Logical Methods in Computer Science
Vol. 9(3:21)2013, pp. 1–17
www.lmcs-online.org

Submitted Oct. 20, 2012
Published Sep. 17, 2013

REACHABILITY UNDER CONTEXTUAL LOCKING *

RÉMI BONNET [a], ROHIT CHADHA [b], P. MADHUSUDAN [c], AND MAHESH VISWANATHAN [d]

[a] LSV, ENS Cachan & CNRS, France
e-mail address: remi.bonnet@cs.ox.ac.uk

[b,c,d] University of Missouri, U.S.A.
e-mail address: chadhar@missouri.edu, madhu@illiniois.edu, vmahesh@illinois.edu

Abstract. The pairwise reachability problem for a multi-threaded program asks, given control locations in two threads, whether they can be simultaneously reached in an execution of the program. The problem is important for static analysis and is used to detect statements that are concurrently enabled. This problem is in general undecidable even when data is abstracted and when the threads (with recursion) synchronize only using a finite set of locks. Popular programming paradigms that limit the lock usage patterns have been identified under which the pairwise reachability problem becomes decidable. In this paper, we consider a new natural programming paradigm, called contextual locking, which ties the lock usage to calling patterns in each thread: we assume that locks are released in the same context that they were acquired and that every lock acquired by a thread in a procedure call is released before the procedure returns. Our main result is that the pairwise reachability problem is polynomial-time decidable for this new programming paradigm as well. The problem becomes undecidable if the locks are reentrant; reentrant locking is a *recursive locking* mechanism which allows a thread in a multi-threaded program to acquire the reentrant lock multiple times.

1. Introduction

In static analysis of sequential programs [8], such as control-flow analysis, data-flow analysis, points-to analysis, etc., the semantics of the program and the data that it manipulates is *abstracted*, and the analysis concentrates on computing fixed-points over a lattice using the control-flow in the program. For instance, in flow-sensitive context-sensitive points-to analysis, a finite partition of the heap locations is identified, and the analysis keeps track of the set of possibilities of which variables point may point to each heap-location partition, propagating this information using the control-flow graph of the program. In fact, several static analysis questions can be formulated as reachability in a pushdown

2012 ACM CCS: [**Theory of computation**]: Semantics and reasoning—Program reasoning—Program verification; [**Software and its engineering**]: Software organization and properties—Software functional properties—Formal methods—Software verification.

Key words and phrases: Static analysis, Pushdown reachability, Locks, Reentrant locks.

* An extended abstract of the paper appeared in [2].

DOI:10.2168/LMCS-9(3:21)2013

system that captures the control-flow of the program (where the stack is required to model recursion) [11].

In concurrent programs, abstracting control-flow is less obvious, due to the various synchronization mechanisms used by threads to communicate and orchestrate their computations. One of the most basic questions is *pairwise reachability*: given two control locations pc_1 and pc_2 in two threads of a concurrent program, are these two locations simultaneously reachable? This problem is very basic to static analysis, as many analysis techniques would, when processing pc_1, take into account the *interference* of concurrent statements, and hence would like to know if a location like pc_2 is concurrently reachable. Data-races can also be formulated using pairwise reachability, as it amounts to asking whether a read/write to a location (or an abstract heap region) is concurrently reachable with a write to the same location (or region). More sophisticated verification techniques like deductive verification can also utilize such an analysis. For instance, in an Owicki-Gries style proof [9] of a concurrent program, the invariant at pc_1 must be proved to be stable with respect to concurrent moves by the environment, and hence knowing whether pc_2 is concurrently reachable will help determine whether the statement at pc_2 need be considered for stability.

Pairwise reachability of control locations is hence an important problem. Given that individual threads may employ recursion, this problem can be *modeled* as reachability of *multiple* pushdown systems that synchronize using the synchronization constructs in the concurrent program, such as locks, barriers, etc. However, it turns out that even when synchronization is limited to using just locks, pairwise reachability is *undecidable* [10]. Consequently, recently, many natural restrictions have been identified under which pairwise reachability is decidable.

One restriction that yields a decidable pairwise reachability problem is *nested locking* [6, 5]: if each thread performs only nested locking (i.e. locks are released strictly in the reverse order in which they are acquired), then pairwise reachability is known to be decidable [6]. The motivation for nested locking is that many high-level locking constructs in programming languages naturally impose nested locking. For instance the `synchronize(o)` $\{\ldots\}$ statement in Java acquires the lock associated with o, executes the body, and releases the lock, and hence nested synchronized blocks naturally model nested locking behaviors. The usefulness of the pairwise reachability problem was demonstrated in [6] where the above decision procedure for nested locking was used to find bugs in the Daisy file system. Nested locking has been generalized to the paradigm of *bounded lock chaining* for which pairwise reachability has also been proved to be decidable [3, 4].

In this paper, we study a different restriction on locking, called *contextual locking*. A program satisfies contextual locking if each thread, in every context, acquires new locks and releases all these locks before returning from the context. Within the context, there is *no requirement* of how locks are acquired and released; in particular, the program can acquire and release locks in a non-nested fashion or have unbounded lock chains.

The motivation for contextual locking comes from the fact that this is a very natural restriction. First, note that it's very natural for programmers to release locks in the same context they were acquired; this makes the acquire and release occur in the same syntactic code block, which is a very simple way of managing lock acquisitions.

Secondly, contextual locking is very much encouraged by higher-level locking constructs in programming languages. For example, consider the code fragment of a method, in Java [7] shown in Figure 1. The above code takes the lock associated with *done* followed later by a lock associated with object r. In order to proceed, it wants *done* to be equal to 1 (a signal

```
public void m() {
  synchronized(done) {
    ...
    synchronized(r) {
      ...
      while (done=0)
      try {
         done.wait();
      }
  ...
}
```

Figure 1: Synchronized blocks in Java

from a concurrent thread, say, that it has finished some activity), and hence the thread waits on *done*, which releases the lock for *done*, allowing other threads to proceed. When some other thread issues a *notify*, this thread wakes up, reacquires the lock for *done*, and proceeds.

Notice that despite having synchronized blocks, the wait() statement causes releases of locks in a non-nested fashion (as it exhibits the sequence *acq lock_done; acq lock_r; rel lock_done; acq lock_done; rel lock_r; rel lock_done*). However, note that the code above does satisfy *contextual locking*; the lock m acquires are all released before the exit, because of the synchronized-statements. Thus, we believe that contextual locking is a natural restriction that is adhered to in many programs.

The first result of this paper is that pairwise reachability is decidable under the restriction of contextual locking. It is worth pointing out that this result does not follow from the decidability results for nested locking or bounded lock chains [6, 3]. Unlike nested locking and bounded lock chains, contextual locking imposes no restrictions on the locking patterns in the absence of recursive function calls; thus, programs with contextual locking may not adhere to the nested locking or bounded lock chains restrictions. Second, the decidability of nested locking and bounded lock chains relies on a non-trivial observation that the number of context switches needed to reach a pair of states is bounded by a value that is *independent* of the size of the programs.[1] However, such a result of a bounded number of context switches does not seem to hold for programs with contextual locking. Thus, the proof techniques used to establish decidability are different as well.

We give a brief outline of the proof ideas behind our decidability result. We observe that if a pair of states is simultaneously reachable by some execution, then they are also simultaneously reachable by what we call a *well-bracketed computation*. A concurrent computation of two threads is not well-bracketed, if in the computation one process, say $\mathcal{P}_0$, makes a call which is followed by the other process ($\mathcal{P}_1$) making a call, but then $\mathcal{P}_0$ returns from its call before $\mathcal{P}_1$ does (but after $\mathcal{P}_1$ makes the call). We then observe that every well-bracketed computation of a pair of recursive programs can simulated by a single recursive program. Thus, decidability in polynomial time follows from observations about reachability in pushdown systems [1].

[1]This observation is implicit in the proofs of decidability in [6, 3].

The second result of the paper concerns *reentrant locks*. The standard mutex locks are blocking, *i.e.*, if a lock is held by some thread, then any attempt to acquire it by any thread (including the *owning* thread) fails and the requesting thread blocks. Some programming languages such as Java support (non-blocking) *reentrant locks*. Reentrant locks are recursive locks; if a thread attempts to acquire a reentrant lock it already holds then the thread succeeds. The lock becomes free only when the owning thread releases the lock as many times as it acquires the lock.

We consider the case of multi-threaded programs synchronizing through *reentrant* locks and show that the pairwise reachability problem is undecidable for contextual reentrant locking (even for non-recursive programs). The undecidability result is obtained by a reduction from the emptiness problem of a 2-counter machine. As the locking mechanism is reentrant, a counter can be simulated by a lock. Counter increment and counter decrement can be simulated by lock acquisition and lock release respectively. The "zero-test" in the reduction is simulated by synchronization between the threads.

The rest of the paper is organized as follows. Section 2 introduces the model of concurrent pushdown systems communicating via locks and presents its semantics. Our main decidability result is presented in Section 3. The undecidability result for contextual reentrant locking is presented in Section 4. Conclusions are presented in Section 5.

