International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2021, 11(4): 405–428, doi: 10.21655/ijsi.1673-7288.00254

International Journal of Software and Informatics

Research Article

Reverse Unfolding of Petri Nets and its Application in Program Data Race Detection

Zongyin Hao (郝宗寅)[1,2], Faming Lu (鲁法明)[1]

[1] (College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China)

[2] (School of Informatics, Xiamen University, Xiamen 361005, China)

Corresponding author: Faming Lu, fm_lu@163.com

Abstract The unfolding technique can partially alleviate the state explosion in Petri nets through branching processes. However, all states of a system are still contained in its unfolding net. To deal with some practical problems, only the coverability determination of a specific state is needed. In view of this, reducing the scale of the unfolding net is feasible. This study proposes a target-oriented reverse unfolding algorithm for the coverability determination of 1-safe Petri nets, which combines a heuristic technique to reduce the scale of unfolding nets, thereby improving the efficiency of coverability determination. Furthermore, the reverse unfolding is applied to the formal verification of concurrent programs, and their data race detection is converted into the coverability determination of a specific state in 1-safe Petri nets. The experiment compares the efficiency between forward unfolding and reverse unfolding in the coverability determination of a Petri net. The results show that when the Petri net has more forward branches than backward branches, reverse unfolding is more efficient than forward unfolding. Finally, the key factors influencing the efficiency of reverse unfolding are analyzed.

Keywords Petri nets; coverability determination; reverse unfolding; heuristic optimization; data race detection

Citation Hao ZY, Lu FM. Reverse unfolding of Petri nets and its application in program data race detection, *International Journal of Software and Informatics*, 2021, 11(4): 405–428. http://www.ijsi.org/1673-7288/254.htm

As a modeling and analysis tool for distributed concurrent systems, Petri nets have been widely used in flexible manufacturing systems[1, 2], business process management systems[3, 4], and formal verification of concurrent programs[5, 6]. However, the state explosion prevents the application of Petri nets in the analysis of large-scale concurrent systems. In view of this, McMillan[7] firstly proposed to describe system behaviors by the unfolding of nets and

This is the English version of the Chinese article “Petri 网的反向展开及其在程序数据竞争检测的应用. 软件学报, 2021, 32(6): 1612–1630. doi: 10.13328/j.cnki.jos.006240”.

Funding items: National Natural Science Foundation of China (61602279, 61472229); National Key Research and Development Plan (2016YFC0801406); Taishan Scholars Program of Shandong Province (ts20190936); Excellent Youth Innovation Team Foundation of Shandong Higher School (2019KJN024); Postdoctoral Innovation Foundation of Shandong Province (201603056); Shandong-Chongqing Science and Technology Cooperation Plan (cstc2020jscx-lyjsAX0008); Open Foundation of First Institute of Oceanography, MNR (2018002); Shandong University of Science and Technology Research Fund (2015TDJH102)

Received 2020-08-31; Revised 2020-10-26; Accepted 2020-12-19; IJSI published online 2021-12-23

constructed finite complete prefixes of unfolding nets by branching processes[8] and the partial ordering, effectively alleviating the state explosion in the property analysis of Petri nets. After that, the unfolding technique received much attention.

Esparza *et al*.[9] pointed out that the partial order relation defined by McMillan led to an exponential increase in the size of finite complete prefixes in some cases. They proposed a total order relation for 1-safe Petri nets to minimize the size of finite complete prefixes. Khomenko *et al*.[10] standardized the definition of unfolding and conducted the parameterization of unfolding. Heljanko *et al*.[11] parallelized the unfolding technique to improve the unfolding efficiency. Benito *et al*.[12] extended the unfolding technique to timed Petri nets, and Schwarick *et al*.[13] extended it to colored Petri nets. In the application of unfolding of Petri nets, Lu *et al*.[5] proposed a finite unfolding technique of unbounded Petri nets for deadlock detection of net systems. Xiang *et al*.[6] detected data inconsistency in concurrent systems with the unfolding technique. Dong et al.[14] verified Computation Tree Logic (CTL) utilizing reachability graphs of Petri nets. Liu *et al*.[15] detected the robustness of workflows by branching processes. In terms of the property analysis of Petri nets, Chatain *et al*.[16] designed a goal-driven unfolding technique for the coverability problem of Petri nets to prune redundant transitions by analyzing internal causalities. Bonet *et al*.[17] proposed a semi-adequate ordering approach based on heuristics to improve the efficiency of the unfolding technique in the coverability analysis of Petri nets. Later, they proved that the extension order could be independent of the partial order of cut-off events[18], which broadened the application of the heuristic unfolding technique in property analysis of Petri nets.

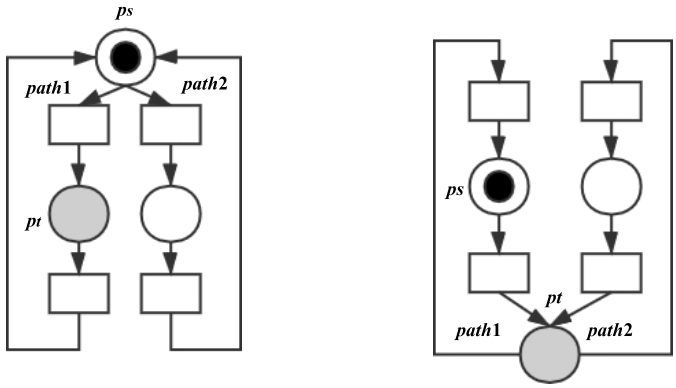
The unfolding of nets alleviates the state explosion in the property analysis of Petri nets to some extent by branching processes and the partial ordering. However, all the states of a system are still contained in its unfolding net. Some practical problems only require to determine the coverability of a specific state. In light of this, simplifying the net unfolding is feasible. To this end, we propose a target-oriented reverse unfolding algorithm for the coverability determination of 1-safe Petri nets. Starting from the target marking that needs coverability determination, reverse unfolding only describes the system states related to coverability determination and reduces the unfolding scale with the heuristic technique, so as to improve the determination efficiency. Further, we apply the reverse unfolding algorithm to formal verification of concurrent programs and convert their data race detection to coverability determination of specific markings in Petri nets. The experiment compares the efficiencies of heuristic reverse unfolding and directed unfolding[17] (a forward unfolding also using the heuristic technique) in coverability determination of Petri nets. The results show that in 415 groups of test data, the scale of reverse unfolding is better than that of directed unfolding on 85 groups of data and is comparable to that of directed unfolding on 26 groups of data. At last, we analyze and summarize the key factors influencing the efficiency of reverse unfolding.

Focusing on the coverability determination of Petri nets, we analyze the application scenarios of forward unfolding and reverse unfolding and use examples to illustrate the advantages of reverse unfolding over forward unfolding in the first section. In the second section, we introduce the reverse unfolding algorithm of Petri nets, including the basic definitions, the algorithm flow, and the heuristic optimization strategy. In Section 3, we apply reverse unfolding to data race detection of concurrent programs. In Section 4, we use experiments to evaluate the efficiency of forward unfolding and reverse unfolding in coverability determination of Petri nets. In Section 5, we make a summarization and a prospect.

1 Examples and Motivation Analysis

In this section, we first discuss the application scenarios of forward unfolding and reverse unfolding in coverability determination of Petri nets. Then we illustrate the advantages of reverse unfolding over forward unfolding by an example.

Here are two simple examples. In Figure 1(a), the initial marking of the Petri net is $\{ps\}$, and the coverability of the target marking $\{pt\}$ should be verified. For simplicity of presentation, the path on the left side in Figure 1(a) is called *path*1, and that on the right side is called *path*2. In this example, the forward unfolding starts from the initial marking $\{ps\}$, and it has difficulty in selecting *path*1 or *path*2. If it unfortunately selects *path*2, it will make plenty of redundant extensions. Unlike the forward unfolding, the reverse unfolding starts from the target marking $\{pt\}$ and only needs to extend reversely along *path*1. Thus, it can easily find a reachable path to the initial marking $\{ps\}$.



(a) Application scenario of reverse unfolding (b) Application scenario of forward unfolding

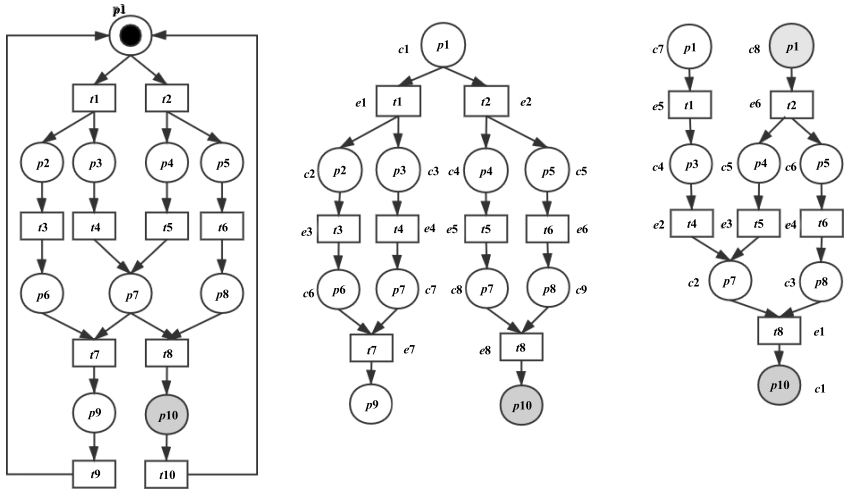
Figure 1 Application scenarios of forward unfolding and reverse unfolding

Figure 1(b) can be seen as an "inversion" of Figure 1(a). In this example, the reverse unfolding starts from the target marking $\{pt\}$, and it has difficulty in selecting *path*1 or *path*2; while the forward unfolding starts from the initial marking $\{ps\}$, and it easily reaches the target marking $\{pt\}$ along *path*1.

As indicated by the above two examples, reverse unfolding is suitable to Petri nets that have more forward branches. When a Petri net has more reverse branches, the forward unfolding is more applicable. Specifically, forward unfolding starts from the initial marking of a Petri net to describe the system, which implies the system's complete behaviors; while reverse unfolding starts from the target marking whose coverability needs to be determined and only describes the system states related to coverability determination. In this paper, we design and realize the reverse unfolding algorithm from this perspective.

Then, we use an example to further illustrate the advantages of reverse unfolding over forward unfolding. In Figure 2(a), the initial marking of the Petri net is $\{p1\}$, and the coverability of the target marking $\{p10\}$ needs to be verified. Figure 2(b) shows the forward unfolding of the Petri net[9], and Figure 2(c) shows its reverse unfolding. The two both use the adequate order, which is based on the breadth-first strategy, as the extension order. Once the coverability of the target marking is verified, the extension ends. In this example, the forward unfolding generates 19 nodes and 20 flow relations. The reverse unfolding yields 14 nodes and 14 flow relations. The scale of the reverse unfolding is better than that of the forward unfolding. This is because the

contribution of the transition $\{t1, t3, t4, t7\}$ to the coverability of $\{p10\}$ is redundant (namely that the path on the left is redundant). The forward unfolding implies complete behaviors of the system, and it inevitably analyzes the redundant behaviors of $\{t1, t3, t4, t7\}$. Regarding the reverse unfolding, only the system states related to coverability determination are described, which avoids the redundant description of $\{t3, t7\}$. Although the reverse unfolding still includes redundancies, its overall scale is better than that of forward unfolding.



