

## RESEARCH ARTICLE

# Petri Net Unfolding-Based Detection and Replay of Program Deadlocks

FAMING LU<sup>1</sup>, FENGHUA LV<sup>1</sup>, MINGHAO CUI<sup>1</sup>, YUNXIA BAO<sup>2</sup>, AND QINGTIAN ZENG<sup>1</sup><sup>1</sup>College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China<sup>2</sup>College of Mathematics and System Science, Shandong University of Science and Technology, Qingdao 266590, China

Corresponding author: Yunxia Bao (baoyunxia98@163.com)

This work was supported in part by the National Science and Technology Major Project under Grant 2022ZD0119501; in part by NSFC under Grant 52374221; in part by the Science and Technology Development Fund of Shandong Province of China under Grant ZR2022MF288 and Grant ZR2023MF097; in part by the Taishan Scholar Program of Shandong Province under Grant ts20190936; and in part by the Science and Technology Program Special Project of Qingdao West Coast New Area under Grant 202209.

**ABSTRACT** Program deadlock detection plays an important role in ensuring the stability of software systems. In view of the high efficiency and low false alarm rate, dynamic deadlock analysis methods are widely studied. However, existing dynamic analysis methods use lock graph or its extended variants to model program behavior. Since there are information loss in existing lock graphs, deadlock false positives still exist. To solve the problem, a novel 2-stage dynamic deadlock analysis method is proposed. In the first stage, a Petri net model of the program is constructed by analyzing program running traces. Then, unfolding-based Petri net dead marking detection methods are used to identify potential program deadlocks. In the second stage, based on Petri net unfolding techniques, a deterministic program scheduling scheme that can be used for deadlock replay is derived for each potential deadlock. Each successfully replayed program deadlock is a real deadlock. It is shown that the proposed deadlock detection and replay method can eliminate more false positives compared to general dynamic methods, and the deadlock replay scheme is deterministic and easy to understand compared with existing random scheduling strategies.

**INDEX TERMS** Petri net, deadlock detection, net unfolding, discrete event system, program verification.

## I. INTRODUCTION

Synchronization primitives are widely used in today's multi-threaded programs [1], [2], [3]. However, many concurrency bugs including data races [4], [5], [6], and deadlocks [7], [8], [9] may happen if synchronization mechanisms are not properly designed. For example, when each thread in a thread set is waiting to acquire a lock held by another thread in the set, all these threads will block forever and a deadlock occurs [10]. Deadlocks are common in today's software. For example, Sun's bug database at <http://bugs.sun.com/> shows that 6,500 bug reports out of 198,000 contain the keyword "deadlock".

Deadlocks are usually difficult to be detected because they happen under very specific thread schedules. Both static [7], [9], [11], [12] and dynamic [13], [14], [15], [16], [17] deadlock detection techniques are developed. Static techniques

analyze program codes to infer lock acquisition cycles. They may guarantee that all executions of a program are deadlock-free. However, static analysis may report a large number of false positives (deadlocks that cannot actually occur in a practical execution) [18]. For example, the static deadlock detector developed by Williams et al. [12] reports 100,000 deadlocks in Sun's JDK 1.4, out of which only 7 are real deadlocks. Furthermore, it often fails to scale for large multi-threaded programs due to the exponential increase of program state space. Dynamic techniques are performed by analyzing one or just a few program executions. Due to the full use of runtime information, they generally produce fewer false positives. This is a significant advantage because eliminating false positives is usually time-consuming [19]. Moreover, since one running trace contains only a part of program behavior information, dynamic techniques are usually efficient. Meanwhile, this also brings a disadvantage, which is false negatives (deadlocks that can actually occur in practical executions, but cannot be detected). However, in view of

The associate editor coordinating the review of this manuscript and approving it for publication was Zhiwu Li<sup>1</sup>.

the difficulty in eliminating false positives, as well as the advantages of automation and high efficiency of dynamic analysis, dynamic analysis becomes the mainstream method of program deadlock detection. This paper focuses on a new dynamic deadlock analysis method.

Generally speaking, dynamic analysis detects potential deadlocks by extracting specific lock authorization patterns from program's running traces. For example, the literature [20] proposed a deadlock dynamic analysis tool called Visual Threads based on lock graphs,<sup>1</sup> which regarded each loop in the lock graph as a potential deadlock. This method is simple and effective, but it usually reports many false positives. For example, false positives may be generated due to thread-inner loops, loops protected by door locks, or loops caused by causal dependent operations belong to different threads. Reference [21] proposed the concept of lock trees and presented a deadlock detection algorithm called GoodLock. It can eliminate false positives caused by thread-inner loops and door-lock protected loops. However, it can only detect deadlocks caused by two threads. Reference [7] improved GoodLock and enabled it to detect deadlocks caused by arbitrary number of threads. Reference [22] proposed the concept of cyclic lock dependency chain, which expanded thread ID, currently-held-lock set on the basis of lock graph. It can eliminate false positives caused by thread-inner loops and door-lock protected loops, and has no limit on the number of threads which form a deadlock. By eliminating removable lock dependency in a cyclic lock dependency chain, [13] proposed a deadlock detection algorithm called Magiclock, which has higher scalability and efficiency compared to that in [22]. The above deadlock detection methods cannot eliminate false positives caused by causal dependent operations belong to different threads. To address this issue, [23] segmented the program operations based on thread start and join operations, and proposed the concept of segmentation graph to model the causal relation among program segments. Meanwhile, extended lock graphs are proposed there by expanding thread ID, currently-held-lock set and segment ID to traditional lock graphs. Finally, a new deadlock detection method was proposed based on the segmentation graph and extended lock graph in [23], which can eliminate all the false positives caused by thread-inner loops, door-lock protected loops, and loops caused by multiple operations which are causal dependent due to thread start or join operations. In addition, [24] developed a type of timestamp and vector clock to describe the causal relations among threads caused by thread start and join operations. Then, it proposed a new deadlock detection method based on vector clocks and cyclic lock dependency chain, which can also eliminate all the false positives caused by the above-mentioned loops.

<sup>1</sup>Take each lock object as a node in the graph. Each time a thread applies for a lock B while holding a lock A, a directed arc from A to B is added to the graph. The resulted graph is called a lock graph.

However, all the above-mentioned deadlock detection methods have the following problems: (1) the different execution of the same lock acquisition statement in a loop module cannot be distinguished in existing models like lock graphs, lock trees, cyclic lock dependency chains, or extended lock graphs. However, some executions may cause deadlocks, while not for others. The latter situation will cause false positives. (2) Causal dependency of operations belong to different threads can be caused by thread start or join operations. However, that is not all. The release and acquisition of the same lock object may also lead to causal dependency. Segmentation graphs or other existing tools cannot correctly model the latter case. The root cause of the above problems is that existing models cannot model the acquisition/release operations and their dependency relations accurately. To address this issue, by mining a program running trace, this paper constructs a Petri net [25] to model program behavior in a finer granularity, where each execution of a lock acquisition/release statement is modeled by a unique transition (even though they are different executions of the same statement), and the causal dependency caused by the aforementioned various synchronization primitives are all contained. Furthermore, considering the high efficiency of unfolding technology in the field of Petri net analysis [26], an unfolding-based deadlock detection method toward concurrent programs is developed. The refinement of the model and the high efficiency of unfolding technology make the proposed method more reliable.

Furthermore, neither the proposed Petri net models nor existing models mined from program running traces can model program behavior without loss of information. For example, conditional thread blocking statements and execution time of operations are both difficult to be described in the model. Both of them may lead to false positives. To ensure the authenticity of the detected deadlocks, deadlock replay technology has been proposed in [27], [28], and [29]. Through active intervention when a program is running, deadlock replay techniques schedule the program execution in order that a program defect can be triggered as much as possible. A potential deadlock that is successfully replayed is sure to be a real deadlock. Reference [27] proposed a random scheduling method to find program bugs, which tried to trigger concurrent defects such as deadlocks by lots of repetitive running under different scheduling schemes. The scheduling policy is completely random and blind. Since deadlock is usually a low probability phenomenon, this method is lack of efficiency and reliability. In contrast, [10], [24], [29], [30], [31], [32] first detect potential deadlocks, and then derive a specific scheduling scheme to increase the triggering probability of potential deadlocks. Specifically, [10], [24], [29] proposed a heuristic scheduling strategy for potential deadlocks, suspending the thread when it reaches a potential deadlock point, thereby increasing the probability of deadlock triggering. However, this method is still random in nature. It usually requires lots of runs to replay a real deadlock successfully. Considering that the trigger of a deadlock is related to both the deadlock point and some lock acquisition

operations before it, [30] proposed a new deadlock replay scheduling algorithm based on barrier. It optimized the intervention time of program scheduling. In addition, [30] and [31] constructed a set of constraints for program scheduling to replay a potential deadlock. It suspends a thread when its execution does not meet those constraints. The above methods improve the probability to successfully replay a real deadlock. However, the scheduling scheme they generated is not intuitive and cannot be used to trace the root cause of deadlocks. In contrast to them, the proposed method can generate an intuitive and deterministic scheduling scheme for each detected potential deadlock.

Generally speaking, this paper presents a novel dynamic deadlock detection method which works in two stages. In the first stage, a Petri net-based model of a program, called a program trace net, is mined from the running trace of a multithreaded program. Then, a deadlock-adjoint trace net is proposed and its unfolding is used to detect potential program deadlocks. The proposed method can avoid several kinds of false positives reported by existing common methods. In the second stage, for each detected potential deadlock, we derive an intuitive and deterministic scheduling scheme to replay a potential program deadlock. A detailed scheduling algorithm based on the scheme is also developed. With the deterministic scheduling, each potential deadlock replayed successfully is sure to be a real program deadlock. In this way, the authenticity of the deadlock can be guaranteed.

The rest of the paper is organized as follows. Section II introduces the concept of multithreaded programs. Also in this section, a Petri net-based program model is proposed to describe the program behavior implied in a program running trace. After that, the mining algorithm of the Petri net-based program model is developed in Section III. Section IV presents the concept of deadlock-adjoint program trace nets, and proposes a deadlock detection method based on its unfolding. Then, a deadlock replay algorithm is developed in Section V to check the authenticity of the detected deadlocks. The next two sections make an experimental evaluation and a conclusion.

## II. CONCEPT AND MODELING OF MULTITHREADED PROGRAMS

Taking Java programs as an example, this section presents the basic concepts related to multithreaded programs. Meanwhile, a Petri net-based model of multithreaded programs is proposed in this section.

### A. MULTITHREADED PROGRAMS IN JAVA

Java is a general purpose, object-oriented language that has received great interest as a means to develop concurrent and distributed applications in the Internet environment [33]. Concurrency is supported in Java by the thread concept. In java, threads are special objects representing independent executions of Java codes which operate on shared objects residing in the main memory. The behavior of a thread is described by the statements contained in its run() method.

Each thread offers a set of public methods by which other threads can control or inspect its execution. For example, if  $t$  is a thread reference,  $t.start()$ ,  $t.stop()$  are statements that respectively start and stop the execution of the thread referenced by  $t$ , whereas calling  $t.join()$  has the effect of waiting until the thread referenced by  $t$  terminates its execution. These methods are widely used for inter-thread communication.

To synchronize threads, Java mainly uses the monitor model, which is a structured way of using semaphores to access shared data. This mechanism can be explained in terms of locks. Each Java object is endowed with a lock that enables only one thread at a time to execute in the protected code regions of the object. Lock and unlock operations are not provided as separate statements in Java, but as implicit operations occurring respectively at the beginning and at the end of synchronized blocks. A synchronized block is introduced syntactically by the declaration `synchronized(o)`, where  $o$  is a reference to the associated object that must be locked. In addition, class `LockSupport` is also a basic thread blocking primitives for creating locks and other synchronization classes. It associates, with each thread that uses it, a permit. A call to `LockSupport.park` will return immediately if the permit is available, consuming it in the process; otherwise it may block. A call to `LockSupport.unpark` makes the permit available, if it was not already available. Methods `LockSupport.park` and `LockSupport.unpark` provide efficient means of blocking and unblocking threads.

Table 1 presents a pseudo-code of a Java multithreaded program. It contains three threads (`MainThread`, `ThreadA`, `ThreadB`) and three lock objects ( $G, o1, o2$ ). `ThreadA` has a loop module containing several lock authorization statements. During each execution of the loop, `ThreadA` first acquires object  $G$ . On the premise of holding lock  $G$ , it successively acquires  $o1$  and  $o2$ , then releases them in reverse order. During the first execution of the loop, `ThreadA` starts `ThreadB`. After `ThreadB` started, it acquires  $G$ , release  $G$ , acquires  $o2$  and  $o1$ , and then release  $o1$  and  $o2$  sequentially. Obviously, “the nested acquisition of locks  $o1$  and  $o2$  in the first loop execution of `ThreadA`” and “the nested acquisition of locks  $o2$  and  $o1$  in `ThreadB`” cannot cause a deadlock even if they seem to form a circular dependency. This is because `ThreadA` always holds  $G$  during the first execution of the loop and this blocks all the operations in `ThreadB`. In other words, `ThreadA`’s first release of  $G$  and `ThreadB`’s acquisition of  $G$  lead to a causal dependency between the above two italicized operations. This makes them impossible to trigger a deadlock. Conversely, if `ThreadA` execute the second loop after `ThreadB` acquires and releases  $G$ , then “the nested acquisition of locks  $o1$  and  $o2$  in the second loop execution of `ThreadA`” and “the nested acquisition of locks  $o2$  and  $o1$  in `ThreadB`” will no longer have a causal dependency, which can lead to a deadlock. Existing detection methods based on segmentation graphs or extended lock graphs identify both cases as deadlocks although only the second one are real. The proposed detection method aims to address such false positives.

**TABLE 1.** The pseudo-code of a multithreaded program, denoted by program 1.

<b>MainThread:</b>
01: public static void main(String[] args) throws InterruptedException {
02:   new ThreadA.start();
03:   ThreadA.join();
04: }
<b>ThreadA:</b>
05: public void run{
06:   for (int i=0, int flag=0; i<2; i++){
07:     synchronized(G){
08:       if (flag==0){
09:          new ThreadB.start();
10:          flag=1;
11:       }
12:     synchronized(o1){
13:       synchronized(o2){
14:       }
15:     }
16: }
17: // LockSupport.unpark(ThreadB);
18: }
<b>ThreadB:</b>
19: public void run{
20:   // LockSupport.park();
21:   synchronized(G){}
22:   synchronized(o2){
23:    synchronized(o1){}
24: }
25: }

In addition, if we uncomment lines 17 and 20 of Program 1, ThreadB can only acquire locks after ThreadA executed its loop module two times. In this case, the above-mentioned deadlock will also be disabled. It can also be correctly identified by the deadlock detection method proposed next.

## B. PROGRAM RUNNING TRACE

We assume that a multithreaded program is composed of a finite set of threads. Each thread has a unique identifier. These threads communicate with each other through shared objects (which correspond to locks in Java programs). A running trace  $\alpha$  captures one execution of a multithreaded program by listing the sequence of synchronization operations performed by the program. Its formal definition is as follows.

**Definition 1 (program running trace):** A running trace of a multithreaded program, denoted by  $\alpha$ , is a sequence of synchronization operations defined as follows:

$$\alpha \in \text{Trace} ::= \text{SynOperation}^*$$

$$\begin{aligned} op \in \text{SynOperation} ::= & c : \text{fork}(u, v) \mid c : \text{join}(u, v) \\ & c : \text{stop}(u) \mid c : \text{acq}(u, l) \mid c : \text{rel}(u, l) \end{aligned}$$

where

- $Tid$  is the set of threads.  $Lock$  is the set of lock objects that appeared in the program.  $u, v \in Tid$  are threads,  $l \in Lock$  is a lock;
- $c$  is the label identifier of a program statement;
- $\text{fork}(u, v)$  means  $u$  start  $v$ ;
- $\text{join}(u, v)$  means  $u$  blocks until  $v$  terminates;
- $\text{stop}(u)$  denotes the termination of thread  $u$ ;
- $\text{acq}(u, l)$  and  $\text{rel}(u, l)$  mean  $u$  acquire and release  $l$ .

Consider the multithreaded java program in Table 1. One of its running traces is given in Table 2. This running trace

does not trigger a deadlock. However, from this trace, we can reconstruct a Petri net model for this program. It can reveal a real deadlock of Program 1.

## C. PROGRAM TRACE NET

Petri nets are widely used for modeling and analysis of concurrent systems, such as flexible manufacturing systems [34], [35], [36], process aware systems [37], [38], [39], [40]. There are also many Petri-net-based methods to model concurrent programs [41], [42], [43], [44]. For example, [45] developed a toolkit for translating an Ada program into a type of Petri net called Ada net. In an Ada net, the control flow of each concurrent branch is modeled by a Petri-net-formatted state machine. Communications among concurrent tasks are represented by extra nodes connecting different state machines. An Ada net may imply the complete behavior of a program. However, it is in general inefficient or even intractable to do program analysis.

When we detect program deadlocks from a program running trace, we only need to construct a Petri net model that describes the program behavior related to this running trace. In this way, those complex Petri net structures used to model decision statements including if-else, case, and loop statements are no longer needed. As a result, the resulting Petri net model will be simpler, easier to construct and analyze. Next, the concepts related to this type of model are given.

