An Automata-Theoretic Approach for Model Checking Threads for LTL Properties

Vineet Kahlon and Aarti Gupta
NEC Labs America,
Princeton, NJ 08540, USA.

Abstract

In this paper, we propose a new technique for the verification of concurrent multi-threaded programs. In general, the problem is known to be undecidable even for programs with just two threads [1]. However, we exploit the observation that, in practice, a large fraction of concurrent programs can either be modeled as Pushdown Systems communicating solely using locks or can be reduced to such systems by applying standard abstract interpretation techniques or by exploiting separation of data from control. Moreover, standard programming practice guidelines typically recommend that programs use locks in a nested fashion. In fact, in languages like Java and C#, locks are guaranteed to be nested. For such a framework, we show, by using the new concept of Lock Constrained Multi-Automata Pair (LMAP), that pre^**-closures of regular sets of states can be computed efficiently. This is accomplished by reducing the* pre^**-closure computation for a regular set of states of a concurrent program with nested locks to those for its individual threads. Leveraging this new technique then allows us to formulate a fully automatic, efficient and exact (sound and complete) decision procedure for model checking threads communicating via nested locks for indexed linear-time temporal logic formulae.*

1 Introduction

The widespread use of concurrent multi-threaded programs in operating systems, embedded systems and databases coupled with their behavioral complexity necessitates the use of formal methods to debug such systems. Unfortunately most existing methods for verifying concurrent programs suffer from at least one of the following drawbacks: the technique does not scale to large programs due to state explosion; it is sound but not guaranteed complete thus resulting in bogus error traces; or it relies on manual and hence time-consuming abstractions to compress the state space enough to make verification amenable. Moreover, a lot of existing techniques cater only to the verification of reachability properties whereas the debugging of reactive concurrent programs like operating systems requires the verification of a much richer class of properties. In this paper, we give a new technique for model checking threads communicating via nested locks that is fully automatic, efficient, sound and complete, and works for a broad range of linear-time properties.

We consider concurrent multi-threaded programs where each thread is modeled as a Pushdown System (PDS). A PDS has a finite control part corresponding to the valuation of the variables of the thread it represents and a stack which provides a means to model recursion. It is thus a natural and therefore a widely used formalism for modeling sequential programs[1] (cf. [2]). While for a single PDS the model checking problem is efficiently decidable for very expressive logics – both linear and branching time ([2, 3]), it was shown in [1] that even simple properties like reachability become undecidable for systems with only two PDSs communicating using CCS-style pairwise rendezvous.

However, in a large fraction of real-world concurrent software used, for example, in file systems, databases or device drivers, the key issue is to resolve conflicts between different threads competing for access to shared resources. Conflicts are typically resolved using locks which allow mutually exclusive access to a shared resource. Before a thread can gain access to a shared resource it has to acquire the lock associated with that resource which is released after executing all the intended operations. For such software, the interaction between concurrently executing threads is very limited making them loosely coupled. For instance, in a standard file system the control flow in the implementation of the various file operations is usually independent of the data being written to or read from a file. Consequently such programs can either be directly modeled as systems comprised of PDSs communicating via locks or can be reduced to such systems either by applying standard abstract interpretation techniques or by exploiting separation of control and data. A case in point is the Daisy file system [4]. There-

[1]Henceforth we shall use the terms thread and PDS interchangeably

fore, in this paper, we consider the model checking problem for PDSs interacting using locks.

Correctness properties are expressed using Indexed Linear Temporal Logic (LTL) which is a rich formalism that can encode most properties of interest including safety, e.g., presence of conflicts like data races, as well as liveness. In addition, we also consider deadlockability. In general, the model checking problem for even *reachability*, and hence more broadly LTL, is undecidable for systems with just *two* PDSs communicating via locks [5]. However, most real-world concurrent programs use locks in a nested fashion, viz., each thread can only release the lock that it acquired last and that has not yet been released. Indeed, practical programming guidelines used by software developers often require that locks be used in a nested fashion. In fact, in Java and C# locking is syntactically guaranteed to be nested. For the case of nested locks, it was shown in [5] that the model checking problems for pairwise reachability, viz., $\mathsf{EF}(a_i \wedge b_j)$; single-index properties interpreted solely over finite paths; and deadlockability are decidable.

In this paper, we propose a new efficient technique for computing pre^*-closures of regular sets of states of systems comprised of multiple PDSs interacting via nested locks. Towards that end, we introduce the new concept of *Lock-Constrained Multi-Automata Pair (LMAP)* which allows us to decompose the computation of pre^*-closures of regular sets of configurations of a multi-PDS system communicating via nested locks to that of its constituent PDSs for which existing efficient techniques (see [2]) can be leveraged. This decomposition enables us to avoid the state explosion problem thereby resulting in an efficient procedure for computing pre^*-closures. An LMAP $\mathcal{A}$ accepting a regular set of configurations C of a system $\mathcal{CP}$ comprised of PDSs T_1 and T_2 is a pair of multi-automata $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_i$ is a multi-automaton accepting the set of local configurations of T_i occurring in the global configurations of $\mathcal{CP}$ in C. The lock interaction among the PDSs is encoded in the acceptance criterion for an LMAP which filters out those pairs of local configurations of T_1 and T_2 which are not simultaneously reachable due to lock enforced mutual exclusion and are therefore not in C. Specifically, to capture lock interaction, we track patterns of lock acquisitions and releases using the new concepts of *backward acquisition history (BAH)* and *forward acquisition history (FAH)*.

Next, we show how to apply our pre^*-closure computation technique to formulate an efficient model checking procedure for threads interacting via nested locks for linear time properties. Towards that end, we invoke the automata-theoretic paradigm to first construct a Büchi system $\mathcal{BP}$ from the given multi-threaded program $\mathcal{CP}$ and linear time formula f. The model checking problem then reduces to deciding whether $\mathcal{BP}$ has an accepting path, viz., a path along which final states occurs infinitely often. For finite state systems, along any infinite path some state must occur infinitely often and so along any accepting path p of $\mathcal{BP}$ there must be an occurrence of a final state along a subsequence p_s of p that starts and ends at the same state s. This observation reduces the problem of deciding the non-emptiness of $\mathcal{BP}$ to showing the existence of a finite *lollipop*, viz., a finite path of $\mathcal{BP}$ leading to a cycle with an accepting state which can be pumped indefinitely to yield an accepting computation. For infinite-state systems, however, the above observation is no longer valid. Indeed, in our case each thread has a stack which could potentially grow to be of unbounded depth and so the existence of a cycle of global states as discussed above is not guaranteed.

To overcome this problem, we first formulate the *Dual Pumping Lemma*[2], which allows us to reduce the problem of deciding whether there exists an accepting computation of $\mathcal{BP}$, to showing the existence of a finite (pseudo-)lollipop like witness with a special structure. The witness is essentially comprised of a stem u 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 v 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 v's execution. These two properties ensure that we can pump the local computations of the individual threads along v by executing them back-to-back, even though each pumping may increase the stack depth of the thread. This allows us to then construct a valid accepting sequence of $\mathcal{BP}$ by interleaving the local pumping sequences of the individual threads via an intricate scheduling of the transitions of the threads occurring along v to effectively pump the pseudo-lollipop, i.e., execute the sequence u followed by the sequence v back-to-back indefinitely. The intricate scheduling is required in order to accommodate lock interaction among threads. The special structure of the witness guarantees the existence of such a scheduling.