(a) Petri net (b) Forward unfolding of Petri net (c) Reverse unfolding of Petri net

Figure 2 An example manifesting the advantage of reverse unfolding over forward unfolding

In this section, we discuss the advantages of forward unfolding and reverse unfolding. It should be noted that these advantages are generalized as they are determined by properties of algorithms. For forward unfolding such as goal-driven unfolding[16] and directed unfolding[17], although they can use internal causalities to prune or use heuristics to improve the algorithm efficiency, they inevitably analyze redundant system behaviors when there are more forward branches. Similarly, even if reverse unfolding is equipped with the heuristic technique, it cannot assure the algorithm efficiency when there are more reverse branches. Under this premise, we further illustrate that reverse unfolding outperforms the forward unfolding in some cases through examples. Then we will introduce the reverse unfolding algorithm in detail.

2 Reverse Unfolding of Petri Nets

2.1 Concept of reverse unfolding

2.1.1 *Petri nets*

A net can be defined as a triple (P, T, F), where P is a place set, T a transition set, F the flow relation between P and T with $F \subset (P \times T) \cup (T \times P)$. We define the preset and post set of the node x as ${}^\bullet x = \{y \in P \cup T | F(y, x) = 1\}$ and $x^\bullet = \{y \in P \cup T | F(x, y) = 1\}$, respectively. The marking of the net (P, T, F) is a multiset established on P. In graphical representation, we present the markings of a net by adding the corresponding number of tokens to each place.

A net system can be defined as a quadruple (P, T, F, M_0), where M_0 is the initial marking of the net (P, T, F). If $\forall p \in P : F(p, t) \leq M(p)$, the transition t under the marking M

enables, and the enabled transition is executable. The execution of t makes the system enter into a new marking M', which is denoted as $M \xrightarrow{t} M'$, i.e., for each place p, $M'(p) = M(p) - F(p,t) + F(t,p)$. A transition sequence $\sigma = t_1 t_2 \cdots t_n$ is denoted as a triggered sequence when and only when there are markings $M_1, M_2, \cdots, M_{n-1}, M_n$ satisfying $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} M_{n-1} \xrightarrow{t_n} M_n$. The above formula can also be indicated as $M_0 \xrightarrow{\sigma} M_n$, and M_n is a reachable marking for the triggered sequence σ.

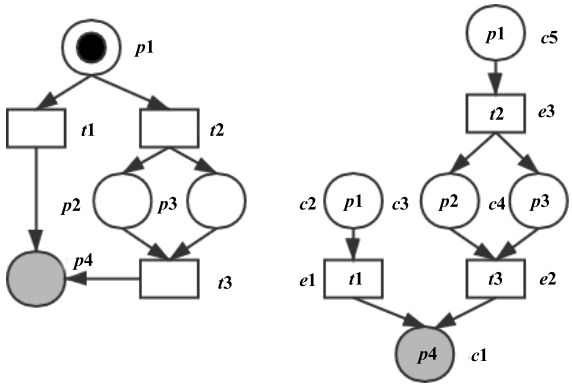
For the marking M_f, if there are markings M and M' as well as the triggered sequence σ satisfying $M \xrightarrow{\sigma} M' \wedge M_f \subseteq M'$, it is called that M_f can be covered by M, which is denoted as $M \mapsto M_f$.

If the reachable marking M satisfies $\forall p : M(p) \leq n$, M is called n-safe. A net system is n-safe when and only when all of its reachable markings are n-safe. Particularly, the 1-safe net systems are called as safe net systems. In this paper, we only focus on the coverability problem of 1-safe Petri nets and denote the target marking of coverability determination as M_f, namely that we verify whether $M_0 \mapsto M_f$ holds.

2.1.2 *Reverse occurrence nets*

Definition 1 (Reverse occurrence nets)**.** Reverse occurrence nets are a subclass of occurrence nets, which are used to determine the coverability of the target marking M_f in a Petri net. A reverse occurrence net is corresponding to a quadruple $\boldsymbol{RON} = (C, E, F', \boldsymbol{CM}_f)$, where C is a condition set, with each condition corresponding to a token of place in the Petri net; E is an event set, with each event corresponding to an execution of a transition in the Petri net; F' is the flow relation between C and E, which corresponds to the flow relation of the Petri net; and $\boldsymbol{CM}_f$ is the corresponding condition set of the target marking M_f of the Petri net in ***RON*** and satisfies $\forall c \in \boldsymbol{CM}_f : c^\bullet = \varnothing$.

With the Petri net shown in Figure 3(a) and the target marking $M_f = \{p4\}$ as an example, its corresponding reverse occurrence net is present in Figure 3(b), where $\boldsymbol{CM}_f = \{c1\}$.



(a) Petri net (b) Reverse occurrence net

Figure 3 An example of Petri net and its reverse occurrence net

There are three relationships between two nodes x_1 and x_2 in RON.

(1) Reverse causality: If a path starting from x_1 can reach x_2, it is denoted as $x_2 \leq x_1$. Particularly, for a node x, there is $x = x$. In Figure 3(b), there are $c1 \leq c5$ and $e2 \leq e3$.

(2) Reverse conflict: If there are two different events $e_1, e_2 \in E, e_1^\bullet \cap e_2^\bullet \neq \varnothing \wedge e_1 \leq x_1 \wedge e_2 \leq x_2, x_1$ and x_2 are called reverse conflict, which is denoted as $x_1 \# x_2$. In Figure 3(b), $c2$

and $c3$ are subjected to reverse conflict because $e1 \leq c2, e2 \leq c3$, and $e1^{\bullet} \cap e2^{\bullet} = \{c1\} \neq \varnothing$. Similarly, reverse conflicts are found between $c2$ and $c4$ and between $e1$ and $e2$.

(3) Reverse concurrency: If $\neg(x_1 \leq x_2 \vee x_2 \leq x_1 \vee x_1 \# x_2)$, x_1 and x_2 are subjected to reverse concurrency, which is denoted as $x_1 \| x_2$. In Figure 3(b), reverse concurrency is found between $c3$ and $c4$.

RON satisfies the following three properties.

(1) $\forall c \in C : |c^{\bullet}| \leq 1$;

(2) There is no reverse self-conflict in RON, namely that there is no event $e \in E$ making $e \# e$.

(3) F' is loop-free, namely that the reflexive and (irreflexive) transitive closure of F' is a partial order.

Definition 2 (Reverse configuration)**.** The reverse configuration *Cfg* of *RON*, which is a set of several events, satisfies the following two properties.

(1) If an event $e \in$ *Cfg*, then $\forall e' < e : e' \in$ *Cfg*;

(2) There are no events subjected to reverse conflict in *Cfg*, namely $\forall e, e' \in$ *Cfg* : $\neg(e \# e')$.

Definition 3 (Reverse local configuration)**.** $\{e' | e' \in E \wedge e' < e\}$ is defined as the reverse local configuration of the event e, which is denoted as $[e]$.

In Figure 3(b), $[e1] = \{e1\}, [e3] = \{e2, e3\}$.

Definition 4 (Reverse cut)**.** For a configuration *Cfg*, its reverse cut is defined as *Cut*(*Cfg*) $=$ ($CM_f \cup {}^{\bullet}$*Cfg*)$\backslash$*Cfg*$^{\bullet}$. In addition, for a condition set, if any two elements are reversely concurrent, one of the two elements is called a co-set. It is not difficult to find that *Cut*(*Cfg*) is a co-set.

In Figure 3(b), *Cut*($[e2]$) $= \{c3, c4\}$, *Cut*($[e3]$) $= \{c5\}$.

Reverse configuration and reverse cut are used to establish the mapping relationship of markings between the reverse occurrence net and the Petri net in Section 2.1.3.

2.1.3 *Reverse unfolding*

For a given Petri net $\Sigma = (P, T, F, M_0)$, the coverability of the target marking M_f should be verified. The mapping relationship $\mu : C \cup E \to P \cup T$ between nodes in Σ and *RON* $= (C, E, F', CM_f)$ is defined as follows.

(1) If $c \in C$, then $\mu(c) \in P$; if $e \in E$, then $\mu(e) \in T$;

(2) $\forall e \in E$, ${}^{\bullet}e$ to ${}^{\bullet}\mu(e)$ satisfies the bijective relationship under the constraint of μ, and $e^{\bullet}$ to $\mu(e)^{\bullet}$ satisfies the injective relationship under the constraint of μ. Differently from forward unfolding, in reverse unfolding, $e^{\bullet}$ to $\mu(e)^{\bullet}$ may not satisfy the surjective relationship.

(3) CM_f and M_f satisfy the bijective relationship under the constraint of μ.

Definition 5 (Reverse marking)**.** *Mark*(*Cfg*) $= \mu($*Cut*(*Cfg*)$)$ is defined as the reverse marking of the configuration *Cfg*.

Mark(*Cfg*) can be viewed as an intermediate marking of *RON*, and the coverability determination of M_f can be converted to the coverability determination of *Mark*(*Cfg*)[19].

In Figure 3(b), *Mark*($[e2]$) $= \mu(\{c3, c4\}) = \{p2, p3\}$, *Mark*($[e3]$) $= \mu(\{c5\}) = \{p1\}$.

Definition 6 (Reverse unfolding)**.** On the basis of the above concepts, the reverse unfolding of the target marking M_f in the Petri net Σ is defined as a 2-tuple *RUnf*(*RON*, μ), which satisfies the following properties:

(1) *RUnf* is complete: Let the initial marking of Σ be M_0. If $M_0 \mapsto M_f$, there is a reverse configuration *Cfg* in *RUnf* which satisfies *Mark*(*Cfg*) $\subseteq M_0$. Here we just need to make sure that the coverability of M_f is not broken, and we do not need to obtain all triggered sequences.

(2) *RUnf* is finite, namely that *RUnf* includes finite conditions and events.

2.2 Reverse unfolding algorithm

With a given Petri net $\Sigma = (P, T, F, M_0)$ and a given target marking M_f whose coverability needs to be determined, the basic principle of the reverse unfolding algorithm is as follows. First, for each place of M_f, a corresponding condition is added to *RUnf*, that is, create CM_f. Specifically, after CM_f is created, reverse extension is conducted from CM_f. The events that can generate these conditions as well as the conditions that enable these events are added, and the redundant events are cut off. The above steps are repeated until a reverse marking can be covered by the initial marking of Σ, or until the target marking is proved to be uncoverable.

Next, we provide the exact definitions of reverse extension and reverse cut-off events and introduce the approach of determining the coverability of target markings with the example shown in Figure 4. It is assumed that the target marking to be determined in Figure 4 is $\{p6, p7\}$.

