Reasoning about Threads with Bounded Lock Chains

Vineet Kahlon, NEC Laboratories America, USA

Abstract. The problem of model checking threads interacting purely via the standard synchronization primitives is key for many concurrent program analyses, particularly dataflow analysis. Unfortunately, it is undecidable even for the most commonly used synchronization primitive, i.e., mutex locks. Lock usage in concurrent programs can be characterized in terms of lock chains, where a sequence of mutex locks is said to be chained if the scopes of adjacent (non-nested) mutexes overlap. Although the model checking problem for fragments of Linear Temporal Logic (LTL) is known to be decidable for threads interacting via nested locks, i.e., chains of length one, these techniques don't extend to programs with non-nested locks used in crucial applications like databases. We exploit the fact that lock usage patterns in real life programs do not produce unbounded lock chains. For such a framework, we show, by using the new concept of Lock Causality Automata (LCA), that pre^*-closures of regular sets of states can be computed efficiently. Leveraging this new technique then allows us to formulate decision procedures for model checking threads communicating via bounded lock chains for fragments of LTL. Our new results narrow the decidability gap for LTL model checking of threads communicating via locks by providing a more refined characterization for it in terms of boundedness of lock chains rather than the current state-of-the-art, i.e., nestedness of locks (chains of length one).

1 Introduction

With the increasing prevalence of multi-core processors and concurrent multi-threaded software, it is highly critical that dataflow analysis for concurrent programs, similar to the ones for the sequential domain, be developed. For sequential programs, Pushdown Systems (PDSs) have emerged as a powerful, unifying framework for efficiently encoding many inter-procedural dataflow analyses [15, 5]. Given a sequential program, abstract interpretation is first used to get a finite representation of the control part of the program while recursion is modeled using a stack. Pushdown systems then provide a natural framework to model such abstractly interpreted structures. Analogous to the sequential case, inter-procedural dataflow analysis for concurrent multi-threaded programs can be formulated as a model checking problem for interacting PDSs. While for a single PDS the model checking problem is efficiently decidable for very expressive logics, it was shown in [18] that even simple properties like reachability become undecidable for systems with only two threads but where the threads synchronize using CCS-style pairwise rendezvous.

However, it has recently been demonstrated that, in practice, concurrent programs have a lot of inherent structure that if exploited leads to decidability of many important problems of practical interest. These results show that there are important fragments of temporal logics and useful models of interacting PDSs for which efficient decidability results can be obtained. Since formulating efficient procedures for model checking interacting PDSs lies at the core of scalable data flow analysis for concurrent programs, it is important that such fragments be identified for the standard synchronization primitives. Furthermore, of fundamental importance also is the need to delineate precisely the decidability boundary of the model checking problem for PDSs interacting via the standard synchronization primitives.

Nested locks are a prime example of how programming patterns can be exploited to yield decidability of the model checking problem for several important temporal logic fragments for interacting pushdown systems [13, 11]. However, even though the use of nested locks remains the most popular lock usage paradigm there are niche applications, like databases, where lock chaining is required. Chaining occurs when the scopes of two mutexes overlap. When one mutex is acquired the code enters a region where another mutex is required. After successfully locking that second mutex, the first one is no longer needed and is released. Lock chaining is an essential tool that is used for enforcing serialization, particularly in database applications. For instance, the two-phase commit protocol [14] which lies at the heart of serialization in databases uses lock chains of length 2. Other classic examples where non-nested locks occur frequently are programs that use both mutexes and (locks associated with) Wait/Notify primitives (condition variables). It is worth pointing out that the lock usage pattern of bounded lock chains covers almost all cases of practical interest encountered in real-life programs.

We consider the model checking problem for pushdown systems synchronizing via bounded lock chains for LTL properties. Decidability of a sub-logic of LTL hinges on whether it is expressive enough to encode, as a model checking problem, the disjointness of the context-free languages accepted by the PDSs in the given multi-PDS system - an undecidable problem. This, in turn, depends on the temporal operators allowed by the sub-logic thereby providing a natural way to characterize LTL-fragments for which the model checking problem is decidable. We use $L(Op_1, ..., Op_k)$, where $Op_i \in \{\mathsf{X}, \mathsf{F}, \mathsf{U}, \mathsf{G}, \overset{\infty}{\mathsf{F}}\}$, to denote the fragment comprised of formulae of the form $\mathsf{E}f$, where f is an LTL formula in positive normal form (PNF), viz., only atomic propositions are negated, built using the operators $Op_1, ..., Op_k$ and the Boolean connectives $\vee$ and $\wedge$. Here X "next-time", F "sometimes", U, "until", G "always", and $\overset{\infty}{\mathsf{F}}$ "infinitely-often" denote the standard temporal operators and E is the "existential path quantifier". Obviously, $L(\mathsf{X}, \mathsf{U}, \mathsf{G})$ is the full-blown LTL.

It has recently been shown that pairwise reachability is decidable for threads interacting via bounded lock chains [10]. In this paper, we extend the envelope of decidability for concurrent programs with bounded lock chains to richer logics. Specifically, we show that the model checking problem for threads interacting via bounded lock chains is decidable not just for reachability but also the fragment of LTL allowing the temporal operators X, F, $\overset{\infty}{\mathsf{F}}$ and the boolean connectives $\wedge$ and $\vee$, denoted by $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$. It is important to note that while pairwise reachability is sufficient for reasoning about simple properties like data race freedom, for more complex properties one needs to reason about richer formulae. For instance, detecting atomicity violations requires reasoning about the fragment of LTL allowing the operators **F**, $\wedge$ and $\vee$ (see [14]).

Moreover, we also delineate precisely the decidability/undecidability boundary for the problem of model checking dual-PDS systems synchronizing via bounded lock chains. Specifically, we show the following.

1. the model checking problem is undecidable for $L(\mathsf{U})$ and $L(\mathsf{G})$. This implies that in order to get decidability for dual-PDS systems interacting via bounded lock chains, we have to restrict ourselves to the sub-logic $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$. Since systems comprised of PDSs interacting via bounded lock chains are more expressive than those interacting

via nested locks (chains of length one) these results follow immediately from the undecidability results for PDSs interacting via nested locks [11].

2. for the fragment $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ of LTL we show that the model checking problem is decidable.

This settles the model checking problem for threads interacting via bounded lock chains for LTL. The prior state-of-the-art characterization of decidability vs. undecidability for threads interacting via locks was in terms of nestedness vs. non-nestedness of locks. We show that decidability can be re-characterized in terms of boundedness vs. unboundedness of lock chains. Since nested locks form chains of length one, our results are strictly more powerful than the existing ones. Thus, our new results narrow the decidability gap by providing a more refined characterization for the decidability of LTL for threads interacting via locks.

A key contribution of the paper is the new notion of a *Lock Causality Automaton (LCA)* that is used to represent sets of states of the given concurrent program so as to allow efficient temporal reasoning about programs with bounded lock chains. To understand the motivation behind an LCA, we recall that when model checking a single PDS, we exploit the fact that the set of configurations satisfying any given LTL formula is regular and can therefore be captured via a finite automaton or, in the terminology of [5], a multi-automaton. For a concurrent program with two PDSs T_1 and T_2, however, we need to reason about pairs of regular sets of configuration - one for each thread. An LCA is a pair of automata (M_1, M_2), where M_i accepts a regular set of configurations of T_i. The usefulness of an LCA stems from the fact that not only does it allow us to reason about $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ properties for concurrent programs with bounded lock chains, but that it allows us to do so in a compositional manner. Compositional reasoning allows us to reduce reasoning about the concurrent program at hand to each of its individual threads. This is crucial in ameliorating the state explosion problem. The main challenge in reducing model checking of a concurrent program to its individual threads lies in tracking relevant information about threads locally that enables us to reason globally about the concurrent program. For an LCA this is accomplished by tracking regular lock access patterns in individual threads.

To sum up, the key contributions of the paper are

1. the new notion of an LCA that allows us to reason about concurrent programs with bounded lock chains in a compositional manner.

2. a model checking procedure for the fragment $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ of LTL that allows us to narrow the decidability gap for model checking LTL properties for threads communicating via locks.

3. delineation of the decidability boundary for the LTL model checking problem for threads synchronizing via bounded lock chains.