Note: An extended abstract of the paper co-authored by Rohit Chadha, P. Madhusudan and Mahesh Viswanathan appeared in [2] and contains the decidability result for pairwise reachability under contextual (non-reentrant) locking. The undecidability result for contextual reentrant locking was obtained subsequently by Rémi Bonnet and Rohit Chadha, and has not been presented elsewhere.

2. Model

The set of natural numbers shall be denoted by $\mathbb{N}$. The set of functions from a set A to B shall be denoted by B^A. Given a function $f \in B^A$, the function $f|_{a \mapsto b}$ shall be the unique function g defined as follows: $g(a) = b$ and for all $a' \neq a$, $g(a') = f(a')$. If $\bar{a} = (a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$ then $\pi_i(a) = a_i$ for each $1 \leq i \leq n$.

2.1. Pushdown Systems.

For static analysis, recursive programs are usually modeled as pushdown systems. Since we are interested in modeling threads in concurrent programs we will also need to model locks for communication between threads. Formally,

Definition: Given a finite set Lcks, a pushdown system (PDS) $\mathcal{P}$ using Lcks is a tuple (Q, Γ, qs, δ) where

- Q is a finite set of control states.
- Γ is a finite stack alphabet.
- qs is the initial state.
- $\delta = \delta_{\mathsf{int}} \cup \delta_{\mathsf{cll}} \cup \delta_{\mathsf{rtn}} \cup \delta_{\mathsf{acq}} \cup \delta_{\mathsf{rel}}$ is a finite set of transitions where
  - $\delta_{\mathsf{int}} \subseteq Q \times Q$.
  - $\delta_{\mathsf{cll}} \subseteq Q \times (Q \times \Gamma)$.
  - $\delta_{\mathsf{rtn}} \subseteq (Q \times \Gamma) \times Q$.
  - $\delta_{\mathsf{acq}} \subseteq Q \times (Q \times \mathsf{Lcks})$.
  - $\delta_{\mathsf{rel}} \subseteq (Q \times \mathsf{Lcks}) \times Q$.

For a PDS $\mathcal{P}$, the semantics is defined as a transition system. The configuration of a PDS $\mathcal{P}$ is the product of the set of control states Q and the stack which is modeled as a word over the stack alphabet Γ. For a thread $\mathcal{P}$ using Lcks, we have to keep track of the locks being held by $\mathcal{P}$. Thus the set of configurations of $\mathcal{P}$ using Lcks is $\mathsf{Conf}_{\mathcal{P}} = Q \times \Gamma^* \times 2^{\mathsf{Lcks}}$ where 2^{Lcks} is the powerset of Lcks.

Furthermore, the transition relation is no longer just a relation between configurations but a binary relation on $2^{\mathsf{Lcks}} \times \mathsf{Conf}_{\mathcal{P}}$ since the thread now *executes* in an *environment*, namely, the free locks (i.e., locks not being held by any other thread). Formally,

Definition: A PDS $\mathcal{P} = (Q, \Gamma, qs, \delta)$ using Lcks gives a labeled transition relation $\longrightarrow_{\mathcal{P}} \subseteq (2^{\mathsf{Lcks}} \times (Q \times \Gamma^* \times 2^{\mathsf{Lcks}})) \times \mathsf{Labels} \times (2^{\mathsf{Lcks}} \times (Q \times \Gamma^* \times 2^{\mathsf{Lcks}}))$ where $\mathsf{Labels} = \{\mathsf{int}, \mathsf{cll}, \mathsf{rtn}\} \cup \{\mathsf{acq}(l), \mathsf{rel}(l) \mid l \in \mathsf{Lcks}\}$ and $\longrightarrow_{\mathcal{P}}$ is defined as follows.

- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{int}}_{\mathcal{P}} \mathsf{fr} : (q', w, \mathsf{hld})$ if $(q, q') \in \delta_{\mathsf{int}}$.
- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{cll}}_{\mathcal{P}} \mathsf{fr} : (q', wa, \mathsf{hld})$ if $(q, (q', a)) \in \delta_{\mathsf{cll}}$.
- $\mathsf{fr} : (q, wa, \mathsf{hld}) \xrightarrow{\mathsf{rtn}}_{\mathcal{P}} \mathsf{fr} : (q', w, \mathsf{hld})$ if $((q, a), q') \in \delta_{\mathsf{rtn}}$.
- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{acq}(l)}_{\mathcal{P}} \mathsf{fr} \setminus \{l\} : (q', w, \mathsf{hld} \cup \{l\})$ if $(q, (q', l)) \in \delta_{\mathsf{acq}}$ and $l \in \mathsf{fr}$.
- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{rel}(l)}_{\mathcal{P}} \mathsf{fr} \cup \{l\} : (q', w, \mathsf{hld} \setminus \{l\})$ if $((q, l), q') \in \delta_{\mathsf{rel}}$ and $l \in \mathsf{hld}$.

2.2. **Multi-pushdown systems.** Concurrent programs are modeled as multi-pushdown systems. For our paper, we assume that threads in a concurrent program communicate only through locks which leads us to the following definition.

Definition: Given a finite set Lcks, a n-pushdown system (n-PDS) $\mathcal{CP}$ communicating via Lcks is a tuple $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ where each $\mathcal{P}_i$ is a PDS using Lcks.

Given a n-PDS $\mathcal{CP}$, we will assume that the set of control states and the stack symbols of the threads are mutually disjoint.

Definition: The semantics of a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ communicating via Lcks is given as a labeled transition system $T = (S, s_0, \longrightarrow)$ where

- S is said to be the set of configurations of $\mathcal{CP}$ and is the set $(Q_1 \times \Gamma_1^* \times 2^{\mathsf{Lcks}}) \times \cdots \times (Q_n \times \Gamma_n^* \times 2^{\mathsf{Lcks}})$, where Q_i is the set of states of $\mathcal{P}_i$ and Γ_i is the stack alphabet of $\mathcal{P}_i$.
- s_0 is the initial configuration and is $((qs_1, \epsilon, \emptyset), \cdots, (qs_m, \epsilon, \emptyset))$ where qs_i is the initial state of $\mathcal{P}_i$.
- The set of labels on the transitions is $\mathsf{Labels} \times \{1, \ldots, n\}$ where $\mathsf{Labels} = \{\mathsf{int}, \mathsf{cll}, \mathsf{rtn}\} \cup \{\mathsf{acq}(l), \mathsf{rel}(l) \mid l \in \mathsf{Lcks}\}$. The labeled transition relation $\xrightarrow{(\lambda, i)}$ is defined as follows
$$((q_1, w_1, \mathsf{hld}_1), \cdots (q_n, w_n, \mathsf{hld}_n)) \xrightarrow{(\lambda, i)} ((q_1', w_1', \mathsf{hld}_1'), \cdots (q_n', w_n', \mathsf{hld}_n'))$$
iff
$$\mathsf{Lcks} \setminus \cup_{1 \leq r \leq n} \mathsf{hld}_r : (q_i, w_i, \mathsf{hld}_i) \xrightarrow{\lambda}_{\mathcal{P}_i} \mathsf{Lcks} \setminus \cup_{1 \leq r \leq n} \mathsf{hld}_r' : (q_i', w_i', \mathsf{hld}_i')$$
and for all $j \neq i$, $q_j = q_j'$, $w_j = w_j'$ and $\mathsf{hld}_j = \mathsf{hld}_j'$.

Notation: Given a configuration $s = ((q_1, w_1, \mathsf{hld}_1), \cdots, (q_n, w_n, \mathsf{hld}_n))$ of a n-PDS $\mathcal{CP}$, we say that $\mathsf{Conf}_i(s) = (q_i, w_i, \mathsf{hld}_i), \mathsf{CntrlSt}_i(s) = q_i, \mathsf{Stck}_i(s) = w_i, \mathsf{LckSt}_i(s) = \mathsf{hld}_i$ and $\mathsf{StHt}_i(s) = |w_i|$, the length of w_i.

2.3. **Computations.** A *computation* of the n-PDS $\mathcal{CP}$, Comp, is a sequence $s_0 \stackrel{(\lambda_1,i_1)}{\longrightarrow} \cdots \stackrel{(\lambda_m,i_m)}{\longrightarrow} s_m$ such that s_0 is the initial configuration of $\mathcal{CP}$. The *label of the computation* Comp, denoted $\mathsf{Label}(\mathsf{Comp})$, is said to be the word $(\lambda_1, i_1)\cdots(\lambda_m, i_m)$. The transition $s_j \stackrel{(\mathsf{cll},i)}{\longrightarrow} s_{j+1}$ is said to be a *procedure call by thread* i. Similarly, we can define *procedure return*, *internal action*, *acquisition of lock* l and *release of lock* l by thread i. A procedure return $s_j \stackrel{(\mathsf{rtn},i)}{\longrightarrow} s_{j+1}$ is said to *match* a procedure call $s_\ell \stackrel{(\mathsf{cll},i)}{\longrightarrow} s_{\ell+1}$ iff $\ell < j$, $\mathsf{StHt}_i(s_\ell) = \mathsf{StHt}_i(s_{j+1})$ and for all $\ell + 1 \le p \le j$, $\mathsf{StHt}_i(s_{\ell+1}) \le \mathsf{StHt}_i(s_p)$.