**Definition 2 (Petri net):** [46] A Petri net is a 4-tuple  $\Sigma = (P, T; F, M_0)$ , where: 1)  $P$  and  $T$  are finite sets of places and transitions, respectively; 2)  $F \subseteq (P \times T) \cup (T \times P)$  is a finite set of flow relations; 3)  $M_0 : P \rightarrow \{0, 1, 2, \dots\}$  is the initial marking of  $\Sigma$ ; 4)  $P \cap T = \emptyset$  and  $P \cup T \neq \emptyset$ .

For each  $x \in P \cup T$ ,  $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$  and  $x \bullet = \{y \in P \cup T \mid (x, y) \in F\}$  denote the preset and post-set of  $x$ . Given that a marking  $M$  and a transition  $t \in T$ ,  $t$  is enabled at  $M$  if  $M(p) \geq 1$  holds for every  $p \in \bullet t$ . An enabled transition can fire. The firing of  $t$  leads to a new marking  $M' = M - \bullet t + t \bullet$  (where  $+$  and  $-$  denote the addition and subtraction of multi-set, respectively), which is denoted by  $M[t > M']$ . For a transition sequence  $\sigma \in T^*$ , if  $M_0[\sigma_1 > M_1[\sigma_2 > M_2[\sigma_3 > \dots M_{k-1}[\sigma_k > M, \sigma$  is said to be firable, and  $M$  is called a reachable marking.

**Definition 3 (program trace net):** Given that a program running trace  $\alpha$ , an extended Petri net  $\Sigma(\alpha) = (P, T; F, M_0, \rho, \tau)$  is called a program trace net if the following hold:

- 1)  $P = P_S \cup P_L, P_S \cap P_L = \emptyset$ .  $P_S$  represents state places corresponding to the control flow states of each thread.  $P_L$  represents lock places corresponding to lock objects appeared in  $\alpha$ ; Let  $P_I \subseteq P_S$  be the initial control flow states of each thread, and  $P_T \subseteq P_S$  be the termination states of each thread;
- 2)  $T = T_F \cup T_J \cup T_L \cup T_T$ , where: a)  $T_F$  represents fork operations appeared in  $\alpha$ ; b)  $T_J$  represents join operations; c)  $T_L$  represents the set of lock acquisition and release operations; and  $T_T$  represents the set of thread termination operations (i.e., stop operation) appearing in  $\alpha$ ;

TABLE 2. A running trace of the program in table 1.

No.	MainThread	TreadA	TreadB
1	2:fork( MainThread, ThreadA)		
2		7:acq( ThreadA, G)	
3		9:fork( ThreadA, ThreadB)	
4		12:acq( ThreadA, o1)	
5		13:acq( ThreadA, o2)	
6		14:rel( ThreadA, o2)	
7		15:rel( ThreadA, o1)	
8		16:rel( ThreadA, G)	
9		7:acq( ThreadA, G)	
10		12:acq( ThreadA, o1)	
11		13:acq( ThreadA, o2)	
12		13:rel( ThreadA, o2)	
13		14:rel( ThreadA, o1)	
14		15:rel( ThreadA, G)	
15			21:acq( ThreadB, G)
16			22:rel( ThreadB, G)
17			23:acq( ThreadB, o2)
18			23:acq( ThreadB, o1)
19			24:rel( ThreadB, o1)
20			25:rel( ThreadB, o2)
21		18: stop( ThreadA)	
22			25: stop( ThreadB)
23	3:join( MainThread, ThreadA)		
24	4:stop( MainThread)		

3)  $F = F_S \cup F_L \cup F_F \cup F_J$ , where: a)  $F_S \subseteq (P_S \times T) \cup (T \times P_S)$  defines, for each operation in  $T$ , the predecessor and subsequent control flow states within a thread; b)  $F_L \subseteq (P_L \times T_L) \cup (T_L \times P_T)$  shows the lock acquisition and release relations; c)  $F_F \subseteq T_F \times P_I$  defines the causal relations between a fork operation and the initial control flow state of the forked thread; and d)  $F_J \subseteq P_T \times T_J$  defines the causal relations between the termination state of a thread and a join operation which waits for the termination;

4)  $\rho : P \rightarrow Locks(\alpha) \cup Threads(\alpha)$  is a function which maps each lock place to a lock object and maps each state place to a thread, where  $Locks(\alpha)$  and  $Threads(\alpha)$  denote the set of lock objects and threads that appeared in  $\alpha$ ;

5)  $\tau : T \rightarrow SynOperations(\alpha)$  is a 1-1 mapping function which maps each transition to a synchronization operation that appeared in  $\alpha$ , where  $SynOperations(\alpha)$  denote the set of operations that appeared in  $\alpha$ ;

6)  $M_0(p) = 1$  if (a)  $p \in P_L$  holds, or (b)  $p$  represents the initial control flow state of the main thread. For the other places,  $M_0(p) = 0$ .

For the sake of simplicity, sometimes a program trace net is referred to as a trace net. Consider for instance the program running trace in Table 2. Its corresponding trace net is given in Figure 1, where red circles are lock places, and black circles are state places. For each place or transition in the trace net, the blue font labels show its mapped lock object or synchronization operation that appeared in Table 2. The black font labels are their node identifier.

Next section will give the algorithm for mining a program trace net from a running trace.

### III. MINING OF PROGRAM TRACE NET AND ITS PROPERTIES

Given a running trace of a multithreaded program, its corresponding trace net can be constructed according to the rules given in Table 3. Generally speaking, at the beginning, only an initial control flow place corresponding to the ready state of the main thread is added. It is marked with one token, meaning that the main thread is ready to execute. Then, for each operation appeared in the trace one by one, a transition along with its input/output places and flows should be added according to the operation information.

For example, for the operation  $fork(u, v)$ , a new transition  $t$  should be generated to describe the operation. Simultaneously, a flow relation  $\langle p_1, t \rangle$  should be added to model the prerequisite for this operation to execute, where  $p_1$  is the last control flow place of thread  $u$  before executing  $fork(u, v)$ . Place  $p_2$  should also be generated to model the new control flow state of thread  $u$  after  $fork(u, v)$  completed. In addition, place  $p_3$  should be generated to model the ready state of thread  $v$ . Finally, flow relations  $\langle t, p_2 \rangle$  and  $\langle t, p_3 \rangle$  should be generated to model the causal relations between  $fork(u, v)$  and its consequent states.

For the operation  $acq(u, l)$ , a new transition  $t$  and a flow relation  $\langle p_1, t \rangle$  should be added to model the prerequisite for this operation to execute, where  $p_1$  is the last control flow place of thread  $u$  before executing  $acq(u, l)$ . Place  $p_2$  and flow relation  $\langle t, p_2 \rangle$  should also be generated to model the new control flow state of thread  $u$  after this operation is executed. In addition, if  $l$  appears for the first time in the trace, a lock place  $p_3$  corresponding to the object  $l$  should be added

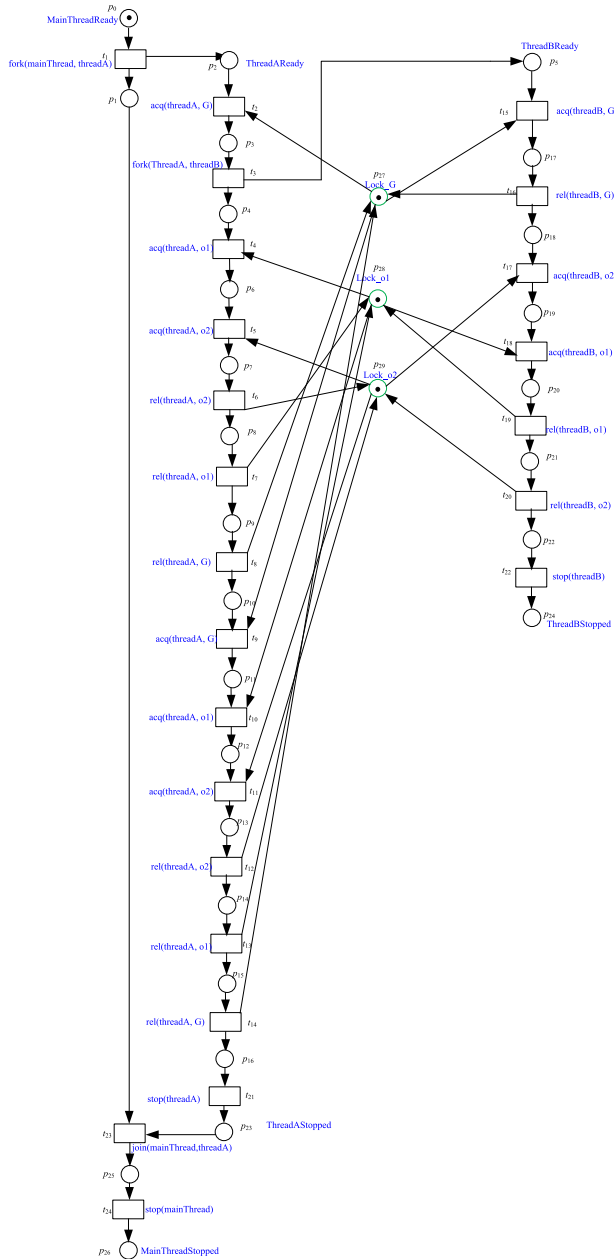


FIGURE 1.  $\Sigma_1$ , the program trace net mined from the running trace in Table 2.

and marked with one token. No matter  $p_3$  is newly added or not, a flow relation  $\langle p_3, t \rangle$  should be added to model the lock acquisition relation. The other operations should be handled in a similar way.

The detailed algorithm to mine a program trace net from a running trace is given in Algorithm 1. For each operation in the running trace, the main work includes (1) adding a new transition and at most two new places, (2) finding one or two places that meet some special conditions, and (3) adding several flow relations associated to the newly-added net nodes. Its time complexity is  $O(|\alpha|^2 + |\alpha| * |Locks(\alpha)|)$ , where  $|\alpha|$  is the number of operations in  $\alpha$ , and  $|Locks(\alpha)|$

denotes the number of lock objects that appeared in  $\alpha$ . Its space complexity is  $O(|\alpha| * |Locks(\alpha)|)$ .



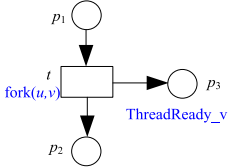
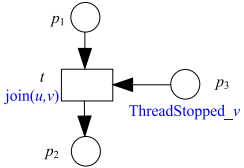
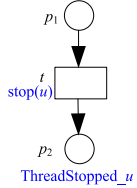
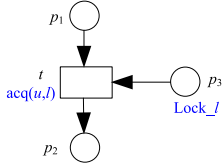
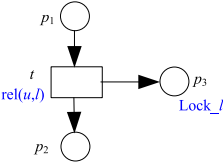
**Theorem 1:** For each firable transition sequence  $\sigma$  in the program trace net  $\Sigma(\alpha) = (P, T; F, M_0, \rho, \tau)$ ,  $\tau(\sigma)$  is a potential program running trace, where  $\tau(\sigma)$  means the program operation sequence obtained by replace each transition  $t$  in  $\sigma$  with  $\tau(t)$ .

**Proof:** According to the mining algorithm of a program trace net, a causal relation among Petri net transitions can be generated only in the following cases: (1) two operations in the same thread are executed one after another, then there will be a causal dependency between their corresponding transitions; (2) thread  $u$  executes operation  $fork(u, v)$ . Then, a causal relation between the transition corresponding to  $fork(u, v)$  and the one corresponding to the first operation of thread  $v$  will be added; (3) thread  $u$  executes operation  $join(u, v)$ . In this case, a causal relation between the transition corresponding to  $stop(u)$  and the one corresponding to  $join(u, v)$  will be generated. In addition, a causal relation from resource place  $l$  to each transition representing  $acq(u, l)$ ; and a causal relation from each transition representing  $rel(u, l)$  to the lock place  $l$ , will also be generated. Except for these cases, no other causal relations among transitions will be generated event if their corresponding program operations are executed one after another in  $\alpha$ . As can be seen, the mined Petri net models only the causal relations that do exist among program operations. Furthermore, the initial state of  $\Sigma(\alpha)$  models exactly the initial program state where only the main thread is ready and all the lock objects are available. Therefore, for each firable transition sequence  $\sigma$  in  $\Sigma(\alpha)$ ,  $\tau(\sigma)$  is a potential program running trace. However, since  $\alpha$  captures only part of program synchronization operations (the common synchronization primitives given in Definition 1), some operation causal relations that exist in the program may not be included in  $\Sigma(\alpha)$ . Hence,  $\tau(\sigma)$  is a potentially executable running trace with high probability, but it is not necessarily executable.

In addition, one should note that the predecessor and successor relations among program operations appeared in a running trace is compatible with the causal relations contained in the trace net. Hence, for a program running trace, its corresponding transition sequence must be firable in the trace net. For example, in Figure 1,  $\sigma = t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13} t_{14} t_{16} t_{17} t_{18} t_{19} t_{20} t_{21} t_{22} t_{23} t_{24}$  is the transition sequence corresponding to the running trace in Table 1. It is firable in  $\Sigma_1$  and results in the marking  $\{p_{24}, p_{26}, p_{27}, p_{28}, p_{29}\}$ . At the resulting marking, all threads are in a termination state. All the resource places are marked with one token. This means that the program terminates normally.

More importantly, the conflict and concurrent structures in the trace net make it implying many other potential program running traces. This makes it possible to detect more program deadlocks. For example,  $t_{15}$  (ThreadB's acquisition of lock G) executes after  $t_9$  (ThreadA's second acquisition of lock G) in Table 2. However, in Figure 1, there is no

TABLE 3. Petri net fragments used to model program objects and operations.

Program objects or operations	Petri net	Notes
the main thread object		$p_0$ is used to model the ready state of the main thread. At the initial marking, it has a token, meaning that the program is ready to execute.
a lock object		$p_i$ is used to model a lock object $Lock_l$ . At the initial marking, it has a token, meaning that it is available at the initial state.
$fork(u, v)$		$p_1$ is the last state place of thread $u$ before it execute $fork(u, v)$ . $p_2, p_3$ , transition $t$ , and all the flows are newly generated. $t$ is used to model this execution of $fork(u, v)$ . $p_3$ represents the ready state of the thread $v$ . $p_2$ represents the new control flow state of the thread $u$ after it execute $fork(u, v)$ .
$join(u, v)$		$p_1$ is the last state place of thread $t$ before it execute $join(u, v)$ . $p_3$ is the stop state of the thread $v$ . $p_2, t$ and all the flows are newly generated. $t$ is used to model this execution of $join(u, v)$ . $p_2$ represents the new control flow state of the thread $u$ after it execute $join(u, v)$ .
$stop(u)$		$p_1$ is the last state place of thread $u$ before it execute $stop(u)$ . $p_2, t$ and all the flows are newly generated. $t$ is used to model this execution of $stop(u)$ . $p_2$ represents the stop state of the thread $u$ .
$acq(u, l)$		$p_1$ is the last state place of thread $u$ before it execute $acq(u, l)$ . $p_3$ is the lock place representing the lock $l$ . $p_2, t$ and all the flows are newly generated. $t$ is used to model this execution of $acq(u, l)$ . $p_2$ represents the new control flow state of the thread $u$ after it execute $acq(u, l)$ .
$rel(u, l)$		$p_1$ is the last state place of thread $u$ before it execute $rel(u, l)$ . $p_3$ is the lock place representing the lock $l$ . $p_2, t$ and all the flows are newly generated. $t$ is used to model this execution of $rel(u, l)$ . $p_2$ represents the new control flow state of the thread $u$ after it execute $rel(u, l)$ .

causal dependency between them. In other firable transition sequences, their execution order may be reversed. For example,  $\sigma' = t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_{15} t_{16} t_9 t_{10} t_{17}$  is such a firable transition sequence where  $t_{15}$  fires before  $t_9$ . Its corresponding program running trace  $\tau(\sigma')$  is shown in Table 4. The transition sequence leads to a dead marking  $\{p_1, p_{12}, p_{19}\}$  in the trace net. Meanwhile, the program in Table 1 will fall into a deadlock state if it runs according to the trace in Table 4. It can be seen that there is a direct correspondence between a dead marking of the trace net and a potential deadlock of the program. Next, we will present the detection method of such potential deadlocks.

Furthermore, given a trace net mined from a program running trace, it is easy to find that (1) the trace net is

1-safe, i.e., a place has at most 1 token in the reachable markings. This is because each control flow state can either be active or not, and each lock object can either be available or not. The former correspond to a place marked with one token, while the latter corresponds to an empty place. In both cases, a place can have at most one token; (2) if all the places modeling the termination state of non-joint threads are marked with one token, all the other control flow state places must be empty and all the lock places must be marked with 1 token. This is because a thread has at least one active state. Once its termination state is active, other state places of the thread must be empty. Meanwhile, all the lock objects are sure to be released when all the program threads terminate correctly.