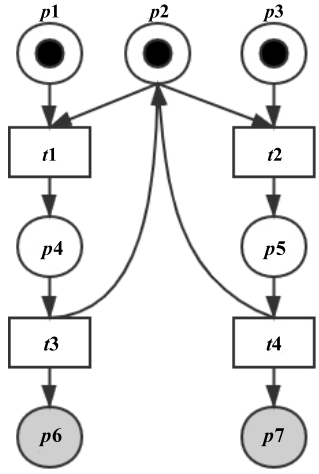


Figure 4 An example of Petri net

2.2.1 *Reverse extension*

Definition 7 (Reverse extension)**.** A reverse extension is a 2-tuple $rext = (t, C)$, where C is a co-set, and $\mu(C) \subseteq t^\bullet$. The set composed of reverse extensions is denoted as *RExt*.

For each reverse extension $rext = (t, C)$ selected from *RExt* and used to expand *RUnf*, an event $e = (t, C)$ needs to be added to *RUnf*. Meanwhile, a condition is added to *RUnf* for each place of ${}^\bullet t$. Then *RExt* is recalculated. This process is denoted as *NE*(*RUnf*, e).

In *NE*(*RUnf*, e), the candidate extension set is calculated as follows. The introduction of ${}^\bullet e$ adds a new co-set for *RUnf*. For a newly added co-set C, if there is a transition t satisfying $\mu(C) \subseteq t^\bullet$, then a new candidate extension $rext = (t, C)$ is added to *RExt*.

According to the above definitions, there are massive redundant extensions in *RExt*. For example, if it is assumed that $\{c1, c2, c3\}$ is a co-set, according to the definition of a co-set, $\{c1, c2\}$, $\{c1, c3\}$, $\{c2, c3\}$, $\{c1\}$, $\{c2\}$, and $\{c3\}$ are all co-sets. If there is a reverse extension $(t, \{c1, c2, c3\})$, there are reverse extensions $(t, \{c1, c2\})$, $(t, \{c1, c3\})$, $(t, \{c2, c3\})$, $(t, \{c1\})$, $(t, \{c2\})$, and $(t, \{c3\})$. Thus, if there are no additional constraints, the scale of *RExt* will be huge. As a result, we add the two following conditions for *RExt*.

(1) For an event e in *RUnf*, there is no reverse extension $rext = (t, C)$ in *RExt* which makes $\mu(e) = t \wedge e^{\bullet} = C$;

(2) For two reverse extensions $rext_1 = (t_1, C_1)$ and $rext_2 = (t_2, C_2)$ in *RExt*, if $t_1 = t_2 \wedge C_1 \subset C_2 \wedge Mark([rext_1]) \geq Mark([rext_2])$, we delete the extension $rext_1$ from *RExt*. Here we assume e is the corresponding event of *rext* in *RUnf*, and $[rext]$ can be viewed as $[e]$.

In condition (2), $Mark([rext_1]) \geq Mark([rext_2])$ seems to be unnecessary. In fact, Parosh proposed the concept of reverse unfolding in Reference [19], and only used the constraint $t_1 = t_2 \wedge C_1 \subset C_2$ in condition (2). However, this breaks the completeness of reverse unfolding and causes errors in some cases applying the reverse unfolding algorithm. Appendix B provides the analysis of relevant counterexamples. For this reason, we add the constraint $Mark([rext_1]) \geq Mark([rext_2])$. Many cases prove that it is effective.

In Figure 5, we use the reverse extension $rext = (t1, \{c3\})$ to generate the event $e2$ and yield the precondition $\{c4, c5\}$ of $e2$ at the same time. The addition of $\{c4, c5\}$ generates new co-sets as well as the corresponding reverse extensions $(t3, \{c5\})$, $(t4, \{c5\})$, and $(t4, \{c2, c5\})$. For $rext_1 = (t4, \{c5\})$ and $rext_2 = (t4, \{c2, c5\})$, there is $Mark([rext_1]) = \{p1, p5, p7\} \geq \{p1, p5\} = Mark([rext_2])$, and we delete $rext_1$ according to the condition (2) of *RExt*. We finally obtain $NE(RUnf, e) = \{(t3, \{c5\}), (t4, \{c2, c5\})\}$.

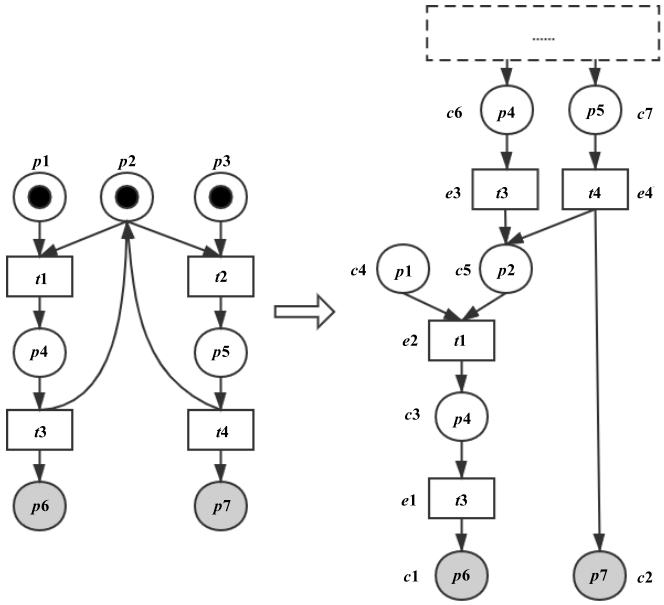


Figure 5 Reverse extension of Petri net shown in Figure 4 (partial)

2.2.2 *Reverse cut-off events*

The above reverse extension rule can assure the completeness of *RUnf*, but cannot guarantee its finiteness. Thus, the rule of the recognizing reverse cut-off events will be provided below to assure the terminability of the reverse unfolding processes.

Definition 8 (Reverse cut-off events)**.** The event e is a reverse cut-off event when and only when there is an event e' in *RUnf* satisfying the following two conditions.

(1) $Mark([e']) \leq Mark([e])$.

(2) $[e'] \prec [e]$.

where $\prec$ is a partial order defined on configurations, which is called an adequate order. It satisfies the following three conditions.

(1) $\prec$ is well-founded.

(2) $\prec$ is a refinement of $\subset$, and $Cfg_1 \subset Cfg_2$ means that $Cfg_1 \prec Cfg_2$.

(3) If $Mark(Cfg_1) \leq Mark(Cfg_2)$ and $Cfg_1 \prec Cfg_2$, for a prefix E_2 of Cfg_2, there is E_1 satisfying $Mark(Cfg_1 \oplus E_1) \leq Mark(Cfg_2 \oplus E_2)$ and $Cfg_1 \oplus E_1 \prec Cfg_2 \oplus E_2$. For a configuration Cfg, $Cfg \oplus E$ indicates that there is an event set E satisfying $Cfg \cap E = \varnothing$, where $Cfg \cup E$ is also a configuration. E is also called the prefix of Cfg.

In this paper, we use $\prec_r$ as an adequate order, and Theorem 1 in Appendix A can be referred to for the relevant proof. $Cfg_1 \prec_r Cfg_2$ is defined as

(1) $|Cfg_1| < |Cfg_2|$.

(2) $|Cfg_1| = |Cfg_2| \wedge Lex(\mu(Cfg_1)) < Lex(\mu(Cfg_2))$, where $\mu(Cfg)$ is a transition set of the configuration Cfg_2 mapping to Σ, and it is a multiset. $Lex(\mu(Cfg))$ sorts the transitions in $\mu(Cfg)$ by ID from small to large. This can be understood as when two configurations are of the same size, the lexicographical order of their corresponding transition sets is compared.

In Figure 5, the event $e3$ is cut off due to the event $e1$, as shown in Figure 6. This is because $|[e1]| < |[e3]|$, i.e., $[e1] \prec_r [e3]$, and $Mark([e1]) = \{p4, p7\} \leq \{p1, p4, p7\} = Mark([e3])$.

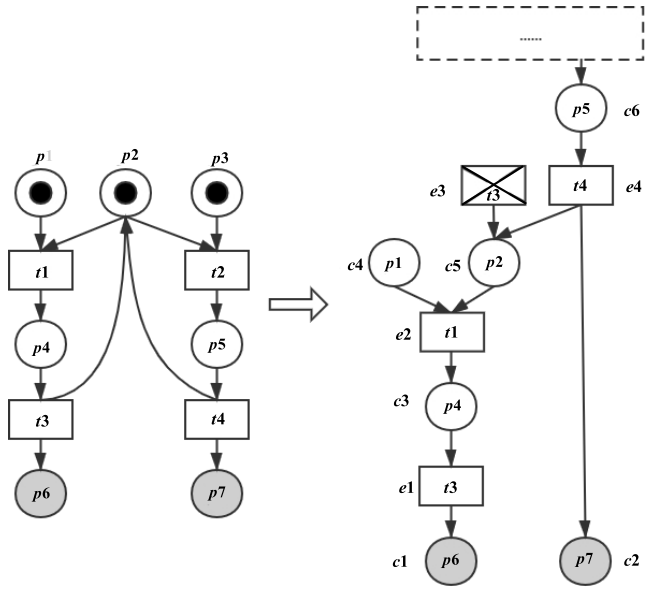


Figure 6 Cutting off event $e3$ by event $e1$

Reverse extension and reverse cut-off events guarantee the completeness and finiteness of *RUnf*, and theorems 2 and 3 in Appendix A can be referred to for the relevant proof.

2.2.3 *Coverability determination based on reverse unfolding*

In this paper, we only focus on 1-safe Petri nets, and we can add source place *ps*, source transition *ts*, and flow relation (*ps*, *ts*) to Petri nets. Moreover, we add the flow relation (ts, p) to each $p \in M_0$ to convert solving $M_0 \mapsto M_f$ to solving $\{p_s\} \mapsto M_f$. This only needs to determine whether there is a configuration *Cfg* in *RUnf* satisfying $Mark(Cfg) = \{ps\}$, after which the coverability of M_f can be verified. With the Petri net in Figure 4 as an example, the Petri net after the addition of source nodes is as shown in Figure 7.

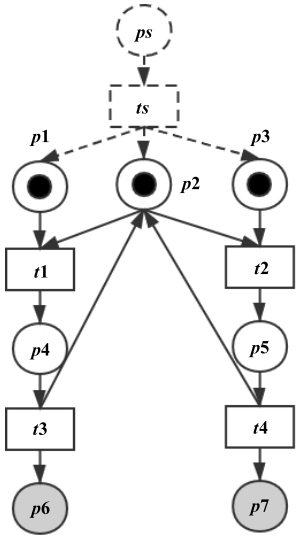


Figure 7 Petri net after source node addition

2.2.4 *Reverse unfolding algorithm and examples of coverability determination*

The flow of coverability determination of reverse unfolding can be summarized as follows. Initially, there is only the condition set CM_f corresponding to M_f in *Runf*, and the algorithm calculates the initial extension set *RExt* on the basis of CM_f. Subsequently, as long as *RExt* is not empty, the algorithm will continue to conduct reverse extension. In each extension, a $rext = (t, C)$ is randomly selected from *RExt*, and a corresponding event $e = (t, C)$ is created in *RUnf*. If e is not a reverse cut-off event, a relevant condition $c = (p, e)$ is created in *RUnf* for each place p in $^\bullet t$. Then, we update *RExt* by calculating $RExt = RExt \cup NE(RUnf, e)$. If there is an event e satisfying $Mark([e]) = \{ps\}$, it indicates that M_f is coverable, and the algorithm ends. If *RExt* is empty finally and there is no event e satisfying $Mark([e]) = \{ps\}$, it indicates M_f is uncoverable. The pseudo-code of the reverse unfolding algorithm is as follows.