2 System Model

We consider concurrent programs comprised of threads modeled as Pushdown Systems (PDSs) [5] that interact with each other using synchronization primitives. PDSs are a natural model for abstractly interpreted programs used in key applications like dataflow analysis [15]. A PDS has a finite control part corresponding to the valuation of the variables of a thread and a stack which provides a means to model recursion.

Formally, a PDS is a five-tuple $P = (Q, Act, \Gamma, c_0, \Delta)$, where Q is a finite set of *control locations*, *Act* is a finite set of *actions*, Γ is a finite *stack alphabet*, and $\Delta \subseteq$

$(Q \times \Gamma) \times Act \times (Q \times \Gamma^*)$ is a finite set of *transitions*. If $((p, \gamma), a, (p', w)) \in \Delta$ then we write $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', w \rangle$. A *configuration* of P is a pair $\langle p, w \rangle$, where $p \in Q$ denotes the control location and $w \in \Gamma^*$ the *stack content*. We call c_0 the *initial configuration* of P. The set of all configurations of P is denoted by $\mathcal{C}$. For each action a, we define a relation $\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{C}$ as follows: if $\langle q, \gamma \rangle \xhookrightarrow{a} \langle q', w \rangle$, then $\langle q, \gamma v \rangle \xrightarrow{a} \langle q', wv \rangle$ for every $v \in \Gamma^*$ – in which case we say that $\langle q', wv \rangle$ results from $\langle q', \gamma v \rangle$ by firing the transition $\langle q, \gamma \rangle \xhookrightarrow{a} \langle q', w \rangle$ of P.

We model a concurrent program with n threads and m locks[1] $\mathsf{l}_1, ..., \mathsf{l}_m$ as a tuple of the form $\mathcal{CP} = (T_1, ..., T_n, L_1, ..., L_m)$, where T_1,...,T_n are pushdown systems (representing threads) with the same set *Act* of non-*acquire* and non-*release* actions, and for each i, $L_i \subseteq \{\bot, 1, ..., n\}$ is the possible set of values that lock l_i can be assigned. A global configuration of $\mathcal{CP}$ is a tuple $c = (t_1, ..., t_n, l_1, ..., l_m)$ where $t_1, ..., t_n$ are, respectively, the configurations of threads $T_1, ..., T_n$ and $l_1, ..., l_m$ the values of the locks. If no thread holds the lock l_i in configuration c, then $l_i = \bot$, else l_i is the index of the thread currently holding l_i. The initial global configuration of $\mathcal{CP}$ is $(c_1, ..., c_n, \bot, ..., \bot)$, where c_i is the initial configuration of thread T_i. Thus all locks are *free* to start with. We extend the relation $\xrightarrow{a}$ to pairs of global configurations of $\mathcal{CP}$ in the standard way by encoding the interleaved parallel composition of $T_1, ..., T_n$ (see the full paper [1] for the precise definition).

Correctness Properties. We consider correctness properties expressed as double-indexed Linear Temporal Logic (LTL) formulae. Here atomic propositions are interpreted over pairs of control states of different PDSs in the given multi-PDS system.

Conventionally, $\mathcal{CP} \models f$ for a given LTL formula f if and only if f is satisfied along all paths starting at the initial state of $\mathcal{CP}$. Using path quantifiers, we may write this as $\mathcal{CP} \models \mathsf{A}f$. Equivalently, we can model check for the dual property $\neg\mathsf{A}f = \mathsf{E}\neg f = \mathsf{E}g$. Furthermore, we can assume that g is in *positive normal form (PNF)*, viz., the negations are pushed inwards as far as possible using DeMorgan's Laws: $(\neg(p \vee q)) = \neg p \wedge \neg q$, $\neg(p \vee q) = \neg p \wedge \neg q$, $\neg Fp \equiv Gq$, $\neg(pUq) \equiv G\neg q \vee \neg qU(\neg p \wedge \neg q)$.

For Dual-PDS systems, it turns out that the model checking problem is not decidable for the full-blown double-indexed LTL but only for certain fragments. Decidability hinges on the set of temporal operators that are allowed in the given property which, in turn, provides a natural way to characterize such fragments. We use $L(Op_1, ..., Op_k)$, where $Op_i \in \{\mathsf{X}, \mathsf{F}, \mathsf{U}, \mathsf{G}, \overset{\infty}{\mathsf{F}}\}$, to denote the fragment of double-indexed LTL comprised of formulae in positive normal form (where only atomic propositions are negated) built using the operators $Op_1, ..., Op_k$ and the Boolean connectives $\vee$ and $\wedge$. Here X "next-time", F "sometimes", U, "until", G "always", and $\overset{\infty}{\mathsf{F}}$ "infinitely-often" denote the standard temporal operators (see [8]). Obviously, $L(\mathsf{X}, \mathsf{U}, \mathsf{G})$ is the full-blown double-indexed LTL.

Outline of Paper. In this paper, we show decidability of the model checking problem for the fragment $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ of LTL for concurrent programs with bounded lock chains. Given an $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ formula f, we build automata accepting global states of the given concurrent program satisfying f. Towards that end, we first show how to construct automata for the basic temporal operators F, $\overset{\infty}{\mathsf{F}}$ and X, and the boolean connectives $\wedge$

[1] We do not allow recursive/re-entrant locks

and $\vee$. Then to compute an automaton for the given property f, we start by building for each atomic proposition *prop* of f, an automata accepting the set of states of the given concurrent program satisfying *prop*. Leverage the constructions for the basic temporal operators and boolean connectives we then recursively build the automaton accepting the set of states satisfying f via an inside out traversal of f. Then if the initial state of the given concurrent program is accepted by the resulting automaton, the program satisfies f. The above approach, which is standard for LTL model checking of finite state and pushdown systems, exploits the fact that for model checking it suffices to reason about regular sets of configurations of these systems. These sets can be captured using regular automata which then reduces model checking to computing regular automata for each of the temporal operators and boolean connectives. However, for concurrent programs the sets of states that we need to reason about for model checking are not regular and cannot therefore be captured via regular automata. We therefore propose the new notion of a *Lock Causality Automaton (LCA)* that is well suited for reasoning about concurrent programs with bounded lock chains. A key contribution of the paper lies is showing how to construct LCAs for the basic temporal operators and the boolean connectives.

The constructions of LCAs for the various temporal operators depend on computing an LCA accepting the pre^*-closure of the set of states accepted by a given LCA. This in turn, hinges on deciding pairwise CFL-reachability (see sec. 3) of a pair $\mathbf{c}_1$ and $\mathbf{c}_2$ of configurations from another pair $\mathbf{d}_1$ and $\mathbf{d}_2$ of configurations of T_1 and T_2, respectively. Our decision procedure for pairwise CFL-reachability relies on the notion of a *Bidirectional Lock Causality Graph* introduced in the next section. This leads naturally to the notion of an LCA defined in sec. 4. Finally the constructions of LCAs for the basic temporal operators are given in sec. 5 which leads to the model checking procedure for $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ formulated in sec. 5.1.

3 Pairwise CFL-Reachability

A key step in the computation of pre^*-closure of LCAs is deciding *Pairwise CFL-Reachability*.

Pairwise CFL-Reachability. *Let $\mathcal{CP}$ be a concurrent program comprised of threads T_1 and T_2. Given pairs (c_1, c_2) and (d_1, d_2), with c_i and d_i being control locations of T_i, does there exist a path of $\mathcal{CP}$ leading from a global state with T_i in c_i to one with T_i in d_i in the presence of recursion and scheduling constraints imposed by locks.*

It is known that pairwise CFL-reachability is undecidable for two threads interacting purely via locks but decidable if the locks are nested [12] and, more generally, for programs with bounded length lock chains [10], where a lock chain is defined as below.

Lock Chains. *Given a computation x of a concurrent program, a lock chain of thread T is a sequence of lock acquisition statements $acq_1, ..., acq_n$ fired by T along x in the order listed such that for each i, the matching release of acq_i is fired after acq_{i+1} and before acq_{i+2} along x.*

However, the decision procedures for programs with bounded lock chains [10] only apply to the case wherein c_1 and c_2 are *lock-free*, i.e., no lock is held by T_i at c_i. In order to decide the pairwise CFL-reachability problem for the general case, we propose the notion of a *Bi-directional Lock Causality Graph* which is a generalization of the (unidirectional) lock causality graph presented in [10].