**Algorithm 1** The Algorithm for Mining a Trace Net From a Program Running Trace**Input:**  $\alpha$ , a running trace of a multithreaded program**Output:** the program trace net corresponding to  $\alpha$ , denoted by  $\sum(\alpha) = (P, T; F, M_0, \rho, \tau)$ **Steps:**

1.  $P_S := \emptyset, P_L := \emptyset$ ; //Initialize the state place set and lock place set to be empty sets.
2.  $T_F := \emptyset, T_J := \emptyset, T_L := \emptyset, T_T := \emptyset$ ; //Initialize the set of various type of operations to be empty sets.
3.  $F_S := \emptyset, F_L := \emptyset, F_F := \emptyset, F_J := \emptyset$ ; // Initialize the set of various type of flow relations to be empty sets.
4. Let  $p_0$  be the initial control flow place corresponding to the ready state of the main thread. Add it to  $P_S$  and  $P_I$ . Set  $\rho(p_0)$  to MainThread. Set  $M_0(p_0)$  to 1;
5. **FOR EACH** operation op in  $\alpha$ {
6. **IF**(op is in the type of  $fork(u, v)$ ){
7. Add a new transition  $t$  to  $T_F$ , and Set  $\tau(t) = fork(u, v)$ ;
8. Let  $p_1$  be the place satisfying  $\rho(p_1) = u \wedge p_1^\bullet = \emptyset$ . Add a new state pace  $p_2$  to  $P_S$ . Set  $\rho(p_2)$  to  $u$ . Add  $(p_1, t)$  and  $(t, p_2)$  to  $F_S$ . Set  $M_0(p_2)$  to 1;
9. Add a new state pace  $p_3$  to  $P_S$  and  $P_I$ . Set  $\rho(p_3)$  to  $v$ . Set  $M_0(p_2)$  to 1. Add  $(t, p_3)$  to  $F_F$ ;
10. }//fork
11. **ELSE IF**(op is in the type of  $join(u, v)$  ){
12. Add a new transition  $t$  to  $T_J$ , and Set  $\tau(t) = join(u, v)$ ;
13. Let  $p_1$  be the place satisfying  $\rho(p_1) = u \wedge p_1^\bullet = \emptyset$ . Add a new state pace  $p_2$  to  $P_S$ . Set  $\rho(p_2)$  to  $u$ . Set  $M_0(p_2)$  to 1. Add  $(p_1, t)$  and  $(t, p_2)$  to  $F_S$ ;
14. Let  $p_3$  be the place satisfying  $\rho(p_3) = v \wedge p_3^\bullet = \emptyset$ . Add  $(p_3, t)$  to  $F_J$ ;
15. }//join
16. **ELSE IF** (op is in the type of  $stop(u)$ ){
17. Add a new transition  $t$  to  $T_T$ , and Set  $\tau(t) = stop(u)$ ;
18. Let  $p_1$  be the place satisfying  $\rho(p_1) = u \wedge p_1^\bullet = \emptyset$ . Add a new state pace  $p_2$  to  $P_S$  and  $P_T$ . Set  $\rho(p_2)$  to  $u$ . Set  $M_0(p_2)$  to 1. Add  $(p_1, t)$  and  $(t, p_2)$  to  $F_S$ ;
19. }//stop
20. **ELSE IF** (op is in the type of  $acq(u, l)$ ){
21. Add a new transition  $t$  to  $T_L$ , and Set  $\tau(t) = acq(u, l)$ ;
22. Let  $p_1$  be the place satisfying  $\rho(p_1) = u \wedge p_1^\bullet = \emptyset$ . Add a new state pace  $p_2$  to  $P_S$ . Set  $\rho(p_2)$  to  $u$ . Set  $M_0(p_2)$  to 1. Add  $(p_1, t)$  and  $(t, p_2)$  to  $F_S$ ;
23. **IF** (there exists lock place  $p_3 \in P_L$  satisfying  $\rho(p_3) = l$ ) Add  $(p_3, t)$  to  $F_L$ ;
24. **ELSE** {Add a new lock place  $p_3$  to  $P_L$ . Set  $\rho(p_3)$  to  $l$ . Set  $M_0(p_3)$  to 1. Add  $(p_3, t)$  to  $F_L$ .};
25. }//acq
26. **ELSE IF**(op is in the type of  $rel(u, l)$ ){
27. Add a new transition  $t$  to  $T_L$ , and Set  $\tau(t) = rel(u, l)$ ;
28. Let  $p_1$  be the place satisfying  $\rho(p_1) = u \wedge p_1^\bullet = \emptyset$ . Add a new state pace  $p_2$  to  $P_S$ . Set  $\rho(p_2)$  to  $u$ . Set  $M_0(p_2)$  to 1. Add  $(p_1, t)$  and  $(t, p_2)$  to  $F_S$ ;
29. Let  $p_3$  be the place satisfying  $\rho(p_3) = l$ . Add  $(t, p_3)$  to  $F_L$ ;
30. }//rel
31. }//FOREACH
32.  $P := P_S \cup P_L; T := T_F \cup T_J \cup T_L \cup T_T; F = F_S \cup F_L \cup F_F \cup F_J$
33. return  $\sum(\alpha) := (P, T; F, M_0, \rho, \tau)$

**IV. PROGRAM DEADLOCK DETECTION BASED ON UNFOLDING OF DEADLOCK-ADJOINT TRACE NETS**

Giving that a program running trace  $\alpha$ , if it triggers a program deadlock, then the deadlock can be easily located and repaired. We mainly study the scenario that the program terminates normally without falling into a deadlock state when it runs in the trace of  $\alpha$ . To detect potential program deadlocks in this scenario, we give the following concepts and algorithms next.

**A. DEADLOCK-ADJOINT TRACE NET**

For a program trace net  $\sum(\alpha)$ , denote the set of its reachable markings by  $R(\sum(\alpha))$ . A reachable marking  $R(\sum(\alpha)) M \in$  is called a dead marking if no transitions are enabled by  $M$ . Note that some dead markings represent program deadlocks, while some others represent normal termination states of a program.

For example, if Program 1 execute in the trace of Table 2, all threads will stop normally and all lock objects will be



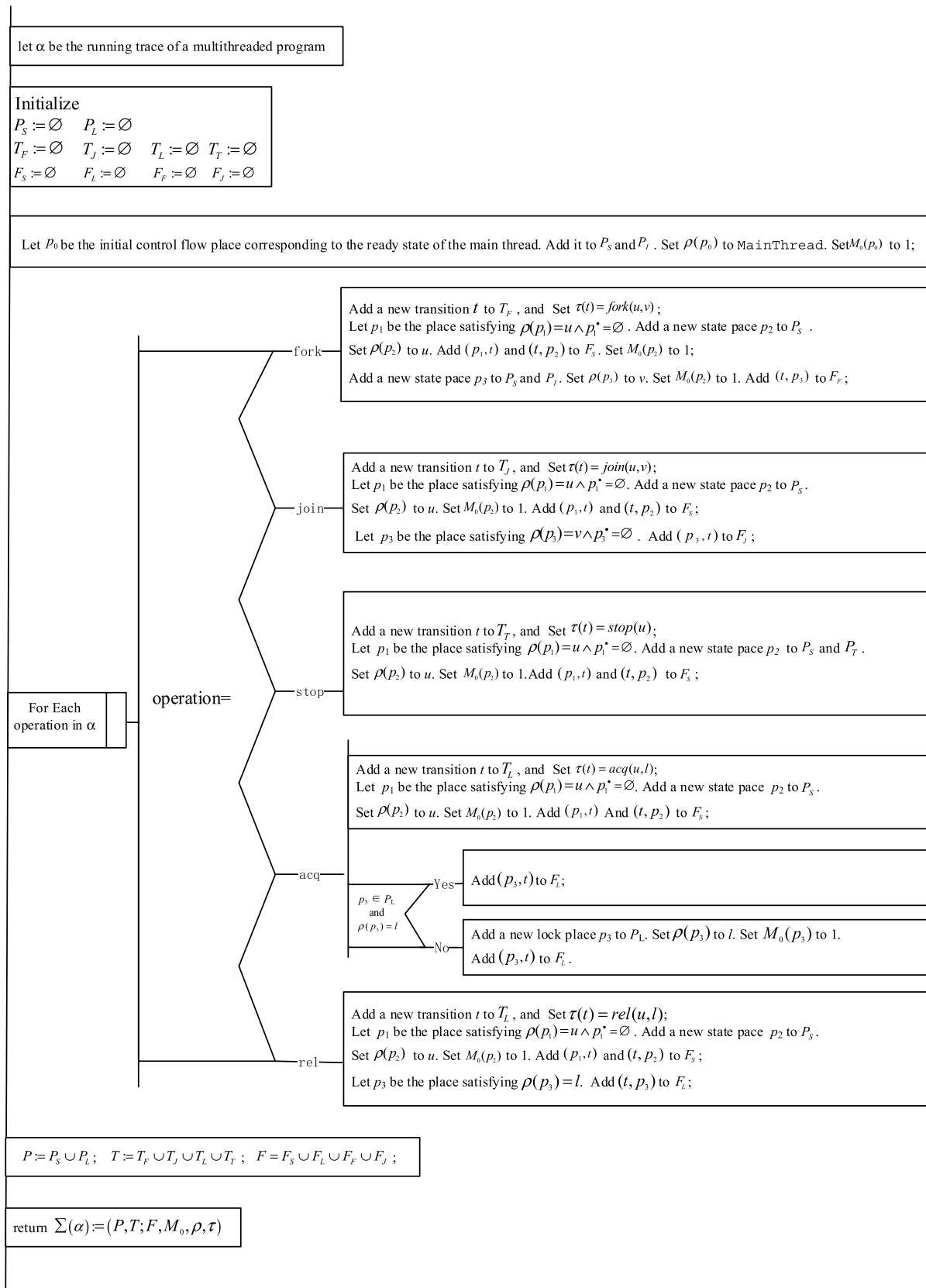


FIGURE 2. Problem analysis diagram of algorithm 1.

**TABLE 4.** Another running trace of program 1 which triggers a program deadlock (red font operations are blocked thread points).

No.	MainThread	TreadA	TreadB
1	2:fork( MainThread, ThreadA)		
2		7:acq( ThreadA, G)	
3		9:start( ThreadA, ThreadB)	
4		12:acq( ThreadA, o1)	
5		13:acq( ThreadA, o2)	
6		13:rel( ThreadA, o2)	
7		14:rel( ThreadA, o1)	
8		15:rel( ThreadA, G)	
9			21:acq( ThreadB, G)
10			22:rel( ThreadB, G)
11		7:acq( ThreadA, G)	
12		12:acq( ThreadA, o1)	
13			23:acq( ThreadB, o2)
	3:join(MainThread,ThreadA)[blocked point]	13:acq( ThreadA, o2) [blocked point]	23:acq(ThreadB,o1)[blocked point]

released. If the trace net execute the transition sequence corresponding to this trace, it will reach a marking where (1) all the places modeling the termination state of non-joint threads and all the places modeling lock objects are marked with one token, (2) all the other places are empty. This marking is dead since no transitions can be enabled at this state. However, this marking corresponds to an expected program state where the program terminates normally. We call such a dead marking a *legal dead marking*. The legal dead marking of  $\Sigma(\alpha)$  in Fig. 1 is  $\{p_{24}, p_{26}, p_{27}, p_{28}, p_{29}\}$ .

Differently, if Program 1 runs in the trace of Table 4, ThreadA will be blocked because it need to acquire lock o1 which is held by ThreadB. ThreadB will also be blocked because it need to acquire lock o2 which is held by ThreadA. The main thread will be blocked too because it need to wait for the termination of ThreadA. In this case, a program deadlock occurs. Correspondingly, if the trace net execute the transition sequence corresponding to this trace, which is  $\sigma' = t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_{15} t_{16} t_9 t_{10} t_{17}$ , the trace net will result in a dead marking  $\{p_{11}, p_{12}, p_{19}\}$ . Such a marking represents a potential program deadlock. We call it an *illegal dead marking*.

Formally speaking, we have the following definition and conclusion.

**Definition 4 (legal/illegal dead marking):** Let  $\Sigma(\alpha) = (P, T; F, M_0, \rho, \tau)$  be a trace net mined from a program running trace  $\alpha$ ,  $R(\Sigma(\alpha)) M \in$  be a dead marking of  $\Sigma(\alpha)$ . If all the places modeling the termination state of non-joint threads and all the places modeling lock objects are marked with one token at  $M$ , and all the other places are empty at  $M$ , then  $M$  is called a legal dead marking of  $\Sigma$ . Otherwise,  $M$  is called an illegal dead marking. In other words, a dead marking is legal if and only if

$$\forall p \in P : M(p) = \begin{cases} 1 & \text{if } (p \in P_L) \vee (p \in P_T \wedge p^\bullet = \emptyset) \\ 0 & \text{otherwise} \end{cases}$$

**Theorem 2:** Let  $\sigma$  be a firable transition sequence of the program trace net  $\Sigma(\alpha) = (P, T; F, M_0, \rho, \tau)$ . If the execution of  $\sigma$  leads to an illegal dead marking of  $\Sigma(\alpha)$ , then the

program may fall into a deadlock when it running in the trace of  $\tau(\sigma)$ .

*Proof:* According to Theorem 1,  $\tau(\sigma)$  is a potential program running trace. The resulting program state after running  $\tau(\sigma)$  is modeled by the resulting marking of  $\Sigma(\alpha)$  after firing  $\sigma$ . If  $\sigma$  results at an illegal dead marking of  $\Sigma(\alpha)$ , then, after the program executes  $\tau(\sigma)$ , no operations can be executed and at least one thread does not terminate. This means that the program may fall into a deadlock state.

According to Theorem 2, detecting the illegal dead markings of a trace net may reveal potential program deadlocks. To detect such illegal dead markings efficiently, we translate a program trace net to another Petri net which retains all the illegal dead markings while excluding all the legal dead markings. To do this, a new transition, denoted by *recover*, should be added. *recover* takes the following places as input: (1) all the places modeling the stop state of non-joint threads; and (2) the places modeling lock objects. Meanwhile, it takes the ready state place of the main thread and the lock places as output. In this way, when the original Petri net reaches a legal dead marking, *recover* will become enabled and it can restore the Petri net to its initial state. We call the resulted Petri net a *deadlock-adjoint trace net* of the original one. Its formal definition is as follows.

**Definition 5 (deadlock-adjoint trace net):** Let  $\Sigma(\alpha) = (P, T; F, M_0, \rho, \tau)$  be a trace net mined from a program running trace  $\alpha$ .  $P_{T\_legal} = \{p | p \in P_T \wedge p^\bullet = \emptyset\}$  denotes the set of state places modeling the termination state of non-joint threads.  $P_L$  denotes the set of lock places.  $p_0$  denotes the state place modeling the ready state of the main thread.  $\Sigma'(\alpha) = (P', T'; F', M_0, \rho, \tau')$  is called a deadlock-adjoint trace net of  $\Sigma(\alpha)$ , where

$$P' = P,$$

$T' = T \cup \{recover\}$ , where *recover* is a newly added transition used to restore the legal dead marking of  $\Sigma$  to its initial marking,

$$F' = F \cup ((P_{T\_legal} \cup P_L) \times \{recover\}) \cup (\{recover\} \times (\{p_0\} \cup P_L));$$

$\tau' : T' \rightarrow \text{SynOperations}(\alpha)$  is a 1-1 mapping function satisfying (1)  $\tau'(t) = \tau(t)$  if  $t \in T$ , and (2)  $\tau'(t) = \varepsilon$  if  $t = \text{recover}$ , where  $\varepsilon$  is the null operation.

Consider for instance the Program trace net in Figure 1. Its deadlock-adjoint trace net is shown in Figure 2, where  $p_{26}$  and  $p_{24}$  belong to  $\bullet \text{recover}$  since they correspond to the termination state of non-joint threads (which are ThreadB and the main thread). On the contrary,  $p_{21}$  (representing the termination state of ThreadA) does not belong to  $\bullet \text{recover}$  because ThreadA is joint by the main thread.  $p_{27}$ ,  $p_{28}$  and  $p_{29}$  belong to  $\bullet \text{recover}$  because they represent lock objects and all of them should be released when the program terminate correctly. In addition,  $\{p_0, p_{27}, p_{28}, p_{29}\}$  constitutes the postset of  $\text{recover}$  since it corresponds to the initial marking of the original program trace net, where the main thread is in ready state and all the lock objects are available.

Obviously, a legal dead marking in a program trace net is no longer dead in its corresponding deadlock-adjoint trace net. However, an illegal dead marking is still dead in the deadlock-adjoint trace net. For example,  $\{p_4, p_{10}, p_{17}, p_{18}\}$ , the legal dead marking of  $\Sigma_1$ , is not dead in  $\Sigma_2$  since  $\text{recover}$  is enabled at this state. Meanwhile, the illegal dead marking  $\{p_1, p_{12}, p_{19}\}$  in  $\Sigma_1$  is still dead in  $\Sigma_2$  since  $\text{recover}$  can be enabled only by the legal dead marking.

**Theorem 3:** Let  $\sigma$  be a firable transition sequence of deadlock-adjoint trace net  $\sum'(\alpha) = (P', T'; F', M_0, \rho, \tau')$ . If the transition  $\text{recover}$  does not appear in  $\sigma$ , and the execution of  $\sigma$  leads to a dead marking of  $\sum'(\alpha)$ , then the program may fall into a deadlock when it runs in the trace of  $\tau'(\sigma)$ .

*Proof:* According to Definition 5, a dead marking of  $\sum'(\alpha)$  is an illegal dead marking of the program trace net  $\sum(\alpha) = (P, T; F, M_0, \rho, \tau)$ . In the case that  $\sigma$  does not contain transition  $\text{recover}$ ,  $\tau'(\sigma)$  is the same as  $\tau(\sigma)$ . According to Theorem 2, the program may fall into a deadlock when it running in the trace of  $\tau'(\sigma)$ .