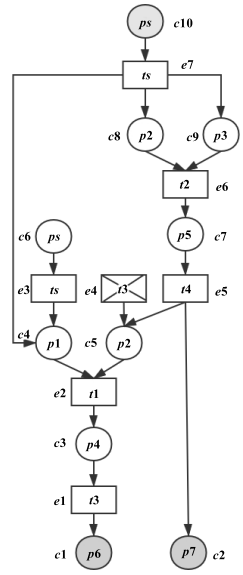
Algorithm 1. Reverse unfolding of Petri net and coverability determination algorithm of target marking

Input: A net system $\Sigma = (P, T, F, \{p_s\})$, with target $M_f = \{p_1, p_2, \cdots, p_n\}$.
Output: The coverability of M_f.

```
RUnf = {(p1, ∅), (p2, ∅), ··· , (pn, ∅)}
RExt = NE(RUnf, ∅)
while RExt ≠ ∅ do
    poll an extension rext = (t, C) from RExt randomly
    create an event e = (t, C) in RUnf
    if e is not a reverse cut-off event then
        for ∀p ∈ •t do
            Append a condition c = (p, e) to RUnf
        end for
        if Mark([e]) = {ps} then
            return true
        end if
        RExt = RExt ∪ NE(RUnf, e)
    end if
end while
return false
```

With the Petri net in Figure 7 as an example, the process of the coverability determination for the target marking $\{p6, p7\}$ by reverse unfolding is shown in Table 1.

Table 1 Example of reverse unfolding

	Process of reverse unfolding	*RExt*	*RUnf*
Step 0.	There are only conditions $c1$ and $c2$ in initial *RUnf*.	$\{rext1 = (t3, \{c1\}),$ $rext2 = (t4, \{c2\})\}$	
Step 1.	Select *rext*1 to extend and create the event $e1 = (t3, \{c1\})$ and its precondition $\{c3\}$ in *RUnf*.	$\{rext2 = (t4, \{c2\}),$ $rext3 = (t1, \{c3\})\}$	
Step 2.	Select *rext*3 to extend and create the event $e2 = (t1, \{c3\})$ and its precondition$\{c4, c5\}$ in *RUnf* to generate new extensions $(ts, \{c4\})$, $(t3, \{c5\})$, $(t4, \{c5\})$, $(t4, \{c2, c5\})$. Delete the extension $(t4, \{c5\})$ according to the constraints in *RExt*.	$\{rext2 = (t4, \{c2\}),$ $rext4 = (ts, \{c4\}),$ $rext5 = (t3, \{c5\}),$ $rext6 = (t4, \{c2, c5\})\}$	
Step 3.	Select *rext*4 to extend and create the event $e3 = (ts, \{c4\})$ and its precondition $\{c6\}$ in *RUnf*. There is no new extension at this point.	$\{rext2 = (t4, \{c2\}),$ $rext5 = (t3, \{c5\}),$ $rext6 = (t4, \{c2, c5\})\}$	
Step 4.	Select *rext*5 to extend and create the event $e4 = (t3, \{c5\})$ in *RUnf*. According to the definition of reverse cut-off events, $e4$ will be cut off due to $e1$, and this extension ends.	$\{rext2 = (t4, \{c2\}),$ $rext6 = (t4, \{c2, c5\})\}$	
Step 5.	Select *rext*6 to extend and create the event $e5 = (t4, \{c2, c5\})$ and its precondition $\{c7\}$ in *RUnf*.	$\{rext2 = (t4, \{c2\}),$ $rext7 = (t2, \{c7\})\}$	
Step 6.	Select *rext*7 to extend and create the event $e6 = (t2, \{c7\})$ and its precondition $\{c8, c9\}$ in *RUnf* to generate new extensions $(ts, \{c8\})$, $(ts, \{c9\})$, $(ts, \{c8, c9\})$, $(ts, \{c4, c8\})$, $(ts, \{c4, c9\})$, $(ts, \{c4, c8, c9\})$, $(t3, \{c8\})$, and $(t4, \{c8\})$. Only reserve extensions $(ts, \{c4, c8, c9\})$, $(t3, \{c8\})$, and $(t4, \{c8\})$ are reserved according to the constraints of *RExt*.	$\{rext2 = (t4, \{c2\}),$ $rext8 = (ts, \{c4, c8, c9\}),$ $rext9 = (t3, \{c8\}),$ $rext10 = (t4, \{c8\})\}$	
Step 7.	Select *rext*8 to extend and create the event $e7 = (ts, \{c4, c8, c9\})$ and its precondition $\{c10\}$ in *RUnf*. The target marking is coverable at this point, and the algorithm ends.		

The above shows the flow of the whole reverse unfolding algorithm. However, the algorithm currently does not specify the extension sequence, and only uses random extension. In fact, the extension sequence has a decisive impact on the efficiency of the reverse unfolding algorithm. Next, we optimize the reverse unfolding with heuristics.

2.3 Heuristic optimization of reverse unfolding algorithm

The extension sequence has a critical impact on the efficiency of coverability determination of target markings. For this purpose, this paper proposes three heuristic techniques based on practices by referring to Reference [17].

(1) The *block* strategy.

In the reverse extension $rext = (t, C)$, C and $t^\bullet$ may do not satisfy a surjective relationship. However, practices show that when the relationship between C and $t^\bullet$ is not surjective, it often means that *rext* is generated too early, and the corresponding conditions have not been yielded. At this point, the extensions guided by it will be redundant. To avoid this situation as far as possible, we add the reverse extensions that do not satisfy the surjective relationship to the blocking queue and select the reverse extensions that satisfy the surjective relationship with priority. Only when all reverse extensions do not satisfy the surjective relationship, a reverse extension is selected from the blocking queue for activation. In practice, the *block* strategy is usually used together with other heuristic strategies.

(2) The *hmax* strategy.

The *hmax* strategy is a distance-based heuristic strategy proposed by Bonet in Reference [17]. $d(M, M')$ is defined as a distance between markings M and M'. In the *hmax* strategy, the distance between M and the transition t is defined as $\max_{p\in t} d(M, \{p\})$, and the distance between M and the place p is defined as $1 + \min_{t\in p} d(M, t)$, and so on. When calculating $d(M, M')$, we only need to solve the maximum distance between M and each place in M', namely to calculate $\max_{p\in M'} d(M, \{p\})$.

The above process can be summarized as

$$d(M, M') = \begin{cases} 0, & M' \subseteq M \\ 1 + \min_{t\in p} d(M, t), & M' = \{p\} \\ \max_{p\in M'} d(M, \{p\}), & \text{otherwise} \end{cases}$$

(3) The *hsum* strategy.

hsum is very similar to *hmax*. The difference is that *hsum* defines the distance between M and the transition t as $\sum_{p\in t} d(M, \{p\})$. Correspondingly, when calculating $d(M, M')$, we sum up the distances between M and each place in M', namely calculate $\sum_{p\in M''} d(M, \{p\})$.

The above process can be summarized as

$$d(M, M') = \begin{cases} 0, & M' \subseteq M \\ 1 + \min_{t\in p} d(M, t), & M' = \{p\} \\ \sum_{p\in t} d(M, \{p\}), & \text{otherwise} \end{cases}$$

In practice, we will take into account the size of configurations when using the *hmax* and *hsum* strategies. Specifically, for two reverse extensions $rext_1 = (t_1, C_1)$ and $rext_2 = (t_2, C_2)$, if $|[rext_1]| + d(M_0, Mark([rext_1])) < |[rext_2]| + d(M_0, Mark([rext_2]))$, we will select $rext_1$ with priority for extension.

3 Data Race Detection Based on Reverse Unfolding

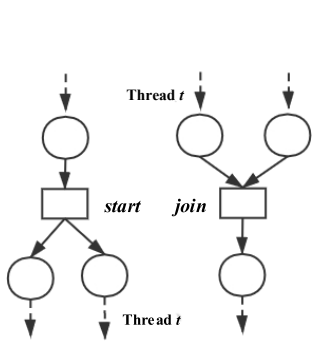
Due to the uncertainty of thread scheduling, multithreaded programs are often accompanied by data race. Data race means that multiple threads access the same memory address space in a non-thread-safe situation. It can affect program results and even lead to system crashes. Since data race usually occurs only in some specific thread traces, it poses a great challenge

for developers to detect data race. Several serious incidents in history, such as the radiation therapy machine Therac-25 accident[20], the massive blackout in North America in 2003[21], and the FaceBook failure in NASDAQ[22], were all related to data race. Data race detection is mainly divided into two categories: static detection[23–25] and dynamic detection[26–28]. In this section, we conduct the static detection of data race in Java concurrent programs based on reverse unfolding of Petri nets.

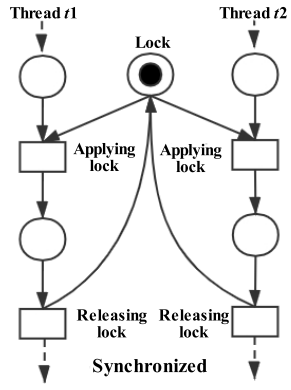
3.1 Building the Petri net model for programs

Krishna[29] built Petri net models for synchronization primitive and flow control statements of C-Pthread programs. In this paper, we apply this modelling method to Java concurrent programs. The models of the following four kinds of statements are built.

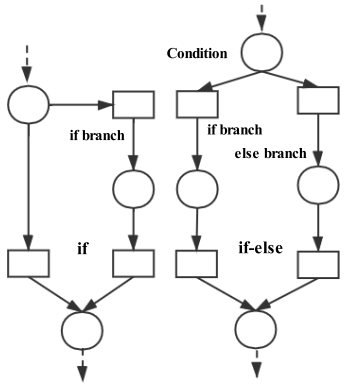
(1) Thread starting and merging: In Java, the starting and merging of the thread t correspond to $t.start()$ and $t.join()$. When $t.start()$ is invoked, the state of the thread t changes to Runnable. After the CPU scheduling is obtained, the thread t runs formally in the Running state. After $t.join()$ is invoked, the state of the current thread changes to Blocked, and changes to Runnable until the thread t is executed. Then it re-waits the CPU scheduling. The corresponding Petri net model of thread starting and merging is shown in Figure 8(a).