Next, by exploiting the special structure of the witness (pseudo-)lollipop, we reduce the problem of deciding its existence to the computation of pre^*-closures of regular sets of configurations of $\mathcal{CP}$. It follows from the Dual Pumping Lemma, that for model checking indexed LTL formulae, it suffices to either (1) compute pre^*-closures for a set C of configurations in which all locks are free, or (2) compute lock-free configurations of the pre^*-closure of a set (that possibly contains configurations in which some locks are held). The notion of BAH is used in the first case while FAH in the second one. Decomposition is then achieved by showing that given an LMAP $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, if $\mathcal{B}_i$ is an

[2]For simplicity we consider programs with two threads only. In general, we can analogously formulate a *Multi-Pumping Lemma*

MA accepting the pre^*-closure of the configurations of the *individual thread* T_i accepted by $\mathcal{A}_i$, then, in the two cases of interest mentioned above, the LMAP $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ accepts the pre^*-closure of the regular set of the *concurrent program* $\mathcal{CP}$ accepted by $\mathcal{A}$. Thus the decomposition results from maintaining the local configurations of the constituent threads separately as multi-automata and computing the pre^*-closures on these multi-automata individually for each thread.

The rest of the paper is organized as follows. Section 2 introduces the system model. LMAPs are introduced in Section 3 while the Dual Pumping Lemma is formulated in section 4. We conclude with some remarks and comparison to related work in section 5.

2 System Model

We consider multi-threaded programs wherein threads communicate using locks. We model each thread as a *pushdown system (PDS)* [2]. A PDS has a finite control part corresponding to the valuation of the variables of the thread it represents and a stack which models recursion. Formally, a PDS is a five-tuple $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$, where P is a finite set of *control locations*, *Act* is a finite set of *actions*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (P \times \Gamma) \times Act \times (P \times \Gamma^*)$ is a finite set of *transition rules*. If $((p, \gamma), a, (p', w)) \in \Delta$ then we write $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$. A *configuration* of $\mathcal{P}$ is a pair $\langle p, w \rangle$, where $p \in P$ denotes the control location and $w \in \Gamma^*$ the *stack content*. We call c_0 the *initial configuration* of $\mathcal{P}$. The set of all configurations of $\mathcal{P}$ is denoted by $\mathcal{C}$. For each action a, we define a relation $\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{C}$ as follows: if $\langle q, \gamma \rangle \overset{a}{\hookrightarrow} \langle q', w \rangle$, then $\langle q, \gamma v \rangle \xrightarrow{a} \langle q', wv \rangle$ for every $v \in \Gamma^*$.

A concurrent program with n threads and m locks $l_1, ..., l_m$ is formally defined as a tuple of the form $\mathcal{CP} = (T_1, ..., T_n,\ L_1, ..., L_m)$, where for each i, $T_i = (P_i, Act_i, \Gamma_i, c_i, \Delta_i)$ is a pushdown system (thread), and for each j, $L_j \subseteq \{\bot, T_1, ..., T_n\}$ is the possible set of values that lock l_j 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 lock l_i in configuration c, then $l_i = \bot$, else l_i is the thread currently holding it. 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 global global configurations of $\mathcal{CP}$ in the usual way. The reachability relation $\Rightarrow$ is the reflexive and transitive closure of the successor relation $\rightarrow$ defined above. A sequence $x = x_0, x_1, ...$ of global configurations of $\mathcal{CP}$ is a *computation* if x_0 is the initial global configuration of $\mathcal{CP}$ and for each i, $x_i \xrightarrow{a} x_{i+1}$, where either for some j, $a \in Act_j$ or for some k, $a = release(l_k)$ or $a = acquire(l_k)$. Given a thread T_i and a reachable global configuration $\mathbf{c} = (c_1, ..., c_n,\ l_1, ..., l_m)$ of $\mathcal{CP}$, we use *Lock-Set*$(T_i, \mathbf{c})$ to denote the set of locks held by T_i in $\mathbf{c}$, viz., the set $\{l_j \mid l_j = T_i\}$. Also, given a thread T_i and a reachable global configuration $\mathbf{c} = (c_1, ..., c_n, l_1, ..., l_m)$ of $\mathcal{CP}$, the *projection* of $\mathbf{c}$ onto T_i, denoted by $c \downarrow T_i$, is defined to be the configuration $(c_i, l'_1, ..., l'_m)$ of the concurrent program comprised solely of the thread T_i, where $l'_i = T_i$ if $l_i = T_i$ and $\bot$, otherwise (locks not held by T_i are freed).

Multi-Automata (see [2]) Multi-Automata are used to capture regular (potentially infinite) sets of configurations of a PDS in a finite form. Let $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$ be a pushdown system, where $P = \{p_1, ..., p_m\}$. A $\mathcal{P}$*-multi-automaton* ($\mathcal{P}$-MA for short) is a tuple $\mathcal{A} = (\Gamma, Q, \delta, I, F)$ where Q is a finite set of states, $\delta \subseteq Q \times \Gamma \times Q$ is a set of transitions, $I = \{s_1, ..., s_m\} \subseteq Q$ is a set of initial states and $F \subseteq Q$ is a set of final states. Each initial state s_i corresponds to a control state p_i of $\mathcal{P}$, and vice versa. We define the transition relation $\longrightarrow \subseteq Q \times \Gamma^* \times Q$ as the smallest relation satisfying the following: (i) if $(q, \gamma, q') \in \delta$ then $q \xrightarrow{\gamma} q'$, (ii) $q \xrightarrow{\epsilon} q$ for every $q \in Q$, and (iii) if $q \xrightarrow{w} q''$ and $q'' \xrightarrow{\gamma} q'$ then $q \xrightarrow{w\gamma} q'$. We say that $\mathcal{A}$ accepts a configuration $\langle p_i, w \rangle$ iff $s_i \xrightarrow{w} q$ for some $q \in F$. The set of configurations recognized by $\mathcal{A}$ is denoted by *Conf*$(\mathcal{A})$.

Nested Lock Access. We say that a concurrent program accesses locks in a *nested* fashion iff along each computation of the program a thread can only release the last lock that it acquired along that computation and that has not yet been released. In this paper, we only consider multi-threaded programs with nested access to locks. This is because even reachability, and hence LTL model checking, is known to be undecidable, in general, for programs with even two threads which allow non-nested access to locks (cf. [5]).

3 Lock-Constrained Multi-Automata Pair

In this section, we introduce the new concept of *Lock-Constrained Multi-Automata Pair (LMAP)* which we use to represent regular sets of configurations of concurrent programs with threads communicating via nested locks. We show that LMAPs are closed under (i) boolean operations, and also, (ii) in certain cases of interest, under computation of pre^* closures. Then we formulate a Dual Pumping Lemma which allows us to reduce the LTL model checking problem for threads interacting via nested locks to computation of pre^*-closures of regular sets of configurations accepted by LMAPs. Thus LMAPs plays the same role as Multi-Automata (MAs) do in the model checking of a single PDS for LTL properties, by allowing us to succinctly (and finitely) represent potentially infinite sets of configurations of the given concurrent program in a way that enables us to compute their pre^*-closure efficiently. This helps us to efficiently decide the LTL model checking problem at hand.