According to Theorem 3, to detect potential deadlocks of a multithread program, we only need to detect dead marking of a deadlock-adjoint trace net.

## B. UNFOLDING BASED DEAD MARKING DETECTION OF DEADLOCK-ADJOINT TRACE NETS

Much work has focused on detecting dead markings of Petri nets. Some are based on reachability trees [39], [47], [48], [49], [50]. Some are based on structural analysis methods [51], [52], [53]. However, unfolding technologies are widely adopted due to its efficiency in tackling with state explosion problems [26], [54], [55], [56]. In this paper, we use the unfolding technique to detect dead markings. A brief introduction to the related concepts is as follows. For more details on Petri net unfolding technology, please refer to [26] and [54].

- **P/T-net:** A P/T-net is 3-tuple  $N = (P, T; F)$  where  $P$  and  $T$  are two finite disjoint set of places and transitions,  $F \subseteq (P \times T) \cup (T \times P)$  is a set of flow relations. The

following relations are defined on the set  $P \cup T$  of nodes in  $N$ :

- 1) the *causality* relation, denoted as  $<$ , is the transitive closure of  $F$ , and  $\leq$  is the reflexive closure of  $<$ ; if  $x < y$ , we say that  $y$  causally depends on  $x$ ;
  - 2) the *conflict* relation, denoted as  $\#$ : nodes  $x, y \in P \cup T$  are in conflict iff  $\exists t, t' \in T : (t \neq t') \wedge (\bullet t \cap \bullet t' \neq \emptyset) \wedge (t \leq x) \wedge (t' \leq y)$ ;
  - 3) the *concurrency* relation, denoted as  $\parallel$ : nodes  $x, y \in P \cup T$  are concurrent if they are not in conflict and neither of them causally depends on the other.
- **Occurrence net:** an *occurrence net* is a P/T-net  $ON = (B, E; G)$  s.t. (1)  $ON$  is acyclic; (2)  $\forall p \in B : |\bullet p| \leq 1$ ; (3) for any  $x \in B \cup E$ , the set  $\{y | y < x\}$  is finite; (4)  $\forall x \in B \cup E : \neg(x\#x)$ , i.e., no node is in self-conflict. In occurrence nets, elements from  $B$  are usually called *conditions* and elements from  $E$  are usually called *events*.
  - **Homomorphism:** Let  $N_1 = (P_1, T_1; F_1)$  and  $N_2 = (P_2, T_2; F_2)$  be two P/T-nets. A *homomorphism* from  $N_1$  to  $N_2$  is a mapping  $h : P_1 \cup T_1 \rightarrow P_2 \cup T_2$  such that (1)  $(h(P_1) \subseteq P_2) \wedge (h(T_1) \subseteq T_2)$ ; (2) for every  $t \in T_1$ , the restriction of  $h$  to  $\bullet t$  is a bijection between  $\bullet t$  (in  $N_1$ ) and  $\bullet h(t)$  (in  $N_2$ ), and similarly for  $t^\bullet$  and  $h(t)^\bullet$ . In other words, a homomorphism is a mapping that preserves the nature of nodes and the environment of transitions.
  - **Branching process:** A Petri net  $\sum = (P, T; F, M_0)$  can also be expressed in the form of  $\sum = (N, M_0)$ , where  $N = (P, T; F)$  is a P/T-net. A *branching process* of Petri net  $\sum = (N, M_0)$  is a pair  $\beta = (N', h)$  where  $N' = (B, E; G)$  is an occurrence net and  $h$  is a homomorphism from  $N'$  to  $N$  such that: (1) the restriction of  $h$  to  $\text{Min}(N')^2$  is a bijection between  $\text{Min}(N')$  and  $M_0$ , (2) for each  $x, y \in E : \bullet x = \bullet y \wedge h(x) = h(y) \rightarrow x = y$  holds;
  - **Unfolding:** Two branching processes  $\beta_1 = (N_1, h_1)$  and  $\beta_2 = (N_2, h_2)$  of a Petri net are isomorphic if there is a bijective homomorphism  $h$  from  $N_1$  to  $N_2$  such that  $h_2 \circ h = h_1$ . Intuitively, two isomorphic branching processes differ only in the names of conditions and events. Up to isomorphism, a Petri net has a unique maximal branching process. We call it the *unfolding* of the Petri net.
  - **Configuration:** A *configuration*  $C$  in an occurrence net  $ON = (B, E; G)$  is a non-conflicting subsets of events satisfying  $(C \subseteq E) \wedge (\forall x, y \in C : \neg(x\#y)) \wedge (x < y \wedge x \in C \wedge y \in C \rightarrow x \in C)$ . For each  $e \in B$ , we define the local configuration of  $e$  to be  $[e] = \{x | x \in E \wedge x < e\}$ .
  - **Configuration-associated Petri net marking:** In an unfolding, each event corresponds to a transition of the original Petri net, and each condition corresponds to a place. We can associate each configuration of the

<sup>2</sup>For a P/T-net  $N = (P, T; F)$ ,  $\text{Min}(N)$  is defined as the set  $\{x | (x \in P \cup T) \wedge \neg(\exists y \in P \cup T : y < x)\}$

unfolding with a Petri net marking by simply identifying the conditions that are produced but are not consumed by the events in the configuration. To do this, we define the *cut* of a configuration  $C$  by  $Cut(C) = (Min(ON) \cup C^*) \setminus C$ . Then, the associated marking in the original Petri net is  $h(Cut(C))$ , which is denoted it by  $Mark(C)$ .

- *complete finite prefix of unfolding and cut-off event*: When a Petri net has arbitrarily long firable transition sequences, its unfolding can be extended infinitely. As a result, the unfolding is infinite. References [26] and [54] proposed the concept of *complete finite prefix of unfolding for a bounded Petri net*, which terminates when an obtained unfolding prefix can represent all the reachable markings of the original net. The key to termination is to identify those events that would not bring new markings into the unfolding. Such events are called *cut-off events*. Formally speaking, an event  $e$  is a *cut-off event* if it satisfies  $\exists e' \in E : Mark([e]) = Mark([e']) \wedge |[e]| > |[e']|$ , where  $[e]$  denotes the local configuration of  $e$  and  $|[e]|$  the cardinality of this latter.

Briefly speaking, unfolding technology uses an occurrence net to describe the behavior of a Petri net. In an unfolding or its complete finite prefix, each event in the occurrence net denotes one firing of a Petri net transition. Each condition denotes a token. Each flow denotes the consumption or the generation of a token. Each configuration leads to a potential partial order run of a Petri net and it yields to a new marking after firing all the events from the initial marking. Each configuration-associated marking corresponds to the resulting marking. Take the program trace net in Figure 2 as an example. One complete finite prefix of its unfolding is shown in Figure 3.

The specific method to unfold a Petri net and compute its complete finite prefix can be found in [54]. For example, to unfold the deadlock-adjoint trace net in Figure 2, conditions  $c_0, c_1$  and  $c_2$  should be generated at the beginning. They correspond to the initial token in  $p_0, p_{17}$  and  $p_{18}$ , respectively.  $c_0$  can enable transition  $t_1$ . Therefore, an event  $e_1$  along with its posterior conditions should be generated to extend the unfolding. In this way, all the other events along with their posterior conditions can be generated one by one. However,  $e_{25}, e_{37}$  and  $e_{51}$  are cut-off events since they reset the adjoint Petri net to a state that previously appeared. No further extensions are needed from their posterior conditions. For more details on Petri net unfolding technology, please refer to [26] and [54].

The literature [54] has proved that a bounded Petri net has a reachable dead marking if and only if its unfolding has a reachable dead marking. Therefore, we can use the unfolding of a deadlock-adjoint trace net to detect potential program deadlocks. Furthermore, [26] also proved that an unfolding has a reachable dead marking if and only if there is a configuration which is in conflict with all the cut-off events. Consider for instance the unfolding prefix in Figure 3. We can find that

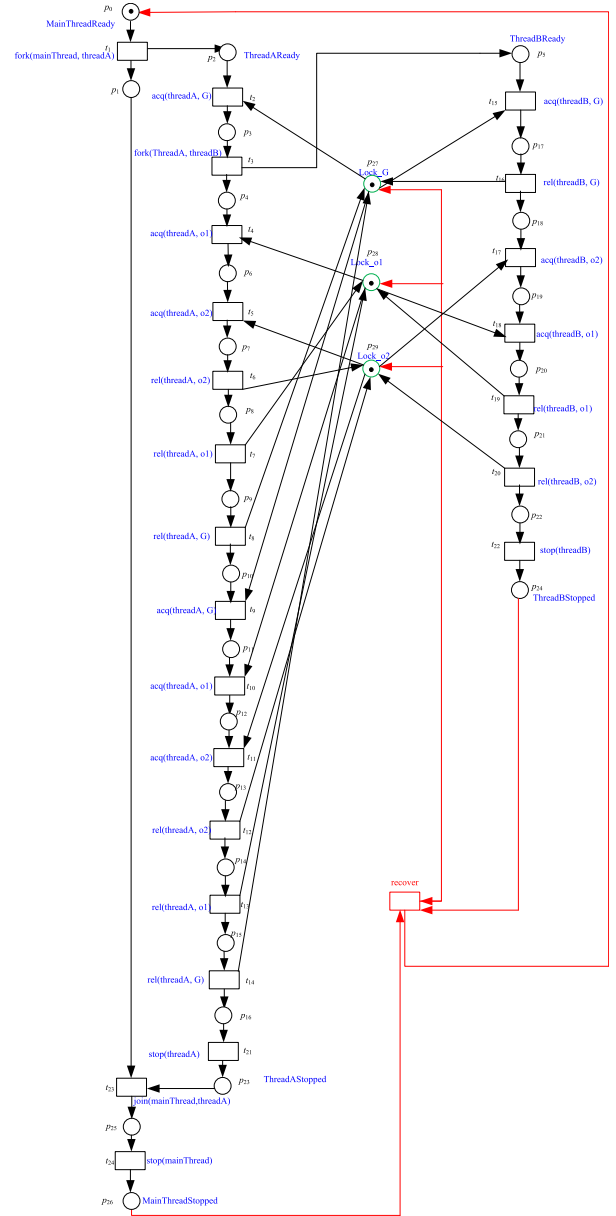


FIGURE 3.  $\Sigma_2$ , the deadlock-adjoint trace net of  $\Sigma_1$ .

$C_1 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_{26}, e_{27}, e_{28}, e_{29}, e_{38}\}$  is such a configuration, since  $e_{26}$  is conflict with  $e_{25}$ , and  $e_{29}$  is conflict with  $e_{51}$ . The resulted Petri net marking after firing the configuration-associated transitions in Figure 2 is  $\{p_1, p_{12}, p_{19}\}$ , which is a dead marking. Furthermore, after executing the operation sequence corresponding to  $C_1$ , Program 1 will reach the state that ThreadA holds lock o1 and waits for lock o2, while ThreadB holds lock o2 and waits for lock o1. Obviously, this leads to a program deadlock. In this way, we got a potential program deadlock. In this process, the configuration leading a deadlock-adjoint trace net to a dead marking is the key. Its formal definition is as follows.

*Definition 6 (deadlock-adjoint configuration)*: Let  $\Sigma'(\alpha) = (P', T'; F', M_0, \rho, \tau')$  be a deadlock-adjoint trace net, and  $Unf(\Sigma'(\alpha)) = (B, E; G, h)$  be one complete finite

**Algorithm 2** The Derivation Algorithm of a Configuration-Associated Lock Scheduler

**Input:**  $\Sigma'(\alpha) = (P', T'; F', M_0, \rho, \tau')$ , a deadlock-adjoint trace net corresponding to the program running trace  $\alpha$ ;  $C$ , a configuration of  $Unf(\Sigma'(\alpha))$ .

**Output:** a program lock set  $LockSet$ , along with a function  $\delta$  which assigns each lock with an authorization thread sequence

**Steps:**

1. Calculate the spanned subnet of configuration  $C$  in  $Unf(\Sigma'(\alpha))$ . Denote it by  $\Sigma = (B', E'; G', h')$
2. **FOR EACH** condition  $c$  satisfying  $\bullet c = \emptyset \wedge \rho(h'(c)) \in P_L$  in  $\Sigma$
3. Let  $o$  be the lock object corresponding to  $c$ , i.e.,  $o := \rho(h'(c))$ ;
4.  $\delta(o) := \epsilon$ , where  $\epsilon$  means an empty thread sequence;
5. **WHILE**  $c$  has an output event in  $\Sigma$
6. Let  $e$  be the output event of  $c$ ;
7. **IF**  $(\tau'(h'(e)))$  is a lock acquisition operation with the form of  $acq(u, o)$
8.  $\delta(o) := \delta(o) \circ u$ , where  $\circ$  means the concatenation operation of a sequence;
9. Reset  $c$  to be the only output condition of  $e$
10. **ENDIF**
11. **ELSE IF**  $(\tau'(h'(e)))$  is a lock release operation with the form of  $rel(u, o)$
12. Reset  $c$  to be the output condition of  $e$  which satisfying  $\rho(h'(c)) = o$ ;
13. **ELSE** // in the case that  $\tau'(h'(e))$  is a thread fork, join, or stop operation
14. Reset  $c$  to be the output condition of  $e$  such that  $\rho(h'(c))$  is the thread which execute  $\tau'(h'(e))$ ;
15. Reset  $e$  to be the output event of the updated condition  $c$ ;
16. **ENDIF**
17. Put the lock object  $o$  into  $LockSet$ , and assign  $\delta(o)$  to its authorized thread sequence;
18. **ENDFOR**

prefix of  $\Sigma'(\alpha)$ 's unfolding. Denote the set of cut-off events in  $Unf(\Sigma'(\alpha))$  by  $\aleph$ . For a configuration  $C$  of  $Unf(\Sigma'(\alpha))$ , if for each  $e \in \aleph$ , there is an event  $e' \in C$  such that  $e' \# e$ , then  $C$  is called a deadlock-adjoint configuration.

As mentioned before,  $C_1 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_{26}, e_{27}, e_{28}, e_{29}, e_{38}\}$  is a deadlock-adjoint configuration in Figure 3. After firing all the transitions corresponding to  $C_1$  from the initial marking in Figure 2,  $\Sigma_2$  results at the dead marking  $\{p_1, p_{12}, p_{19}\}$ . This marking corresponds to a real deadlock of Program 1. However, in Figure 1, if we uncomment line 30 of Program 1, the running trace in Table 2 is also feasible for the modified program. The program trace net mined from the trace is still the same as that in Figure 1. With the method proposed in this section, we will still report a potential deadlock for the modified program. However, as mentioned in Section II-A, the detected deadlock becomes a false positive in this case. In fact, since only a part of synchronization operations are captured in the running trace, a program trace net may lose lots of program behavior information. As a result, it may contain infeasible program running traces and leads to false deadlocks. To address the issue, a schedule algorithm to replay potential deadlocks are proposed next.

## V. DEADLOCK REPLAY BASED ON CONFIGURATION-ASSOCIATED LOCK SCHEDULER

### A. DEADLOCK-ADJOINT TRACE NET

To replay a potential program deadlock, we aim to find a lock scheduler which, for each lock object, assigns a thread sequence in which the lock object should be

authorized one by one. In fact, each configuration of a deadlock-adjoint trace net implies a deterministic lock scheduler which leads the net to the configuration-associated marking.

Specifically, let  $\Sigma'(\alpha) = (P', T'; F', M_0, \rho, \tau')$  be a deadlock-adjoint trace net,  $Unf(\Sigma'(\alpha)) = (B, E; G, h)$  be one complete finite prefix of its unfolding, and  $C$  be a configuration of  $Unf(\Sigma'(\alpha))$ . If a subnet of  $Unf(\Sigma'(\alpha))$  consists of all the events in  $C$  and all their input and output places, we call it the *spanned subnet of configuration C*.

According to the definition of configurations, the spanned subnet must be conflict free. It can be used to obtain a lock scheduler as follows. First, for each program lock object  $o$ , let  $p \in P'$  be the lock place satisfying  $\rho(p) = o$ . Since there are no conflicting events in a spanned subnet, we only need to start from the condition corresponding to the initial token of the lock places, compute the thread that newly acquires the lock object one by one, and append the thread to the thread sequence associated with  $o$ . The resulting thread sequence will be the authorization thread sequence of lock object  $o$ . Denote the authorization thread sequence of  $o$  by  $\delta(o)$ . We call the authorization thread sequences of all the lock objects a *configuration-associated lock scheduler* of the multithreaded program.

According to the program operations associated with  $C$ , the authorization thread sequence for each lock object and the lock scheduler can be obtained with Algorithm 2. Its time complexity is  $O(|P_L|^2 + |P_L| * |C|)$ , where  $P_L \subseteq P'$  is the set of lock places,  $|P_L|$  and  $|C|$  represent the cardinality of their respective sets.