Example 2.1. Consider the two-threaded program showed in Figure 2. For sake of convenience, we only show the relevant actions of the programs. Figure 3 shows computations whose labels are as follows:

$$\mathsf{Label}(\mathsf{Comp1}) = (\mathsf{cll},0)(\mathsf{acq}(l1),0)(\mathsf{cll},1)(\mathsf{acq}(l2),0)(\mathsf{rel}(l1),0)(\mathsf{acq}(l1),1)\\ (\mathsf{rel}(l2),0)(\mathsf{rtn},0)(\mathsf{rel}(l1),1)(\mathsf{rtn},1)$$

and

$$\mathsf{Label}(\mathsf{Comp2}) = (\mathsf{cll},0)(\mathsf{acq}(l1),0)(\mathsf{cll},1)(\mathsf{acq}(l2),0)(\mathsf{rel}(l1),0)(\mathsf{acq}(l1),1)\\ (\mathsf{rel}(l1),1)(\mathsf{rtn},1)(\mathsf{rel}(l2),0)(\mathsf{rtn},0).$$

respectively.

```
int a(){
   acq l1;
   acq l2;
   if (..) then{
      ....
      rel l2;
      ....
      rel l1;
       };
   else{
       .....
       rel l1
       .....
       rel l2
       };
      return i;
};

public void P0() {
  n=a();
}
```

```
int b(){
    acq l1;
    rel l1;
    return j;
 };

 public void P1() {
   m=b();
 }
```

Figure 2: A 2-threaded program with threads P0 and P1

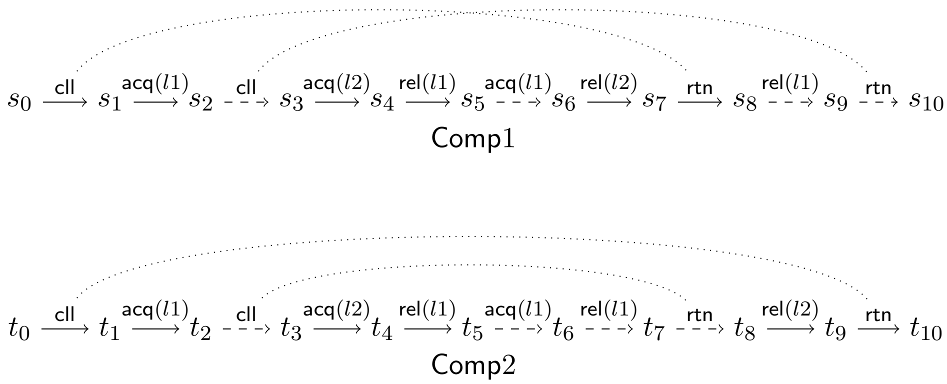

Figure 3: Computations Comp1 and Comp2. Transitions of $P0$ are shown as solid edges while transition of $P1$ are shown as dashed edges; hence the process ids are dropped from the label of transitions. Matching calls and returns are shown with dotted edges.

2.4. **Contextual locking.** In this paper, we are considering the pairwise reachability problem when the threads follow *contextual locking*. Informally, this means that–

- every lock acquired by a thread in a procedure call must be released before the corresponding return is executed, and
- the locks held by a thread just before a procedure call is executed are not released during the execution of the procedure.

Formally,

Definition: A thread i in a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ is said to follow contextual locking if whenever $s_\ell \xrightarrow{(\mathsf{cll},i)} s_{\ell+1}$ and $s_j \xrightarrow{(\mathsf{rtn},i)} s_{j+1}$ are matching procedure call and return along a computation $s_0 \xrightarrow{(\lambda_1,i)} s_1 \cdots \xrightarrow{(\lambda_m,i)} s_m$, we have that

$$\mathsf{LckSt}_i(s_\ell) = \mathsf{LckSt}_i(s_j) \text{ and } \text{ for all } \ell \le r \le j.\ \mathsf{LckSt}_i(s_\ell) \subseteq \mathsf{LckSt}_i(s_r).$$

Example 2.2. Consider the 2-threaded program shown in Figure 2. Both the threads P0 and P1 follow contextual locking. The program P2 in Figure 4 does not follow contextual locking.

Example 2.3. Consider the 2-threaded program in Figure 5. The two threads P3 and P4 follow contextual locking as there is no recursion! However, the two threads do not follow either the discipline of nested locking [6] or of bounded lock chaining [3]. Hence, algorithms of [6, 3] cannot be used to decide the pairwise reachability question for this program. Notice that the computations of this pair of threads require an unbounded number of context switches as the two threads proceed in lock-step fashion. The locking pattern exhibited by these threads can present in any program with contextual locking as long as this pattern is within a single calling context (and not across calling contexts). Such locking patterns when used in a non-contextual fashion form the crux of undecidability proofs for multi-threaded programs synchronizing via locks [6].

```
int a(){
   acq l1;
   rel l2;
   return i;
};
public void P2(){
acq l2;
n=a();
rel l1;
}
```

Figure 4: A program that does not follow contextual locking.

```
public void P3(){
 acq l1;
 while (true){
    acq l2;
    rel l1;
    acq l3;
    rel l2;
    acq l1;
    rel l3;
 }
 }
```

```
public void P4(){
  acq l3;
  while (true){
     acq l1;
     rel l3;
     acq l2;
     rel l1;
     acq l3;
     rel l2;
  }
  }
```

Figure 5: A 2-threaded program with unbounded lock chains

3. Pairwise reachability

The pairwise reachability problem for a multi-threaded program asks whether two given states in two threads can be simultaneously reached in an execution of the program. Formally,

Definition: Given a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ communicating via Lcks, indices $1 \leq i, j \leq n$ with $i \neq j$, and control states q_i and q_j of threads $\mathcal{P}_i$ and $\mathcal{P}_j$ respectively, let $Reach(\mathcal{CP}, q_i, q_j)$ denote the predicate that there is a computation $s_0 \longrightarrow \cdots \longrightarrow s_m$ of $\mathcal{CP}$ such that $\mathsf{CntrlSt}_i(s_m) = q_i$ and $\mathsf{CntrlSt}_j(s_m) = q_j$. The pairwise control state reachability problem asks if $Reach(\mathcal{CP}, q_i, q_j)$ is true.

The pairwise reachability problem for multi-threaded programs communicating via locks was first studied in [10], where it was shown to be undecidable. Later, Kahlon et. al. [6] showed that when the locking pattern is restricted the pairwise reachability problem is decidable. In this paper, we will show that the problem is decidable for multi-threaded programs in which each thread follows contextual locking. Before we show this result, note that it suffices to consider programs with only two threads [6].

Proposition 3.1. Given a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ communicating via Lcks, indices $1 \leq i, j \leq n$ with $i \neq j$ and control states q_i and q_j of $\mathcal{P}_i$ and $\mathcal{P}_j$ respectively, let $\mathcal{CP}_{i,j}$ be the 2-PDS $(\mathcal{P}_i, \mathcal{P}_j)$ communicating via Lcks. Then $Reach(\mathcal{CP}, q_i, q_j)$ iff $Reach(\mathcal{CP}_{i,j}, q_i, q_j)$.

Thus, for the rest of the section, we will only consider 2-PDS.

3.1. **Well-bracketed computations.** The key concept in the proof of decidability is the concept of well-bracketed computations, defined below.

Definition: Let $\mathcal{CP} = (\mathcal{P}_0, \mathcal{P}_1)$ be a 2-PDS via Lcks and let $\mathsf{Comp} = s_0 \stackrel{(\lambda_1,i_1)}{\longrightarrow} \cdots \stackrel{(\lambda_m,i_m)}{\longrightarrow} s_m$ be a computation of $\mathcal{CP}$. Comp is said to be *non-well-bracketed* if there exist $0 \leq \ell_1 < \ell_2 < \ell_3 < m$ and $i \in \{0,1\}$ such that

- $s_{\ell_1} \stackrel{(\mathsf{cll},i)}{\longrightarrow} s_{\ell_1+1}$ and $s_{\ell_3} \stackrel{(\mathsf{rtn},i)}{\longrightarrow} s_{\ell_3+1}$ are matching call and returns of $\mathcal{P}_i$, and
- $s_{\ell_2} \stackrel{(\mathsf{cll},i)}{\longrightarrow} s_{\ell_2+1}$ is a procedure call of thread $\mathcal{P}_{1-i}$ whose matching return either occurs after ℓ_3+1 or does not occur at all.

Furthermore, the triple (ℓ_1, ℓ_2, ℓ_3) is said to be a *witness* of non-well-bracketing of Comp.

Comp is said to be *well-bracketed* if it is not non-well-bracketed.

Example 3.2. Recall the 2-threaded program from Example 2.1 shown in Figure 2. The computation $\mathsf{Comp1}$ (see Figure 3) is non-well-bracketed, while the computation $\mathsf{Comp2}$ (see Figure 3) is well-bracketed. On the other hand, all the computations of the 2-threaded program in Example 2.3 (see Figure 5) are well-bracketed as the two threads are non-recursive.

