\omega')$ marking sequence of ω' starting from M_0'. $M(\omega)$ and $M(\omega')$ are ψ-stuttering equivalent marking sequences, denoted by $M(\omega) \overset{st_\psi}{\rightsquigarrow} M(\omega')$, if $\downarrow[\psi]M_0 = \downarrow[\psi]M_0'$, and the eliminated transitions from ω are ψ-stuttering equivalent transitions.*

Intuitively, the ψ-non-stuttering equivalent transitions of ω are all preserved in ω'.

Definition D.5 (ψ-stuttering Equivalent PDNets)**.** *Let N be a PDNet, ψ an LTL-$_\chi$ formula of N, N' a PDNet with the same LTL-$_\chi$ formula ψ, M_0 the initial marking of N, and M_0' the initial marking of N'. N and N' are ψ-stuttering equivalent PDNets, denoted by $N \overset{st_\psi}{\rightsquigarrow} N'$, if*

1) $\downarrow[\psi]M_0 = \downarrow[\psi]M_0'$, and

2) for each marking sequence $M(\omega)$ of N starting from M_0, there exists a marking sequence $M(\omega')$ of N' starting from M_0' such that $M(\omega) \overset{st_\psi}{\rightsquigarrow} M(\omega')$, and

3) for each marking sequence $M(\omega')$ of N' starting from M_0', there exists a marking sequence $M(\omega)$ of N starting from M_0 such that $M(\omega') \overset{st_\psi}{\rightsquigarrow} M(\omega)$.

Theorem D.1. *[34] Let N be a PDNet, ψ an LTL-$_\chi$ formula of N, and N' a PDNet with the same LTL-$_\chi$ formula ψ. An LTL-$_\chi$ formula is invariant under stuttering, denoted by $N \overset{st_\psi}{\rightsquigarrow} N'$, iff $N \vDash \psi \Leftrightarrow N' \vDash \psi$.*

The above theorem shows that an LTL-$_\chi$ formula ψ is invariant under stuttering [34].

Theorem D.2. *Let N be a PDNet, M_0 the initial marking of N, ψ an LTL-$_\chi$ formula of N, $Crit$ be extracted from ψ, N' w.r.t. $Crit$ the final PDNet slice from N by Algorithm 1 and C.1, and M_0' the initial marking of N'. $N \overset{st_\psi}{\rightsquigarrow} N'$.*

Proof. According to Definition D.5's 1), $\downarrow[\psi]M_0 = \downarrow[\psi]M_0'$ holds, because we keep all places in $Crit$, and Algorithm 1 does not change the initial marking of N.

Then, we prove Definition D.5's 2). Let ω be an arbitrary occurrence sequence of N arbitrarily. There exists an occurrence sequence ω' that is generated by eliminating some binding elements from ω by Algorithm 1. $M(\omega)$ is the marking sequence corresponding to ω, and $M(\omega')$ is the marking sequence corresponding to ω'. We prove $M(\omega) \overset{st_\psi}{\rightsquigarrow} M(\omega')$ by using structural induction on the length of occurrence sequence of ω, denoted by $|\omega|$.

Base case: Let $|\omega|$=1. $\downarrow[\psi]M_0 = \downarrow[\psi]M_0'$ holds.

Induction: Assume that it holds when $|\omega|=k$. That is, $\omega=(t_{i1}, b_{i1}), (t_{i2}, b_{i2}), \cdots, (t_{ik}, b_{ik})$ of N, $\omega'=(t_{j1}, b_{j1}), (t_{j2}, b_{j2}), \cdots, (t_{jm}, b_{jm})$ ($m \le k$), and $M(\omega) \overset{st_\psi}{\rightsquigarrow} M(\omega')$. Then, we consider whether it holds when $|\omega|=k+1$. Suppose that the last transition is t_{k+1}, and $\omega(t_{k+1}, b_{k+1})$ is the extended occurrence sequence. There are two cases for ω': t_{k+1} is sliced and t_{k+1} is not sliced.

Case 1: t_{k+1} is sliced by Algorithm 1. That is, $\nexists p \in Crit$: $t_{k+1} \xrightarrow{d}{}^{*} p$. Consider the two proposition forms. Let po be a proposition from ψ arbitrarily. If po is in the form of $token-value(p_t)\ rop\ c$, $t_{k+1} \notin {}^{\bullet}p_t$ or $E(t_{k+1}, p_t)=E(p_t, t_{k+1})$, and the evaluation of this proposition under each marking is not change. If po is in the form of $is\text{-}fireable(t)$, t_{k+1} does not affect the enabling condition of t according to Definition 12 and Algorithm 1, and the evaluation of this proposition under each marking is not change. Thus, t_{k+1} is a ψ-stuttering equivalent transition. As a result, $M(\omega(t_{k+1}, b_{k+1})) \overset{st_\psi}{\rightsquigarrow} M(\omega')$.