3.1 Lock Constrained Multi-Automata Pair

To motivate the concept of *Lock-Constrained Multi-Automata Pair (LMAP)* we revisit the key idea behind multi-automata (see section 2) that are used in the LTL model checking of a single PDS. An MA $\mathcal{A}$ accepting a (regular) set of configurations C of a PDS $T = (P, Act, \Gamma, \mathbf{c_0}, \Delta)$ has an *initial* state s_i of $\mathcal{A}$ corresponding to each control state $p_i \in P$ and transitions labeled with stack symbols from the set Γ such that $\langle p_i, w\rangle \in C$ iff there is a path in $\mathcal{A}$ starting at s_i labeled with w and leading to a final state of $\mathcal{A}$. Thus the stack content in each configuration of C is stored as a sequence of labels along an accepting path of $\mathcal{A}$.

In a concurrent program $\mathcal{CP}$ comprised of the two threads $T_1 = (P_1, Act_1, \Gamma_1, \mathbf{c}_1, \Delta_1)$ and $T_2 = (P_2, Act_2, \Gamma_2, \mathbf{c}_2, \Delta_2)$, for each configuration we need to keep track of the contents of the stacks of both T_1 and T_2. This is accomplished by using a pair of multi-automata $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_i$ is a multi-automaton accepting a set of regular configurations of T_i. The broad idea is that a global configuration $\mathbf{c}$ is accepted by $\mathcal{A}$ iff the local configurations of T_1 and T_2 in $\mathbf{c}$ are accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively.

However, we also need to factor in lock interaction among the threads that prevents them from simultaneously reaching certain pairs of local configurations. To capture lock interaction, we introduce the new concept of *backward acquisition history (BAH)* and generalize the existing concept of *acquisition history* [5] to that of *forward acquisition history (FAH)*. It turns out (using the Dual Pumping Lemma) that for model checking LTL formulae, we need to either (1) compute pre^*-closures for a set C of configurations in which all locks are free, or (2) compute those configurations of the pre^*-closure of a set (that possibly contains configurations in which some locks are held), in which all locks are free. The notion of BAH is used in the first case while FAH in the second one.

Key Insight. By tracking patterns of lock acquisition and releases, we will show that an LMAP accepting the pre^*-closure of the set of configurations accepted by the LMAP $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ is the pair $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$, where $\mathcal{B}_i$ is a multi-automaton accepting the pre^*-closure of the set of configurations of thread T_i accepted by $\mathcal{A}_i$ (which can be computed efficiently using the techniques given in [2]). This reduces the pre^*-closure computation of a set of configurations of a concurrent program with threads interacting via nested locks to its individual threads and thereby not only avoids the state explosion problem but, as we shall show, makes our procedure efficient.

3.1.1 Backward Acquisition History

We motivate the notion of BAH using the example concurrent program $\mathcal{CP}$ comprised of the two PDSs shown in figure 1 with locks l_1 and l_2. Suppose that we are in-

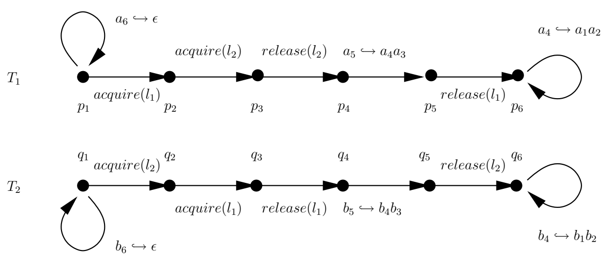

Figure 1. Example Concurrent Program

terested in computing the pre^*-closure of the set $LC = \{(\langle p_6, a_1a_2a_3\rangle, \langle q_6, b_1b_2b_3\rangle,\ \perp, \perp)\}$, i.e., the set of configurations $\mathbf{c}$ of $\mathcal{CP}$ such there is a path from $\mathbf{c}$ to a configuration $\mathbf{d}$ in LC. Note that in each configuration of LC all locks are free. The key insight is that by tracking patterns of backward lock releases we can reduce this to evaluating the pre^*-closures of regular sets of configurations $LC_1 = \{\langle p_6, a_1a_2a_3\rangle, \perp, \perp)\}$ of thread T_1 and $LC_2 = \{\langle q_6, b_1b_2b_3\rangle, \perp, \perp)\}$ of T_2 for the concurrent programs comprised *solely of the individual threads* T_1 and T_2, respectively, instead of the original multi-threaded program.

Consider, for example, the configuration $(\langle p_2, a_5\rangle, T_1, \perp)$, which belongs to $pre^*_{T_1}(LC_1)$[3] the path $(\langle p_2, a_5\ \rangle, T_1, \perp) \overset{acquire(l_2)}{\longrightarrow} (\langle\ p_3, a_5\ \rangle, T_1, T_1) \overset{release(l_2)}{\longrightarrow} (\langle\ p_4, a_5\ \rangle, T_1, \perp) \longrightarrow (\langle\ p_5,\ a_4a_3\ \rangle, T_1, \perp) \overset{release(l_1)}{\longrightarrow} (\langle\ p_6,\ a_4a_3\ \rangle, \perp, \perp) \longrightarrow (\langle\ p_6,\ a_1a_2a_3\rangle, \perp, \perp)$ of T_1; and the configuration $(\langle\ q_2,\ b_5\ \rangle, \perp, T_2)$ which belongs to $pre^*_{T_2}(LC_2)$ via the path $(\langle q_2,\ b_5\ \rangle, \perp, T_2) \overset{acquire(l_1)}{\longrightarrow} (\langle\ q_3,\ b_5\ \rangle, T_2, T_2) \overset{release(l_1)}{\longrightarrow} (\langle\ q_4,\ b_5\ \rangle, \perp, T_2) \longrightarrow (\langle\ q_5,\ b_4b_3\ \rangle, \perp, T_2) \overset{release(l_2)}{\longrightarrow} (\langle\ q_6,\ b_4b_3\ \rangle, \perp, \perp) \longrightarrow (\langle\ q_6,\ b_1b_2b_3\rangle, \perp, \perp)$ of T_2. Note that even though T_1 and T_2 hold different sets of locks, i.e., $\{l_1\}$ and $\{l_2\}$, respectively, at control locations p_2 and q_2, there does not exist a global configuration of $\mathcal{CP}$ with T_1 and T_2 in the local configurations $(\langle\ p_2,\ a_5\ \rangle, T_1, \perp)$ and $(\langle\ q_2,\ b_5\ \rangle, \perp, T_2)$, respectively, that is backward reachable in $\mathcal{CP}$ from $(\langle\ p_6,\ a_1a_2a_3\ \rangle, \langle\ q_6,\ b_1b_2b_3\ \rangle, \perp, \perp)$. The reason is that in order for T_1 to reach p_6 from p_2 it first has to acquire (and release) lock l_2. However, in order to do that T_2, which currently holds lock l_2, must release it. But for T_2 to release l_2, it first has to acquire (and release) l_1 which is currently held by T_1. This creates an unresolvable cyclic dependency.