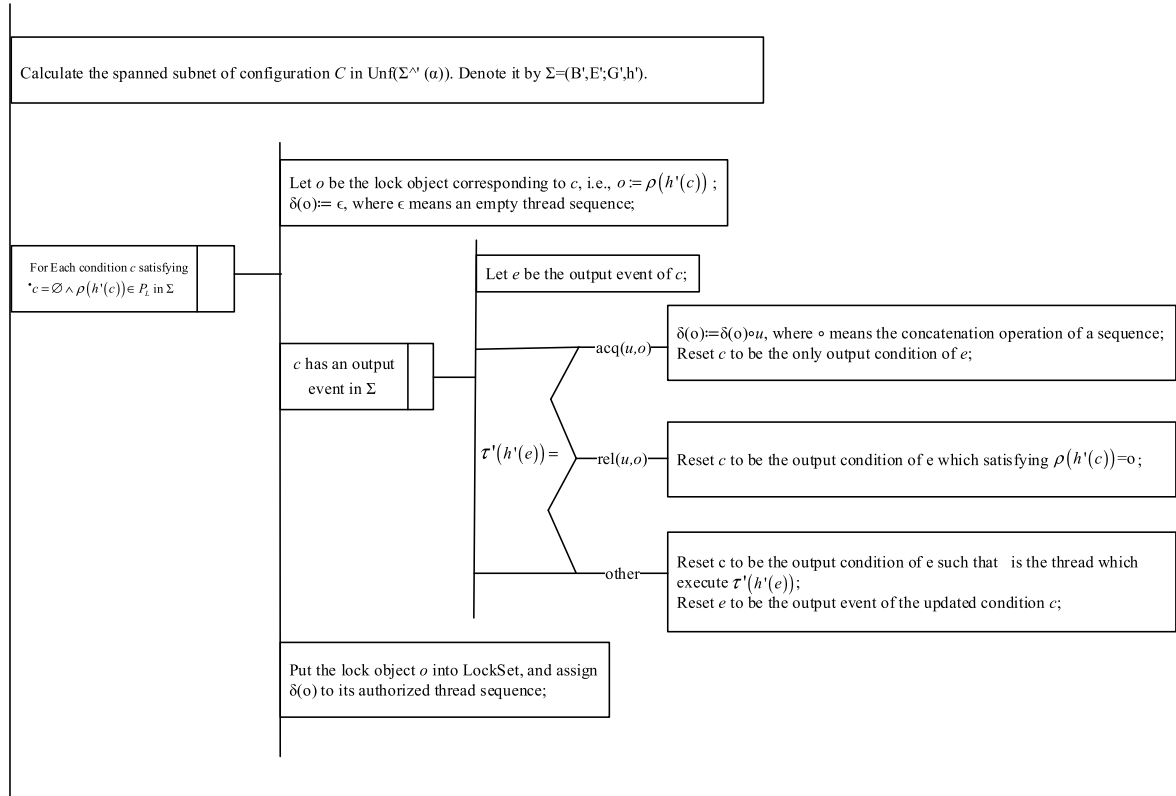


FIGURE 4. Problem analysis diagram of algorithm 2.

Consider for instance the deadlock-adjoint configuration  $C_1 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_{26}, e_{27}, e_{28}, e_{29}, e_{38}\}$  of  $\Sigma_2$ . Its spanned subnet is shown in Figure 4. With Algorithm 2, we can find that  $\delta(G) = \text{ThreadA} \rightarrow \text{ThreadB} \rightarrow \text{ThreadA}$ ,  $\delta(o_1) = \text{ThreadA} \rightarrow \text{ThreadA}$ , and  $\delta(o_2) = \text{ThreadA} \rightarrow \text{ThreadB}$ . It's easy to verify that, if Program 1 runs according to this scheduler, it will fall into a deadlock state. In this way, we can obtain a deterministic lock scheduler for each deadlock-adjoint configuration and its associated potential deadlock. Obviously, the obtained subnet spanned by a deadlock-adjoint configuration clearly describes the process of the program falling into a deadlock. This makes it easy to trace the root cause of a program deadlock. Meanwhile, the obtained lock scheduler is deterministic and easy to understand.

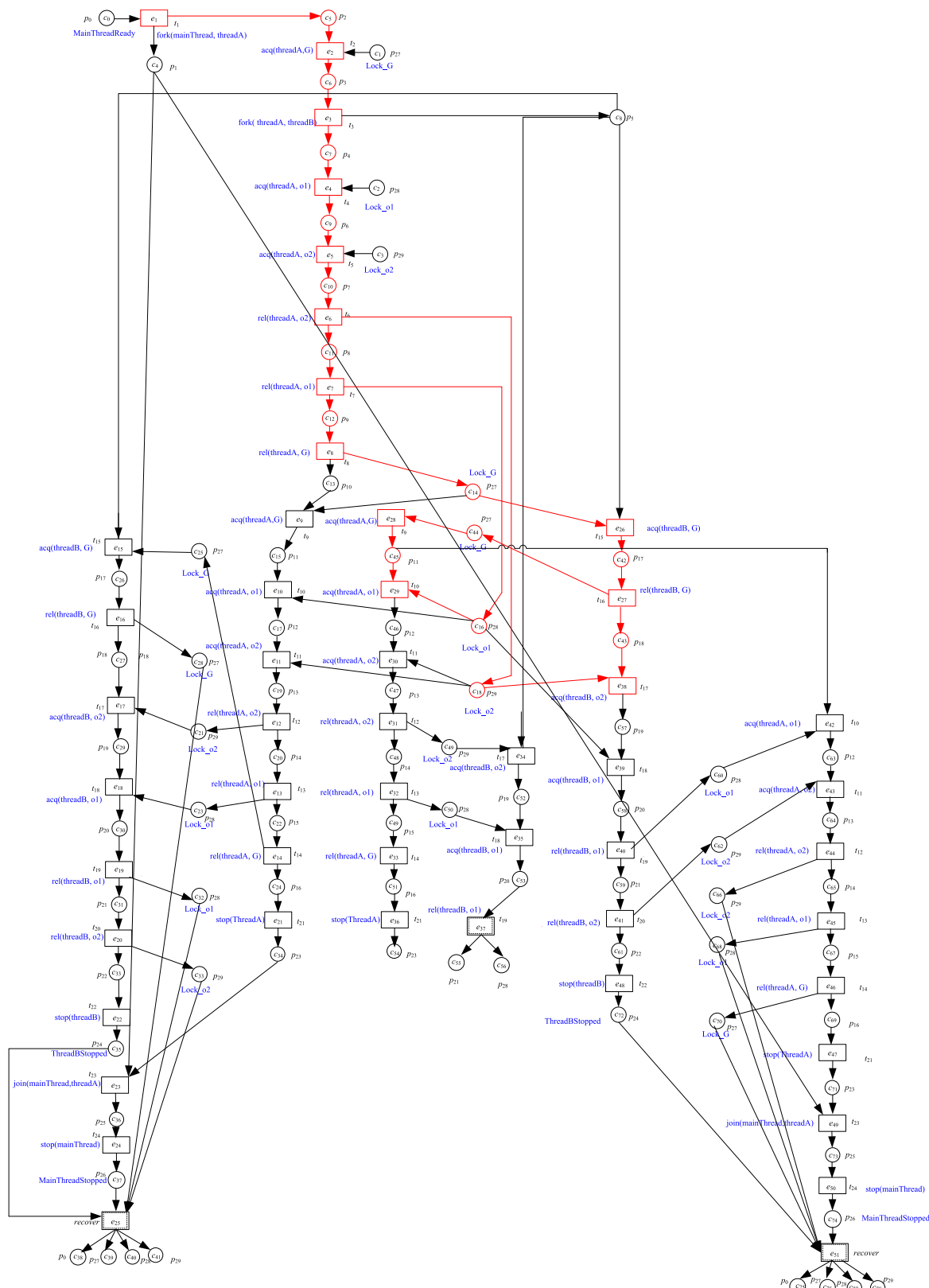
Take the configuration  $\{e_1, e_2, e_3, \dots, e_{25}\}$  of  $\Sigma_2$  as another example. Its spanned subnet is shown in Figure 5. Obviously, all the lock objects  $G, o_1$  and  $o_2$  are authorized to  $\text{ThreadA} \rightarrow \text{ThreadA} \rightarrow \text{ThreadB}$  sequentially. Therefore, its associated lock scheduler is  $\{\delta(G) = \text{ThreadA} \rightarrow \text{ThreadA} \rightarrow \text{ThreadB}, \delta(o_1) = \text{ThreadA} \rightarrow \text{ThreadA} \rightarrow \text{ThreadB}, \delta(o_2) = \text{ThreadA} \rightarrow \text{ThreadB}\}$ . The reached marking under the scheduling of this scheduler is  $\{h(c_{35}), h(c_{37}), h(c_{28}), h(c_{32}), h(c_{33})\} = \{p_{24}, p_{26}, p_{27}, p_{28}, p_{29}\}$ , which is a legal dead marking and corresponds to a normal program termination state.

## B. LOCK SCHEDULER-BASED REPLAY OF PROGRAM POTENTIAL DEADLOCKS

Based on the lock scheduler obtained in previous section, this section develops a scheduling algorithm for deadlock replay. Each time a thread tries to acquire a lock object during the running of a program, the program should decide whether or not the lock should be authorized to the thread according to the lock scheduler.

Specifically, before each acquisition of a lock object  $o$ , we check whether the requesting thread should be authorized according to the first element of  $\delta(o)$ . There are 3 cases: (1) If  $\delta(o)$  is not empty and the requesting thread is not the first element of  $\delta(o)$ , which means that the requesting thread is not the expected one at this moment, we park the requesting thread and put it into the waiting thread set associated with  $o$ ; (2) If  $\delta(o)$  is not empty and the requesting thread is the first element of  $\delta(o)$ , we authorize  $o$  to the requesting thread. Then, update  $\delta(o)$  by deleting its first element. Meanwhile, unpark all the threads in the waiting thread set associated with  $o$ ; As a last case, (3) if  $\delta(o)$  is empty, no intervention is needed. In addition, each time a lock acquisition happens, we check whether a cycle appears in the lock graph spanned by  $C$ . If a cycle is found, it indicates that a program deadlock is triggered. In this case, we set *CycleDetected* to TRUE.

<sup>3</sup>The lock graph spanned by a configuration  $C$  means the sub-graph of the traditional lock graph consisting of the lock acquisition operations associated with  $C$ .



**FIGURE 5.** one complete finite prefix of  $\Sigma 2'$ 's unfolding (The label inner each node is its identifier in the unfolding. The label next to each node is the identifier of its associated node in Figure 2. Events represented by 2-sided rectangles are cut-off events. Events represented in red rectangles constitute a dead-marking associated configuration).

**Algorithm 3** The Scheduling Algorithm of a Multithreaded Program on the Basis of a Configuration-Associated Lock Scheduler

**Input:**  $\Sigma$ , a deadlock-adjoint trace net,  
 $C$ , a deadlock-adjoint configuration,  
 $Program$ , a multithreaded program;

**Output:** Whether or not the potential deadlock associated with  $C$  is a real deadlock

**Steps:**

1. **Call** Algorithm 2 with  $\Sigma$  and  $C$  as the input, and denote the obtained lock set as  $LockSet$ . Let  $\delta$  be the function mapping each lock in  $LockSet$  to its authorization thread sequence.
2. **FOREACH** lock  $o$  in  $LockSet$ , let  $WaitingThreadSet(o)$  be empty, which is used to store the threads parked by the scheduler due to the unexpected acquisition of  $o$ ;
3. Start  $Program$ ;
4. **WHILE** there are threads in RUNNABLE or TIMED\_WAITING state {
5. Let  $op$  be an operation that is scheduled to be executed by the operating system in current state;
6. **IF**  $op$  is a lock acquisition operation in the form of  $acq(threadX, o)$  {
7. **IF**  $\delta(o)$  is not empty and  $threadX$  is the same with the first thread element in  $\delta(o)$  {
8. Execute the operation  $acq(threadX, o)$ ;
9. Delete the first thread element from  $\delta(o)$  immediately after  $acq(threadX, o)$  completed;
10. unpark all the threads in  $WaitingThreadSet(o)$ ;
11. }//IF
12. **ELSE IF**  $\delta(o)$  is not empty and  $threadX$  is not the same with the first thread in  $\delta(o)$
13. Park  $threadX$  and put  $threadX$  into  $WaitingThreadSet(o)$ ;
14. **ELSE** {//in case that  $\delta(o)$  is empty
15. Execute the operation  $acq(threadX, o)$ ;
16. Check whether or not there is a cycle in the lock graph spanned by  $C$ . If there is a loop, let  $CycleDetected$  be true and exit the loop module.
17. }//ELSE
18. }**ELSE** //  $op$  is not a lock acquisition operation
19. Execute the operation  $acq(threadX, o)$  and no intervention is needed;
20. }//WHILE
21. **IF**  $\delta(o)$  is empty for each lock  $o$  and  $CycleDetected$  is true
22. **OUTPUT** that the potential deadlock associated with  $C$  is a real deadlock
23. **ELSE IF**  $\delta(o)$  is empty for each lock  $o$  and all thread of the program terminates normally
24. **OUTPUT** that the potential deadlock associated with  $C$  is a false deadlock;
25. **ELSE**
26. **OUTPUT** that the potential deadlock is not successfully replayed. Whether it is a real deadlock or not is unknown.

During the running of a multithreaded program, we schedule its execution with the policy given above until no threads are in RUNNABLE or TIMED\_WAITING state (which means that no operations can execute). Finally, we check whether  $\delta(o)$  becomes empty for each lock  $o \in LockSet(C)$  and whether  $CycleDetected$  is TRUE. If both are true, the replay procedure succeeds. The potential deadlock associated with  $C$  is a real deadlock. Otherwise, if  $\delta(o)$  becomes empty for each lock  $o \in LockSet$  and all threads of the program terminates normally, the potential deadlock associated with  $C$  is a false deadlock. For the rest cases, we don't know whether it is a true or false deadlock.

The detailed scheduling algorithm to replay a potential deadlock is given in Algorithm 3. Its time complexity is  $O(|C|^2 + |P_L| * |C|)$ , where  $P_L$  is the set of lock places,  $|P_L|$  and  $|C|$  represent the cardinality of their respective sets.

Consider for instance the replay of the deadlock associated with configuration  $C_1$ . We have derived that

$$\delta(G) = ThreadA \rightarrow ThreadB \rightarrow ThreadA, \delta(o_1) = ThreadA \rightarrow ThreadA, \text{ and } \delta(o_2) = ThreadA \rightarrow ThreadB.$$

They constitute a scheduling schema. A possible program running scenario and its scheduling procedure is given in Table 5, where lockBefore, lockAfter, unlockAfter are CalFuzzer<sup>4</sup> probe functions with which we schedule the execution of the program.

As another case, if we uncomment line 30 from Program 1, the obtained potential deadlock along with its adjoint configuration remains the same when analyzing the trace in Table 2. Furthermore, the derived configuration-associated lock scheduler by Algorithm 2 remains unchanged, i.e.,  $\delta(G) = ThreadA \rightarrow ThreadB \rightarrow ThreadA$ ,  $\delta(o_1) = ThreadA \rightarrow ThreadA$ , and  $\delta(o_2) = ThreadA \rightarrow ThreadB$ . Unfortunately, the detected deadlock becomes a false positive this time.

<sup>4</sup>CalFuzzer is an extensible tool for active testing of concurrent Java programs. See literature [32] for details.



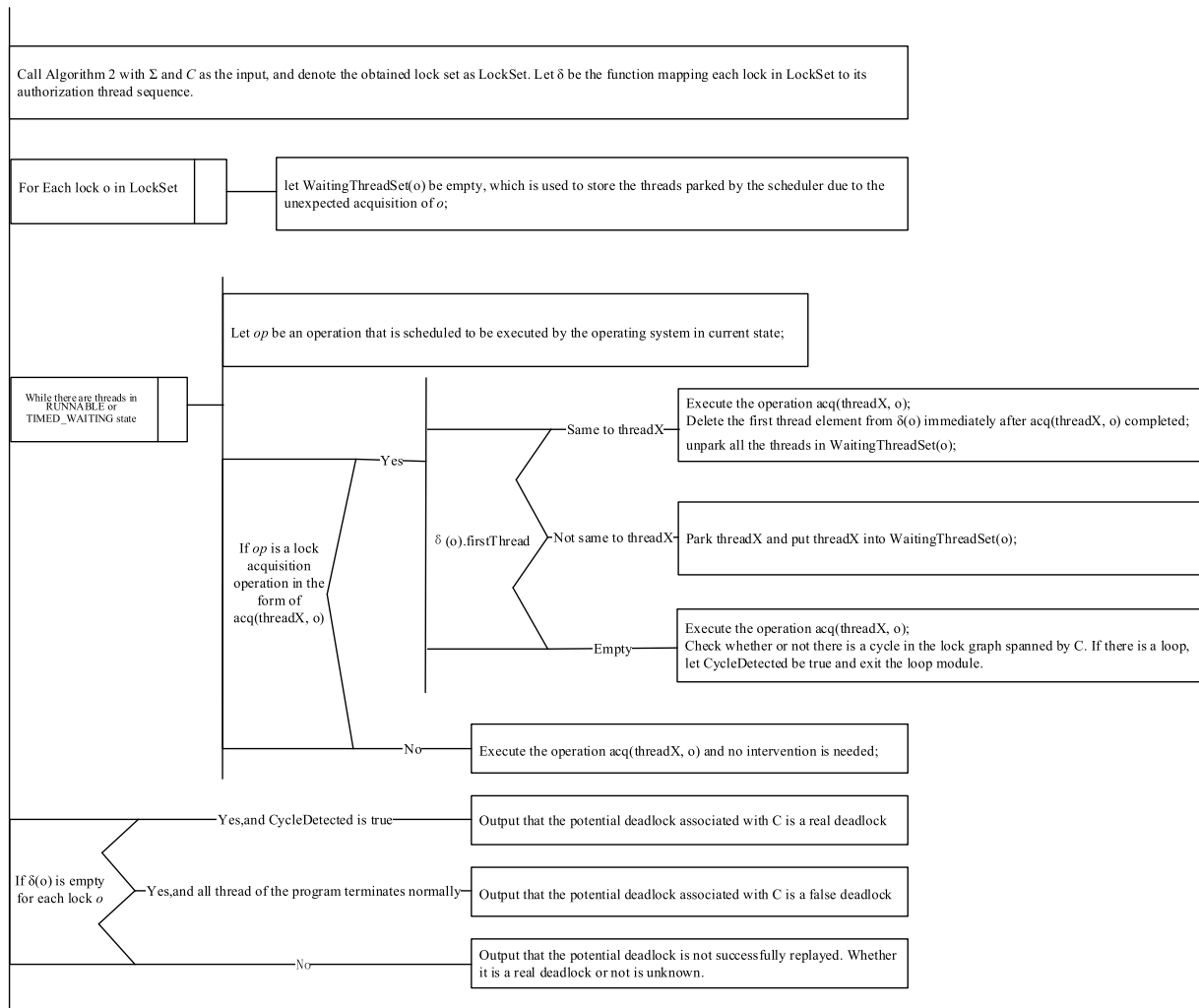


FIGURE 6. Problem analysis diagram of algorithm 3.