Case 2: t_{k+1} is not sliced, and $\omega'(t_{k+1}, b_{k+1})$ is the extended occurrence sequence. According to $M(\omega) \overset{st_\psi}{\rightsquigarrow} M(\omega')$, t_{k+1} produces the same effect for ω and ω'. As a result, $M(\omega(t_{k+1}, b_{k+1})) \overset{st_\psi}{\rightsquigarrow} M(\omega'(t_{k+1}, b_{k+1}))$.

Finally, we prove Definition D.5's 3). Let ω' be an arbitrary occurrence sequence of N'. There exists an occurrence sequence ω that is generated by adding some binding elements to ω' that are identified by Algorithm 1. $M(\omega')$ is the marking sequence corresponding to ω', and $M(\omega)$ is the marking sequence corresponding to ω. Then, we prove that $M(\omega') \overset{st_\psi}{\rightsquigarrow} M(\omega)$ by using structural induction on the length of occurrence sequence ω', denoted by $|\omega'|$.

Base case: Let $|\omega'|$=1. $\downarrow[\psi]M_0' = \downarrow[\psi]M_0$ holds.

Induction: Assume that it holds when $|\omega'|=k'$. That is, $\omega'=(t_{i1}, b_{i1}), (t_{i2}, b_{i2}), \cdots, (t_{ik'}, b_{ik'})$ of N, $\omega=(t_{j1}, b_{j1}), (t_{j2}, b_{j2}), \cdots, (t_{jm'}, b_{jm'})$ ($m' \ge k'$), and $M(\omega') \overset{st_\psi}{\rightsquigarrow} M(\omega)$. Then, we consider whether it holds when $|\omega'|=k'+1$. Suppose the last transition is $t_{k'+1}$, and $\omega'(t_{k'+1}, b_{k'+1})$ is the extended occurrence sequence. Obviously, $t_{k'+1}$ belongs to ω, and $\omega(t_{k'+1}, b_{k'+1})$ is the extended occurrence sequence. $t_{k'+1}$ produces the same effect for ω and ω'. As a result, $M(\omega'(t_{k'+1}, b_{k'+1})) \overset{st_\psi}{\rightsquigarrow} M(\omega(t_{k'+1}, b_{k'+1}))$.

Thus, $N \overset{st_\psi}{\rightsquigarrow} N'$ holds based on Definition D.5. □

Theorem D.3. *Let N be a PDNet, ψ an LTL-$_\chi$ formula, $Crit$ be extracted from ψ, and N' w.r.t. $Crit$ be a reduced PDNet by Algorithm 1. $N \vDash \psi \Leftrightarrow N' \vDash \psi$.*

Proof. It is obvious that $N \overset{st_\psi}{\rightsquigarrow} N'$ proved by Theorem D.2. Thus, $N \vDash \psi \Leftrightarrow N' \vDash \psi$ iff $N \overset{st_\psi}{\rightsquigarrow} N'$ according to Theorem D.1. Therefore, this corollary holds. □

Hence, it is concluded that the PDNet slice obtained by our methods is correct based on Theorem D.3.

Journal Pre-proof

Highlights

Program Dependence Net and On-demand Slicing for Property Verification of Concurrent System and Software

Zhijun Ding, Shuo Li, Cheng Chen, Cong He

• Property verification of concurrent system and software is a long-standing and challenging task.

• Program Dependence Net as a unified model combines a control-flow structure with control-flow dependencies.

• An on-demand slicing method based on PDNet captures data-flow dependencies in an on-demand way.

• *DAMER* can automatically verify concurrent software of LTL properties without any manual intervention.

Journal Pre-proof

Journal Pre-proof

Zhijun Ding received the PhD degree from Tongji University, Shanghai, China, in 2007. Currently he is a 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.

Shuo Li received the B.S. degree in Software Engineering from the Shandong University of Science and Technology, Qingdao, China, in 2017, and received the PhD degree from Tongji University, Shanghai, China, in 2024. Her current research interests include model checking, Petri nets, and formal methods. She has published 5 papers in domestic and international academic journals and conference proceedings

Cheng Chen received the B.S. and M.S. degree from Tongji University, Shanghai, China, in 2019 and 2022. His current research interests include model checking and formal methods.

Cong He received the B.S. and M.S. degree from Tongji University, Shanghai, China, in 2019 and 2022. His current research interests include model checking and formal methods.

Journal Pre-proof

Journal Pre-proof

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Journal Pre-proof