If there is a computation that simultaneously reaches control states $p \in \mathcal{P}_0$ and $q \in \mathcal{P}_1$ then there is a well-bracketed computation that simultaneously reaches p and q:

Lemma 3.3. Let $\mathcal{CP} = (\mathcal{P}_0, \mathcal{P}_1)$ be a 2-PDS communicating via Lcks such that each thread follows contextual locking. Given control states $p \in \mathcal{P}_0$ and $q \in \mathcal{P}_1$, we have that $\mathit{Reach}(\mathcal{CP}, p, q)$ iff there is a well-bracketed computation $s_0^{wb} \longrightarrow \cdots \longrightarrow s_r^{wb}$ of $\mathcal{CP}$ such that $\mathsf{CntrlSt}_0(s_r^{wb}) = p$ and $\mathsf{CntrlSt}_1(s_r^{wb}) = q$.

Proof. Let $\mathsf{Comp}_{nwb} = s_0 \stackrel{(\lambda_1,i_1)}{\longrightarrow} \cdots \stackrel{(\lambda_m,i_m)}{\longrightarrow} s_m$ be a non-well-bracketed computation that simultaneously reaches p and q. Let ℓ_{mn} be the smallest ℓ_1 such that there is a witness (ℓ_1, ℓ_2, ℓ_3) of non-well-bracketing of Comp_{nwb}. Observe now that it suffices to show that there is another computation Comp_{mod} of the same length as Comp_{nwb} that simultaneously reaches p and q and

- either Comp_{mod} is well-bracketed,
- or if Comp_{mod} is non-well-bracketed then for each witness $(\ell_1', \ell_2', \ell_3')$ of non-well-bracketing of Comp_{mod}, it must be the case $\ell_1' > \ell_{\mathsf{mn}}$.

We show how to construct Comp_{mod}. Observe first that any witness $(\ell_{\mathsf{mn}}, \ell_2, \ell_3)$ of non-well-bracketing of Comp_{nwb} must necessarily agree in the third component ℓ_3. Let ℓ_{rt} denote this component. Let ℓ_{sm} be the smallest ℓ_2 such that $(\ell_{\mathsf{mn}}, \ell_2, \ell_{\mathsf{rt}})$ is a witness of non-well-bracketing of Comp_{mod}. Thus, the transition $s_{\ell_{\mathsf{mn}}} \longrightarrow s_{\ell_{\mathsf{mn}}+1}$ and $s_{\ell_{\mathsf{rt}}} \longrightarrow s_{\ell_{\mathsf{rt}}+1}$ are matching procedure call and return of some thread $\mathcal{P}_r$ while the transition $s_{\ell_{\mathsf{sm}}} \longrightarrow s_{\ell_{\mathsf{sm}}+1}$ is a procedure call of c' by thread $\mathcal{P}_{1-r}$ whose return happens only after ℓ_{rt}. Without loss of generality, we can assume that $r = 0$.

Let $u, (\mathsf{cll}, 0), v_1, (\mathsf{cll}, 1), v_2, (\mathsf{rtn}, 0)$ and w be such that $\mathsf{Label}(\mathsf{Comp}_{nwb}) = u(\mathsf{cll}, 0)v_1(\mathsf{cll}, 1)$ $v_2(\mathsf{rtn}, 0)w$. and length of u is ℓ_{mn}, of $u(\mathsf{cll}, 0)v_1$ is ℓ_{sm} and of $u(\mathsf{cll}, 0)v_1(\mathsf{cll}, 1)v_2$ is ℓ_{rt}. Thus, $(\mathsf{cll}, 0)$ and $(\mathsf{rtn}, 0)$ are matching call and return of thread $\mathcal{P}_0$ and $(\mathsf{cll}, 1)$ is a call of the thread $\mathcal{P}_1$ whose return does not happen in v_2.

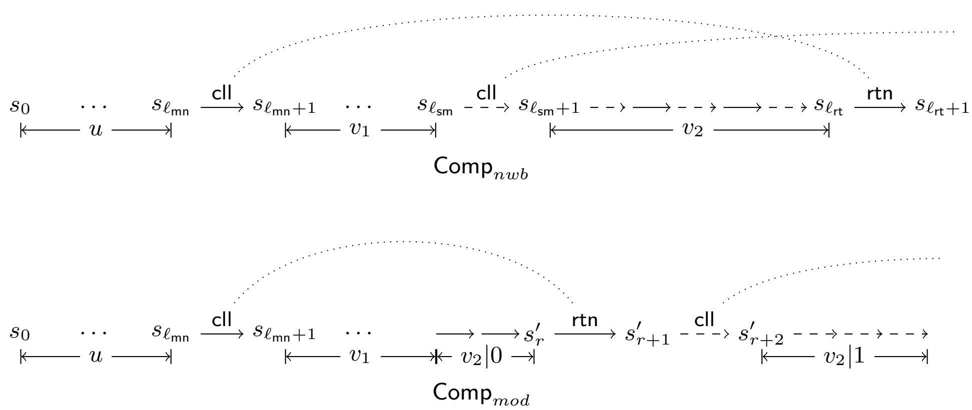

Figure 6: Computations Comp_{nwb} and Comp_{mod}. Transitions of $\mathcal{P}_0$ are shown as solid edges and transitions of $\mathcal{P}_1$ are shown as dashed edges; hence process ids are dropped from the label of transitions. Matching calls and returns are shown with dotted edges. Observe that all calls of $\mathcal{P}_1$ in v_1 have matching returns within v_1.

We construct Comp_{mod} as follows. Intuitively, Comp_{mod} is obtained by "rearranging" the sequence $\mathsf{Label}(\mathsf{Comp}_{nwb}) = u(\mathsf{cll},0)v_1(\mathsf{cll},1)v_2(\mathsf{rtn},0)w$ as follows. Let $v_2|0$ and $v_2|1$ denote all the "actions" of thread $\mathcal{P}_0$ and $\mathcal{P}_1$ respectively in v_2. Then Comp_{mod} is obtained by rearranging $u(\mathsf{cll},0)v_1(\mathsf{cll},1)v_2(\mathsf{rtn},0)w$ to $u(\mathsf{cll},0)v_1(v_2|0)(\mathsf{rtn},0)(\mathsf{cll},1)(v_2|1)w$. This is shown in Figure 6.

The fact that if Comp_{mod} is non-well-bracketed then there is no witness $(\ell'_1,\ell'_2,\ell'_3)$ of non-well-bracketing with $\ell'_1 \leq \ell_{\mathsf{mn}}$ will follow from the following observations on $\mathsf{Label}(\mathsf{Comp}_{nwb})$.

† v_1 cannot contain any returns of $\mathcal{P}_1$ which have a matching call that occurs in u (by construction of ℓ_{mn}).

†† All calls of $\mathcal{P}_1$ in v_1 must return either in v_1 or after c' is returned. But the latter is not possible (by construction of ℓ_{sm}). Thus, all calls of $\mathcal{P}_1$ in v_1 must return in v_1.

Formally, Comp_{mod} is constructed as follows. We fix some notation. For each $0 \leq j \leq m$, let $\mathsf{Conf}_0^j = \mathsf{Conf}_0(s_j)$ and $\mathsf{Conf}_1^j = \mathsf{Conf}_1(s_j)$. Thus $s_j = (\mathsf{Conf}_0^j, \mathsf{Conf}_1^j)$.

(1) The first ℓ_{sm} transitions of Comp_{mod} are the same as Comp_{nwb}, i.e., initially $\mathsf{Comp}_{mod} = s_0 \longrightarrow \cdots \longrightarrow s_{\ell_{\mathsf{sm}}}$.

(2) Consider the sequence of transitions $s_{\ell_{\mathsf{sm}}} \overset{(\lambda_{\mathsf{sm}+1}, i_{\mathsf{sm}+1})}{\longrightarrow} \cdots \overset{(\lambda_{\mathsf{rt}+1}, i_{\mathsf{rt}+1})}{\longrightarrow} s_{\ell_{\mathsf{rt}+1}}$ in Comp. Let k be the number of transitions of $\mathcal{P}_0$ in this sequence and let $\ell_{\mathsf{sm}} \leq j_1 < \cdots < j_k \leq \ell_{\mathsf{rt}}$ be the indices such that $s_{j_n} \overset{(\lambda_{j_n+1},0)}{\longrightarrow} s_{j_n+1}$. Note that it must be the case that for each $1 \leq n < k$,

$$\mathsf{Conf}_0^{\ell_{\mathsf{sm}}} = \mathsf{Conf}_0^{j_1},\ \mathsf{Conf}_0^{j_n+1} = \mathsf{Conf}_0^{j_{n+1}} \text{ and } \mathsf{Conf}_0^{j_k+1} = \mathsf{Conf}_0^{\mathsf{rt}+1}.$$

For $1 \leq n \leq k$, let

$$s'_{\ell_{\mathsf{sm}}+n} = (\mathsf{Conf}_0^{j_n+1}, \mathsf{Conf}_1^{\ell_{\mathsf{sm}}}).$$

Observe now that, thanks to contextual locking, the set of locks held by $\mathcal{P}_1$ in $\mathsf{Conf}_1^{\ell_{\mathsf{sm}}}$ is a subset of the locks held by $\mathcal{P}_1$ in $\mathsf{Conf}_1^{\ell_{j_n}}$ for each $1 \leq n \leq k$. Thus we can extend

Comp_{mod} by applying the k transitions of $\mathcal{P}_0$ used to obtain $s_{j_n} \longrightarrow s_{j_n+1}$ in Comp_{nwb}. In other words, Comp_{mod} is now

$$s_0 \longrightarrow \cdots \longrightarrow s_{\ell_{\mathsf{sm}}} \stackrel{(\lambda_{j_1+1},0)}{\longrightarrow} s'_{\ell_{\mathsf{sm}}+1} \cdots \stackrel{(\lambda_{j_k+1},0)}{\longrightarrow} s'_{\ell_{\mathsf{sm}}+k}.$$

Note that $s'_{\ell_{\mathsf{sm}}+k} = (\mathsf{Conf}_0^{\mathsf{rt}+1}, \mathsf{Conf}_1^{\ell_{\mathsf{sm}}})$. Thus the set of locks held by $\mathcal{P}_0$ in $s'_{\ell_{\mathsf{sm}}+k}$ is exactly the set of locks held by $\mathcal{P}_0$ at $\mathsf{Conf}_0^{\ell_{\mathsf{mn}}}$.