Algorithm 1 Bi-Directional Lock Causality Graph

```
Input: Local paths $x^1$ and $x^2$ of $T_1$ and $T_2$ leading from $c_1$ and $c_2$ to $d_1$ and $d_2$, respectively
for each lock $l$ held at location $d_i$ do
    If $c$ and $c'$ are the last statements to acquire and release $l$ occurring along $x^i$ and $x^{i'}$,
    respectively, Add edge $c' \rightsquigarrow c$ to $G_{(x^1,x^2)}$.
end for
for each lock $l$ held at location $c_i$ do
    If $c$ and $c'$ are the first statements to release and acquire $l$ occurring along $x^i$ and $x^{i'}$,
    respectively, add edge $c \rightsquigarrow c'$ to $G_{(x^1,x^2)}$.
end for
repeat
    for each lock $l$ and each edge $d_{i'} \rightsquigarrow d_i$ of $G_{(x^1,x^2)}$ do
        Let $a_{i'}$ be the last statement to acquire $l$ before $d_{i'}$ along $x^{i'}$ and $r_{i'}$ the matching
        release for $a_{i'}$ and let $r_i$ be the first statement to release $l$ after $d_i$ along $x^i$ and $a_i$ the
        matching acquire for $r_i$
        if $l$ is held at either $d_i$ or $d_{i'}$ then
            if there does not exist an edge $b_{i'} \rightsquigarrow b_i$ such that $r_{i'}$ lies before $b_{i'}$ along $x^{i'}$ and $a_i$
            lies after $b_i$ along $x^i$ then
                add edge $r_{i'} \rightsquigarrow a_i$ to $G_{(x^1,x^2)}$
            end if
        end if
    end for
until no new statements can be added to $G_{(x^1,x^2)}$
for $i \in [1..2]$ do
    Add edges among locations of $x^i$ in $G_{(x^1,x^2)}$ to preserve their relative ordering along $x^i$
end for
```

Bidirectional Lock Causality Graph. Consider the example concurrent program comprised of threads T_1 and T_2 shown in fig. 1. Suppose that we are interested in deciding whether *a7* and *b7* are pairwise reachable starting from the locations *a1* and *b1* of T_1 and T_2, respectively. Note that the set of locks held at *a1* and *b1* are $\{l_1\}$ and $\{l_3, l_5\}$, respectively. For *a7* and *b7* to be pairwise reachable there must exist local paths x^1 and x^2 of T_1 and T_2 leading to *a7* and *b7*, respectively, along which locks can be acquired and released in a consistent fashion. We start by constructing a *bi-directional lock causality graph* $G_{(x^1,x^2)}$ that captures the constraints imposed by locks on the order in which statements along x^1 and x^2 need to be executed in order for T_1 and T_2 to simultaneously reach *a7* and *b7*. The nodes of this graph are (the relevant) locking/unlocking statements fired along x^1 and x^2. For statements c_1 and c_2 of $G_{(x^1,x^2)}$, there exists an edge from c_1 to c_2, denoted by $c_1 \rightsquigarrow c_2$, if c_1 must be executed before c_2 in order for T_1 and T_2 to simultaneously reach *a7* and *b7*.

$G_{(x^1,x^2)}$ has two types of edges (i) *Seed* edges and (ii) *Induced* edges.

Seed Edges: Seed edges, which are shown as bold edges in fig. 1(c), can be further classified as (a) *Backward* and (b) *Forward* seed edges.

(a) **Forward Seed Edges:** Consider lock l_1 held at *b7*. Note that once T_2 acquires l_1 at location *b4*, it is not released along the path from *b4* to *b7*. Since we are interested in the pairwise CFL-reachability of $a7$ and $b7$, T_2 cannot progress beyond location *b7* and therefore cannot release l_1. Thus we have that once T_2 acquires l_1 at *b4*, T_1

void T_1(){
a1: lock(l_4);
a2: lock(l_5);
a3: unlock(l_4);
a4: unlock(l_5);
a5: unlock(l_1);
a6: lock(l_3);
a7: $Race_0$;
}

void T_2(void){
b1: lock(l_4);
b2: unlock(l_5);
b3: unlock(l_3);
b4: lock(l_1);
b5: lock(l_2);
b6: unlock(l_4);
b7: $Race_1$;
}

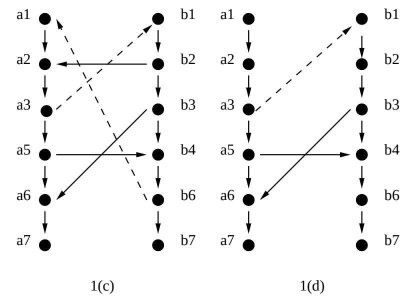

Fig. 1. An Example Program and its Bi-directional Lock Causality Graph

cannot acquire it thereafter. If T_1 and T_2 are to simultaneously reach *a7* and *b7*, the last transition of T_1 that releases l_1 before reaching *a7*, i.e., *a5*, must be executed before *b4*. Thus $a5 \rightsquigarrow b4$.

(b) **Backward Seed Edges:** Consider lock l_5 held at *b1*. In order for T_1 to acquire l_5 at *a2*, l_5 must first be released by T_2. Thus the first statement of T_1 acquiring l_5 starting at *a1*, i.e., *a2*, must be executed after *b2*. Thus $b2 \rightsquigarrow a2$.

The interaction of locks and seed causality edges can be used to deduce further causality constraints that are captured as *induced* edges (shown as dashed edges in the BLCG in fig. 1(c)). These induced edges are key in guaranteeing both soundness and completeness of our procedure.

Induced Edges: Consider the constraint *b2* $\rightsquigarrow$ *a2*. At location *b2*, lock l_4 is held which was acquired at *b1*. Also, once l_4 is acquired at *b1* it is not released till after T_2 exits *b6*. Thus since l_4 has been acquired by T_2 before reaching *b2* it must be released before *a1* (and hence *a2*) can be executed. Thus, *b6* $\rightsquigarrow$ *a1*.

Computing the Bidirectional Lock Causality Graph. Given finite local paths x^1 and x^2 of threads T_1 and T_2 starting at control locations c_1 and c_2 and leading to control locations d_1 and d_2, respectively, the procedure (see alg. 1) to compute $G_{(x^1,x^2)}$ adds the causality constraints one-by-one (forward seed edges via steps 2-6, backward seed edges via steps 7-11 and induced edges via steps 12-24) till we reach a fixpoint. Throughout the description of alg. 1, for $i \in [1..2]$, we use i' to denote an integer in $[1..2]$ other than i. Note that condition 18 in alg. 1 ensures that we do not add edges representing causality constraints that can be deduced from existing edges.

Necessary and Sufficient Condition for CFL-Reachability Let x^1 and x^2 be local computations of T_1 and T_2 leading to c_1 and c_2. Since each causality constraint in $G_{(x^1,x^2)}$ is a *happens-before* constraint, we see that in order for c_1 and c_2 to be pairwise reachable $G_{(x^1,x^2)}$ has to be acyclic. In fact, it turns out that acyclicity is also a sufficient condition (see [1] for the proof).

Theorem 1. (Acyclicity). *Locations d_1 and d_2 are pairwise reachable from locations c_1 and c_2, respectively, if there exist local paths x^1 and x^2 of T_1 and T_2, respectively, leading from c_1 and c_2 to d_1 and d_2, respectively, such that (1) $L_{T_1}(c_1) \cap L_{T_2}(c_2) = \emptyset$ (disjointness of* **backward locksets***), (2) $L_{T_1}(d_1) \cap L_{T_2}(d_2) = \emptyset$ (disjointness of* **forward locksets***), and (3) $G_{(x^1,x^2)}$ is acyclic. Here $L_T(e)$ denotes the set of locks held by thread T at location e.*

Synergy Between Backward and Forward Lock Causality Edges. Note that in order to deduce that *a7* and *b7* are not pairwise reachable it is important to consider causality edges induced by both backward and forward seed edges ignoring either of which may cause us to incorrectly deduce that $a7$ and $b7$ are reachable. In the above example if we ignore the backward seed edges then we will construct the unidirectional lock causality graph $L_{(x^1,x^2)}$ shown in fig. 1(d) which is acyclic. Thus the lock causality graph construction of [10] is inadequate in reasoning about bi-directional pairwise reachability.