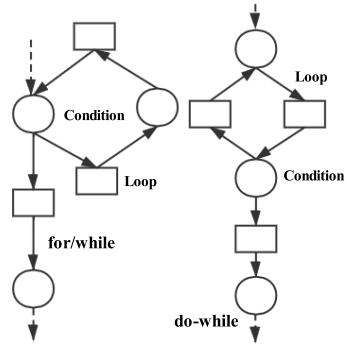
(a) Thread starting and merging



(b) Lock application and release



(c) Branches in programs



(d) Loops in programs

Figure 8 Petri net models of Java concurrent programs

(2) Lock application and release: Java uses synchronized statements to achieve mutual

exclusion of threads, including synchronized methods and synchronized code blocks, which are essentially the same. When the thread $t1$ and the thread $t2$ access the synchronized code block at the same time, only one thread can get access. The Petri net model of lock application and release is shown in Figure 8(b).

(3) Branches: Branches in Java are equivalent to if and if-else branches logically. The corresponding Petri net model is shown in Figure 8(c).

(4) Loops: Loops in Java programs are divided into for/while loops and do-while loops. The two have similar structures, and they both consist of a control condition and a loop body. The corresponding Petri net model is shown in Figure 8(d).

Figure 9 shows an example of Java multithreaded program and its Petri net model constructed in line with the above rules. The transition with an asterisk corresponds to a program statement, and other transitions are merely used for indicating program structure. Specifically,

(1) For the program statements, $t1$ indicates creating the thread $t1$; $t6$ indicates creating the thread $t2$; $t14$ indicates merging the thread $t1$ to the main thread; $t7$ and $t8$ indicate applying the lock; $t15$ and $t16$ indicates releasing the lock; $t9$ corresponds to the statement $x = 2$; $t10$ corresponds to the statement $x = 1$; $t11$ corresponds to the statement *System.out.println*(x).

(2) In terms of program structure, $t2$ and $t4$ indicates the entry into the if structure; $t13$ and $t17$ indicates the exit from the if structure; $t3$ indicates the beginning of a loop; $t5$ indicates the end of a loop; $t12$ indicates the exit from a loop.

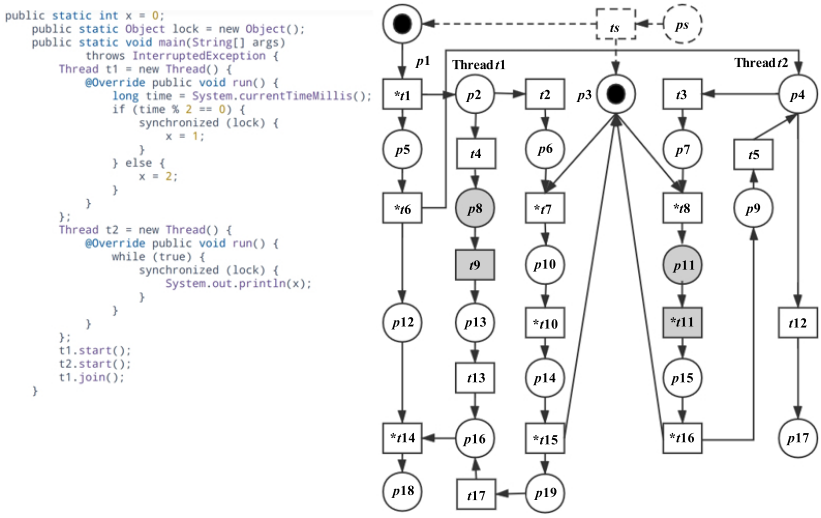


Figure 9 Java source code (left) and its Petri net model (right)

3.2 Data race detection based on coverability determination

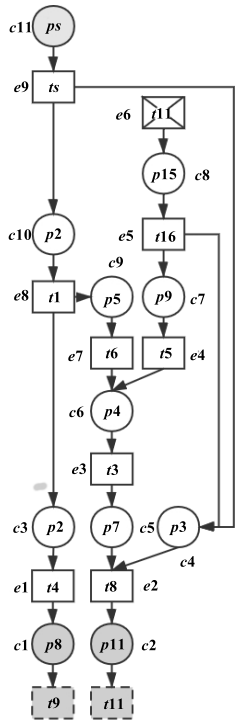
Two read/write or write/write operations on a same shared variable in a program do not result in data race if they have sequential dependencies in the execution order. On the contrary, if they are concurrent, it will lead to data race. In view of this, we can analyze the Petri net model of the program to determine whether the transitions corresponding to the two operations have the concurrent relationship in some states. In fact, for any two transitions t and t', we only need to determine whether there is a reachable marking M satisfying $M \geq {}^\bullet t + {}^\bullet t'$, namely to determine the coverability of the marking ${}^\bullet t + {}^\bullet t'$.

For the example shown in Figure 9, data race occurs if the operation of reading the shared variable *System.out.println*(x) and the operation of writing the shared variable $x = 2$ do not

have a same lock. The write operation $x = 2$ of the shared variable x corresponds to the transition $t9$, and the read operation *System.out.println*(x) corresponds to the transition $t11$. To determine data race, we only need to verify the coverability of the target marking $\{p8, p11\}$.

The reverse unfolding flow of the Petri net model of the program is shown in Table 2.

Table 2 Example of data race detection for concurrent programs

Process of reverse unfolding	*RExt*	*RUnf*
Step 0. There are only conditions $c1$ and $c2$ in initial RUnf.	$\{rext1 = (t4, \{c1\}),$ $rext2 = (t8, \{c2\})\}$	
Step 1. Randomly select *rext*1 to extend and create $e1$ and $c3$.	$\{rext2 = (t8, \{c2\}),$ $rext3 = (t1, \{c3\})\}$	
Step 2. Select *rext*2 to extend and create $e2$, $c4$, and $c5$.	$\{rext3 = (t1, \{c3\}),$ $rext4 = (t15, \{c4\}),$	
Step 3. Select *rext*7 to extend and create $e3$ and $c6$.	$rext5 = (t16, \{c4\}),$ $rext6 = (ts, \{c4\}),$	
Step 4. Select *rext*9 to extend and create $e4$ and $c7$. The extensions $(t16, \{c7\})$ and $(t16, \{c4, c7\})$ are newly added. Delete $(t16, \{c7\})$ and *rext*5 according to the conditions of *RExt*.	$rext7 = (t3, \{c5\})\}$ $\{rext3 = (t1, \{c3\}),$ $rext4 = (t15, \{c4\}),$ $rext5 = (t16, \{c4\}),$ $rext6 = (ts, \{c4\}),$ $rext8 = (t6, \{c6\}),$	
Step 5. Select *rext*10 to extend and create $e5$ and $c8$.	$rext9 = (t5, \{c6\})\}$ $\{rext3 = (t1, \{c3\}),$ $rext4 = (t15, \{c4\}),$ $rext6 = (ts, \{c4\}),$ $rext8 = (t6, \{c6\}),$	
Step 6. Select *rext*11 to extend and create $e6$. According to the definition of reverse cut-off events, $e6$ will be cut off due to the initial marking of *RUnf*, and this extension ends.	$rext10 = (t16, \{c4, c7\})\}$ $\{rext3 = (t1, \{c3\}),$ $rext4 = (t15, \{c4\}),$ $rext6 = (ts, \{c4\}),$ $rext8 = (t6, \{c6\}),$ $rext11 = (t11, \{c8\})\}$	
Step 7. Select *rext*8 to extend and create $e7$ and $c9$. The extensions $(t1, \{c9\})$ and $(t1, \{c3, c9\})$ are newly added. Delete $(t1, \{c9\})$ according to the conditions of *RExt*.	$\{rext3 = (t1, \{c3\}),$ $rext4 = (t15, \{c4\}),$ $rext6 = (ts, \{c4\}),$ $rext8 = (t6, \{c6\}),$ $rext10 = (t16, \{c4, c7\})\}$	
Step 8. Select *rext*12 to extend and create $e8$ and $c10$. The extensions $(ts, \{c10\})$ and $(ts, \{c4, c10\})$ are newly added. Delete $(ts, \{c10\})$ and *rext*6 according to the conditions of *RExt*.	$\{rext3 = (t1, \{c3\}),$ $rext4 = (t15, \{c4\}),$ $rext6 = (ts, \{c4\}),$ $rext8 = (t6, \{c6\})\}$ $\{rext3 = (t1, \{c3\}),$ $rext4 = (t15, \{c4\}),$ $rext6 = (ts, \{c4\}),$	
Step 9. Select *rext*13 to extend and create $e9$ and $c11$. The target marking is coverable at this point, and the algorithm ends.	$rext12 = (t1, \{c3, c9\})\}$ $\{rext3 = (t1, \{c3\}),$ $rext4 = (t15, \{c4\}),$ $rext13 = (ts, \{c4, c10\})\}$	

It is not difficult to find that $Mark([e9]) = \{ps\}$. Thus the target marking $\{p8, p11\}$ is coverable. The transitions corresponding to the read and write operations can be concurrent, and the program has potential data race.

In this section, we build the Petri net model of a Java concurrent program and transform the data race detection to the coverability verification of relevant transitions. With an example, we show the application of the reverse unfolding algorithm in data race detection. However, this paper only provides the Petri net modelling method for Java multithreaded programs, but does not design a modelling tool. In practice, Soot[30] can be used for the automatic modelling of Java concurrent programs. Soot is a Java bytecode optimization framework, by which the Java source code can be converted into Jimple intermediate code for analysis. For example, we can use JInvokeStmt to analyze the starting and merging of threads and to invoke many functions; we can adopt JEnterMonitorStmt and JExitMonitorStmt to analyze lock application and release;

we can leverage JIfStmt and JGotoStmt to analyze branches and loops in programs. Limited by space, we will study this in our future work.

4 Experimental Evaluation

In this section, we compare the efficiency of heuristic reverse unfolding and directed unfolding[17] (a forward unfolding also using heuristics) on the coverability problem of Petri nets. In this paper, the algorithm efficiency is evaluated by the total number of events and operation time. The reference library used in the experiment contains 415 test cases, including 358 cases of directed unfolding and 57 cases constructed by us. All of them are coverable. The number of transitions of Petri nets varies from 6 to 9,462, and there are 261,694 transitions in total. The experiment is conducted on a machine equipped with AMD Ryzen 7 4800U and 16 GB RAM. The source code and test cases can be acquired from Reference [31].

4.1 Comparison between reverse unfolding and forward unfolding without heuristic strategy

As the heuristic strategy has a great impact on the algorithm efficiency, it is unfavorable to visualize the property advantages of reverse unfolding. Thus, this paper first uses the basic breadth-first and depth-first strategies to compare the two algorithms (Reference [18] can be referred to for the forward unfolding). The case set randomtree in the experiment constructs 37 Petri nets in a tree structure. Each node in the tree is a cyclic model similar to that in Figure 8(d). We conduct coverability determination by randomly selecting markings of leaf nodes. The maximum number of extension times of algorithms is set as 10,000, and the maximum operation time is 35 s. The experimental results are shown in Table 3.

The results show that the reverse unfolding based on the depth-first strategy performs best in randomtree. The scale of $|E|$ is optimal on all 37 cases. Meanwhile, the efficiency of reverse unfolding is higher than that of forward unfolding no matter depth-first or breadth-first strategies are used. This is because randomtree has a tree structure. The forward unfolding implicitly portrays the complete behavior of the system and covers all the branches of the tree structure. In contrast, the reverse unfolding only describes the system states related to coverability determination and covers only a branch of the tree.