(3) Consider the sequence of transitions $s_{\ell_{\mathsf{sm}}} \stackrel{(\lambda_{\mathsf{sm}+1}, i_{\mathsf{sm}+1})}{\longrightarrow} \cdots \stackrel{(\lambda_{\mathsf{rt}+1}, i_{\mathsf{rt}+1})}{\longrightarrow} s_{\ell_{\mathsf{rt}+1}}$ in Comp. Let t be the number of transitions of $\mathcal{P}_1$ in this sequence and let $\ell_{\mathsf{sm}} \le j_1 < \cdots < j_t \le \ell_{\mathsf{rt}}$ be the indices such that $s_{j_n} \stackrel{(\lambda_{j_n+1},1)}{\longrightarrow} s_{j_n+1}$. Note that it must be the case that for each $1 \le n < t$,

$$\mathsf{Conf}_1^{j_1} = \mathsf{Conf}_1^{\ell_{\mathsf{sm}}},\ \mathsf{Conf}_1^{j_n+1} = \mathsf{Conf}_1^{j_{n+1}} \text{ and } \mathsf{Conf}_1^{j_t+1} = \mathsf{Conf}_1^{\mathsf{rt}+1}.$$

For $1 \le n \le t$, let

$$s'_{\ell_{\mathsf{sm}}+k+n} = (\mathsf{Conf}_0^{\mathsf{rt}+1}, \mathsf{Conf}_1^{j_n+1}).$$

Observe now that, thanks to contextual locking, the set of locks held by $\mathcal{P}_0$ in $\mathsf{Conf}_0^{\ell_{\mathsf{rt}}+1}$ is exactly the set of locks held by $\mathcal{P}_0$ at $\mathsf{Conf}_0^{\ell_{\mathsf{mn}}}$ and the latter is a subset of the locks held by $\mathcal{P}_0$ in $\mathsf{Conf}_0^{\ell_{j_n}}$ for each $1 \le n \le t$. Thus we can extend Comp_{mod} by applying the t transitions of $\mathcal{P}_1$ used to obtain $s_{j_n} \longrightarrow s_{j_n+1}$ in Comp_{nwb}. In other words, Comp_{mod} is now

$$s_0 \longrightarrow \cdots \longrightarrow s'_{\ell_{\mathsf{sm}}+k} \stackrel{(\lambda_{j_1+1},1)}{\longrightarrow} s'_{\ell_{\mathsf{sm}}+k+1} \cdots \stackrel{(\lambda_{j_t+1},1)}{\longrightarrow} s'_{\ell_{\mathsf{sm}}+k+t}.$$

Observe now that the extended Comp_{mod} is a sequence of $\mathsf{rt}+1$ transitions and that the final configuration of Comp_{mod}, $s'_{\ell_{\mathsf{sm}}+k+t} = (\mathsf{Conf}_0^{\mathsf{rt}+1}, \mathsf{Conf}_1^{\mathsf{rt}+1})$ is exactly the configuration $s_{\mathsf{rt}+1}$.

(4) Thus, now we can extend Comp_{mod} as

$$s_0 \longrightarrow \cdots \longrightarrow s'_{\ell_{\mathsf{sm}}+k+t} = s_{\mathsf{rt}+1} \stackrel{(\lambda_{\mathsf{rt}+2}, i_{\mathsf{rt}+2})}{\longrightarrow} \cdots \stackrel{(\lambda_m, i_m)}{\longrightarrow} s_m.$$

Clearly, Comp_{mod} has the same length as Comp_{nwb} and simultaneously reaches p and q. There is also no witness $(\ell'_1, \ell'_2, \ell'_3)$ of non-well-bracketing in Comp_{mod} with $\ell'_1 \le \ell_{\mathsf{mn}}$. The lemma follows. □

3.2. **Deciding pairwise reachability.** We are ready to show that the problem of checking pairwise reachability is decidable.

Theorem 3.4. There is an algorithm that given a 2-threaded program $\mathcal{CP} = (\mathcal{P}_0, \mathcal{P}_1)$ communicating via Lcks such that $\mathcal{P}_0$ and $\mathcal{P}_1$ follow contextual locking, and control states p and q of $\mathcal{P}_0$ and $\mathcal{P}_1$ respectively decides if $Reach(\mathcal{P}, p, q)$ is true or not. Furthermore, if m and n are the sizes of the programs $\mathcal{P}_0$ and $\mathcal{P}_1$ and ℓ the number of elements of Lcks, then this algorithm has a running time of $2^{O(\ell)}O((mn)^3)$.

Proof. The main idea behind the algorithm is to construct a single PDS $\mathcal{P}_{comb} = (Q, \Gamma, qs, \delta)$ which simulates all the well-bracketed computations of $\mathcal{CP}$. $\mathcal{P}_{comb}$ simulates a well-bracketed computation as follows. The set of control states of $\mathcal{P}_{comb}$ is the product of control states of $\mathcal{P}_0$ and $\mathcal{P}_1$. The single stack of $\mathcal{P}_{comb}$ keeps track of the stacks of $\mathcal{P}_0$ and $\mathcal{P}_1$: it is the sequence of those calls of the well-bracketed computation which have not been returned. Furthermore, if the stack of $\mathcal{P}_{comb}$ is w then the stack of $\mathcal{P}_0$ is the projection of w onto the stack symbols of $\mathcal{P}_0$ and the stack of $\mathcal{P}_1$ is the projection of w onto the stack symbols of $\mathcal{P}_1$. Thus, the top of the stack is the most recent unreturned call and if it belongs to $\mathcal{P}_i$, well-bracketing ensures that no previous unreturned call is returned without returning this call.

Formally, $\mathcal{P}_{comb} = (Q, \Gamma, qs, \delta)$ is defined as follows. Let $\mathcal{P}_0 = (Q_0, \Gamma_0, qs_0, \delta_0)$ and $\mathcal{P}_1 = (Q_1, \Gamma_1, qs_1, \delta_1)$. Without loss of generality, assume that $Q_0 \cap Q_1 = \emptyset$ and $\Gamma_0 \cap \Gamma_1 = \emptyset$.

- The set of states Q is $(Q_0 \times 2^{\mathsf{Lcks}}) \times (Q_1 \times 2^{\mathsf{Lcks}})$.
- $\Gamma = \Gamma_0 \cup \Gamma_1$.
- $qs = ((qs_0, \emptyset), (qs_1, \emptyset))$.
- δ consists of three sets $\delta_{\mathsf{int}}, \delta_{\mathsf{cll}}$ and δ_{rtn} which simulate the internal actions, procedure calls and returns, and lock acquisitions and releases of the threads as follows. We explain here only the simulation of actions of $\mathcal{P}_0$ (the simulation of actions of $\mathcal{P}_1$ is similar).
  - *Internal actions.* If (q_0, q_0') is an internal action of $\mathcal{P}_0$, then for each $\mathsf{hld}_0, \mathsf{hld}_1 \in 2^{\mathsf{Lcks}}$ and $q_1 \in Q_1$
  $$(((q_0, \mathsf{hld}_0), (q_1, \mathsf{hld}_1)), ((q_0', \mathsf{hld}_0), (q_1, \mathsf{hld}_1))) \in \delta_{\mathsf{int}}.$$
  - *Lock acquisitions.* Lock acquisitions are also modeled by δ_{int}. If $(q_0, (q_0', l))$ is a lock acquisition action of thread $\mathcal{P}_0$, then for each $\mathsf{hld}_0, \mathsf{hld}_1 \in 2^{\mathsf{Lcks}}$ and $q_1 \in Q_1$,
  $$(((q_0, \mathsf{hld}_0), (q_1, \mathsf{hld}_1)), ((q_0', \mathsf{hld}_0 \cup \{l\}), (q_1, \mathsf{hld}_1))) \in \delta_{\mathsf{int}} \text{ if } l \notin \mathsf{hld}_0 \cup \mathsf{hld}_1.$$
  - *Lock releases.* Lock releases are also modeled by δ_{int}. If $((q_0, l), q_0')$ is a lock release action of thread $\mathcal{P}_0$, then for each $\mathsf{hld}_0, \mathsf{hld}_1 \in 2^{\mathsf{Lcks}}$ and $q_1 \in Q_1$,
  $$(((q_0, \mathsf{hld}_0), (q_1, \mathsf{hld}_1)), ((q_0', \mathsf{hld}_0 \setminus \{l\}), (q_1, \mathsf{hld}_1))) \in \delta_{\mathsf{int}} \text{ if } l \in \mathsf{hld}_0.$$
  - *Procedure Calls.* Procedure calls are modeled by δ_{cll}. If $(q_0, (q_0', a))$ is a procedure call of thread $\mathcal{P}_0$ then for each $\mathsf{hld}_0, \mathsf{hld}_1 \in 2^{\mathsf{Lcks}}$ and $q_1 \in Q_1$,
  $$(((q_0, \mathsf{hld}_0), (q_1, \mathsf{hld}_1)), (((q_0', \mathsf{hld}_0), (q_1, \mathsf{hld}_1)), a)) \in \delta_{\mathsf{cll}}.$$
  - *Procedure Returns.* Procedure returns are modeled by δ_{rtn}. If $((q_0, a), q_0')$ is a procedure return of thread $\mathcal{P}_0$ then for each $\mathsf{hld}_0, \mathsf{hld}_1 \in 2^{\mathsf{Lcks}}$ and $q_1 \in Q_1$,
  $$((((q_0, \mathsf{hld}_0), (q_1, \mathsf{hld}_1)), a), ((q_0', \mathsf{hld}_0), (q_1, \mathsf{hld}_1))) \in \delta_{\mathsf{rtn}}.$$