Bounding the Size of the Lock Causality Graph. Under the assumption of bounded lock chains, we show that the size of the bidirectional lock causality graph is bounded. From alg. 1 it follows that each causality edge is induced either by an existing induced causality edge or a backward or forward seed edge. Thus for each induced causality edge e, there exists a sequence $e_0, ..., e_n$ of causality edges such that e_0 is a seed edge and for each $i \geq 1$, e_i is induced by e_{i-1}. Such a sequence is referred to as a lock causality sequence. Under the assumption of bounded lock chains it was shown in [10] that the length of any lock causality sequence is bounded. Note that the number of seed edges is at most $4|L|$, where $|L|$ is the number of locks in the given concurrent program. Since the number of seed edges is bounded, and since the length of each lock causality sequence is bounded, the number of induced edges in each bi-directional lock causality graph is also bounded leading to the following result.

Theorem 2. (Bounded Lock Causality Graph). *If the length of each lock chain generated by local paths x^1 and x^2 of threads T_1 and T_2, respectively, is bounded then the size (number of vertices) of $G_{(x^1,x^2)}$, is also bounded.*

4 Lock Causality Automata

When model checking a single PDS, we exploit the fact that the set of configurations satisfying a given LTL formula is regular and can therefore be captured via a finite automaton also called a multi-automaton [5]. For a concurrent program with two PDSs, however, we need to reason about pairs of regular sets of configurations. Thus instead of performing pre^*-closures over multi-automata, we need to perform pre^*-closures over automata pairs.

Suppose that we are given a pair (R_1, R_2) of sets, where R_i is a regular set of configurations of thread T_i. The set S_i of configurations of T_i that are (locally) backward reachable from R_i forms a regular set [5]. However, given a pair of configurations (a_1, a_2), where $a_i \in S_i$, even though a_i is backward reachable from some $b_i \in R_i$ in T_i, there is no guarantee that a_1 and a_2 are pairwise backward reachable from b_1 and b_2 in the concurrent program $\mathcal{CP}$. That happens only if there exist local paths x^1 and x^2 of threads T_1 and T_2, respectively, from a_i to b_i such that $G_{(x^1,x^2)}$ is acyclic. Thus in computing the pre^*-closure S_i of R_i in thread T_i, we need to track relevant lock access patterns that allow us to deduce acyclicity of the lock causality graph $G_{(x^1,x^2)}$.

In order to capture the set of global states of $\mathcal{CP}$ that are backward reachable from (R_1, R_2), we introduce the notion of a *Lock Causality Automaton (LCA)*. An LCA is a pair of automata $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_i$ accepts the regular set of configurations of T_i that are backward reachable from R_i. For $\mathcal{L}$ to accept precisely the set of global states (a_1, a_2) that are pairwise backward reachable from $(b_1, b_2) \in (R_1, R_2)$, we encode the existence of a pair of local paths x^i from a_i to b_i generating an acyclic lock causality graph in the acceptance condition of $\mathcal{L}$. For concurrent programs with nested locks, this was accomplished by tracking forward and backward acquisition histories and incorporating a consistency check for these acquisition histories (a necessary and sufficient condition for pairwise reachability) in the acceptance condition of $\mathcal{L}$ [12]. A key feature of acquisition histories that we exploited was that they are defined locally for each thread and could therefore be tracked during the (local) computation of the pre^*-closure of R_i. In contrast, the lock causality graph depends on lock access patterns of both threads. Thus we need to locally track relevant information about lock accesses in a manner that allows us to re-construct the (global) lock causality graph. Towards that end, the following result is key. Let L be the set of locks in the given concurrent program and let $\Sigma_L = \cup_{l \in L}\{a_l, r_l\}$, where a_l and r_l denote labels of transitions acquiring and releasing lock l, respectively, in the given program.

Theorem 3. (Regular Decomposition) *Let G be a directed bipartite graph over nodes labeled with lock acquire/release labels from the set Σ_L. Then there exist regular automata $G_{11}, ..., G_{1n}, G_{21}, ..., G_{2n}$ over Σ_L such that the set $\{(x^1, x^2)|x^1 \in \Sigma_L^*, x^2 \in \Sigma_L^*, G_{(x^1,x^2)} = G\}$ can be represented as $\bigcup_i L(G_{i1}) \times L(G_{i2})$, where $L(G_{ij})$ is the language accepted by G_{ij}.*

To prove this result, we introduce the notion of a lock schedule. The motivation behind the definition a lock schedule is that not all locking events, i.e., lock/unlock statements, along a local computation x of a thread T need occur in a lock causality graph involving x. A lock schedule u is intended to capture only those locking events $u : u_0, ..., u_m$ that occur in a lock causality graph. The remaining locking events, i.e., those occurring between u_i and u_{i+1} along x are specified in terms of its complement set F_i, i.e., symbols from Σ_L that are forbidden to occur between u_i and u_{i+1}. We require that if u_i is the symbol a_l, representing the acquisition of lock l and if it matching release r_l is executed along x then that matching release also occurs along the sequence u, i.e., $u_j = r_l$ for some $j > i$. Also, since l cannot be acquired twice, in order to preserve locking semantics the letters a_l and r_l cannot occur between u_i and u_j along x. This is captured by including a_l and r_l in each of the forbidden sets $F_i, ..., F_{j-1}$.

Definition (Lock Schedule). *A lock schedule is a sequence $u_0, ..., u_m \in \Sigma_L^*$ having for each i, a set $F_i \subseteq \Sigma_L$ associated with u_i such that if $u_i = a_l$ and u_j it matching release, then for each k such that $i \le k < j$ we have $r_l, a_l \in F_k$. We denote such a lock schedule by $u_0F_0u_1...u_mF_m$.*

We say that a sequence $x \in \Sigma_L^*$ satisfies a given lock schedule $sch = u_0F_0u_1...u_mF_m$, denoted by $sch \models x$, if $x \in u_0(\Sigma_L \setminus F_0)^*u_1...u_m(\Sigma_L \setminus F_m)^*$. The following is an easy consequence of the above definition.

Lemma 4. *The set of sequences in Σ_L^* satisfying a given lock schedule is regular.*

The proof of thm. 3 then follows easily from the following (see [1] for all the proofs).

Theorem 5. *Given a lock causality graph G, we can construct a finite set SCH_G of pairs of lock schedules such that the set of pairs of sequences in Σ_L^* generating G is precisely the set of pairs of sequences in Σ_L^* satisfying at least one schedule pair in SCH_G, i.e., $\{(x^1, x^2)|x^1, x^2 \in \Sigma_L^*, G_{(x^1,x^2)} = G\} = \{(y^1, y^2)|\, y^1, y^2 \in \Sigma_L^*$, for some $(\mathsf{sch}_1, \mathsf{sch}_2) \in \mathsf{SCH}_G, \mathsf{sch}_1 \models y^1$ and $\mathsf{sch}_2 \models y^2\}$.*

Lock Causality Automata. We now formally define the notion of a Lock Causality Automata. Since for programs with bounded lock chains the number of lock causality graphs is bounded (thm. 2), so is the number of acyclic lock causality graphs. With each acyclic lock causality graph G we can, using thm. 5, associate a finite set $\mathsf{ACYC_G}$ of automata pairs that accept all pairs of sequences in $\Sigma_L^* \times \Sigma_L^*$ generating G. By taking the union over all acyclic lock causality graphs G, we construct the set of all automata pairs that accept all pairs of sequences in $\Sigma_L^* \times \Sigma_L^*$ generating acyclic lock causality graphs. We denote all such pairs by ACYC. Let $(\mathsf{G}_{11}, \mathsf{G}_{21}), ..., (\mathsf{G}_{1n}, \mathsf{G}_{2n})$ be an enumeration of all automata pairs of ACYC.