In general, when testing for backward reachability of $\mathbf{c}$ from $\mathbf{d}$ in $\mathcal{CP}$, it suffices to test whether there exist local paths x and y in the individual threads from states $\mathbf{c}_1 = \mathbf{c} \downarrow T_1$ to $\mathbf{d}_1 = \mathbf{d} \downarrow T_1$ and from $\mathbf{c}_2 = \mathbf{c} \downarrow T_2$ to $\mathbf{d}_2 = \mathbf{d} \downarrow T_2$, respectively, such that along x and y locks can be acquired in a compatible fashion. Compatibility ensures that we do not end up with an unresolvable cyclic dependency as above. This allows us to reconcile x and y to get

[3]Unless clear from the context, we use $pre^*_{T_i}(C)$ to denote the pre^*-closures for a set C of configurations of thread T_i.

a valid path of $\mathcal{CP}$ from $\mathbf{c}$ to $\mathbf{d}$. Next we define the notion of backward acquisition history, which captures patterns of lock releases from $\mathbf{d}$ to $\mathbf{c}$ that are used to test compatibility. Our discussion above is then formalized in theorem 2.

Definition 1 (Backward Acquisition History). *Let x be a computation of a concurrent program $\mathcal{CP}$ leading from configurations $\mathbf{c}$ to $\mathbf{d}$. Then for thread T_i and lock l_j of $\mathcal{CP}$, if $l_j \notin$ Lock-Set$(T_i, \mathbf{c})$ then $BAH(T_i, \mathbf{c}, l_j, x)$ is defined to be the empty set $\emptyset$. If $l_j \in$ Lock-Set$(T_i, \mathbf{c})$, then $BAH(T_i, \mathbf{c}, l_j, x)$ is the set of locks that were released (and possibly acquired) by T_i after the last release of l_j by T_i in traversing backward along x from $\mathbf{d}$ to $\mathbf{c}$. If $l_j \in$ Lock-Set$(T_i, \mathbf{c})$ and l_j wasn't released along x, then $BAH(T_i, \mathbf{c}, l_j, x)$ is the set of locks that were released (and possibly acquired) by T_i in traversing backwards along x.*

Theorem 2 (Backward Decomposition Result) *Let $\mathcal{CP}$ be a concurrent program comprised of the two threads T_1 and T_2 with nested locks. Then configuration $\mathbf{c}$ of $\mathcal{CP}$ is backward reachable from configuration $\mathbf{d}$ in which all locks are free iff configurations $\mathbf{c}_1 = \mathbf{c} \downarrow T_1$ of T_1 and $\mathbf{c}_2 = \mathbf{c} \downarrow T_2$ of T_2 are backward reachable from configurations $\mathbf{d}_1 = \mathbf{d} \downarrow T_1$ and $\mathbf{d}_2 = \mathbf{d} \downarrow T_2$, respectively, via computation paths x and y of programs comprised solely of threads T_1 and T_2, respectively, such that*

1. Lock-Set$(T_1, \mathbf{c}_1) \cap$ Lock-Set$(T_2, \mathbf{c}_2) = \emptyset$

2. there do not exist locks $l \in$ Lock-Set$(T_1, \mathbf{c}_1)$ and $l' \in$ Lock-Set$(T_2, \mathbf{c}_2)$ such that $l \in BAH(T_2, \mathbf{c}_2, l', y)$ and $l' \in BAH(T_1, \mathbf{c}_1, l, x)$.

BAH enhanced pre^*-computation To make use of the above result while computing $pre^*(C)$ for a set C such that all locks are free in each configuration of $\mathcal{CP}$ in C, we augment the configurations of each individual thread with a BAH entry for each lock. Thus a configuration of a program comprised solely of thread T_i is now of the form $(\langle c, w\rangle, l_1, ..., l_m, BAH_1, ..., BAH_m)$ where BAH_i tracks the BAH of lock l_i.

Reworking our example (fig 1) with BAH-augmented configurations, we see that augmented configuration $(\langle p_2, a_5\rangle, T_1, \bot, \{l_2\}, \emptyset)$ of thread T_1 belongs to $pre^*_{T_1}(\{\mathbf{d}_1\})$ via the path (of augmented configurations) x : $(\langle p_2, a_5\rangle, T_1, \bot, \{l_2\}, \emptyset) \xrightarrow{acquire(l_2)} (\langle p_3, a_5\rangle, T_1, T_1, \{l_2\}, \emptyset) \xrightarrow{release(l_2)} (\langle p_4, a_5\rangle, T_1, \bot, \emptyset, \emptyset) \longrightarrow (\langle p_5, a_4a_3\rangle, T_1, \bot, \emptyset, \emptyset) \xrightarrow{release(l_1)} (\langle p_6, a_4a_3\rangle, \bot, \bot, \emptyset, \emptyset) \longrightarrow (\langle p_6, a_1a_2a_3\rangle, \bot, \bot, \emptyset, \emptyset)$. Note that in traversing backwards from the configuration $(\langle p_6, a_4a_3\rangle, \bot, \bot, \emptyset, \emptyset)$ via the transition $release(l_1)$, we set $l_1 = T_1$ indicating that l_1 is now held by T_1. Next, in traversing backwards from the configuration $(\langle p_4, a_5\rangle, T_1, \bot, \emptyset, \emptyset)$ via the transition $release(l_2)$ and set $l_2 = T_1$ and add lock l_2 to the backward acquisition history of lock l_1 as it is currently held by T_1. Similarly we can see that configuration $(\langle q_2, b_5\rangle, \bot, T_2, \emptyset, \{l_1\})$ of the augmented thread T_2 belongs to $pre^*_{T_2}(\{\mathbf{d}_2\})$ via the path y : $(\langle q_2, b_5\rangle, \bot, T_2, \emptyset, \{l_1\}) \xrightarrow{acquire(l_1)} (\langle q_3, b_5\rangle, T_2, T_2, \emptyset, \{l_1\}) \xrightarrow{release(l_1)} (\langle q_4, b_5\rangle, \bot, T_2, \emptyset, \emptyset) \longrightarrow (\langle q_5, b_4b_3\rangle, \bot, T_2, \emptyset, \emptyset) \xrightarrow{release(l_2)} (\langle q_6, b_4b_3\rangle, \bot, \bot, \emptyset, \emptyset) \longrightarrow (\langle q_6, b_1b_2b_3\rangle, \bot, \bot, \emptyset, \emptyset)$. Since the states $\mathbf{c}_1 = (\langle p_2, a_5\rangle, T_1, \bot, \{l_2\}, \emptyset)$ and $\mathbf{c}_2 = (\langle q_2, b_5\rangle, \bot, T_2, \emptyset, \{l_1\})$ are not BAH-compatible as $l_2 \in BAH(T_1, \mathbf{c}_1, l_1, x) = \{l_2\}$ and $l_1 \in BAH(T_2, \mathbf{c}_2, l_2, y) = \{l_1\}$, by theorem 2, global configuration $\mathbf{c}$ is not backward reachable from $\mathbf{d}$ in $\mathcal{CP}$.