Next, let’s see how this false positive is identified by the deadlock replay procedure.

Specifically, in the case that line 30 is uncommented from Program 1, ThreadB can acquire locks only after ThreadA has acquired and subsequently released all the lock objects. However, as shown in Table 6, when ThreadA acquires G for the second time, ThreadA is parked since it is not the first element of  $\delta(G)$ . Now, ThreadA, ThreadB and MainThread are all in a waiting state. Moreover, at least one of  $\delta(o_1)$ ,  $\delta(o_2)$  and  $\delta(G)$  is not empty. This means that we cannot continue scheduling the program’s running as we wish. Therefore, the potential deadlock cannot be replayed successfully. We cannot determine whether the potential deadlock is real or not.

Note that the deadlock detection and replay method proposed in this paper needs to run a program two times. During the first run of the program, a Program trace net and a deadlock-adjoint trace net are constructed by mining the program running trace. On this basis, some potential deadlocks along with their corresponding lock schedulers are derived. During the second run, for each potential deadlock, we sched-

ule the program’s execution according to its corresponding lock scheduler. However, it needs to identify the thread or lock objects that are the “same” in the two different program runs. This identification cannot be done using thread/lock memory address because their addresses may change across different program runs. To address this issue, we need to find the mapping relation between threads and locks in different runs. The detailed mapping method is given in Appendix I.

VI. EXPERIMENTAL EVALUATION

To evaluate the proposed method, we develop a prototype system according to the framework in Figure 6. The working process is as follows: (1) For a given multithreaded program, use CalFuzzer to generate a program running trace at first. As an extensible tool for active testing of concurrent Java programs, CalFuzzer provides a set of clean API for generating an event stream, where an event may be acquisition or releasing of a lock, thread fork/join etc. In addition, the thread stop event of a thread can be captured by checking its thread state repeatedly; (2) Based on the program running

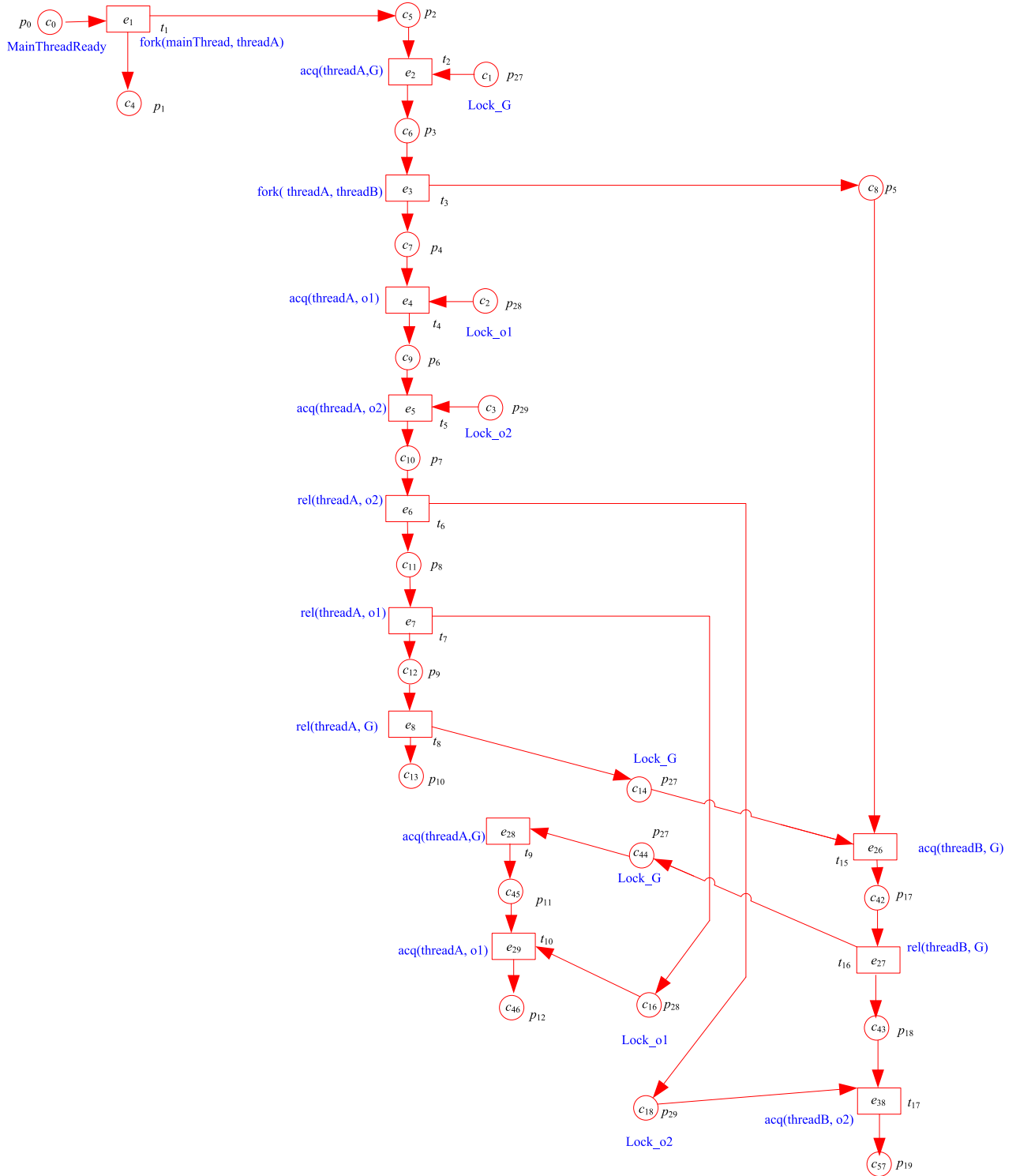
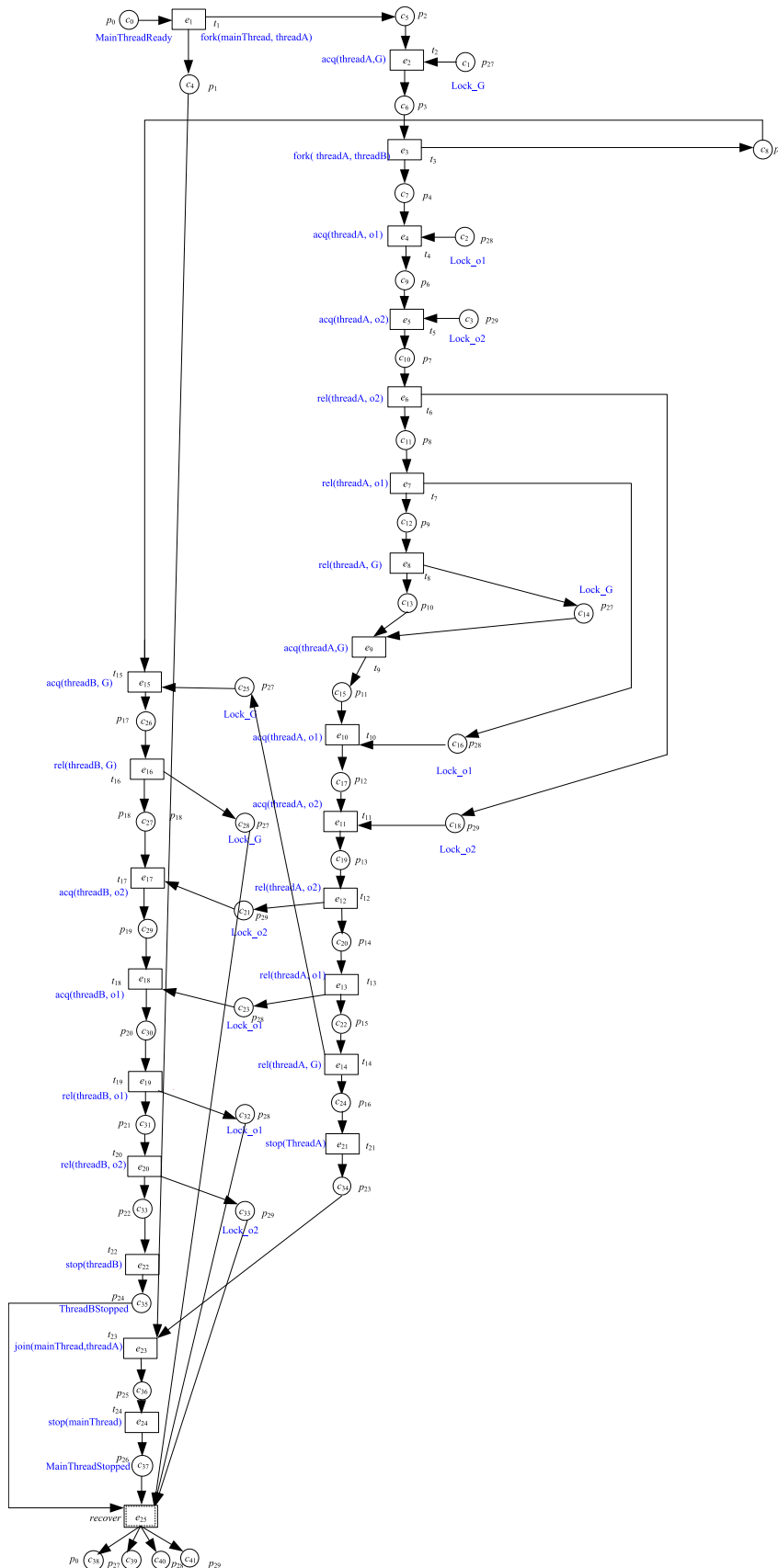


FIGURE 7. The unfolding subnet spanned by the deadlock-adjoint configuration  $C_1$ , which leads to a deadlock of program 1.

trace just obtained, the prototype system construct its corresponding program trace net and deadlock-adjoint trace net with Algorithm 1; (3) Detect the reachable dead markings of the deadlock-adjoint trace net using Petri net unfolding technology. Each of the dead markings means a potential program deadlock; (4) Derive the configuration-associated lock

scheduler for each potential deadlock using Algorithm 2; (5) Replay the potential deadlock on CalFuzzer with Algorithm 3.

We have conducted experiments on the Program 1 presented in Table 1 and the multithreaded program examples disclosed in the Calfuzzer open source project [57]. The



**FIGURE 8.** The unfolding subnet spanned by configuration  $\{e_1, e_2, e_3, \dots, e_{25}\}$ , which leads program 1 to a normal termination state.

**TABLE 5.** A scheduling procedure for a possible running trace of program 1 (operations in yellow filled cells are suspended by the scheduler. red font operations are blocked points of each thread).

No.	MainThread	ThreadA	ThreadB	Scheduling measures based on CalFuzzer	Remarks
1	2:fork(MainThread, ThreadA)				For operations that are not lock acquisition or release, no intervention is needed.
2		7:acq(ThreadA,G)		$\begin{aligned} &\text{lockBefore:}\{ \} \\ &\text{lockAfter:}\{ \\ &\delta(G):=\delta(G)-threadA \\ &\} \end{aligned}$	The initial value of $\delta(G)$ is $\langle \text{ThreadA}, \text{ThreadB}, \text{ThreadA} \rangle$ . Since ThreadA is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(G)$ , and update it to $\langle \text{ThreadB}, \text{ThreadA} \rangle$ .
3		9:fork(ThreadA,ThreadB)			No intervention is needed for operations that are not lock acquisition or release.
4		12: acq(ThreadA,o1)		$\begin{aligned} &\text{lockBefore:}\{ \} \\ &\text{lockAfter:}\{ \\ &d(o_1)=d(o_1)-threadA \\ &\} \end{aligned}$	The initial value of $\delta(o_1)$ is $\langle \text{ThreadA}, \text{ThreadA} \rangle$ . Since ThreadA is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(o_1)$ , and update it to $\langle \text{ThreadA} \rangle$ .
5		13: acq(ThreadA,o2)		$\begin{aligned} &\text{lockBefore:}\{ \} \\ &\text{lockAfter:}\{ \\ &d(o_2)=d(o_2)-threadA \\ &\} \end{aligned}$	The initial value of $\delta(o_2)$ is $\langle \text{ThreadA}, \text{ThreadB} \rangle$ . Since ThreadA is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(o_2)$ , and update it to $\langle \text{ThreadB} \rangle$ .
6		13:rel(ThreadA,o2)		<b>unlockAfter:</b> { }	Since the current waiting thread pool associated with $o_2$ is empty, no threads need to be awakened in the <b>unlockAfter</b> function.
7		14:rel(ThreadA,o1)		<b>unlockAfter:</b> { }	Since the current waiting thread pool associated with $o_1$ is empty, no threads need to be awakened in the <b>unlockAfter</b> function.
8		15:rel(ThreadA,G)		<b>unlockAfter:</b> { }	Since the current waiting thread pool associated with $G$ is empty, no threads need to be awakened in the <b>unlockAfter</b> function.
9			21:acq(ThreadB,G)	$\begin{aligned} &\text{lockBefore:}\{ \} \\ &\text{lockAfter:}\{ \\ &\delta(G):=\delta(G)-threadB \\ &\} \end{aligned}$	The current value of $\delta(G)$ is $\langle \text{ThreadB}, \text{ThreadA} \rangle$ . Since ThreadB is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(G)$ , and update it to $\langle \text{ThreadA} \rangle$ .
10			21:rel(ThreadB,G)	<b>unlockAfter:</b> { }	Since the current waiting thread pool associated with $G$ is empty, no threads need to be awakened in the <b>unlockAfter</b> function.
11		07:acq(ThreadA,G)		$\begin{aligned} &\text{lockBefore:}\{ \} \\ &\text{lockAfter:}\{ \\ &\delta(G):=\delta(G)-threadA \\ &\} \end{aligned}$	The current value of $\delta(G)$ is $\langle \text{ThreadA} \rangle$ . Since ThreadA is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(G)$ , and update it to be empty.
12			22:acq(ThreadB,o2)	$\begin{aligned} &\text{lockBefore:}\{ \} \\ &\text{lockAfter:}\{ \\ &\delta(o_2):=\delta(o_2)-threadB \\ &\} \end{aligned}$	The current value of $\delta(o_2)$ is $\langle \text{ThreadB} \rangle$ . Since ThreadB is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(o_2)$ , and update it to be empty.
13			23:acq(ThreadB,o1)	$\begin{aligned} &\text{lockBefore:}\{ \\ &\text{LockSupport.park}(\text{ThreadB}) \\ &\quad ; \\ &o1.\text{WaitingThreads} \leftarrow \text{Thread} \\ &\quad \text{B} \\ &\} \end{aligned}$	The current value of $\delta(o_1)$ is $\langle \text{ThreadA} \rangle$ . Since ThreadB is not the first element, ThreadB is parked in the <b>lockBefore</b> function, and added ThreadB to the waiting thread pool of $o_1$ . The <b>lockAfter</b> function cannot execute Since the thread is parked.
14		12: acq(ThreadA,o1)		$\begin{aligned} &\text{lockBefore:}\{ \} \\ &\text{lockAfter:}\{ \\ &d(o_1)=d(o_1)-threadA ; \\ &\text{LockSupport.unpark}(\text{ThreadB}) \\ &\quad ; \end{aligned}$	The current value of $\delta(o_1)$ is $\langle \text{ThreadA} \rangle$ . Since ThreadA is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(o_1)$ , and update it to be empty.