We recall that a key motivation in defining LCAs is to capture the pre^*-closure, i.e., the set of pairs of configurations that are pairwise backward reachable from a pair of configurations in (R_1, R_2), where R_i is a regular set of configurations of T_i. We therefore define an LCA to be a pair of the form $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_i$ is a multi-automaton accepting the set of configurations of T_i that are backward reachable from configurations in R_i. Note that if (a_1, a_2) is pairwise backward reachable from $(b_1, b_2) \in (R_1, R_2)$ then a_i is accepted by $\mathcal{L}_i$. However, due to constraints imposed by locks not all pairs of the form (c_1, c_2), where c_i is accepted by $\mathcal{L}_i$, are pairwise backward reachable from (b_1, b_2). In order for $\mathcal{L}$ to accept precisely the set of global configurations (a_1, a_2) that are pairwise backward reachable from (b_1, b_2), we encode the existence of local paths x^i from a_i to b_i generating an acyclic lock causality graph in the acceptance condition of $\mathcal{L}$. Towards that end, when performing the backward pre^*-closure in computing $\mathcal{L}_i$ we track not simply the set of configurations c of T_i that are backward reachable from R_i but also the lock schedules encountered in reaching c.

In deciding whether configurations c_1 and c_2 are pairwise backward reachable from b_1 and b_2, where $(b_1, b_2) \in (R_1, R_2)$, we only need to check whether for each $i \in [1..2]$, there exist lock schedules sch_i from c_i to b_i such that $G_{(\mathsf{sch}_1, \mathsf{sch}_2)}$ is acyclic, i.e., for some j, $(\mathsf{sch}_1, \mathsf{sch}_2) \in L(\mathsf{G}_{1j}) \times L(\mathsf{G}_{2j})$. Since, in performing backward pre^*-closure for each thread T_i, we track local computation paths and hence lock schedules in the reverse manner, we have to consider the reverse of the regular languages accepted by G_{ij}. Motivated by this, for each i, j, we let G_{ij}^r be a regular automata accepting the language resulting by reversing each word in the language accepted by G_{ij}. Then c_1 and c_2 are pairwise backward reachable from b_1 and b_2 if there exists for each i, a (reverse) lock schedule rsch_i along a path y^i from b_i to c_i, such that for some j, rsch_1 is accepted by G_{1j}^r and rsch_2 is accepted by G_{2j}^r. Thus when computing the backward pre^*-closure in thread T_i, instead of tracking the sequence z^i of lock/unlock statements encountered thus far, it suffices to track for each j, the set of possible current local states of the regular automata G_{ij}^r reached by traversing z^i starting at its initial state. Indeed, for each i, j, let $\mathsf{G}_{ij}^r = (Q_{ij}, \delta_{ij}, \mathsf{in}_{ij}, F_{ij})$, where Q_{ij} is the set of states of G_{ij}^r, δ_{ij} its transition relation, in_{ij} its initial state and F_{ij} its set of final states. Let $S_{ij}(\mathsf{rsch}_i) = \delta_{ij}(\mathsf{in}_{ij}, \mathsf{rsch}_i)$. Then the above condition can be re-written as follows: c_1 and c_2 are pairwise backward reachable from b_1 and b_2 if there exists for each i, a lock schedule rsch_i along a path y^i from b_i to c_i, such that for some j, $S_{1j}(\mathsf{rsch}_1) \cap F_{1j} \neq \emptyset$ and $S_{2j}(\mathsf{rsch}_2) \cap F_{2j} \neq \emptyset$.

Thus in performing pre^*-closure in thread T_i, we augment the local configurations of T_i to track for each i, j, the current set of states of G_{ij} induced by the lock/unlock sequence seen so far. Hence an augmented configuration of T_i now has the form $\langle (c, FLS, BLS, GS_{i1}, ..., GS_{in}), u \rangle$, where FLS and BLS are the forward and backward locksets (see thm. 1) at the start and end points and GS_{ij} is the set of states of G_{ij}^r induced by the lock/unlock sequences seen so far in reaching configuration $\langle c, u \rangle$. To start with GS_{ij} is set to $\{\mathsf{in}_{ij}\}$, the initial state of G_{ij}^r.

Lock Augmented Multi-Automata. Formally, a lock augmented multi-automaton can be defined as follow: Let T_i be the pushdown system $(Q_i, Act_i, \Gamma_i, c_{i0}, \Delta_i)$. A *Lock*

Augmented T_i-Multi-Automaton is a tuple $\mathcal{M}_i = (\Gamma_i, P_i, \delta_i, I_i, F_i)$, where P_i is a finite set of states, $\delta_i \subseteq P_i \times \Gamma_i \times P_i$ is a set of transitions, $I_i = \{(c, FLS, BLS, GS_{i1}, ..., GS_{in}) \mid c \in Q_i, BLS, FLS \subseteq L, GS_{ij} \subseteq Q_{ij}\} \subseteq P_i$ is a set of initial states and $F_i \subseteq P_i$ is a set of final states. $\mathcal{M}_i$ accepts an augmented configuration $\langle (c, FLS, BLS, GS_{i1}, ..., GS_{in})), u \rangle$ if starting at the initial state $(c, FLS, BLS, GS_{i1}, ..., GS_{in}))$ there is a path in $\mathcal{M}_i$ labeled with u and leading to a final state of $\mathcal{M}_i$. Note that the only difference between a lock augmented multi-automation and the standard multi-automaton as defined in [5] is that the control state is augmented with the lockset information BLS and FLS, and the subsets GS_{ij} used to track lock schedules.

A lock causality automaton is then defined as follows:

Definition (Lock Causality Automaton) *Given threads $T_1 = (Q_1, Act_1, \Gamma_1, \mathbf{c}_1, \Delta_1)$ and $T_2 = (Q_2, Act_2, \Gamma_2, \mathbf{c}_2, \Delta_2)$, a lock causality automaton is a pair $(\mathcal{L}_1, \mathcal{L}_2)$ where $\mathcal{L}_i$ is a lock augmented T_i-multi-automaton.*

The acyclicity check (thm. 1) for pairwise reachability is encoded in the acceptance criterion of an LCA.

Definition (LCA-Acceptance). *We say that LCA $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2)$ accepts the pair $(\mathbf{c}_1, \mathbf{c}_2)$, where $\mathbf{c}_i = \langle c_i, u_i \rangle$ is a configuration of T_i, if there exist lock sets BLS_i and FLS_i, and sets $GS_{ij} \subseteq Q_{ij}$, such that*

1. for each i, the augmented configuration $\langle (c_i, FLS_i, BLS_i, GS_{i1}, ..., GS_{in}), u_i \rangle$ is accepted by $\mathcal{L}_i$,

2. $FLS_1 \cap FLS_2 = \emptyset$ and $BLS_1 \cap BLS_2 = \emptyset$, and

3. there exists k such that $GS_{1k} \cap F_{1k} \neq \emptyset$ and $GS_{2k} \cap F_{2k} \neq \emptyset$.

Intuitively, condition 1 checks for local thread reachability, condition 2 checks for disjointness of lock sets and condition 3 checks for acyclicity of the lock causality graph induced by the lock schedules leading to $\langle c_1, u_1 \rangle$ and $\langle c_2, u_2 \rangle$.

5 Computing LCAs for Operators

We now show how to construct LCAs for (i) *Boolean Operators:* $\vee$ and $\wedge$, and (ii) *Temporal Operators:* F, $\overset{\infty}{\mathsf{F}}$ and X.

Computing LCA for F. Given an LCA $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2)$ our goal is to compute an LCA $\mathcal{M}$, denote by $pre^*(\mathcal{L})$, accepting the pair $(\mathbf{b}_1, \mathbf{b}_2)$ of augmented configurations that is pairwise backward reachable from some pair $(\mathbf{a}_1, \mathbf{a}_2)$ accepted by $\mathcal{L}$. In other words, $\mathcal{M}$ must accept the pre^*-closure of the set of states accepted by $\mathcal{L}$. We first show how to compute the pre^*-closure of a lock augmented T_i-multi-automaton.