However, it can be seen that $\mathbf{c}'_1 = (\langle p_1, a_5\rangle, \bot, \bot, \emptyset, \emptyset)$ is backward reachable from $\mathbf{d}_1$ in T_1 and $\mathbf{c}'_2 = (\langle q_1, b_5\rangle, \bot, \bot, \emptyset, \emptyset)$ from $\mathbf{d}_2$ in T_2. Note that since all locks of $\mathcal{CP}$ are free in $\mathbf{c}'_1$ and $\mathbf{c}'_2$, the BAH of each lock is the empty set in these configurations. In this case, however, since $\mathbf{c}'_1$ and $\mathbf{c}'_2$ are trivially BAH-compatible, $\mathbf{c}' = (\langle p_1, a_5\rangle, \langle q_1, b_5\rangle, \bot, \bot)$ is backward reachable from $\mathbf{d}$ in $\mathcal{CP}$.

BAH-enhanced pre^*-computation To construct an MA accepting the pre^*-closure of a regular set of BAH-enhanced configurations accepted by a given MA (used later in the pre^*-closure computation of LMAPs), we slightly modify the procedure given in [2] for constructing an MA accepting the pre^*-closure of a regular set of (non-enhanced) configurations accepted by a given MA. The only difference is that since now we are also tracking BAHs, whenever we encounter a lock operation, we modify the BAHs accordingly. Thus using results from [2], we have,

Lemma 3 *Given a PDS $\mathcal{P}$, and a regular set of BAH-augmented configurations accepted by a $\mathcal{P}$-MA $\mathcal{A}$, we can construct a $\mathcal{P}$-MA $\mathcal{A}_{pre^*}$ recognizing $pre^*(Conf(\mathcal{A}))$ in time polynomial in the sizes of $\mathcal{A}$ and the control states of $\mathcal{P}$ and exponential in the number of locks of $\mathcal{P}$.*

3.1.2 Forward Acquisition History

The notion of *Forward Acquisition History (FAH)* is motivated by our goal of using backward reachability to compute those configurations in the pre^*-closure of a set C of global configurations of $\mathcal{CP}$ in which all locks are free. It is an extension of the concept of acquisition histories, defined in [5]. The basic difference is that while acquisition histories were defined for computation paths starting at the initial state, FAHs are defined for paths starting at arbitrary states. The only property of the initial state that was used in developing the theory of [5] was that all locks are free in the initial state. Since we are only interested in those configurations of $pre^*(C)$ is which all locks are free, we need to consider only those computation paths that start at configurations of $\mathcal{CP}$ in which all locks are free and so the decomposition result in [5] also extends to this case with minor modifications. This is formulated below as the Forward Decomposition Result.

Definition 4 (Forward Acquisition History). *Let x be a computation of a concurrent program $\mathcal{CP}$ leading from configurations* $\mathbf{c}$ *to* $\mathbf{d}$. *For thread T_i and lock l_j of $\mathcal{CP}$, if $l_j \notin$ Lock-Set$(T_i, \mathbf{d})$ then $FAH(T_i, \mathbf{c}, l_j, x)$ is defined to be the empty set $\emptyset$. If $l_j \in$ Lock-Set$(T_i, \mathbf{d})$, then we define $FAH(T_i, \mathbf{c}, l_j, x)$ to be the set of locks that were acquired (and possibly released) by T_i after the last acquisition of l_j by T_i in traversing forward along x from* $\mathbf{c}$ *to* $\mathbf{d}$. *If $l_j \in$ Lock-Set$(T_i, \mathbf{d})$ but l_j was not acquired along x, then $FAH(T_i, \mathbf{c}, l_j, x)$ is the set of locks that were acquired (and possibly released) by T_i along x.*

Theorem 5 (Forward Decomposition Result) *Let $\mathcal{CP}$ be a concurrent program comprised of the two threads T_1 and T_2 with nested locks. Then configuration* $\mathbf{c}$ *of $\mathcal{CP}$ in which all locks are free is backward reachable from* $\mathbf{d}$ *iff configurations $\mathbf{c}_1 = \mathbf{c} \downarrow T_1$ of T_1 and $\mathbf{c}_2 = \mathbf{c} \downarrow T_2$ of T_2 are backward reachable from configurations $\mathbf{d}_1 = \mathbf{d} \downarrow T_1$ and $\mathbf{d}_2 = \mathbf{d} \downarrow T_2$, respectively, via computation paths x and y of programs comprised solely of threads T_1 and T_2, respectively, such that*

1. *Lock-Set$(T_1, \mathbf{d}_1) \cap$ Lock-Set$(T_2, \mathbf{d}_2) = \emptyset$, and*
2. *there do not exist locks $l \in$ Lock-Set$(T_1, \mathbf{d}_1)$ and $l' \in$ Lock-Set$(T_2, \mathbf{d}_2)$ such that $l \in FAH(T_2, \mathbf{c}_2, l', y)$ and $l' \in FAH(T_1, \mathbf{c}_1, l, x)$.*

Unlike pre^*-closure for BAH-augmented configurations, an important issue that arises when computing pre^*-closure for FAH-augmented configurations, is that we need to compute FAHs while performing a backward reachability analysis. For that we need to augment the configurations of each thread with two extra fields as we now illustrate.

FAH-enhanced pre^*-computation Suppose that we want to compute the lock free configurations of $pre^*(\{\mathbf{d}\})$, where $\mathbf{d}$ is the configuration $(\langle p_5, a_4a_3\rangle, \langle q_5, b_4b_3\rangle, T_1, T_2)$ of the concurrent program shown in figure 1. Let $\mathbf{d}_1 = \mathbf{d} \downarrow T_1 = (p_5, a_4a_3, T_1, \perp)$ and $\mathbf{d}_2 = \mathbf{d} \downarrow T_2 = (q_5, b_4b_3, \perp, T_2)$. By theorem 5, it suffices to compute the set of all pairs of lock-free configurations $\mathbf{c}_1$ and $\mathbf{c}_2$ of T_1 and T_2, respectively, such that the FAHs of $\mathbf{c}_1$ and $\mathbf{c}_2$ along some paths of T_1 and T_2 starting at $\mathbf{c}_1$ and $\mathbf{c}_2$ and ending at $\mathbf{d}_1$ and $\mathbf{d}_2$, respectively, are compatible. Note that, by definition, the FAH of l along a path x from $\mathbf{c}_1$ to $\mathbf{d}_1$ is the set of locks that were acquired and released since the last acquisition of l in traversing forward along x. Thus while traversing x backwards, we stop updating the FAH of l after encountering the first acquisition of l along x as all lock operations on l encountered after that are immaterial. To ensure that, we maintain two extra entries in the FAH-augmented configurations. The first entry LHI is the set of locks held initially in $\mathbf{d}_1$ when starting the backward reachability. The second entry LR is the set of locks from LHI that have been acquired so far in the backward search. For a lock $l \in LHI$, once a transition acquiring l is encountered for the first time while performing backward reachability, we add it to LR and stop modifying it's FAH even if it is acquired or released again during the backward search. Thus an FAH-augmented configuration is of the form $(\langle p, w\rangle, l_1, ..., l_m, FAH_1, ..., FAH_m, LHI, LR)$.