**TABLE 5. (Continued.) A scheduling procedure for a possible running trace of program 1 (operations in yellow filled cells are suspended by the scheduler, red font operations are blocked points of each thread).**

					At this time, since $\delta(o_1)$ , $\delta(o_2)$ and $\delta(G)$ all becomes empty, <b>awaken all those parked threads in the waiting thread pool associated with each lock object.</b> No further scheduling is required.
15			23:acq(ThreadB,o1)	lockBefore: { }	Since the authorized thread sequence of all lock objects becomes empty, no further scheduling is required. Note that o1 is held by ThreadA. In this case, ThreadB is blocked.
16		13: acq(ThreadA,o2)		lockBefore: { }	Since the authorized thread sequence of all lock objects becomes empty, no further scheduling is required. Since o2 is held by ThreadB, ThreadA is blocked. At this time, due to the cyclic waiting of ThreadA and ThreadB on lock objects o2 and o1, a deadlock is triggered. Obviously, both lock acquisition operations appeared in $C_1$ . Moreover, considering that $\delta(o_1)$ , $\delta(o_2)$ and $\delta(G)$ are all empty, <b>the replay procedure succeeded and the potential deadlock corresponding to <math>C_1</math> is a real one.</b>

**TABLE 6. A scheduling procedure for a possible running trace of the uncommented program 1 (operations in yellow filled cells are suspended ones).**

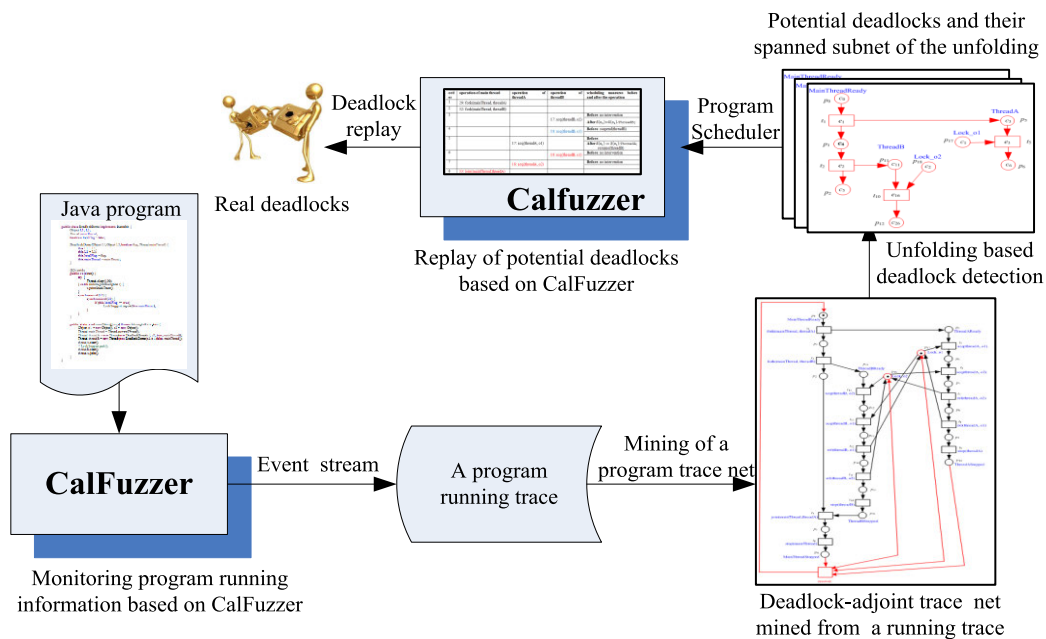
No.	MainThread	ThreadA	ThreadB	Scheduling measures based on CalFuzzer	Remarks
1	2:fork(MainThread, ThreadA)				For operations that are not lock acquisition or release, no intervention is needed.
2		7:acq(ThreadA,G)		lockBefore: { } lockAfter: { } $\delta(G) := \delta(G) - threadA$	The initial value of $\delta(G)$ is <ThreadA, ThreadB, ThreadA>. Since ThreadA is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(G)$ , and update it to <ThreadB, ThreadA>.
3		9:fork(ThreadA, Thread)			No intervention is needed for operations that are not lock acquisition or release.
4		12: acq(ThreadA ,o1)		lockBefore: { } lockAfter: { } $d(o_1) := d(o_1) - threadA$	The initial value of $\delta(o_1)$ is <ThreadA, ThreadA>. Since ThreadA is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(o_1)$ , and update it to <ThreadA>
5		13: acq(ThreadA ,o2)		lockBefore: { } lockAfter: { } $d(o_2) := d(o_2) - threadA$	The initial value of $\delta(o_2)$ is <ThreadA, ThreadB>. Since ThreadA is the first element, no intervention is required in the <b>lockBefore</b> function. In the <b>lockAfter</b> function, delete the first element of $\delta(o_2)$ , and update it to <ThreadB>.
6		13:rel(ThreadA ,o2)		unlockAfter: { }	Since the current waiting thread pool associated with $o_2$ is empty, no threads need to be awakened in the <b>unlockAfter</b> function.
7		14:rel(ThreadA ,o1)		unlockAfter: { }	Since the current waiting thread pool associated with $o_1$ is empty, no threads need to be awakened in the <b>unlockAfter</b> function.
8		15:rel(ThreadA ,G)		unlockAfter: { }	Since the current waiting thread pool associated with $G$ is empty, no threads need to be awakened in the <b>unlockAfter</b> function.
9	00	07:acq(ThreadA ,G)		lockBefore: { } LockSupport.park(ThreadA); $G.WaitingThreads \leftarrow ThreadA$	The current value of $\delta(G)$ is <ThreadB, ThreadA>. Since ThreadA is not the first element, ThreadA is parked in the <b>lockBefore</b> function, and added ThreadA to the waiting thread pool of $o_1$ . The <b>lockAfter</b> function cannot execute since the thread is parked. Now, ThreadA, ThreadB and MainThread are all in waiting state. Moreover, at least one of $\delta(o_1)$ , $\delta(o_2)$ and $\delta(G)$ is not empty. Therefore, <b>the potential deadlock is not successfully replayed. We cannot determine whether the potential deadlock is real or not.</b>

machine configuration during the experiment is: Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz, memory 2 GB,

and the operating system is Ubuntu 5.4.0. The comparison of the experimental results is shown in Table 7.

**TABLE 7. Comparison of experiment results (where iGoodlock is a deadlock detection and replay tool implemented in calfuzzer. PNUlock means the proposed deadlock detection and replay method based on petri net unfolding).**

Program instance	number of detected potential deadlocks		time consumed in deadlock detection stage(s)		number of successfully replayed deadlocks		time consumed in deadlock replay stage (s)	
	iGoodlock	PNUlock	iGoodlock	PNUlock	iGoodlock	PNUlock	iGoodlock	PNUlock
Demo2	2	1	0.135	0.291	1	1	0.968	0.125
Test1a	1	1	0.140	0.682	1	1	0.469	0.126
Test1b	1	1	0.129	0.256	1	1	0.488	0.123
Test2a	0	0	0.141	0.768	0	0	0	0
Test3	1	1	0.142	0.283	1	1	0.513	0.123
Test4	1	1	0.144	0.284	1	1	0.484	0.124
Test6	0	0	0.127	0.927	0	0	0	0
Test7	1	1	0.147	0.260	1	1	0.494	0.123
Test8	4	4	0.161	0.639	4	4	2.441	0.522
Program 1	2	1	0.123	0.539	1	1	0.895	0.129



**FIGURE 9. The unfolding subnet spanned by configuration {e1,e2,e3,...,e25}, which leads Program 1 to a normal termination state.**

Among the existing deadlock detection methods participating in the comparison, iGoodlock uses cyclic lock dependency chain to detect potential deadlocks. It usually produces more false positives compared to our proposed method. The program Demo2 in Table 1 and Program 1 in Table 1 confirmed this point. iGoodlock algorithm report 1 false positive while our proposed method has no false positives.

In addition, for each potential deadlock, iGoodlock performs deadlock replay based on DeadlockFuzzer, which is a random replay strategy. On the contrary, the scheduling scheme proposed in this article is deterministic. As a result, for the same real deadlock, iGoodlock and DeadlockFuzzer usually need multiple runs to replay a real deadlock successfully. E.g., for program Test1a and Test1b, iGoodlock

replayed more than 3 times to trigger the program deadlock in our experiment. However, our proposed method only needs to replay one time.

From the perspective of time performance, in the first stage of deadlock detection, our proposed method needs to consume more time. This is because the program trace net constructed in this paper contains more program behavioral information compared to lock graphs, cyclic lock dependency chain, segmentation graphs, and other existing program models that are mined from a running trace. Moreover, the deadlock detection algorithm of Petri net is usually time consuming due to the problem of state explosion. This decrease in time performance is to obtain higher deadlock detection accuracy. In the second state of deadlock replay, the replay method proposed in this article is more efficient compared to

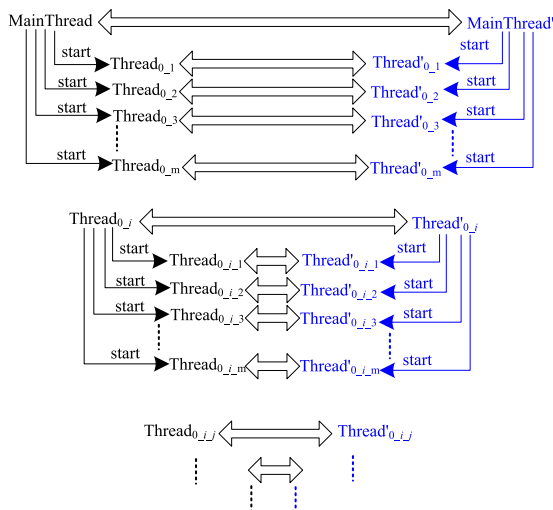


FIGURE 10. The mapping relation between thread identifiers in two different program runs.

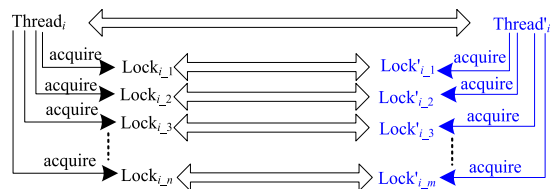


FIGURE 11. The mapping relation between lock identifiers in two different program runs.

iGoodlock, because it is a deterministic scheduling scheme. However, DeadlockFuzzer’s scheduling scheme is random and usually requires multiple runs to replay a deadlock successfully.

VII. CONCLUSION

This work proposed a new method for program deadlock detection and replay. Firstly, it defined the concept of program trace nets to model the program behavior implied in a running trace, and developed the algorithm for mining the trace net from a program running trace. Then, the program deadlock detection problem was transformed to the dead marking detection of a deadlock-adjoint trace net. Each of the dead markings indicates a potential program deadlock. After that, an algorithm was developed to derive a configuration-associated lock scheduler for each potential deadlock. Based on the obtained lock scheduler, a program scheduling algorithm is proposed to replay the potential deadlocks and check their authenticity. It was shown that the proposed deadlock detection and replay method can eliminate more false positives compared to common dynamic deadlock analysis methods. Meanwhile, the obtained deadlock replay scheme is deterministic and easy to understand compared to existing ones.

At the same time, there are still many shortcomings to be improved in this work. First, when a program trace net

contains a lot of branch structures, its unfolding may have the problem of state explosion. In the future work, we aim to design a new program model which contains the same program behavior information but is more efficient in analysis. Secondly, this paper only detects program deadlocks from one running trace. Multiple running traces can contain more program structure and behavior information. When discovering a Petri net-based program model from multiple running traces, the decision statements including if-else, case, and loop statements should be modeled by unobservable transitions. On this issue, [58] proposes an optimization-based approach to discover the unobservable behavior of a discrete event system through interpreted Petri nets. It improves previous approaches by evaluating the accuracy of the identified model with respect to sequences of transitions and markings. This work is expected to solve the above-mentioned problem of mining unobservable transitions when discover a program model from multiple running traces. Furthermore, the identification method of time Petri net models presented in [59] can also help us to mine a program model that incorporates execution time information of program operations. The time information can improve the accuracy of program deadlock detection.

In addition, this paper mainly deals with resource deadlocks. Except for them, communication deadlocks may also occur due to improper use of wait, notify, park, unpark and other concurrency primitives [60]. Communication deadlocks can also be detected if all of the primitives are modeled correctly with Petri nets. However, due to the large randomness in the execution order of these operations, the state space of the program and its Petri net model will increase dramatically. Since [61] and [62] have proved that the deadlock detection problem is PSPACE-complete for RRNs (Petri nets of Resource Allocation), the unified detection of resource deadlocks and communication deadlocks may lead to a greater time complexity. Therefore, we tend to study the two types of problems separately, and this paper only focuses on detection of resource deadlocks. Moreover, deadlock fixes [63] and many deadlock-related issues in flexible manufacturing systems [34], [35], [64], [65], [66], particularly phantom deadlocks that may occur in the current state but disappear in subsequent states, deserve further attention. It’s crucial to note that phantom deadlocks have the potential to be triggered from the initial state, whereas the false positives discussed in this paper are not. Instead, false positives occur due to inaccuracies in the detection algorithm’s analysis of program behavior.

Finally, except for the deadlock freedom property, data consistency and soundness are also important aspects of concurrent systems [67], [68], [69], [70], [71], [72], [73]. References [44], [74], and [75] have proposed several Petri net unfolding-based methods to verify these properties, including verifying CTL (Computation Tree Logic), detecting data inconsistency, and checking soundness of workflow systems. They all analyze the static model of the systems and have made good progress. In the follow-up, we will carry out

dynamic analysis similar to the method proposed in this article.

## APPENDIX A

The proposed deadlock detection method in this paper needs to run a program two times. In the first run, the Petri net model is constructed by analyzing the program trace. Then, the potential deadlock along with its corresponding lock scheduler is obtained. In the second run, according to the lock scheduler, we schedule the program's execution according to the lock scheduler. However, it needs to identify the thread or lock objects that are the "same" in the two different program runs. This identification cannot be done using thread/lock memory address because their addresses may change across different program runs. To address this issue, we need to find the mapping relation between threads and locks in different runs.

Firstly, let's derive the thread mapping relation between two different program runs. In most dynamic analysis tools of concurrent programs (e.g., CalFuzzer [36]), the main thread usually has a special identifier (denoted by MainThread). Hence, it is easy to derive the mapping relation between the two main threads in different runs. Next, we assume that the two different program runs have the same input and that both of them terminate normally without running into a deadlock. Moreover, without loss of generality, we can suppose that both runs perform the same set of operations except that the operations' execution order may be different. Then, the sequence of threads started by the main thread is consistent in order even in different program runs. Based on this consistency, we can obtain the mapping relation for those threads started by the main thread. In the same way, for each thread we have obtained its mapping relation, we can derive the mapping relation for those threads started by it. The principle to generate the thread mapping relation is shown in Figure 7. The thread identifiers listed in the left part are from one program run, the others from another run. A one-way arrow pointing from Thread<sub>x</sub> to Thread<sub>x,i</sub> means Thread<sub>x,i</sub> is the *i*-th thread started by Thread<sub>x</sub>. A two-way arrow pointing from Thread<sub>x</sub> to Thread'<sub>x</sub> means they are the same thread in different program runs.

As for the mapping relation of lock objects between two different program runs, the derivation is similar to that of threads. On the premise that the two different program runs have the same input and that both of them terminate normally without running into a deadlock, we can suppose that a thread acquire the same set of locks in the same order even though it is executed in two different program runs. Based on this consistency, we can obtain the mapping relation for those locks acquired by a same thread. The generation principle of the lock mapping relation is shown in Figure 8, where Thread<sub>*i*</sub> and Thread'<sub>*i*</sub> are the same thread in two different runs, Lock<sub>*i,j*</sub> is the *j*-th lock object acquired by Thread<sub>*i*</sub>, and Lock'<sub>*i,j*</sub> the *j*-th lock object acquired by Thread'<sub>*i*</sub>. Obviously Lock<sub>*i,j*</sub> and Lock'<sub>*i,j*</sub> are the same object in different program runs.