Computing the pre^*-closure of a Lock Augmented Multi-Automaton. Given a lock augmented T_i-multi-automaton $\mathcal{A}$, we show how to compute another lock augmented T_i-multi-automaton $\mathcal{B}$, denoted by $pre^*(\mathcal{A})$, accepting the pre^*-closure of the set of augmented configurations of T_i accepted by $\mathcal{A}$. We recall that each augmented configuration of $\mathcal{A}$ is of the form $\langle (c, FLS, BLS, GS_{i1}, ..., GS_{in}), u \rangle$, where c is a control state of T_i, u its stack content, FLS and BLS are locksets, and GS_{ij} is the set of states of G_{ij} induced by the lock schedules seen so far in reaching configuration $\langle c, u \rangle$. We set $\mathcal{A}_0 = \mathcal{A}$ and construct a finite sequence of lock-augmented multi-automata $\mathcal{A}_0, ..., \mathcal{A}_p$ resulting in $\mathcal{B} = \mathcal{A}_p$. Towards that end, we use $\rightarrow_i$ to denote the transition relation of

$\mathcal{A}_i$. For every $i \geq 0$, $\mathcal{A}_{i+1}$ is obtained from $\mathcal{A}_i$ by conserving the sets of states and transitions of $\mathcal{A}_i$ and adding new transitions as follows

1. for each *stack* transition $(c, \gamma) \hookrightarrow (c', w)$ and state q such that $(c', FLS, BLS, GS_{i1}, ..., GS_{in}) \xrightarrow{w}_i q$ we add $(c, FLS, BLS, GS_{i1}, ..., GS_{in}) \xrightarrow{\gamma}_{i+1} q$.

2. for each *lock release* operation $c \overset{r_l}{\hookrightarrow} c'$ and for every state $(c', FLS, BLS, GS_{i1}, ..., GS_{in})$ of $\mathcal{A}_i$, we add a transition $(c, FLS, BLS', GS'_{i1}, ..., GS'_{in}) \xrightarrow{\epsilon}_{i+1} (c', FLS, BLS, GS_{i1}, ..., GS_{in})$ to $\mathcal{A}_{i+1}$, where ϵ is the empty symbol; $BLS' = BLS \cup \{l_i\}$; and for each j, $GS'_{ij} = \delta_{ij}(GS_{ij}, r_l)$.

3. for every *lock acquire* operation $c \overset{a_l}{\hookrightarrow} c'$ and for every state $(c', FLS, BLS\ GS_{i1}, ..., GS_{in})$ of $\mathcal{A}_i$ we add a transition $(c, FLS', BLS'\ GS'_{i1}, ..., GS'_{in}) \xrightarrow{\epsilon}_{i+1} (c', FLS, BLS, GS_{i1}, ..., GS_{in})$ to $\mathcal{A}_{i+1}$, where ϵ is the empty symbol; $BLS' = BLS \setminus \{l\}$; $FLS' = (FLS \cup \{l\}) \setminus BLS$; and for each j, $GS'_{ij} = \delta_{ij}(GS_{ij}, a_l)$.

In the above pre^*-closure computation, the stack transitions do not affect the 'lock-augmentations' and are therefore handled in the standard way. For a lock acquire(release) transitions labeled with $a_l(r_l)$ we need to track the access patterns in order to determine acyclicity of the induced LCGs. Thus in steps 2 and 3 for each GS_{ij}, we compute the set $\delta_{ij}(GS_{ij}, a_l)$ of its successor states via the symbol $r_l(a_l)$ in the regular automaton G^r_{ij} tracking reverse schedules. Moreover, the backward lockset in any configuration is simply the set of locks for which release statements have been encountered during the backward traversal but not the matching acquisitions. Thus if a release statement r_l for lock l is encountered, l is included in BLS (step 2). If later on the acquisition statement a_l in encountered then l is dropped from the BLS (step 3). Finally, the forward lockset is simply the set of locks acquired along a path that are not released. Thus a lock is included in FLS if a lock acquisition symbol is encountered during the backward traversal such that its release has not yet been encountered, i.e., $r_l \notin BLS$. Thus $FLS' = (FLS \cup \{l\}) \setminus BLS$ (step 3).

LCA for F. Given an LCA $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, we define $pre^*(\mathcal{A})$ to be the LCA $(pre^*(\mathcal{A}_1), pre^*(\mathcal{A}_2))$.

Computation of $\wedge$. Let A and B be sets of pairs of configurations accepted by LCAs $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$, respectively. We show how to construct an LCA accepting $A \cap B$ via the standard product construction.

For $1 \leq i \leq 2$, let $T_i = (Q_i, Act_i, \Gamma_i, \mathbf{c}_i, \Delta_i)$, $\mathcal{A}_i = (\Gamma_i^{\mathcal{A}}, P_i^{\mathcal{A}}, \delta_i^{\mathcal{A}}, I_i^{\mathcal{A}}, F_i^{\mathcal{A}})$ and $\mathcal{B}_i = (\Gamma_i^{\mathcal{B}}, P_i^{\mathcal{B}}, \delta_i^{\mathcal{B}}, I_i^{\mathcal{B}}, F_i^{\mathcal{B}})$. Note that for $1 \leq i \leq 2$, $\Gamma_i^{\mathcal{A}} = \Gamma_i^{\mathcal{B}} = \Gamma_i$ and $I_i^{\mathcal{A}} = I_i^{\mathcal{B}} = I_i$. Then we define the LCA $\mathcal{N} = (\mathcal{N}_1, \mathcal{N}_2)$, where $\mathcal{N}_i$ is a multi-automaton accepting $A \cap B$, as the tuple $(\Gamma_i^{\mathcal{N}}, P_i^{\mathcal{N}}, \delta_i^{\mathcal{N}}, I_i^{\mathcal{N}}, F_i^{\mathcal{N}})$, where

(i) $\Gamma_i^{\mathcal{N}} = \Gamma_i$, (ii) $P_i^{\mathcal{N}} = P_i^{\mathcal{A}} \times P_i^{\mathcal{B}}$, (iii) $I_i^{\mathcal{N}} = I_i$, (iv) $F_i^{\mathcal{N}} = F_i^{\mathcal{A}} \times F_i^{\mathcal{B}}$, and (v) $\delta_i^{\mathcal{N}} = \{(s_1, s_2) \xrightarrow{a} (t_1, t_2) |\ s_1 \xrightarrow{a} t_1 \in \delta_i^{\mathcal{A}}, s_2 \xrightarrow{a} t_2 \in \delta_i^{\mathcal{B}}\}$.

A minor technicality is that in order to satisfy the requirement in the definition of a lock-augmented multi-automaton that $I_i \subseteq P_i^{\mathcal{N}}$, we 're-name' states of the form (s, s), where $s \in I_i^{\mathcal{A}}$ as simply s. The correctness of the construction follows from the fact that it is merely the standard product construction with minor changes.

Computation of $\vee$. Similar to the above case (see appendix).

Dual Pumping. Let $\mathcal{CP}$ be a concurrent program comprised of the threads $T_1 = (P_1, Act, \Gamma_1, c_1, \Delta_1)$ and $T_2 = (P_2, Act, \Gamma_2, c_2, \Delta_2)$ and let f be an LTL property. Let

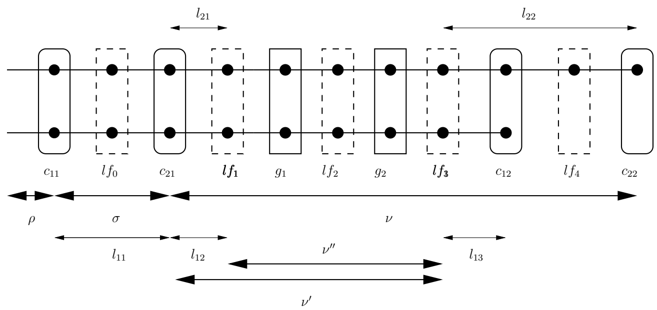

Fig. 2. Pumpable Witness

$\mathcal{BP}$ denote the Büchi system formed by the product of $\mathcal{CP}$ and $\mathcal{B}_{\neg f}$, the Büchi automaton corresponding to $\neg f$. Then LTL model checking reduces to deciding whether there exists an accepting path of $\mathcal{BP}$.