It is easy to see that (p, q) is reachable in $\mathcal{CP}$ by a well-bracketed computation iff there is a computation of $\mathcal{P}_{comb}$ which reaches $((p, \mathsf{hld}_0), (q, \mathsf{hld}_1))$ for some $\mathsf{hld}_0, \mathsf{hld}_1 \in 2^{\mathsf{Lcks}}$. The complexity of the results follows from the observations in [1] and the size of $\mathcal{P}_{comb}$. □

4. Reentrant locking

We now turn our attention to reentrant locking. Recall that a reentrant lock is a recursive lock which allows the thread owning a lock to acquire the lock multiple times; the owning thread must release the lock an equal number of times before another thread can acquire the lock. Thus, the set of configurations of a thread $\mathcal{P}$ using *reentrant locks* is the set $Q \times \Gamma^* \times \mathbb{N}^{\mathsf{Lcks}}$ (and not the set $Q \times \Gamma^* \times 2^{\mathsf{Lcks}}$ as in non-reentrant locks). Intuitively, the elements of a configuration (q, w, hld) of $\mathcal{P}$ now have the following meaning: q is the control state of $\mathcal{P}$, w the contents of the stack and $\mathsf{hld} : \mathsf{Lcks} \to \mathbb{N}$ is a function that tells the number of times each lock has been acquired by $\mathcal{P}$. The semantics of a PDS $\mathcal{P}$ using reentrant Lcks is formally defined as:[2]

Definition: A PDS $\mathcal{P} = (Q, \Gamma, qs, \delta)$ using reentrant Lcks gives a labeled transition relation $\longrightarrow_{\mathcal{P}} \subseteq (2^{\mathsf{Lcks}} \times (Q \times \Gamma^* \times \mathbb{N}^{\mathsf{Lcks}})) \times \mathsf{Labels} \times (2^{\mathsf{Lcks}} \times (Q \times \Gamma^* \times \mathbb{N}^{\mathsf{Lcks}}))$ where $\mathsf{Labels} = \{\mathsf{int}, \mathsf{cll}, \mathsf{rtn}\} \cup \{\mathsf{acq}(l), \mathsf{rel}(l) \mid l \in \mathsf{Lcks}\}$ and $\longrightarrow_{\mathcal{P}}$ is defined as follows.

- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{int}}_{\mathcal{P}} \mathsf{fr} : (q', w, \mathsf{hld})$ if $(q, q') \in \delta_{\mathsf{int}}$.
- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{cll}}_{\mathcal{P}} \mathsf{fr} : (q', wa, \mathsf{hld})$ if $(q, (q', a)) \in \delta_{\mathsf{cll}}$.
- $\mathsf{fr} : (q, wa, \mathsf{hld}) \xrightarrow{\mathsf{rtn}}_{\mathcal{P}} \mathsf{fr} : (q', w, \mathsf{hld})$ if $((q, a), q') \in \delta_{\mathsf{rtn}}$.
- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{acq}(l)}_{\mathcal{P}} \mathsf{fr} \setminus \{l\} : (q', w, \mathsf{hld}|_{l \mapsto \mathsf{hld}(l)+1})$ if $(q, (q', l)) \in \delta_{\mathsf{acq}}$ and either $l \in \mathsf{fr}$ or $\mathsf{hld}(l) > 0$.
- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{rel}(l)}_{\mathcal{P}} \mathsf{fr} : (q', w, \mathsf{hld}|_{l \mapsto \mathsf{hld}(l)-1})$ if $((q, l), q') \in \delta_{\mathsf{rel}}$ and $\mathsf{hld}(l) > 1$.
- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{rel}(l)}_{\mathcal{P}} \mathsf{fr} \cup \{l\} : (q', w, \mathsf{hld}|_{l \mapsto 0})$ if $((q, l), q') \in \delta_{\mathsf{rel}}$ and $\mathsf{hld}(l) = 1$.

The semantics of n-PDS $\mathcal{CP} = (\mathcal{P}_1, \dots, \mathcal{P}_n)$ communicating via reentrant Lcks is given as a transition system on $(Q_1 \times \Gamma_1^* \times \mathbb{N}^{\mathsf{Lcks}}) \times \cdots \times (Q_n \times \Gamma_n^* \times \mathbb{N}^{\mathsf{Lcks}})$ where Q_i and Γ_i are the set of states and the stack alphabet of process $\mathcal{P}_i$ respectively. Formally,

Definition: The semantics of a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \dots, \mathcal{P}_n)$ communicating via reentrant Lcks is given as a labeled transition system $T = (S, s_0, \longrightarrow)$ where

- S is said to be the set of configurations of $\mathcal{CP}$ and is the set $(Q_1 \times \Gamma_1^* \times \mathbb{N}^{\mathsf{Lcks}}) \times \cdots \times (Q_n \times \Gamma_n^* \times \mathbb{N}^{\mathsf{Lcks}})$, where Q_i is the set of states of $\mathcal{P}_i$ and Γ_i is the stack alphabet of $\mathcal{P}_i$.
- The initial configuration s_0 is $(qs_1, \epsilon, \overline{0}), \cdots, (qs_m, \epsilon, \overline{0})$ where qs_i is the initial local state of $\mathcal{P}_i$ and $\overline{0} \in \mathbb{N}^{\mathsf{Lcks}}$ is the function which takes the value 0 for each $l \in \mathsf{Lcks}$.
- The set of labels on the transitions is $\mathsf{Labels} \times \{1, \dots, n\}$ where $\mathsf{Labels} = \{\mathsf{int}, \mathsf{cll}, \mathsf{rtn}\} \cup \{\mathsf{acq}(l), \mathsf{rel}(l) \mid l \in \mathsf{Lcks}\}$. The labeled transition relation $\xrightarrow{(\lambda,i)}$ is defined as follows.

$$(q_1, w_1, \mathsf{hld}_1), \cdots, (q_n, w_n, \mathsf{hld}_n)) \xrightarrow{(\lambda,i)} ((q_1', w_1', \mathsf{hld}_1'), \cdots, (q_n', w_n', \mathsf{hld}_n'))$$

iff for all $j \neq i$, $q_j = q_j'$, $w_j = w_j'$ and $\mathsf{hld}_j = \mathsf{hld}_j'$ and

$$\mathsf{Lcks} \setminus \cup_{1 \le r \le n}\{l \mid \mathsf{hld}_r(l) > 0\} : ((q, q_i), w_i, \mathsf{hld}_i) \xrightarrow{\lambda}_{\mathcal{P}_i} \mathsf{Lcks} \setminus \cup_{1 \le r \le n}\{l \mid \mathsf{hld}_r'(l) > 0\} : ((q, q_i'), w_i', \mathsf{hld}_i').$$

Notation: Given a configuration $s = ((q_1, w_1, \mathsf{hld}_1), \cdots, (q_n, w_n, \mathsf{hld}_n))$ of a n-PDS $\mathcal{CP}$ using reentrant Lcks, we say that $\mathsf{held}_i(s) = \mathsf{hld}_i$.

4.1. **Contextual locking.** We now adapt contextual locking to reentrant locks. Informally, contextual reentrant locking means that–

- each instance of a lock acquired by a thread in a procedure call must be released before the corresponding return is executed, and
- the instances of locks held by a thread just before a procedure call is executed are not released during the execution of the procedure.

Formally,

[2]The definition of PDS $\mathcal{P}$ using reentrant locks $\mathcal{P}$ is exactly the definition of PDS $\mathcal{P}$ using locks $\mathcal{P}$ (see Section 2).