Going back to our example, we see that the FAH-augmented configuration $(\langle p_1, a_5\rangle, \perp, \perp, \{l_2\}, \emptyset, \{l_1\}, \{l_1\})$ of the augmented thread T_1 belongs to $pre^*_{T_1}(\{\mathbf{d}_1\})$ via the backwardly traversed path x : $(\langle p_5, a_4a_3\rangle, T_1, \perp, \emptyset, \emptyset, \{l_1\}, \emptyset) \leftarrow (\langle p_4, a_5\rangle, T_1, \perp, \emptyset, \emptyset, \{l_1\}, \emptyset) \overset{release(l_2)}{\longleftarrow} (\langle p_3, a_5\rangle, T_1, T_1, \{l_2\}, \emptyset, \{l_1\}, \emptyset) \overset{acquire(l_2)}{\longleftarrow} (\langle p_2, a_5\rangle, T_1, \perp, \{l_2\}, \emptyset, \{l_1\}, \emptyset) \overset{acquire(l_1)}{\longleftarrow} (\langle p_1, a_5\rangle, \perp, \perp, \{l_2\}, \emptyset, \{l_1\}, \{l_1\})$. Similarly, the FAH-augmented configuration $(\langle q_1, b_5\rangle, \perp, \perp, \emptyset, \{l_1\}, \{l_2\}, \{l_2\})$ of the thread T_2 belongs to $pre^*_{T_2}(\{\mathbf{d}_2\})$ via the backwardly traversed path y : $(\langle q_5, b_4b_3\rangle, \perp, T_2, \emptyset, \emptyset, \{l_2\}, \emptyset) \leftarrow (\langle q_4, b_5\rangle, \perp, T_2, \emptyset, \emptyset, \{l_2\}, \emptyset) \overset{release(l_1)}{\longleftarrow} (\langle q_3, b_5\rangle, T_2, T_2, \emptyset, \{l_1\}, \{l_2\}, \emptyset) \overset{acquire(l_1)}{\longleftarrow} (\langle q_2, b_5\rangle, \perp, T_2, \emptyset, \{l_1\}, \{l_2\}, \emptyset) \overset{acquire(l_2)}{\longleftarrow} (\langle q_1, b_5\rangle, \perp, \perp, \emptyset, \{l_1\}, \{l_2\}, \{l_2\})$. Since augmented states $\mathbf{c}_1 = (\langle p_1, a_5\rangle, \perp, \perp, \{l_2\}, \emptyset, \{l_1\}, \{l_1\})$ and $\mathbf{c}_2 = (\langle q_1, b_5\rangle, \perp, \perp, \emptyset, \{l_1\}, \{l_2\}, \{l_2\})$ are not FAH-compatible as $l_2 \in \{l_2\} = FAH(T_1, \mathbf{c}_1, l_1, x)$ and $l_1 \in \{l_1\} = FAH(T_2, \mathbf{c}_2, l_2, y)$, by theorem 5 global configuration $\mathbf{c}$ is not backward reachable from $\mathbf{d}$ in $\mathcal{CP}$.

Given an MA $\mathcal{A}$ accepting a regular set of FAH-augmented configurations of a PDA $\mathcal{P}$, we can, as for BAH-augmented configurations, efficiently construct an MA accepting $pre^*(Conf(\mathcal{A}))$. As for the BAH case, using results from [2], we have the following.

Lemma 6 *Given a PDS $\mathcal{P}$, and a regular set of FAH-augmented configurations accepted by a $\mathcal{P}$-MA $\mathcal{A}$, we can construct a $\mathcal{P}$-MA $\mathcal{A}_{pre^*}$ recognizing $pre^*(Conf(\mathcal{A}))$ in time polynomial in the sizes of $\mathcal{A}$ and the control states of $\mathcal{P}$ and exponential in the number of locks of $\mathcal{P}$.*

3.1.3 Lock Constrained Multi-Automata Pair

Given a concurrent program $\mathcal{CP}$ comprised of the two threads $T_1 = (P_1, Act_1, \Gamma_1, \mathbf{c}_1, \Delta_1)$ and $T_2 = (P_2, Act_2, \Gamma_2, \mathbf{c}_2, \Delta_2)$, we define a *Lock-Constrained Multi-Automata Pair (LMAP)* for $\mathcal{CP}$, denoted by $\mathcal{CP}$-LMAP, as a pair $(\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_i = (\Gamma_i, Q_i, \delta_i, I_i, F_i)$ is a multi-automaton accepting a (regular) set of configurations of thread T_i. To track lock interactions, we augment the configurations of each thread with their respective backward and forward acquisition histories and the LHI and LR fields as discussed in the last two subsections. Thus an augmented configuration of thread T_i is of the form $(\langle c, w\rangle, l_1, ..., l_m, BAH_1, ..., BAH_m, FAH_1, ..., FAH_m, LHI, LR)$, where BAH_i and FAH_i are used to track the backward and forward acquisition histories, respectively, of lock l_i. As in the case of a multi-automaton, we have an initial state of $\mathcal{A}_i$ corresponding to each configuration of T_i, and vice versa. Since in this case the configurations are augmented with FAHs and BAHs, each initial state of $\mathcal{A}_i$ is of the form $(\langle s_i, w\rangle, l_1, ..., l_m, BAH_1, ..., BAH_m, FAH_1, ..., FAH_m, LHI, LR)$, where $(\langle p_i, w\rangle, l_1, ..., l_m, BAH_1, ..., BAH_m, FAH_1, ..., FAH_m, LHI, LR)$ is an augmented configuration of T_i.

Motivated by theorems 2 and 5, we say that augmented configurations $s = (\langle c, w\rangle, l_1, ..., l_m, BAH_1, ..., BAH_m, FAH_1, ..., FAH_m, LHI, LR)$ and $t = (\langle c', w'\rangle, l'_1, ..., l'_m, BAH'_1, ..., BAH'_m, FAH'_1, ..., FAH'_m, LHI', LR')$ of T_1 and T_2, respectively, are *FAH-compatible* iff there do not exist locks l_i and l_j such that $l_i = T_1$, $l'_j = T_2$, $l_i \in FAH'_j$ and $l_j \in FAH_i$. Analogously, we say that s and t are *BAH-compatible* iff there do not exist locks l_i and l_j such that $l_i = T_1$, $l'_j = T_2$, $l_i \in BAH'_j$ and $l_j \in BAH_i$.

Definition 7 *Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be a $\mathcal{CP}$-*LMAP. *We say that $\mathcal{A}$ accepts global configuration $(\langle p_i, w\rangle, \langle q_j, v\rangle, l_1, ..., l_m)$ of $\mathcal{CP}$ iff there exist sets $FAH_1, ..., FAH_m, BAH_1, ..., BAH_m, LHI, LR, FAH'_1, ..., FAH'_m, BAH'_1, ..., BAH'_m, LHI', LR'$ such that if $\mathbf{s}_1 = (\langle p_i, w\rangle, l'_1, ..., l'_m, BAH_1, ..., BAH_m, FAH_1, ..., FAH_m, LHI, LR)$ and $\mathbf{s}_2 = (\langle q_j, v\rangle, l''_1, ..., l''_m, BAH'_1, ..., BAH'_m, FAH'_1, ..., FAH'_m, LHI', LR')$, where $l'_i = T_1$ if $l_i = T_1$ and $\perp$ otherwise and $l''_i = T_2$ if $l_i = T_2$ and $\perp$ otherwise, then*

1. *$\mathcal{A}_i$ accepts $\mathbf{s}_i$, and*
2. Lock-Set$(T_1, \mathbf{s}_1) \cap$ Lock-Set$(T_2, \mathbf{s}_2) = \emptyset$ *and* $LHI \cap LHI' = \emptyset$.
3. *$\mathbf{s}_1$ and $\mathbf{s}_2$ are BAH-compatible and FAH-compatible.*

Given a *$\mathcal{CP}$-LMAP $\mathcal{A}$*, we use $Conf(\mathcal{A})$ to denote the set of configurations of $\mathcal{CP}$ accepted by $\mathcal{A}$. A set of configurations C of $\mathcal{CP}$ is called *lock-constrained regular* if there exists a *$\mathcal{CP}$-LMAP $\mathcal{A}$* such that $C = Conf(\mathcal{A})$. For model checking LTL properties of concurrent programs interacting via nested locks we need two key properties of LMAPs
(i) closure under boolean operations, and
(ii) closure under pre^*-computation.