The Dual Pumping Lemma allows us to reduce the problem of deciding whether there exists an accepting computation of $\mathcal{BP}$, to showing the existence of a finite lollipop-like witness with a special structure comprised of a stem ρ which is a finite path of $\mathcal{BP}$, and a pseudo-cycle which is a sequence v of transitions with an accepting state of $\mathcal{BP}$ having the following two properties (i) executing v returns each thread of the concurrent program to the same control location with the same symbol at the top of its stack as it started with, and (ii) executing it does not drain the stack of any thread, viz., any symbol that is not at the top of the stack of a thread to start with is not popped during the execution of the sequence. For ease of exposition we make the assumption that along all infinite runs of $\mathcal{BP}$ any lock that is acquired is eventually released. This restriction can be dropped in the same manner as in [12].

Theorem 6. (Dual Pumping Lemma). *$\mathcal{BP}$ has an accepting run starting from an initial configuration c if and only if there exist $\alpha \in \Gamma_1, \beta \in \Gamma_2$; $u \in \Gamma_1^*, v \in \Gamma_2^*$; an accepting configuration g; configurations lf_0, lf_1, lf_2 and lf_3 in which all locks are free; lock values $l_1, ..., l_m, l'_1, ..., l'_m$; control states $p', p''' \in P_1$, $q', q'' \in P_2$; $u', u'', u''' \in \Gamma_1^*$; and $v', v'', v''' \in \Gamma_2^*$ satisfying the following conditions*

1. $c \Rightarrow (\langle p, \alpha u\rangle, \langle q', v'\rangle, l_1, ..., l_m)$
2. $(\langle p, \alpha\rangle, \langle q', v'\rangle, l_1, ..., l_m) \Rightarrow lf_0 \Rightarrow (\langle p', u'\rangle, \langle q, \beta v\rangle, l'_1, ..., l'_m)$
3. $(\langle p', u'\rangle, \langle q, \beta\rangle, l'_1, ..., l'_m)$
   $\Rightarrow lf_1 \Rightarrow g \Rightarrow lf_2$
   $\Rightarrow (\langle p, \alpha u''\rangle, \langle q'', v''\rangle, l_1, ..., l_m) \Rightarrow lf_3$
   $\Rightarrow (\langle p''', u'''\rangle, \langle q, \beta v'''\rangle, l'_1, ..., l'_m)$

Let ρ, σ, ν be the sequences of global configurations realizing conditions 1, 2 and 3, respectively. We first define sequences of transitions spliced from ρ, σ and ν that we will concatenate to construct an accepting path of $\mathcal{BP}$: (1) $\mathbf{l_{11}}$: the local sequence of T_1 fired along σ. (2) $\mathbf{l_{12}}$: the local sequence of T_1 fired along ν between $c_{21} = (\langle p', u'\rangle, \langle q, \beta\rangle, l'_1, ..., l'_m)$ and lf_1. (3) $\mathbf{l_{13}}$: the local sequence of T_1 fired along ν between lf_2 and $c_{12} = (\langle p, \alpha u''\rangle, \langle q'', v''\rangle, l_1, ..., l_m)$. (4) $\mathbf{l_{21}}$: the local sequence of T_2 fired along ν between $c_{21} = (\langle p', u'\rangle, \langle q, \beta\rangle, l'_1, ..., l'_m)$ and lf_1. (5) $\mathbf{l_{22}}$: the local sequence of T_2 fired along ν between lf_2 and $c_{22} = (\langle p''', u'''\rangle, \langle q, \beta\ v'''\rangle, l_1, ..., l_m)$.

(6) ν': the sequence of global transitions fired along ν till lf_2. (7) ν'': the sequence of global transitions fired along ν between lf_1 and lf_2.

Then $\pi : \rho \; \sigma \; \nu' \; (\; l_{13} \; l_{11} \; l_{12} \; l_{22} \; l_{21} \; \nu'' \;)^{\omega}$ is a scheduling realizing an accepting valid run of $\mathcal{BP}$. Intuitively, thread T_1 is pumped by firing the sequence $l_{13}l_{11}l_{12}$ followed by the local computation of T_1 along ν''. Similarly, T_2 is pumped by firing the sequence $l_{22}l_{21}$ followed by the local computation of T_2 along ν''. The lock free configurations $lf_0, ..., lf_3$ are *breakpoints* that help in scheduling to ensure that π is a valid path. Indeed, starting at lf_2, we first let T_1 fire the local sequences l_{31}, l_{11} and l_{12}. This is valid as T_2 which currently does not hold any lock does not execute any transition and hence does not compete for locks with T_1. Executing these sequences causes T_1 to reach the local configuration of T_1 in lf_1 which is lock free. Thus T_2 can now fire the local sequences l_{22} and l_{21} to reach the local configuration of T_2 in lf_1 after which we let $\mathcal{CP}$ fire ν'' and then repeat the procedure.

It is worth noting that if the lock chains are unbounded in length then the existence of breakpoints as above is not guaranteed.

Constructing an LCA for $\overset{\infty}{\mathsf{F}}$. Conditions 1, 2 and 3 in the statement of the Dual Pumping Lemma can easily be re-formulated via a combination of $\cap$, $\cup$ and pre^*-closure computations for regular sets of configurations. This immediately implies that the computation of an LCA for $\overset{\infty}{\mathsf{F}}$ can be reduced to that for F, $\wedge$ and $\vee$ (see [12] for details).

Computation of X can be handled exactly as in [12].

5.1 The Model Checking Procedure for $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$

Given an LCA $\mathcal{L}_g$ accepting the set of states satisfying a formula g of $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$, we formulated for each operator $\mathsf{Op} \in \{\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}}\}$, a procedure for computing an LCA $\mathcal{L}_{\mathsf{Op}g}$ accepting the set of all configurations that satisfy $\mathsf{Op}g$. Given a property f, by recursively applying these procedures starting from the atomic propositions and proceeding inside out in f we can construct the LCA $\mathcal{L}_f$ accepting the set of states of $\mathcal{CP}$ satisfying f In composing LCAs for different operators a technical issue that arises is of maintaining consistency across the various operators. This has already been handled before in the literature and is discussed in more detail in the appendix. Finally, $\mathcal{CP}$ satisfies f if the initial global state of $\mathcal{CP}$ is accepted by $\mathcal{L}_f$.

6 Conclusion

Among prior work on the verification of concurrent programs, [7] attempts to generalize the techniques given in [5] to model check pushdown systems communicating via CCS-style pairwise rendezvous. However, since even reachability is undecidable for such a framework, the procedures are not guaranteed to terminate, in general, but only for certain special cases, some of which the authors identify. The key idea here is to restrict interaction among the threads so as to bypass the undecidability barrier. Another natural way to obtain decidability is to explore the state space of the given concurrent multi-threaded program for a bounded number of context switches among the threads both for model checking [17, 3] and dataflow analysis [16] or by restricting the allowed set of schedules [2].

The framework of Asynchronous Dynamic Pushdown Networks has been proposed recently [6]. It allows communication via shared variables which makes the model checking problem undecidable. Decidability is ensured by allowing only a bounded number of updates to the shared variables. Networks of pushdown systems with varying topologies for which the reachability problem is decidable have also been studied [4]. Dataflow analysis for asynchronous programs wherein threads can fork off other threads but where threads are not allowed to communicate with each other has also been explored [19, 9] and was shown to be EXPSPACE-hard, but tractable in practice.

In this paper, we have identified fragments of LTL for which the model checking problem is decidable for threads interacting via bounded lock chains thereby delineating precisely the decidability boundary for the problem. A desirable feature of our technique is that it enables compositional reasoning for the concurrent program at hand thereby ameliorating the state explosion problem. Finally, our new results enable us to provide a more refined characterization of the decidability of LTL model checking in terms of boundedness of lock chains as opposed to nestedness of locks.