4.2 Comparison of reverse unfolding and forward unfolding in the presence of heuristic strategies

In this section, we compare the efficiencies of heuristic reverse unfolding and directed unfolding in the coverability problem of Petri nets. In realizing directed unfolding, we use the heuristic strategies *hmax* and *hsum* in Reference [17] and the property that the extension sequence can be separated from the partial order of cut-off events in Reference [18]. In implementing reverse unfolding, we use the heuristic strategies *block*, *hmax*, and *hsum* stated in Section 2.3. In addition, implementation details of the two methods are kept consistent as far as possible. For comparison purposes, we only select the optimal heuristic strategy in directed unfolding and reverse unfolding for each group of experiments. The relevant experiments are as follows.

(1) randomtree

In Section 4.1, we only compare the efficiencies of forward unfolding and reverse unfolding without heuristics. Herein, we compare them under the heuristic technology. The maximum number of extension times is set as 10,000, and the maximum operation time is 35 s. The experimental results are shown in Table 4 and Figure 10(a).

The results show that the reverse unfolding based on the *block*+ depth-first strategy performs the best in randomtree. The scale of $|E|$ is optimal in all 37 cases. It should be noted that the

efficiency of the forward unfolding combined with the *hmax* strategy is improved significantly compared with that without the heuristic strategy. This indicates, to some extent, the heuristic technique can compensate for property deficiencies of algorithms.

Table 3 Results of forward unfolding and reverse unfolding in randomtree without heuristic strategy

Test case	$\|T\|$	*Unf-bfs*		*Unf-dfs*		*RUnf-bfs*		*RUnf-dfs*	
		$\|E\|$	Time (ms)	$\|E\|$	Time (ms)	$\|E\|$	Time (ms)	$\|E\|$	Time (ms)
randomtree100	899	500	64	448	125	36	34	**29**	42
randomtree125	1,194	318	20	1,291	309	23	25	**18**	15
randomtree150	1,387	656	98	428	21	38	15	**29**	18
randomtree175	1,729	1,165	241	1,082	115	42	14	**34**	17
randomtree200	1,854	886	60	1,721	197	36	5	**27**	12
randomtree225	2,130	134	3	4,640	2,575	15	3	**12**	2
randomtree250	2,429	1,278	106	1,125	96	34	44	**26**	4
randomtree275	2,575	1,390	148	2,309	415	43	8	**34**	5
randomtree300	2,898	1,690	248	1,231	132	46	7	**37**	23
randomtree325	3,132	1,359	143	1,238	129	33	22	**28**	4
randomtree350	3,492	2,128	330	383	20	40	12	**31**	8
randomtree375	3,615	2,134	404	2,082	361	39	5	**31**	5
randomtree400	3,712	1,658	206	5,942	7,139	37	4	**31**	4
randomtree425	4,076	1,166	91	9,642	20,789	38	3	**29**	4
randomtree450	4,304	2,030	296	2,600	611	45	10	**33**	17
randomtree475	4,523	613	30	4,801	2,654	25	4	**20**	2
randomtree500	4,732	1,476	141	2,887	887	34	24	**27**	5
randomtree525	5,014	1,828	204	5,781	6,730	37	4	**30**	4
randomtree550	5,160	2,059	308	159	4	38	3	**30**	18
randomtree575	5,394	1,008	77	8,720	13,284	36	5	**27**	14
randomtree600	5,739	2,610	483	–	–	42	7	**32**	18
randomtree625	5,835	1,314	123	891	70	29	4	**24**	7
randomtree650	6,238	2,760	651	9,494	24,708	43	4	**31**	4
randomtree675	6,423	2,829	630	6,615	10,874	41	4	**31**	5
randomtree700	6,529	1,427	133	6,257	8,508	34	4	**26**	5
randomtree725	7,026	4,759	3,183	–	–	63	8	**51**	10
randomtree750	7,133	2,853	688	2,061	280	52	9	**38**	7
randomtree775	7,415	5,005	5,516	2,340	525	63	12	**49**	9
randomtree800	7,673	294	12	232	7	24	16	**20**	3
randomtree825	7,689	2,557	665	–	–	37	7	**29**	6
randomtree850	8,030	267	10	5,549	6,056	21	3	**15**	5
randomtree875	8,224	2,182	373	–	–	34	4	**25**	6
randomtree900	8,402	3,507	1,247	9,323	20,989	34	5	**28**	8
randomtree925	8,762	5,305	4,789	2,551	442	60	8	**47**	10
randomtree950	9,135	6,247	7,353	–	–	75	10	**59**	12
randomtree975	9,491	6,274	8,860	5,583	6,217	72	10	**52**	11
randomtree1000	9,462	6,025	8,618	7,241	12,960	59	7	**43**	9

(2) threadlock

The case set threadlock simulates a thread-lock model according to the following rules. It is assumed that there are x threads and y locks. The x threads successively apply lock 1, lock $2, \cdots$, lock y, and then successively release lock y, lock $y-1, \cdots$, lock 1. The target marking is composed of the final places of these threads. The maximum number of extension times is set as 10,000, and the maximum operation time is 70 s. The experimental results are shown in Table 5 and Figure 10(b).

The results show that the forward unfolding based on the depth-first strategy performs the best in threadlock. Among 20 cases, the scale of $|E|$ is the optimal in 15 cases and is similar to that of the reverse unfolding based on the *block* + depth-first strategy in other 5 cases. In addition, with the increase in the number of cases, the operation time of forward unfolding is

much superior to that of reverse unfolding. However, in threadlock, the number of forward branches of Petri nets is similar to that of reverse branches, which makes their efficiency close in theory. However, this is not consistent with the experimental results. The further analysis demonstrates that as the number of cases rises, the number of redundant extensions in *RExt* increases rapidly, resulting in a slow operation speed of reverse unfolding. Additionally, a huge extension set is harmful for heuristics to make the correct selection. Thus, how to effectively reduce the number of redundant extensions is a key factor to improve the efficiency of reverse unfolding.

Table 4 Results of directed unfolding and reverse unfolding in randomtree

Test case	$\|T\|$	*Unf-hmax*		*RUnf-blcok+dfs*	
		$\|E\|$	Time (ms)	$\|E\|$	Time (ms)
randomtree100	899	30	55	**29**	40
randomtree125	1,194	19	12	**18**	14
randomtree150	1,387	32	21	**29**	11
randomtree175	1,729	38	32	**34**	18
randomtree200	1,854	31	38	**27**	6
randomtree225	2,130	13	23	**12**	29
randomtree250	2,429	28	47	**26**	5
randomtree275	2,575	36	49	**34**	7
randomtree300	2,898	39	32	**37**	5
randomtree325	3,132	29	29	**28**	4
randomtree350	3,492	34	45	**31**	5
randomtree375	3,615	35	45	**31**	4
randomtree400	3,712	33	28	**31**	4
randomtree425	4,076	32	34	**29**	15
randomtree450	4,304	34	55	**33**	44
randomtree475	4,523	22	37	**20**	15
randomtree500	4,732	29	85	**27**	6
randomtree525	5,014	32	90	**30**	3
randomtree550	5,160	33	133	**30**	4
randomtree575	5,394	31	150	**27**	3

Table 5 Results of directed unfolding and reverse unfolding in threadlock

Test case	$\|T\|$	*Unf-dfs*			*RUnf-blcok+dfs*		
		$\|E\|$	Time (ms)	$\|Ext\|$	$\|E\|$	Event (ms)	$\|RExt\|$
threadlock2_1	6	**8**	20	1	**8**	10	5
threadlock3_1	10	**12**	7	3	**12**	8	11
threadlock3_2	16	**18**	27	3	21	14	23
threadlock4_1	14	**16**	19	6	**16**	10	19
threadlock4_2	22	**24**	7	6	28	8	41
threadlock4_3	30	**32**	8	6	40	7	60
threadlock5_1	18	**20**	9	10	**20**	2	29
threadlock5_2	28	**30**	5	10	35	5	62
threadlock5_3	38	**40**	8	10	50	7	91
threadlock6_1	22	**24**	5	15	**24**	3	43
threadlock6_2	34	**36**	20	15	42	5	84
threadlock6_3	46	**48**	23	15	60	25	129
threadlock8_4	78	**80**	29	28	104	39	291
threadlock10_5	118	**120**	34	45	160	87	554
threadlock12_6	166	**168**	87	66	228	105	943
threadlock14_7	222	**224**	85	91	308	223	1,474
threadlock16_8	286	**288**	72	120	400	161	2,186
threadlock18_9	358	**360**	70	153	504	399	3,088
threadlock20_10	438	**440**	131	190	620	497	4,211
threadlock50_25	2,598	**2,600**	3,746	1,225	3,800	49,214	63,785

(3) Reference libraries of directed unfolding

The directed unfolding uses four case sets, i.e., dartes, random, airport, and openstacks. We use them to perform experiments. The maximum number of extension times of the four use case sets is selected as 10,000, and the maximum operation time is 35 s.

The experimental results of the case set dartes are shown in Figure 10(c). In forward unfolding, the *hsum* strategy performs the best, and the coverability of 256/257 cases is verified. In reverse unfolding, the *hmax* strategy has the best effect, which verifies the coverability of 257/257 cases. Further, the scale of $|E|$ in reverse unfolding achieves the best effect in 48 cases and is comparable to that of forward unfolding in 2 cases. In this case set, forward unfolding and reverse unfolding both develop their own advantages.

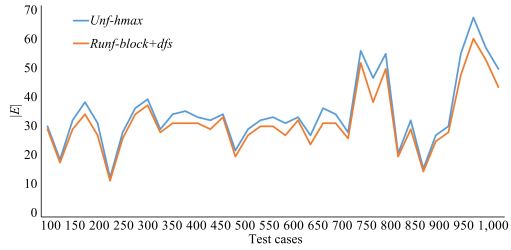
In the case set random, the forward unfolding combined with the *hsum* strategy performs the best, and the coverability of 33/45 use cases is verified. In reverse unfolding, the *block* + *hsum* strategy is outstanding, which verifies the coverability of 26/45 cases. Further, the scale of $|E|$ in forward unfolding performs the best in 33 cases and is comparable to that of reverse unfolding in 12 use cases. In the subsequent tests on airport and openstacks, the forward unfolding combined with the *hsum* strategy verifies the coverability of 19/26 and 30/30 cases, respectively, while reverse unfolding cannot verify the coverability of any cases. The low efficiency or even failure regarding these cases indicates that reverse unfolding is not applicable to all scenarios, which is determined by its property. However, the heuristic technique can compensate for deficiencies of algorithms in properties. Thus, we should design more effective heuristics to improve the low efficiency or even failure of reverse unfolding.