## REFERENCES

- [1] F. A. Bianchi, A. Margara, and M. Pezzè, "A survey of recent trends in testing concurrent software systems," *IEEE Trans. Softw. Eng.*, vol. 44, no. 8, pp. 747–783, Aug. 2018, doi: [10.1109/TSE.2017.2707089](https://doi.org/10.1109/TSE.2017.2707089).
- [2] J. Jahić, T. Bauer, T. Kuhn, N. Wehn, and P. O. Antonino, "FERA: A framework for critical assessment of execution monitoring based approaches for finding concurrency bugs," in *Intelligent Computing: Proceedings of the 2020 Computing Conference*, vol. 1. Springer, 2020, pp. 54–74.
- [3] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2008, pp. 329–339.
- [4] Y. Cai and W. K. Chan, "Lock trace reduction for multithreaded programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 12, pp. 2407–2417, Dec. 2013, doi: [10.1109/TPDS.2013.13](https://doi.org/10.1109/TPDS.2013.13).
- [5] Y. Zhang, H. Liu, and L. Qiao, "Context-sensitive data race detection for concurrent programs," *IEEE Access*, vol. 9, pp. 20861–20867, 2021.
- [6] L. Zheng, X. Liao, S. Wu, X. Fan, and H. Jin, "Understanding and identifying latent data races cross-thread interleaving," *Frontiers Comput. Sci.*, vol. 9, no. 4, pp. 524–539, Aug. 2015.
- [7] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *Proc. Haifa Verification Conf.* Springer, 2005, pp. 191–207.
- [8] W. Duo, X. Jiang, O. Karoui, X. Guo, D. You, S. Wang, and Y. Ruan, "A deadlock prevention policy for a class of multi-threaded software," *IEEE Access*, vol. 8, pp. 16676–16688, 2020, doi: [10.1109/ACCESS.2020.2964312](https://doi.org/10.1109/ACCESS.2020.2964312).
- [9] M. Gómez-Zamalloa and M. Isabel, "Deadlock-guided testing," *IEEE Access*, vol. 9, pp. 46033–46048, 2021, doi: [10.1109/ACCESS.2021.3065421](https://doi.org/10.1109/ACCESS.2021.3065421).
- [10] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 110–120, May 2009.
- [11] S. A. Polyakov and A. E. Borodin, "Deadlock detection using static analysis," *Proc. Inst. Syst. Program. RAS*, vol. 32, no. 5, pp. 21–34, 2020.
- [12] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for Java libraries," in *ECOOP 2005—Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25–29, 2005. Proceedings 19*. Berlin, Germany: Springer, 2005, pp. 602–629.
- [13] Y. Cai and W. K. Chan, "Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs," *IEEE Trans. Softw. Eng.*, vol. 40, no. 3, pp. 266–281, Mar. 2014, doi: [10.1109/TSE.2014.2301725](https://doi.org/10.1109/TSE.2014.2301725).
- [14] Z. D. Luo, R. Das, and Y. Qi, "Multicore SDK: A practical and efficient deadlock detector for real-world applications," in *Proc. 4th IEEE Int. Conf. Softw. Test., Verification Validation*, Mar. 2011, pp. 309–318.
- [15] M. Ghorbani and S. M. Babamir, "Runtime deadlock tracking and prevention of concurrent multithreaded programs: A learning-based approach," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 10, p. e5324, May 2020, doi: [10.1002/cpe.5324](https://doi.org/10.1002/cpe.5324).
- [16] H. C. Tung, U. Mathur, A. Pavlogiannis, and M. Viswanathan, "Sound dynamic deadlock prediction in linear time," in *Proc. ACM Program. Lang.*, vol. 7, 2023, pp. 1733–1758.
- [17] J. Zhou, H. Yang, J. Lange, and T. Liu, "Deadlock prediction via generalized dependency," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2022, pp. 455–466.
- [18] V. K. Shanbhag, "Deadlock-detection in Java-library using static-analysis," in *Proc. 15th Asia-Pacific Softw. Eng. Conf.*, Dec. 2008, pp. 361–368.
- [19] D. Kim and Y. Choi, "Comparison of false alarm detection using KLEE and CBMC for effective multitask program verification," *J. KIISE*, vol. 48, no. 2, pp. 174–182, Feb. 2021.
- [20] J. J. Harrow Jr., "Runtime checking of multithreaded applications with visual threads," in *Proc. Int. SPIN Workshop Model Checking Softw.* Berlin, Germany: Springer, 2000, pp. 331–342.
- [21] K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *Proc. 7th Int. SPIN Workshop SPIN Model Checking Softw. Verification*, 2001, pp. 245–264.
- [22] S. Bensalem and K. Havelund, "Dynamic deadlock analysis of multi-threaded programs," in *Proc. Haifa Verification Conf.*, 2005, pp. 208–223.
- [23] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang, "Detection of deadlock potentials in multithreaded programs," *IBM J. Res. Develop.*, vol. 54, no. 5, pp. 3:1–3:15, Sep. 2010, doi: [10.1147/JRD.2010.2060276](https://doi.org/10.1147/JRD.2010.2060276).

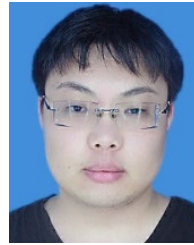


- [24] M. Samak and M. K. Ramanathan, "Trace driven dynamic deadlock detection and reproduction," *ACM SIGPLAN Notices*, vol. 49, no. 8, pp. 29–42, Nov. 2014.
- [25] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, 1981.
- [26] K. L. McMillan, "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits," in *Computer Aided Verification: Fourth International Workshop, CAV'92 Montreal, Canada, June 29–July 1, 1992 Proceedings 4*. Berlin, Germany: Springer, 1993, pp. 164–177.
- [27] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 167–178, Mar. 2010, doi: [10.1145/1735970.1736040](https://doi.org/10.1145/1735970.1736040).
- [28] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. Usenix Conf. Operating Syst. Design Implement.*, vol. 8, 2009, pp. 209–224.
- [29] Y. Cai and W. K. Chan, "MagicFuzzer: Scalable deadlock detection for large-scale applications," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 606–616, doi: [10.1109/ICSE.2012.6227156](https://doi.org/10.1109/ICSE.2012.6227156).
- [30] Y. Cai, C. Jia, S. Wu, K. Zhai, and W. K. Chan, "ASN: A dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 13–23, Jan. 2015, doi: [10.1109/TPDS.2014.2307864](https://doi.org/10.1109/TPDS.2014.2307864).
- [31] Y. Cai and Q. Lu, "Dynamic testing for deadlocks via constraints," *IEEE Trans. Softw. Eng.*, vol. 42, no. 9, pp. 825–842, Sep. 2016, doi: [10.1109/TSE.2016.2537335](https://doi.org/10.1109/TSE.2016.2537335).
- [32] Y. Cai, S. Wu, and W. K. Chan, "ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs," in *Proc. 36th Int. Conf. Softw. Eng.*, Hyderabad, India: ACM, May 2014, pp. 491–502, doi: [10.1145/2568225.2568312](https://doi.org/10.1145/2568225.2568312).
- [33] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *Java Language Specification, Java SE 8 Edition*. Reading, MA, USA: Addison-Wesley, 2014.
- [34] Y. Chen, Z. Li, A. Al-Ahmari, N. Wu, and T. Qu, "Deadlock recovery for flexible manufacturing systems modeled with Petri nets," *Inf. Sci.*, vol. 381, pp. 290–303, Mar. 2017, doi: [10.1016/j.ins.2016.11.011](https://doi.org/10.1016/j.ins.2016.11.011).
- [35] Z. Li, N. Wu, and M. Zhou, "Deadlock control of automated manufacturing systems based on Petri nets—A literature review," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 4, pp. 437–462, Jul. 2012.
- [36] M. Uzam, G. Gelen, and T. L. Saleh, "Think-globally-act-locally approach with weighted arcs to the synthesis of a liveness-enforcing supervisor for generalized Petri nets modeling FMSs," *Inf. Sci.*, vol. 363, pp. 235–260, Oct. 2016.
- [37] Q. Hu, Y. Du, and S. Yu, "Service net algebra based on logic Petri nets," *Inf. Sci.*, vol. 268, pp. 271–289, Jun. 2014.
- [38] B. Vázquez-Barreiros, M. Mucientes, and M. Lama, "ProDiGen: Mining complete, precise and minimal structure process models with a genetic algorithm," *Inf. Sci.*, vol. 294, pp. 315–333, Feb. 2015.
- [39] S. Wang, M. Gan, M. Zhou, and D. You, "A reduced reachability tree for a class of unbounded Petri nets," *IEEE/CAA J. Autom. Sinica*, vol. 2, no. 4, pp. 345–352, Oct. 2015.
- [40] S. Zhang, N. Wu, Z. Li, T. Qu, and C. Li, "Petri net-based approach to short-term scheduling of crude oil operations with less tank requirement," *Inf. Sci.*, vol. 417, pp. 247–261, Nov. 2017.
- [41] M. Brezovan and S. Liana, "Modeling dynamic systems with high-level Petri nets with object-orientation," in *Proc. Int. Symp. Innov. Intell. Syst. Appl. (INISTA)*, Aug. 2016, pp. 1–8, doi: [10.1109/INISTA.2016.7571845](https://doi.org/10.1109/INISTA.2016.7571845).
- [42] S. M. Shatz and W. K. Cheng, "A Petri net framework for automated static analysis of ada tasking behavior," *J. Syst. Softw.*, vol. 8, no. 5, pp. 343–359, Dec. 1988.
- [43] J. Sun, G. Liu, D. Xiang, and C. Jiang, "A Petri-net-based method for detecting bugs in multiple threads," in *Proc. IEEE 16th Int. Conf. Netw., Sens. Control (ICNSC)*, May 2019, pp. 150–156.
- [44] D. Xiang, G. Liu, C. Yan, and C. Jiang, "Detecting data inconsistency based on the unfolding technique of Petri nets," *IEEE Trans. Ind. Informat.*, vol. 13, no. 6, pp. 2995–3005, Dec. 2017.
- [45] S. M. Shatz, K. Mai, C. Black, and S. Tu, "Design and implementation of a Petri net based toolkit for Ada tasking analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 4, pp. 424–441, Oct. 1990.
- [46] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.
- [47] Z. Ding, M. Pan, R. Yang, C. Jiang, and M. Zhou, "Fully expanded tree for property analysis of one-place-unbounded Petri nets," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 47, no. 9, pp. 2574–2585, Sep. 2017, doi: [10.1109/TSMC.2016.2523944](https://doi.org/10.1109/TSMC.2016.2523944).
- [48] J. Li, X. Yu, M. Zhou, and X. Dai, "Lean reachability tree for unbounded Petri nets," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 48, no. 2, pp. 299–308, Feb. 2018.
- [49] F. Lu, Q. Zeng, M. Zhou, Y. Bao, and H. Duan, "Complex reachability trees and their application to deadlock detection for unbounded Petri nets," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 49, no. 6, pp. 1164–1174, Jun. 2019.
- [50] S. Wang, M. Zhou, Z. Li, and C. Wang, "A new modified reachability tree approach and its applications to unbounded Petri nets," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 43, no. 4, pp. 932–940, Jul. 2013.
- [51] K. Barkaoui, J.-M. Couvreur, and K. Klai, "On the equivalence between liveness and deadlock-freeness in Petri nets," in *Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20–25, 2005. Proceedings 26*. Berlin, Germany: Springer, 2005, pp. 90–107.
- [52] K. Barkaoui and J.-F. Pradat-Peyre, "On liveness and controlled siphons in Petri nets," in *Proc. Int. Conf. Appl. Theory Petri Nets*. Berlin, Germany: Springer, 1996, pp. 57–72.
- [53] F. Chu and X.-L. Xie, "Deadlock analysis of Petri nets using siphons and mathematical programming," *IEEE Trans. Robot. Autom.*, vol. 13, no. 6, pp. 793–804, Dec. 1997, doi: [10.1109/70.650158](https://doi.org/10.1109/70.650158).
- [54] J. Esparza, S. Römer, and W. Vogler, "An improvement of McMillan's unfolding algorithm," *Formal Methods Syst. Design*, vol. 20, pp. 285–310, May 2002.
- [55] G. Liu, K. Zhang, and C. Jiang, "Deciding the deadlock and livelock in a Petri net with a target marking based on its basic unfolding," in *Proc. Int. Conf. Algorithms Archit. Parallel Process*. Cham, Switzerland: Springer, 2016, pp. 98–105.
- [56] F. Lu, R. Tao, Y. Du, Q. Zeng, and Y. Bao, "Deadlock detection-oriented unfolding of unbounded Petri nets," *Inf. Sci.*, vol. 497, pp. 1–22, Sep. 2019.
- [57] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "CalFuzzer: An extensible active testing framework for concurrent programs," in *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26–July 2, 2009. Proceedings 21*. Berlin, Germany: Springer, 2009, pp. 675–681.
- [58] F. Basile, G. Faraut, L. Ferrara, and J.-J. Lesage, "An optimization-based approach to discover the unobservable behavior of a discrete-event system through interpreted Petri nets," *IEEE Trans. Autom. Sci. Eng.*, vol. 17, no. 2, pp. 784–798, Apr. 2020, doi: [10.1109/TASE.2019.2944299](https://doi.org/10.1109/TASE.2019.2944299).
- [59] F. Basile, P. Chiacchio, and J. Coppola, "Identification of time Petri net models," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 47, no. 9, pp. 2586–2600, Sep. 2017, doi: [10.1109/TSMC.2016.2523929](https://doi.org/10.1109/TSMC.2016.2523929).
- [60] S. Li, M. Wang, H. Zhang, and Y. Liu, "Program partitioning and deadlock analysis for MPI based on logical clocks," *Parallel Comput.*, vol. 119, Feb. 2024, Art. no. 103061.
- [61] G. Liu, "Complexity of the deadlock problem for Petri nets modeling resource allocation systems," *Inf. Sci.*, vol. 363, pp. 190–197, Oct. 2016.
- [62] G. Liu, "PSPACE-completeness of the soundness problem of safe asymmetric-choice workflow nets," in *Application and Theory of Petri Nets and Concurrency: 41st International Conference, PETRI NETS 2020, Paris, France, June 24–25, 2020, Proceedings 41*. Springer, 2020, pp. 196–216.
- [63] W. Ji, L. Bo, Y. Yuan, and X. Sun, "TDFix: A lightweight tool for fixing deadlocks based on templates," *Sci. Comput. Program.*, vol. 233, Mar. 2024, Art. no. 103073.
- [64] Y. Lu, Y. Chen, Z. Li, and N. Wu, "An efficient method of deadlock detection and recovery for flexible manufacturing systems by resource flow graphs," *IEEE Trans. Autom. Sci. Eng.*, vol. 19, no. 3, pp. 1707–1718, Jul. 2022.
- [65] M. Uzam, Z. Li, M. A. El-Meligy, M. A. Sharaf, and Q. Tang, "Optimality test for control places of Petri net based liveness enforcing supervisors of FMSs," *IEEE Access*, vol. 12, pp. 20031–20046, 2024, doi: [10.1109/ACCESS.2024.3361811](https://doi.org/10.1109/ACCESS.2024.3361811).
- [66] Z. Zhang, G. Liu, K. Barkaoui, and Z. Li, "Adaptive deadlock control for a class of Petri nets with unreliable resources," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 52, no. 5, pp. 3113–3125, May 2022, doi: [10.1109/TSMC.2021.3062469](https://doi.org/10.1109/TSMC.2021.3062469).
- [67] F. Cao, Y. Feng, S. Wang, G. Zhang, and K. Xing, "Deadlock control and hybrid social spider scheduling algorithm for two-stage assembly permutation flowshop with limited buffers," *Expert Syst. Appl.*, vol. 245, Jul. 2024, Art. no. 122744.

- [68] A. L. De Sousa and A. S. De Oliveira, "Finite-time consensus and readjustment three-stage filter for predictive schedules in FMS," *IEEE Access*, vol. 11, pp. 88558–88582, 2023.
- [69] M. S. Elsayed, K. Kefi, and Z. Li, "An optimal transition-based recovery policy for controlling deadlock within flexible manufacturing systems using graph technique," *IEEE Access*, vol. 11, pp. 51723–51739, 2023.
- [70] K. Lee and S. Park, "Geometric zone-control algorithm for collision and deadlock avoidance in AGV system," *IEEE Access*, vol. 11, pp. 131289–131301, 2023.
- [71] H. Li, Q. Liu, M. Liu, B. Sun, and B. Du, "Robust deadlock control for reconfigurable printing manufacturing system based on process algebra," *IEEE Access*, vol. 11, pp. 42473–42484, 2023.
- [72] T. Nakamura, M. Kobayashi, and N. Motoi, "Path planning for mobile robot considering turnabouts on narrow road by deep Q-Network," *IEEE Access*, vol. 11, pp. 19111–19121, 2023.
- [73] T.-C. Row and Y.-L. Pan, "Incomparable single controller for solving deadlock problems of flexible manufacturing systems," *IEEE Access*, vol. 11, pp. 45270–45278, 2023.
- [74] L. Dong, G. Liu, and D. Xiang, "Verifying CTL with unfoldings of Petri nets," in *Algorithms and Architectures for Parallel Processing* (Lecture Notes in Computer Science), J. Vaidya and J. Li, Eds. Cham, Switzerland: Springer, 2018, pp. 47–61, doi: [10.1007/978-3-030-05063-4\\_5](https://doi.org/10.1007/978-3-030-05063-4_5).
- [75] G. Liu, W. Reising, C. Jiang, and M. Zhou, "A branching-process-based method to check soundness of workflow systems," *IEEE Access*, vol. 4, pp. 4104–4118, 2016.



**FENGHUA LV** is currently pursuing the master's degree with the College of Computer Science and Engineering, Shandong University of Science and Technology. Her research interests include program analysis and Petri net.



**MINGHAO CUI** is currently pursuing the master's degree with the College of Computer Science and Engineering, Shandong University of Science and Technology. His research interests include program analysis and Petri net.



**YUNXIA BAO** received the M.S. degree in application mathematics from Shandong University of Science and Technology, Qingdao, China, in 2005. She is currently an Associate Professor with the College of Mathematics and System Science, Shandong University of Science and Technology. Her research interests include statistics and Petri nets.



**QINGTIAN ZENG** received the Ph.D. degree in computer software and theory from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2005.

In 2008, he was a Visiting Professor with the Department of Information Systems, City University of Hong Kong, Hong Kong. He is currently a Professor with the College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao, China. His research

interests include Petri nets, process mining, ontology, and knowledge acquisition and management.

...



**FAMING LU** received the Ph.D. degree in computer software and theory from Shandong University of Science and Technology, Qingdao, China, in 2013.

He is currently a Professor with the Shandong University of Science and Technology. He has published more than 30 articles in academic journals. His research interests include Petri nets and process mining.