Definition: A thread i in a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ is said to follow contextual locking if whenever $s_\ell \xrightarrow{(\mathsf{cll},i)} s_{\ell+1}$ and $s_j \xrightarrow{(\mathsf{rtn},i)} s_{j+1}$ are matching procedure call and return along a computation $s_0 \xrightarrow{(\lambda_1,i)} s_1 \cdots \xrightarrow{(\lambda_m,i)} s_m$, we have that

$$\mathsf{held}_i(s_\ell) = \mathsf{held}_i(s_{j+1}) \text{ and for all } \ell \leq r \leq j.\ \mathsf{held}_i(s_\ell) \leq \mathsf{held}_i(s_r).$$

4.2. **Pairwise reachability problem.** We are ready to show that the pairwise reachability problem becomes undecidable if we have reentrant locks.

Lemma 4.1. The following problem is undecidable: Given a 2-PDS $\mathcal{CP} = (\mathcal{P}_1, \mathcal{P}_2)$ communicating only via reentrant locks Lcks s.t. $\mathcal{P}_1$ and $\mathcal{P}_2$ follow contextual locking, and control states q_1 and q_2 of threads $\mathcal{P}_1$ and $\mathcal{P}_2$, check if $Reach(\mathcal{CP}, q_1, q_2)$ is true. The problem continues to remain undecidable even when P_1 and P_2 are finite state systems, *i.e.*, P_1 and P_2 do not use their stack during any computation.

Proof. We shall show a reduction from the halting problem of a two-counter machine (on empty input) to the pairwise control state reachability problem.

A two-counter machine $\mathcal{M}$ is a tuple (Q, q_s, q_f, Δ): Q is a finite set of states, $q_s \in Q$ is the initial state, $q_f \in Q$ is the final state and Δ is a tuple $(\Delta_{state}, \{\Delta_{inc_i}, \Delta_{dec_i}, \Delta_{z_i}\}_{i=1,2})$ where $\Delta_{state} \subseteq Q \times Q$ is the set of state transitions of $\mathcal{M}$ for each $i = 1, 2$, $\Delta_{inc_i} \subseteq Q \times Q$ is the set of increment transitions of the counter i, $\Delta_{dec_i} \subseteq Q \times Q$ is the set of decrement transitions of the counter i, and $\Delta_{z_i} \subseteq Q \times Q$ is the set of zero-tests of the counter i. A configuration of $\mathcal{M}$ is a triple (q, c_1, c_2) where q is the "current control state", and $c_1 \in \mathbb{N}$ and $c_2 \in \mathbb{N}$ are the values of the counters 1 and 2 respectively. $\mathcal{M}$ is said to *halt* if there is a computation starting from $(q_s, 0, 0)$ and reaching a configuration (q_f, c_1, c_2) for some $c_1, c_2 \in \mathbb{N}$. The halting problem asks if $\mathcal{M}$ halts.

Fix a two-counter machine $\mathcal{M} = (Q, q_s, q_f, \Delta)$. We will construct a finite set Lcks, a 2-PDS $\mathcal{CP} = (\mathcal{P}_1, \mathcal{P}_2)$ communicating via reentrant Lcks and states q_1, q_2 of $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively such that $\mathcal{M}$ halts iff $Reach(\mathcal{CP}, q_1, q_2)$ is true.

The set Lcks will be the set $\{h, h', r_1, r_2, l_1, l_2, t_1, t_2\}$. The 2-PDS $\mathcal{CP}$ will not have any recursion, *i.e.*, its stack alphabet will be empty. $\mathcal{CP}$ will constructed in two steps. In the first step, we construct a 2-PDS $\mathcal{CP}^0 = (\mathcal{P}_1^0, \mathcal{P}_2^0)$ communicating via reentrant Lcks, a configuration $init = ((p, \epsilon, \mathsf{hld}_1), (q, \epsilon, \mathsf{hld}_2))$ of $\mathcal{CP}^0$ and states q_1 and q_2 of $\mathcal{P}_1^0$ and $\mathcal{P}_2^0$ respectively such that there is a computation of $\mathcal{CP}^0$ starting from $init$ and ending in a state $((q_1, \epsilon, \mathsf{hld}'_1), (q_2, \epsilon, \mathsf{hld}'_2))$ for some $\mathsf{hld}'_1, \mathsf{hld}'_2$ iff $\mathcal{M}$ halts. In the next step of the construction, we will show how to "set up" the appropriate values of hld_1 and hld_2 initially.

We construct $\mathcal{CP}^0$ as follows. The locks h and h' will not play a role in the construction of $\mathcal{CP}^0$, but will be used later to initialize hld_1 and hld_2. Intuitively, the program $\mathcal{CP}^0$ simulates a computation of $\mathcal{M}$. The simulation is mostly carried out by the thread $\mathcal{P}_1^0$. $\mathcal{P}_1^0$ maintains the control state of $\mathcal{M}$ as well as the values of the two counters using the locks l_1 and l_2: the contents of the counter i are stored as number of times the lock l_i is acquired by $\mathcal{P}_1^0$. An increment of the counter i is simulated by an acquisition of the lock l_i and a decrement of c_i is simulated by release of the lock l_i. A state transition of $\mathcal{M}$ is modeled by an internal action of $\mathcal{P}_1^0$.

A zero-test of counter i is carried by synchronization of threads $\mathcal{P}_1^0$ and $\mathcal{P}_2^0$ and involves the locks r_i and t_i. The zero test on counter i is carried out as follows. For the zero test to be carried out, $\mathcal{P}_2^0$, must be in a "ready" state q_*. Before a zero-test on counter i is carried out, thread $\mathcal{P}_1^0$ holds one instance of lock r_i and thread $\mathcal{P}_2^0$ holds one instance of lock t_i. To carry out the zero-test, $\mathcal{P}_1^0$ carries out the following sequence of lock acquisitions and releases:

$$\mathsf{acq}(t_i)\mathsf{rel}(r_i)\mathsf{acq}(l_i)\mathsf{rel}(t_i)\mathsf{acq}(r_i)\mathsf{rel}(l_i)$$

and $\mathcal{P}_2^0$ carries out the following sequence:

$$\mathsf{acq}(l_i)\mathsf{rel}(t_i)\mathsf{acq}(r_i)\mathsf{rel}(l_i)\mathsf{acq}(t_i)\mathsf{rel}(r_i).$$

Intuitively, the zero-test "commences" by $\mathcal{P}_2^0$ trying to acquire lock l_i. If the lock acquisition succeeds then $\mathcal{P}_2^0$ "learns" that counter i is indeed zero. $\mathcal{P}_2^0$ then releases the lock t_i in order to "inform" $\mathcal{P}_1^0$ that the counter i is 0 and "waits" for thread $\mathcal{P}_1^0$ to update its state. To test that counter i is indeed 0, thread $\mathcal{P}_1^0$ tries to acquire lock t_i. If it succeeds in acquiring the lock then thread $\mathcal{P}_1^0$ learns that counter i is indeed 0. It updates its state, releases lock r_i to "inform" thread $\mathcal{P}_2^0$ that it has updated the state. Thread $\mathcal{P}_2^0$ learns that $\mathcal{P}_1^0$ has updated its state by acquiring lock r_i. Now, $\mathcal{P}_2^0$ releases lock l_i so that it can "tell" $\mathcal{P}_1^0$ to release the lock t_i. Thread $\mathcal{P}_1^0$ acquires lock l_i; releases lock t_i and waits for lock r_i to be released. $\mathcal{P}_2^0$ can now acquire the lock t_i. After acquiring t_i, $\mathcal{P}_2^0$ releases lock r_i and transitions back to the state q_*. Thread $\mathcal{P}_1^0$ acquires lock r_i and releases lock l_i. Now, we have that $\mathcal{P}_1^0$ holds lock r_i and $\mathcal{P}_2^0$ holds lock t_i as before and nobody holds l_i. Now $\mathcal{P}_1^0$ can continue simulating $\mathcal{M}$.

Let $\mathsf{hld}_1(h) = \mathsf{hld}_1(r_1) = \mathsf{hld}_1(r_2) = 1$ and $\mathsf{hld}_1(h') = \mathsf{hld}_1(t_1) = \mathsf{hld}_1(t_2) = \mathsf{hld}_1(l_1) = \mathsf{hld}_1(l_2) = 0$. Let $\mathsf{hld}_2(h) = \mathsf{hld}_2(r_1) = \mathsf{hld}_2(r_2) = \mathsf{hld}'_2(l_1) = \mathsf{hld}'_2(l_2) = 0$, $\mathsf{hld}_2(h') = \mathsf{hld}_2(t_1) = \mathsf{hld}_2(t_2) = 1$. Let $init = ((q_s, \epsilon, \mathsf{hld}_1), (q_*, \epsilon, \mathsf{hld}_2))$. The construction of $\mathcal{P}^0$ ensures that there is a computation of $\mathcal{CP}^0$ starting from $init$ and ending in a state $((q_f, \epsilon, \mathsf{hld}'_1), (q_*, \epsilon, \mathsf{hld}'_2))$ for some $\mathsf{hld}'_1, \mathsf{hld}'_2$ iff $\mathcal{M}$ halts.