Closure of LMAPs under Boolean Operations. Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$, be given $\mathcal{CP}$-LMAPs and *op* a boolean operation. Then, broadly speaking, the closure of LMAPs under *op* follows from the facts that (1) $\mathcal{A}$ *op* $\mathcal{B} = (\mathcal{A}_1$ *op* $\mathcal{B}_1, \mathcal{A}_2$ *op* $\mathcal{B}_2)$, and (2) MAs are closed under boolean operations (see [2]). In this section, we focus only on the intersection operation which is what we require for the LTL model checking of threads, the rest of the operations being handled similarly.

Proposition 8 (Closure under Intersection). *Given $\mathcal{CP}$-*LMAP*s $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$, we can construct a $\mathcal{CP}$-*LMAP *accepting $Conf(\mathcal{A}) \cap Conf(\mathcal{B})$.*

Computing the pre*-closure of an LMAP. Let LC be a lock-constrained regular set accepted by a $\mathcal{CP}$-LMAP $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$. In this section, we show that we can efficiently, in polynomial time, construct a $\mathcal{CP}$-LMAP $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ accepting (1) $pre^*(LC)$ in case all locks are free in each configuration of LC, or (2) those configurations of $pre^*(LC)$ in which all locks are free.

The Procedure. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ are MAs accepting regular sets of configurations of the individual PDSs T_1 and T_2, respectively, we can construct, using the efficient techniques given in sections 3.1.1 and 3.1.2, multi-automata $\mathcal{B}_1$ and $\mathcal{B}_2$, accepting, respectively, the pre^*-closures, $pre^*_{T_1}(Conf(\mathcal{A}_1))$ and $pre^*_{T_2}(Conf(\mathcal{A}_2))$.

In the first case, since all locks are free in each configuration of LC, the forward acquisition history of each lock as well as the LHI and LR fields are $\emptyset$. Thus these fields do not come into play and so $\mathcal{B}_1$ and $\mathcal{B}_2$ can be computed using the procedure given in section 3.1.1, thereby giving us the following.

Proposition 9 *Let LC be a lock-constrained regular set of configurations of $\mathcal{CP}$ such that all locks are free in every configuration $\mathbf{c} \in LC$. If $\mathcal{A}$ is a $\mathcal{CP}$-LMAP accepting LC and if $\mathcal{B}$ is the $\mathcal{CP}$-LMAP constructed from $\mathcal{A}$ as above, then $Conf(\mathcal{B}) = pre^*(LC)$.*

In the second case, we are interested only in those configurations $\mathbf{c}$ of $pre^*(LC)$ in which all locks are free and due to which each BAH field of $\mathbf{c}$ is the empty set. Thus, in this case, the BAH fields are immaterial, and so $\mathcal{B}_1$ and $\mathcal{B}_2$ can be computed using the procedure in section 3.1.2.

Proposition 10 *If $\mathcal{A}$ is a $\mathcal{CP}$-LMAP accepting a lock-constrained regular set LC and if $\mathcal{B}$ is the $\mathcal{CP}$-LMAP constructed from $\mathcal{A}$ as above, then $Conf(\mathcal{B}) \cap LF = pre^*(LC) \cap LF$, where LF is the set of all configurations of $\mathcal{CP}$ in which all locks are free.*

Requirement of Nestedness of Locks. It is important to note that, in general, we can carry out the above pre^*-closure construction for LMAPs only for threads communicating via *nested* locks. If the locks are not nested then the problems of deciding reachability of a global configuration of $\mathcal{CP}$ to or from a configuration in which all locks are free are both undecidable as was shown in [5]. If, on the other hand, a concurrent program has nested locks, then from theorems 2 and 5 we can see that both these problems are not only decidable but efficiently so.

Complexity Analysis. Note that the computation of an LMAP accepting the pre^*-closure of given LMAP $\mathcal{A} =$

$(\mathcal{A}_1, \mathcal{A}_2)$ reduces to the computation of MAs $\mathcal{B}_i$ accepting the pre^*-closure of $Conf(\mathcal{A}_i)$ for each individual thread T_i, instead of the entire program $\mathcal{CP}$. From lemmas 3 and 6, $\mathcal{B}_i$ can be computed in time polynomial in the sizes of $\mathcal{A}_i$ and the control states of T_i and exponential in the number of locks of T_i. Thus we have the following

Theorem 11 (Efficient pre^*-closure computation) *Given a concurrent program $\mathcal{CP}$ comprised of threads T_1 and T_2 interacting via nested locks, and a $\mathcal{CP}$-LMAP $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, then in the two cases considered above, we can construct a $\mathcal{CP}$-LMAP $\mathcal{A}_{pre^*}$ recognizing $pre^*(Conf(\mathcal{A}))$ in time polynomial in the sizes of $\mathcal{A}_i$ and the control states of T_i and exponential in the number of locks of $\mathcal{CP}$.*

4 Dual Pumping

In this section, we formulate the *Dual Pumping Lemma* which allows to us leverage the efficient pre^*-closure computation procedure for LMAPs given above towards model checking threads for indexed LTL$\backslash$X formulae. Note that the use of stuttering-insensitive indexed logics is natural when reasoning about multi-threaded programs with interleaving semantics. This is because every global transition results from the firing of a local transition of a single thread thereby forcing other threads to stutter. Furthermore, we observe that the model checking problem for doubly-indexed LTL properties wherein atomic propositions are interpreted over the control states of two or more threads is undecidable for system comprised of multiple PDSs even when the PDSs do not interact with each other. Broadly speaking, this is due to the fact that such a formula can be used to restrict the model checking for control state reachability to those sets of computations of $\mathcal{CP}$ that simulate CCS-style pairwise rendezvous thereby yielding undecidability by reduction from the undecidability result of [1]. We now consider the model checking problem for single-index LTL$\backslash$X properties.