In this section, we compare the efficiencies of heuristic reverse unfolding and directed unfolding in coverability determination of Petri nets. The results show that in 415 groups of test data, the scale of reverse unfolding is better than that of forward unfolding for 85 data groups and is comparable to that of forward unfolding for 26 data groups. The experiment verifies that reverse unfolding is effective in some cases. If a Petri net has many forward branches, reverse unfolding can start from the target marking and only describes the system states related to coverability determination, so that the efficiency of coverability determination of target markings can be improved. However, reverse unfolding still has a low efficiency in most cases currently due to the following two reasons. The first is the number of redundant extensions. Too many redundant extensions result in slow operation of the algorithm and the heuristic cannot help to make correct selections. The second is the algorithm properties. When there are too many reverse branches in a Petri net, the efficiency of reverse unfolding is always lower than that of forward unfolding. However, the heuristic technique can compensate for property deficiencies of algorithms. Thus, we should design more effective heuristics to improve the low efficiency or failure of reverse unfolding.

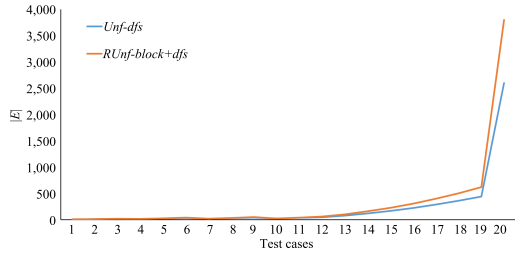
5 Conclusion and Prospect

This paper proposed a target-oriented reverse unfolding algorithm for the coverability determination of safe Petri nets. Starting from the target marking that needs coverability determination, reverse unfolding only describes the system states related to coverability determination. In addition, it reduces the unfolding scale with heuristic techniques such as *block*, *hmax*, and *hsum* strategies, so as to improve the efficiency of coverability determination of target markings. Further, we apply the reverse unfolding algorithm to formal verification of concurrent programs and convert data race detection of concurrent programs to coverability determination of specific markings of Petri nets. We compare the efficiencies of heuristic reverse unfolding and directed unfolding in coverability of Petri nets by experiments. It is verified that reverse unfolding can improve the coverability determination efficiency when Petri nets have many forward branches.

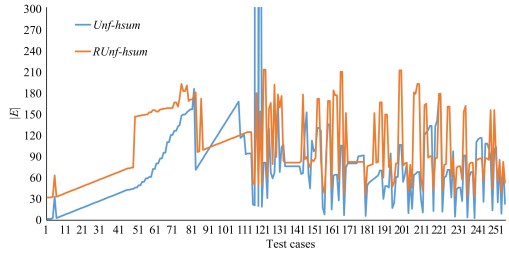
However, this paper still has some deficiencies. In terms of algorithm efficiency, we should further reduce the number of redundant extensions and design more efficient heuristic techniques. Meanwhile, we can try to combine forward unfolding with reverse unfolding to integrate their advantages. In terms of data race detection, other static detection algorithms can be used to initially confirm the positions of statements where data race may occur, and then reverse unfolding algorithm is adopted for verification. Moreover, Reference [32] discussed robustness, compatibility, and deadlock detection of generalized concurrent systems. Subsequently, we will conduct studies on applications of reverse unfolding in these fields.



(a) Randomtree



(b) Threadlock



(c) Dartes

Figure 10 Comparison of $|E|$ in different test cases between directed unfolding and reverse unfolding

References

[1] Han L, Xing K, Chen X, Xiong F. A Petri net-based particle swarm optimization approach for scheduling deadlock-prone flexible manufacturing systems. Journal of Intelligent Manufacturing, 2018, 29(5): 1083–1096. [doi: 10.1007/s10845-015-1161-2].

[2] Hu L, Liu Z, Hu W, Wang Y, Tan J, Wu F. Petri-Net-Based dynamic scheduling of flexible manufacturing system via deep reinforcement learning with graph convolutional network. Journal of Manufacturing Systems, 2020, 55: 1–4. [doi: 10.1016/j.jmsy.2020.02.004].

[3] Fauzan AC, Sarno R, Yaqin MA. Performance measurement based on coloured Petri net simulation

of scalable business processes. Proc. of the 2017 4th Int'l Conf. on Electrical Engineering, Computer Science and Informatics (EECSI). IEEE, 2017. 1–6. [doi: 10.1109/EECSI.2017.8239121].

[4] Liu C, Zeng Q, Duan H, Wang L, Tan J, Ren C, Yu W. Petri net based data-flow error detection and correction strategy for business processes. IEEE Access, 2020, 8: 43265–43276. [doi: 10.1109/ACCESS.2020.2976124].

[5] Lu F, Tao R, Du Y, Zeng Q, Bao Y. Deadlock detection-oriented unfolding of unbounded Petri nets. Information Sciences, 2019, 497: 1–22. [doi: 10.1016/j.ins.2019.05.021].

[6] Xiang D, Liu G, Yan C, Jiang C. Detecting data inconsistency based on the unfolding technique of petri nets. IEEE Trans. on Industrial Informatics, 2017, 13(6): 2995–3005. [doi: 10.1109/TII.2017.2698640].

[7] McMillan KL. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer-Verlag, 1992. 164–177.

[8] Engelfriet J. Branching processes of Petri nets. Acta Informatica, 1991, 28(6): 575–591. [doi: 10.1007/BF01463946].

[9] Esparza J, Römer S, Vogler W. An improvement of McMillan's unfolding algorithm. Formal Methods in System Design, 2002, 20(3): 285–310. [doi: 10.1023/A:1014746130920].

[10] Khomenko V, Koutny M, Vogler W. Canonical prefixes of Petri net unfoldings. Acta Informatica, 2003, 40(2): 95–118. [doi: 10.1007/s00236-003-0122-y].

[11] Heljanko K, Khomenko V, Koutny M. Parallelisation of the Petri net unfolding algorithm. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer-Verlag, 2002. 371–385. [doi: 10.1007/3-540-46002-0_26].

[12] Benito FC, Kunzle LA. Unfolding for time Petri net. IEEE Latin America Transactions, 2017, 15(5): 1001–1008. [doi: 10.1109/TLA.2017.7912599].

[13] Schwarick M, Rohr C, Liu F, Assaf G, Chodak J, Heiner M. Efficient unfolding of coloured Petri nets using interval decision diagrams. Proc. of the Int'l Conf. on Applications and Theory of Petri Nets and Concurrency. Cham: Springer-Verlag, 2020. 324–344. [doi: 10.1007/978-3-030-51831-8_16].

[14] Dong L, Liu G, Xiang D. Verifying CTL with unfoldings of Petri nets. Proc. of the Int'l Conf. on Algorithms and Architectures for Parallel Processing. Cham: Springer-Verlag, 2018. 47–61. [doi: 1007/978-3-030-05063-4_5].

[15] Liu G, Reisig W, Jiang C, Zhou M. A branching-process-based method to check soundness of workflow systems. IEEE Access, 2016, 4: 4104–4118. [doi: 10.1109/ACCESS.2016.2597061].

[16] Chatain T, Paulevé L. Goal-Driven unfolding of Petri nets. arXiv: 1611.01296, 2016.

[17] Bonet B, Haslum P, Hickmott S, Thiébaux S. Directed unfolding of Petri nets. Proc. of the Trans. on Petri Nets and Other Models of Concurrency I. Berlin, Heidelberg: Springer-Verlag, 2008. 172–198. [doi: 10.1007/978-3-540-89287-8_11].

[18] Bonet B, Haslum P, Khomenko V, Thiébaux S, Vogler W. Recent advances in unfolding technique. Theoretical Computer Science, 2014, 551: 84–101. [doi: 10.1016/j.tcs.2014.07.003].

[19] Abdulla PA, Iyer SP, Nylén A. SAT-Solving the coverability problem for Petri nets. Formal Methods in System Design, 2004, 24(1): 25–43. [doi: 10.1023/B:FORM.0000004786.30007.f8].

[20] Leveson NG, Turner CS. An investigation of the Therac-25 accidents. Computer, 1993, 26(7): 18–41. [doi: 10.1109/MC.1993.274940].

[21] Poulsen K. Software bug contributed to blackout. Security Focus. 2004. http://www.securityfocus.com/news/8016.

[22] Joab J. Nasdaq's Facebook glitch came from 'race conditions'. 2012. http://www.computerworld.com/s/article/9227350.

[23] Blackshear S, Gorogiannis N, O'Hearn PW, Sergey I. RacerD: Compositional static race detection. Proc. of the ACM on Programming Languages, 2018.2(OOPSLA): 1–28. [doi: 10.1145/3276514].

[24] Chatarasi P, Shirako J, Kong M, Sarkar V. An extended polyhedral model for SPMD programs and its use in static data race detection. Proc. of the Int'l Workshop on Languages and Compilers for Parallel

Computing. Cham: Springer-Verlag, 2016. 106–120. [doi: 1007/978-3-319-52709-3_10]

[25] Bora U, Das S, Kukreja P, Joshi S, Upadrasta R, Rajopadhye S. Llov: A fast static data-race checker for OpenMP programs. ACM Trans. on Architecture and Code Optimization (TACO), 2020, 17(4): 1–26. [doi: 10.1145/3418597].

[26] Wilcox JR, Flanagan C, Freund SN. VerifiedFT: A verified, high-performance precise dynamic race detector. Proc. of the 23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. 2018. 354–367. [doi: 1145/3178487.3178514].

[27] Gu Y, Mellor-Crummey J. Dynamic data race detection for OpenMP programs. Proc. of the SC18: Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018. 767–778. [doi: 10.1109/SC.2018.00064].

[28] Lidbury C, Donaldson AF. Dynamic race detection for C++ 11. ACM SIGPLAN Notices, 2017, 52(1): 443–457. [doi: 1145/3093333.3009857].

[29] Kavi KM, Moshtaghi A, Chen DJ. Modeling multithreaded applications using Petri nets. Int'l Journal of Parallel Programming, 2002, 30(5): 353–371. [doi: 10.1023/A:1019917329895].

[30] Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot: A Java bytecode optimization framework. Proc. of the CASCON 1st Decade High Impact Papers. 2010. 214–224. [doi: 10.1145/1925805.1925818].

[31] https://github.com/Zongyin-Hao/Coverability

[32] Liu GJ. Meta-unfolding of Petri Nets: A Model Checking Method of Concurrent Systems. Beijing: Science Press, 2020 (in Chinese).

Appendix A

First, we provide the definition of adequate order. A partial order $\prec$ is an adequate order if it satisfies

(1) $\prec$ is well-founded;

(2) $\prec$ is refinement of $\subset$, $Cfg_1 \subset Cfg_2$ means $Cfg_1 \prec Cfg_2$;

(3) If $Mark(Cfg_1) \leq Mark(Cfg_2)$, $Cfg_1 \prec Cfg_2$, then for any prefix E_2 of Cfg_2, there is E_1 satisfying $Mark(Cfg_1 \oplus E_1) \leq Mark(Cfg_2 \oplus E_2)$ and $Cfg_1 \oplus E_1 \prec Cfg_2 \oplus E_2$.

Theorem 1. $\prec_r$ is an adequate order.