Now, $\mathcal{CP} = (\mathcal{P}_1, \mathcal{P}_2)$ is constructed from $\mathcal{CP}^0$ as follows. The thread $\mathcal{P}_1$ initially performs the following sequence of lock acquisitions and releases:

$$\mathsf{acq}(h')\mathsf{acq}(r_1)\mathsf{acq}(r_2)\mathsf{acq}(h)\mathsf{rel}(h');$$

makes a transition to q_s and starts behaving like thread $\mathcal{P}_1^0$. The thread $\mathcal{P}_2$ initially performs the following sequence of lock acquisitions and releases:

$$\mathsf{acq}(h)\mathsf{acq}(t_1)\mathsf{acq}(t_2)\mathsf{rel}(h)\mathsf{acq}(h');$$

makes a transition to q_* and starts behaving like thread $\mathcal{P}_2^0$.

The construction ensures that $Reach(\mathcal{CP}, q_f, q_*)$ is true iff $\mathcal{M}$ halts. The formal construction of $\mathcal{CP}$ has been carried out in the Appendix. □

5. Conclusions

The paper investigates the problem of pairwise reachability of multi-threaded programs communicating using only locks. We identified a new restriction on locking patterns, called contextual locking, which requires threads to release locks in the same calling context in which they were acquired. Contextual locking appears to be a natural restriction adhered to by many programs in practice. The main result of the paper is that the problem of pairwise reachability is decidable in polynomial time for programs in which the locking scheme is contextual. Therefore, in addition to being a natural restriction to follow, contextual locking may also be more amenable to practical analysis. We observe that these results do not follow from results in [6, 5, 3, 4] as there are programs with contextual locking that do not adhere to the nested locking principle or the bounded lock chaining principle. The proof principles underlying the decidability results are also different. Our results can also be mildly extended to handling programs that release locks a bounded stack-depth away from when they were acquired (for example, to handle procedures that call a function that acquires a lock, and calls another to release it before it returns).

There are a few open problems immediately motivated by the results on contextual locking in this paper. First, decidability of model checking with respect to fragments of LTL under the contextual locking restriction remains open. Next, while our paper establishes the decidability of pairwise reachability, it is open if the problem of checking if 3 (or more) threads simultaneously reach given local states is decidable for programs with contextual locking. Finally, from a practical standpoint, one would like to develop analysis algorithms that avoid to construct the cross-product of the two programs to check pairwise reachability.

We also considered the case of reentrant locking mechanism and established that the pairwise reachability under contextual reentrant locking is undecidable. The status of the pairwise reachability problem for the case when the locks are nested (and not necessarily contextual) is open. This appears to be a very difficult problem. Our reasons for believing this is that the problem of checking control state reachability in a PDS with *one* counter and no zero tests can be reduced to the problem of checking pairwise reachability problem in a 2-threaded program communicating via a single (and hence nested) reentrant lock. The latter is a long standing open problem.

For a more complete account for multi-threaded programs, other synchronization primitives such as thread creation and barriers should be taken into account. Combining lock-based approaches such as ours with techniques for other primitives is left to future investigation.

5.1. **Acknowledgements.** P. Madhusudan was supported in part by NSF Career Award 0747041. Mahesh Viswanathan was supported in part by NSF CNS 1016791 and NSF CCF 1016989. Rohit Chadha was at LSV, ENS Cachan & INRIA during the time research was carried out.

References

[1] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of the International Conference on Concurrency Theory*, pages 135–150, 1997.

[2] R. Chadha, P. Madhusudan, and M. Viswanathan. Reachability under contextual locking. In *Proceedings of the International Tools and Algorithms for the Construction and Analysis of Systems*, pages 437–450, 2012.

[3] V. Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-Reachability for threads communicating via locks. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 27–36, 2009.

[4] V. Kahlon. Reasoning about threads with bounded lock chains. In *Proceedings of the International Conference on Concurrency Theory*, pages 450–465, 2011.

[5] V. Kahlon and A. Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 101–110, 2006.

[6] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 505–518, 2005.

[7] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1999.

[8] S.S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[9] S.S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.
[10] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.
[11] T.W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 49–61, 1995.

Appendix A. Construction of $\mathcal{CP}$ in the proof of Theorem 4.1

We carry out the construction of $\mathcal{CP}$ in the proof of Theorem 4.1.

Recall that $\mathcal{M} = (Q, q_s, q_f, \Delta)$ is a two counter machine and Δ is the tuple $(\Delta_{state}, \{\Delta_{inc_i}, \Delta_{dec_i}, \Delta_{z_i}\}_{i=1,2})$. $\mathcal{P}$ is a 2-PDS communicating via reenterant $\mathsf{Lcks} = \{h, h', r_1, r_2, l_1, l_2, t_1, t_2\}$. Recall also that $\mathcal{CP}$ is constructed in two steps. First a 2-PDS $\mathcal{CP} = (\mathcal{P}_1^0, \mathcal{P}_2^0)$ is constructed and then extended to $\mathcal{CP}$.

Formally, the set of states of $\mathcal{P}_1^0$ is $Q^1 = Q \cup (Q \times \{1,2,3,4,5\} \times \{1,2\})$. Intuitively, the state $(q, j, i) \in Q \times \{1,2,3,4,5\} \times \{1,2\}$ means that the counter i is being tested for zero, the thread $\mathcal{P}_1^0$ has completed its j^{th} step in the test and q will be the resulting state after the zero test is completed. The transitions δ^1 of $\mathcal{P}_1^0$ are defined as follows. The set of internal actions of $\mathcal{P}_1^0$, δ_{int}, is the set δ_{state}. The set of lock acquisitions δ_{acq} is the union of sets δ_{acq_1} and δ_{acq_2} where δ_{acq_i} is the set $\{(q^\dagger, (q^\ddagger, l_i)) \mid (q^\dagger, q^\ddagger) \in \delta_{inc_i}\} \cup \{(q^\dagger, ((q^\ddagger, 1, i), t_i)) \mid (q^\dagger, q^\ddagger) \in \delta_{z_i}\} \cup \{((q,2,i), ((q,3,i), l_i)) | q \in Q\} \cup \{((q,4,i), ((q,5,i), r_i)) | q \in Q\}$. The set of lock releases δ_{rel} is the union of sets δ_{rel_1} and δ_{rel_2} where δ_{rel_i} is the set $\{((q^\dagger, l_i), (q^\ddagger)) \mid (q^\dagger, q^\ddagger) \in \delta_{dec_i}\} \cup \{(((q,1,i), r_i), (q,2,i)) \mid q \in Q\} \cup \{(((q,3,i), t_i), (q,4,i)) \mid q \in Q\} \cup \{(((q,5,i), l_i), q) \mid q \in Q\}$.

The set of states of $\mathcal{P}_2^0$ is the set $Q^2 = \{q_*\} \cup (\{0,1,2,3,4,5\} \times \{1,2\})$ where q_* is a new state. Intuitively, the state q_* means that $\mathcal{P}_2^0$ is ready to test a counter. The state (j, i) means that the counter i is being tested for zero and the thread $\mathcal{P}_2^0$ has completed its j^{th} step in the test. The set of transitions δ' of $\mathcal{P}_2^0$ consists of only lock acquisition transitions and lock release transitions. The set of lock acquisitions δ'_{acq} is the union of sets δ'_{acq_1} and δ'_{acq_2}, where $\delta'_{\mathsf{acq}_i} = \{(q_*, ((1,i), l_i)), ((2,i), ((3,i), r_i)), ((4,i), ((5,i), t_i))\}$. The set of lock releases δ'_{rel} is the union of sets δ'_{rel_1} and δ'_{rel_2}, where $\delta'_{\mathsf{rel}_i} = \{(((1,i), t_i), (2,i)), (((3,i), l_i), (4,i)), (((5,i), r_i), q_*)\}$.

Now the 2-PDS $\mathcal{CP} = (\mathcal{P}_1, \mathcal{P}_2)$ is constructed as follows. For $\mathcal{P}_1$, the set of states is $Q_1 = Q^1 \cup \{q_0, q_1, q_2, q_3, q_4\}$ where q_0, q_1, q_2, q_3, q_4 are new states. In addition to δ^1, the set of transitions of $\mathcal{P}_1$ contains the lock acquisition transitions $(q_0, (q_1, h'))$, $(q_1, (q_2, r_1))$, $(q_2, (q_3, r_2))$, $(q_3, (q_4, h))$ and the lock release transition $((q_4, h'), q_s)$. The state q_0 is the initial state of $\mathcal{P}_1$.

For $\mathcal{P}_2$, the set of states is $Q_2 = Q^2 \cup \{q'_0, q'_1, q'_2, q'_3, q'_4\}$ where $q'_0, q'_1, q'_2, q'_3, q'_4$ are new states. In addition to δ^2, the set of transitions of $\mathcal{P}_2$ contains the lock acquisition transitions $(q'_0, (q'_1, h))$, $(q'_1, (q'_2, t_1))$, $(q'_2, (q'_3, t_2))$, $(q'_4, (q'_*, h'))$ and the lock release transition $\{((q'_3, h), q'_4)\}$. The state q'_0 is the initial state of $\mathcal{P}_2$.