References

[1] `www.cs.utexas.edu/users/kahlon/blc`.
[2] M. F. Atig and A. Bouajjani. On the Reachability Problem for Dynamic Networks of Conurrent Pushdown Systems. In *RP*, 2009.
[3] M. F. Atig, A. Bouajjani, and S. Qadeer. Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. In *TACAS*, 2009.
[4] M. F. Atig and T. Touili. Verifying Parallel Programs with Dynamic Communication Structures. In *CIAA*, 2009.
[5] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, LNCS 1243, pages 135–150, 1997.
[6] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS*, 2005.
[7] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *IJFCS*, volume 14(4), pages 551–, 2003.
[8] E. A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Volume B*, pages 997–1072, 1998.
[9] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL*, 2007.
[10] V. Kahlon. Boundedness vs. Unboundedness of Lock Chains: Characterizing CFL-Reachability of Threads Communicating via Locks. In *LICS*, 2009.
[11] V. Kahlon and A. Gupta. An Automata-Theoretic Approach for Model Checking Threads for LTL Properties. In *LICS*, 2006.
[12] V. Kahlon and A. Gupta. On the Analysis of Interacting Pushdown Systems. In *POPL*, 2007.
[13] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, 2005.
[14] Nicholas Kidd, Peter Lammich, Tayssir Touili, and Thomas W. Reps. A decision procedure for detecting atomicity violations for communicating processes with locks. In *SPIN*, 2009.
[15] A. Lal, G. Balakrishnan, and T. Reps. Extended weighted pushdown systems. In *CAV*, 2005.
[16] A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural Dataflow Analysis of Concurrent Programs Under a Context Bound. In *TACAS*, 2008.
[17] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.
[18] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *ACM TOPLAS*, 2000.
[19] Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV*, 2006.

Computation of $\vee$**.** Let A and B be sets of pairs of configurations accepted by LCAs $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$, respectively. We show how to construct an LCA accepting $A \cup B$.

For $1 \leq i \leq 2$, let $T_i = (Q_i, Act_i, \Gamma_i, \mathbf{c}_i, \Delta_i)$, $\mathcal{A}_i = (\Gamma_i^{\mathcal{A}}, P_i^{\mathcal{A}}, \delta_i^{\mathcal{A}}, I_i^{\mathcal{A}}, F_i^{\mathcal{A}})$ and $\mathcal{B}_i = (\Gamma_i^{\mathcal{B}}, P_i^{\mathcal{B}}, \delta_i^{\mathcal{B}}, I_i^{\mathcal{B}}, F_i^{\mathcal{B}})$. Note that for $1 \leq i \leq 2$, $\Gamma_i^{\mathcal{A}} = \Gamma_i^{\mathcal{B}} = \Gamma_i$ and $I_i^{\mathcal{A}} = I_i^{\mathcal{B}} = I_i$. Then we define the LCA $\mathcal{N} = (\mathcal{N}_1, \mathcal{N}_2)$, where $\mathcal{N}_i$ is a multi-automaton accepting $A \cup B$, as the tuple $(\Gamma_i^{\mathcal{N}}, P_i^{\mathcal{N}}, \delta_i^{\mathcal{N}}, I_i^{\mathcal{N}}, F_i^{\mathcal{N}})$, where

(i) $\Gamma_i^{\mathcal{N}} = \Gamma_i$, (ii) $P_i^{\mathcal{N}} = P_i^{\mathcal{A}} \cup P_i^{\mathcal{B}}$, (iii) $I_i^{\mathcal{N}} = I_i$ (iv) $F_i^{\mathcal{N}} = F_i^{\mathcal{A}} \cup F_i^{\mathcal{B}}$, and (v) $\delta_i^{\mathcal{N}} = \delta_i^{\mathcal{A}} \cup \delta_i^{\mathcal{B}}$,

Note that $\mathcal{N}_i$ accepts the union of configurations accepted by $\mathcal{A}_i$ and $\mathcal{B}_i$. However, the above definition, though straightforward, leads to a minor technical issue. According to our definition of LCA acceptance, for $\mathcal{N} = (\mathcal{N}_1, \mathcal{N}_2)$ to accept the pair $(\mathbf{c}_1, \mathbf{c}_2)$, where $\mathbf{c}_i = \langle c_i, u_i \rangle$ is a configuration of T_i, it must satisfy the condition (in addition to two others) that there exist lock sets BLS_i and FLS_i, and sets GS_{ij}, such that for each i, the augmented configuration $\mathbf{ac}_i = \langle (c_i, FLS_i, BLS_i, GS_{i1}, ..., GS_{in}), u_i \rangle$ is accepted by $\mathcal{N}_i$,

In order to make sure that our construction does not accept more than the set $A \cup B$, we have to ensure that $\mathbf{ac}_1$ and $\mathbf{ac}_2$ are accepted via final states of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, or via final states of $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively. In other words, we cannot have mixed acceptance via final states of $\mathcal{A}_1$ and $\mathcal{B}_2$ or $\mathcal{B}_1$ and $\mathcal{A}_2$. This issue is alluded to at the end of section 5. It can easily be handled by augmenting the control states of $\mathcal{A}_i$ and $\mathcal{B}_j$ with a consistency bit such that the bit is set to 0 in the control states of $\mathcal{A}_1$ and $\mathcal{A}_2$ whereas it is set to 1 in the control states of $\mathcal{B}_1$ and $\mathcal{B}_2$. Then the acceptance condition for $\mathcal{N}$ adds the extra check that the consistency bit values need to be equal.

Maintaining Consistency Across Operators In composing LCAs for different operators the following technical issue needs to be handled: Consider the LTL formula $f = \mathsf{F}(a \wedge \mathsf{F}b)$. Then the model checking procedure described above would proceed by first building LCAs $\mathcal{L}_a = (\mathcal{L}_a^1, \mathcal{L}_a^2)$ and $\mathcal{L}_b = (\mathcal{L}_b^1, \mathcal{L}_b^2)$ for the atomic propositions a and b, respectively. Next, using the LCA construction for the F operator, we build an LCA $\mathcal{L}_{\mathsf{F}b} = (\mathcal{L}_{\mathsf{F}b}^1, \mathcal{L}_{\mathsf{F}b}^2)$ for $\mathsf{F}b$. Then leveraging the LCA construction for $\wedge$ we build an LCA $\mathcal{L}_{a \wedge \mathsf{F}b} = (\mathcal{L}_{a \wedge \mathsf{F}b}^1, \mathcal{L}_{a \wedge \mathsf{F}b}^2)$ for $a \wedge \mathsf{F}b$ from $\mathcal{L}_a$ and $\mathcal{L}_{\mathsf{F}b}$. Finally, we again use the LCA construction for F to build an LCA $\mathcal{L}_f = (\mathcal{L}_f^1, \mathcal{L}_f^2)$ for f from $\mathcal{L}_{a \wedge \mathsf{F}b}$.

Using our pre^*-closure computation procedure, we see that $\mathcal{L}_f = (pre^*(\mathcal{L}_{a \wedge \mathsf{F}b}^1), pre^*(\mathcal{L}_{a \wedge \mathsf{F}b}^2))$. Note that $\mathcal{L}_f^i$ captures only local reachability information in thread T_i. In other words, $(\mathbf{a}_1, \mathbf{a}_2)$ is accepted by $\mathcal{L}_f$ if there exists a state $(\mathbf{b}_1, \mathbf{b}_2)$ accepted by $\mathcal{L}_{a \wedge \mathsf{F}b}$ such that $\mathbf{a}_i$ is backward reachable from $\mathbf{b}_i$ in thread T_i irrespective of whether $(\mathbf{b}_1, \mathbf{b}_2)$ satisfies $a \wedge \mathsf{F}b$ or not. Recall that whether $(\mathbf{b}_1, \mathbf{b}_2)$ satisfies $a \wedge \mathsf{F}b$ is encoded in the acceptance condition of $\mathcal{L}_{a \wedge \mathsf{F}b}$. Thus is order to ensure that $(\mathbf{a}_1, \mathbf{a}_2)$ satisfies f we need to perform two checks (i) $(\mathbf{b}_1, \mathbf{b}_2)$ satisfies $a \wedge \mathsf{F}b$ and (ii) (a_1, a_2) is backward reachable from $(\mathbf{b}_1, \mathbf{b}_2)$ in the given concurrent program. By our LCA construction for F, the second check is already encoded in the acceptance condition of $\mathcal{L}_f$. To make sure that the first condition is satisfied we also have to augment this check with the acceptance condition for $\mathcal{L}_{a \wedge \mathsf{F}b}$. In general if there are n operators, temporal or boolean, in the given formula f, we need to perform such a check for each operator encountered in building the LCA bottom up via the above mentioned recursive procedure. This has been handled in the literature using the notion of vectors of consistency conditions - one for each operator (see [14] for details).