Proof: It is obvious that $\prec_r$ satisfies conditions (1) and (2) of adequate order, and we only need to prove $\prec_r$ satisfies condition (3). Specifically, we need to prove when $Mark(Cfg_1) \leq Mark(Cfg_2) \wedge Cfg_1 \prec_r Cfg_2$, for any prefix E_2 of Cfg_2, there is E_1 satisfying $Mark(Cfg_1 \oplus E_1) \leq Mark(Cfg_2 \oplus E_2)$ and $Cfg_1 \oplus E_1 \prec_r Cfg_2 \oplus E_2$.

E_k is defined as a prefix of the configuration Cfg with a size of k, $Cfg^k = Cfg \oplus E^k$. Then we develop proof by induction according to $|E_2|$. When $|E_2| = 0$, the conclusion is established obviously. When $|E_2| = k$, $Mark(Cfg_1^k) \leq Mark(Cfg_2^k)$ and $Cfg_1^k \prec_r Cfg_2^k$ can be obtained from the inductive assumption. When $|E_2| = k + 1$, for any transition t let $e_2 = (t, C_2)$, $\mu(C_2) \subseteq t \wedge C_2 \subseteq Cut(Cfg_2^k)$, so $Cfg_2^{k+1} = Cfg_2^k \cup \{e_2\}$; let $e_1 = (t, C_1)$, $\mu(C_1) \subseteq t \wedge C_1 \subseteq Cut(Cfg_1^k)$, so $Cfg_1^{k+1} = Cfg_1^k \cup \{e_1\}$. $\mu(C_1) \leq \mu(C_2)$ is required and C_1 should be as big as possible (C_1 can be $\varnothing$).

Then we only need to prove $Mark(Cfg_1^{k+1}) \leq Mark(Cfg_2^{k+1})$ and $Cfg_1^{k+1} \prec_r Cfg_2^{k+1}$. There are the following two situations.

(1) $C_1 = \varnothing$: $Cfg_1^k = Cfg_1^{k+1}$. As it is required that C_1 should be as big as possible, it can be known that $\forall p \in \mu(C_2)$, $p \notin Mark(Cfg_1^k)$. Thus, it can be obtained that $Mark(Cfg_1^{k+1}) = Mark(Cfg_1^k) \leq Mark(Cfg_2^k) - \mu(C_2) + t = Mark(Cfg_2^{k+1})$. In addition, due to $|Cfg_2^k| < |Cfg_2^{k+1}|$, according to the definition (1) of $\prec_r$, $Cfg_2^k \prec Cfg_2^{k+1}$, and thereby $Cfg_1^{k+1} = Cfg_1^k \prec_r Cfg_2^k \prec_r Cfg_2^{k+1}$. Thus, the conclusion holds when $C_1 = \varnothing$.

(2) $C_1 \neq \varnothing$: As it is required that C_1 should be as big as possible and $\mu(C_1) \leq \mu(C_2)$, it can be known that $\forall p \in \mu(C_2)$, and $p \in \mu(C_1) \vee p \notin \mathit{Mark}(\mathit{Cfg}_1^k)$. Thus, one can obtain $\mathit{Mark}(\mathit{Cfg}_1^{k+1}) = \mathit{Mark}(\mathit{Cfg}_1^k) - \mu(C_1) + t \leq \mathit{Mark}(\mathit{Cfg}_2^k) - \mu(C_2) + t = \mathit{Mark}(\mathit{Cfg}_2^{k+1})$. When $|\mathit{Cfg}_1^k| < |\mathit{Cfg}_2^k|$, there is $|\mathit{Cfg}_1^{k+1}| = |\mathit{Cfg}_1^k| + 1 < |\mathit{Cfg}_2^k| + 1 = |\mathit{Cfg}_2^{k+1}|$, $\mathit{Cfg}_1^{k+1} \prec_r \mathit{Cfg}_2^{k+1}$. When $|\mathit{Cfg}_1^k| = |\mathit{Cfg}_2^k|$ and $\mathit{Lex}(\mu(\mathit{Cfg}_1^k)) < \mathit{Lex}(\mu(\mathit{Cfg}_2^k))$, there is $|\mathit{Cfg}_1^{k+1}| = |\mathit{Cfg}_1^k| + 1 < |\mathit{Cfg}_2^k| + 1 = |\mathit{Cfg}_2^{k+1}|$. After two ordered strings with the same size are added with the same element and are re-ordered, their precedence relationship in the lexicographical order will not change, so $\mathit{Lex}(\mu(\mathit{Cfg}_1^{k+1})) < \mathit{Lex}(\mu(\mathit{Cfg}_2^{k+1}))$, $\mathit{Cfg}_1^{k+1} \prec_r \mathit{Cfg}_2^{k+1}$. Thus, the conclusion is established when $C_1 \neq \varnothing$.

To sum up, $\mathit{Mark}(\mathit{Cfg}_1 \oplus E_1) \leq \mathit{Mark}(\mathit{Cfg}_2 \oplus E_2)$ and $\mathit{Cfg}_1 \oplus E_1 \prec_r \mathit{Cfg}_2 \oplus E_2$. So $\prec_r$ is an adequate order.

Theorem 2. Reverse cut-off events will not break the completeness of *RUnf*.

Proof: For any reachable marking M satisfying $M \mapsto M_f$, there should be a configuration *Cfg* satisfying $\mathit{Mark}(\mathit{Cfg}) \subseteq M$. Assuming *Cfg* is not included in *RUnf*, *Cfg* must contain a reverse cut-off event e, and there is an event e' in *RUnf* satisfying $\mathit{Mark}([e']) \leq \mathit{Mark}([e]) \wedge [e'] \prec [e]$.

According to the condition (3) of adequate order, for $\mathit{Cfg} = [e] \oplus E$, there is $\mathit{Cfg}' = [e'] \oplus E'$ satisfying $\mathit{Mark}(\mathit{Cfg}') \leq \mathit{Mark}(\mathit{Cfg}) \leq M \wedge \mathit{Cfg}' \prec \mathit{Cfg}$. As $\prec$ is well-founded, the above process will finally find a minimum configuration in *RUnf*, and this is not consistent with the assumption. Thus, reverse cut-off events will not break the completeness of *RUnf*.

Theorem 3. *RUnf* is finite.

Proof: For any event e in *RUnf*, there is a longest chain $e_1 < e_2 < \cdots < e$, whose length is set as $d(e)$. The following three conclusions are drawn.

(1) For any condition c, ${}^\bullet c$ and $c^\bullet$ are finite. For any event e, ${}^\bullet e$ and $e^\bullet$ are finite.

(2) Let the number of reachable markings of a 1-safe Petri net be n; and there is $d(e) \leq n+1$. For a chain $e_1 < e_2 < \cdots < e_{n+1}$ with a size of $n+1$, there are definitely two events e_i and e_j $(i < j)$ which satisfy $\mathit{Mark}([e_i]) \leq \mathit{Mark}([e_j])$. In addition, due to $[e_i] \subset [e_j]$, $[e_i] \prec [e_j]$ are obtained according to the condition (2) of adequate order, where e_j is a reverse cut-off event. Thus there is no e in the chain which satisfies $e_j < e$, namely that the length of the chain cannot be greater than $n + 1$.

(3) For any $k \geq 0$, *RUnf* only includes finite event e satisfying $d(e) \leq k$.

It can be proved inductively that when $k = 0$, the conclusion holds obviously. Let E_k be the event set when $d(e) \leq k$, and the inductive assumption yields that E_k is finite. E_{k+1} is the event set when $d(e) \leq k + 1$. In light of $E_{k+1}^\bullet \subseteq {}^\bullet E_k \cup \mathit{CM}_f$ and conclusion (1), one can know E_{k+1} is finite.

Conclusions (2) and (3) indicate that *RUnf* includes finite events, and conclusion (1) shows that *RUnf* includes finite conditions, so *RUnf* is finite.

Appendix B

First, let us review the two conditions of *RExt*.

(1) For any event e in *RUnf*, there is no reverse extension $\mathit{rext} = (t, C)$ in *RExt* which satisfies $\mu(e) = t \wedge e^\bullet = C$.

(2) For any two different reverse extensions $\mathit{rext}_1 = (t_1, C_1)$ and $\mathit{rext}_2 = (t_2, C_2)$ in *RExt*, if $t_1 = t_2 \wedge C_1 \subset C_2 \wedge \mathit{Mark}([\mathit{rext}_1]) \geq \mathit{Mark}([\mathit{rext}_2])$, the extension rext_1 is deleted from *RExt*.

Parosh first proposed the principle of reverse unfolding in Reference [19], but did not add the constraint $Mark([rext_1]) \geq Mark([rext_2])$ to the extension rule, which thus results in the incompleteness of reverse unfolding. Here we analyze the counterexample.

We use the Petri net shown in Figure 11 and its reverse unfolding to illustrate. The left shows the original Petri net, and the target marking is $\{p12\}$. Obviously, the target marking is coverable. The right shows the reverse unfolding without the constraint $Mark([rext_1]) \geq Mark([rext_2])$. When the reverse unfolding algorithm generates the event $e8$, the corresponding reverse extension is $rext1 = (t8, \{c2, c9\})$, and $rext2 = (t8, \{c2\})$ will be deleted according to Parosh's rule. However, when analyzing the original Petri net, we find that the reverse marking $\{p4, p8\}$ corresponding to $rext1$ is uncoverable, and the reverse marking $\{p8, p11\}$ corresponding to $rext2$ is coverable. Further, the reverse marking corresponding to $rext2$ is necessary for determining the coverability of $\{p12\}$. At this point, it is impossible to verify the coverability of the target marking regardless of subsequent extensions, and the completeness of *RUnf* has been broken.

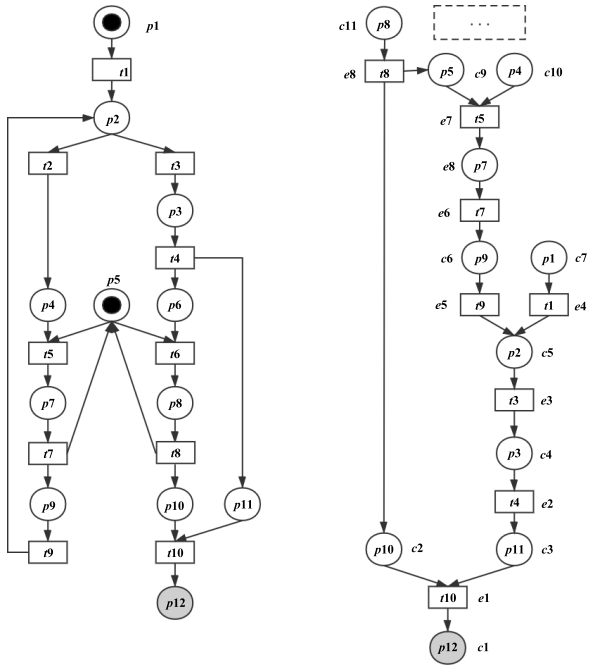


Figure 11 Example on completeness destruction of reverse unfolding

Zongyin Hao, master degree candidate, CCF student member. His research interests include the theory and applications of Petri nets and formal verification of concurrent program.

Faming Lu, Ph.D., associate professor, doctoral supervisor, CCF professional member. His research interests include the theory and applications of Petri nets, modeling and analysis of concurrent systems, business process management, and decision support.