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 $f = \bigwedge_i \mathsf{E}f_i$, where f_i is an LTL$\backslash$X formula interpreted over the control states of T_i (Formulae of the form $f = \bigwedge_i \mathsf{A}f_i$ can be handled by taking their negation and then model checking for each i, the simpler formula $\mathsf{E}\neg f_i$). We use $\mathcal{BP}$ to denote the system resulting from the interleaved parallel composition of the *Büchi-augmented* threads $\mathcal{BP}_i$ formed by taking the product of T_i and $\mathcal{B}_{\neg f_i}$, the Büchi automaton corresponding to $\neg f_i$. We then say that a global configuration $\mathbf{c}$ of $\mathcal{BP}$ is an accepting global configuration of $\mathcal{BP}_i$ iff $\mathbf{c} \downarrow T_i$ is an accepting local configuration of the Büchi-augmented thread $\mathcal{BP}_i$. Then the model checking problem for f reduces to deciding whether there exists an *accepting path* of $\mathcal{BP}$, viz., a path along which for each i, an accepting state of $\mathcal{BP}_i$ occurs infinitely often.

Checking that $\mathcal{BP}$ has an accepting path is complicated by the fact that systems comprised of PDSs have infinitely many states in general. To overcome this problem, we now formulate the *Dual Pumping Lemma*, that 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. These properties enable us to construct a valid accepting sequence of $\mathcal{BP}$ by executing the sequence v repeatedly resulting in the pumping of each of the threads. However the lock interaction among the threads complicates the interleaved execution of the pumping sequences of the individual threads which therefore requires an intricate scheduling of their local transitions. To begin with, for ease of exposition we make the assumption that along all infinite runs of $\mathcal{BP}$ any lock that is acquired is eventually released. Later on we drop this restriction.

Theorem 12 (Dual Pumping Lemma) *$\mathcal{BP}$ has an accepting run starting from an initial configuration c if and only if there exist $\alpha \in \Gamma_1, \beta \in \Gamma_2$; $u \in \Gamma_1^*, v \in \Gamma_2^*$; accepting configurations g_i of $\mathcal{BP}_i$; 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. $(\langle p', u'\rangle, \langle q, \beta\rangle, l'_1, ..., l'_m)$
   $\Rightarrow lf_1 \Rightarrow g_1 \Rightarrow lf_2 \Rightarrow g_2 \Rightarrow lf_3$
   $\Rightarrow (\langle p, \alpha u''\rangle, \langle q'', v''\rangle, l_1, ..., l_m) \Rightarrow lf_4$
   $\Rightarrow (\langle p''', u'''\rangle, \langle q, \beta v'''\rangle, l'_1, ..., l'_m)$

The above result gives us the required witness with the special structure shown in fig 2. Here ρ, σ, ν are the sequences of global configurations realizing conditions 1, 2 and 3, respectively, in the statement of the theorem. We now show how to interleave the execution of the local pumping sequences of the two threads to construct a valid accepting path of $\mathcal{BP}$. Towards that end, we first define sequences of transitions spliced from ρ, σ and ν that we will concatenate appropriately to construct the accepting path.

- l_{11}: the local sequence of T_1 fired along σ.

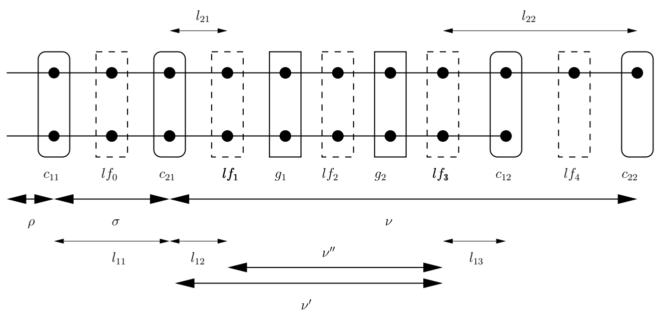

Figure 2. Pumpable Witness

- 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.
- l_{13}: the local sequence of T_1 fired along ν between lf_3 and $c_{12} = (\langle p, \alpha u''\rangle, \langle q'', v''\rangle, l_1, ..., l_m)$.
- 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.
- l_{22}: the local sequence of T_2 fired along ν between lf_3 and $c_{22} = (\langle p''', u'''\rangle, \langle q, \beta v'''\rangle, l_1, ..., l_m)$.
- ν': the sequence of global transitions fired along ν till lf_3.
- ν'': the sequence of global transitions fired along ν between lf_1 and lf_3.

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 local transitions occurring along the sequences $l_{13}l_{11}l_{12}$ followed by the local computation of T_1 along ν''. Similarly, T_2 is pumped by firing the local transitions occurring along the sequences $l_{22}l_{21}$ followed by the local computation of T_2 along ν''. Note that the pumping sequences of the two threads are staggered with respect to each other. The lock free configurations $lf_0, ..., lf_4$ are *breakpoints* that help in scheduling to ensure that π is a valid path. Indeed, starting at lf_3, we first let T_1 fire the local sequences l_{13}, l_{11} and l_{12}. This is valid as T_2, which currently does not hold any locks, 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 $lf_1 \downarrow T_1$ which is lock free. Thus T_2 can now fire the local sequences l_{22} and l_{21} to reach $lf_1 \downarrow T_2$ after which we let $\mathcal{CP}$ fire ν'' and then repeat the procedure.

To drop the lock restriction, we note that if there is a set of locks L' that are acquired along a run π of $\mathcal{BP}$ but never released then all operations on such locks are executed only along some finite prefix π_p of π which, wlog, we can assume to be a prefix of the stem ρ of the witness lollipop. Then the Dual Pumping Lemma is simply modified to add the extra proviso that there exists a set of locks L', all of which are held in c_{11} such that no operation on these locks is executed along σ and ν. Thus in the pre^*-closure computation to check for existence of σ and ν (see below), we do not execute transitions that operate on these locks.

Formulating Dual Pumping as Reachability Next we show how conditions 1, 2 and 3 in the statement of the Dual Pumping Lemma can be re-formulated as a set of reachability problems for regular sets of configurations. Let

$R_0 = pre^*(\{p\} \times \alpha\Gamma_1^* \times P_2 \times \Gamma_2^* \times \{(l_1, ..., l_m)\})$
Then condition 1 can be re-written as $c \in R_0$. Similarly, if
$R_1 = P_1 \times \Gamma_1^* \times \{q\} \times \beta\Gamma_2^* \times \{(l'_1, ..., l'_m)\}$
$R_2 = pre^*(R_1) \cap P_1 \times \Gamma_1^* \times P_2 \times \Gamma_2^* \times \{(\bot, ..., \bot)\}$.
$R_3 = pre^*(R_2) \cap \{p\} \times \{\alpha\} \times P_2 \times \Gamma_2^* \times \{(l_1, ..., l_m)\}$
then condition 2 can be captured as $R_3 \neq \emptyset$. Finally, let
$R_4 = P_1 \times \Gamma_1^* \times \{q\} \times \beta\Gamma_2^* \times \{(l'_1, ..., l'_m)\}$
$R_5 = pre^*(R_4) \cap P_1 \times \Gamma_1^* \times P_2 \times \Gamma_2^* \times \{(\bot, ..., \bot)\}$.
$R_6 = pre^*(R_5) \cap \{p\} \times \alpha\Gamma_1^* \times P_2 \times \Gamma_2^* \times \{(l_1, ..., l_m)\}$.
$R_7 = pre^*(R_6) \cap P_1 \times \Gamma_1^* \times P_2 \times \Gamma_2^* \times \{(\bot, ..., \bot)\}$.
$R_8 = pre^*(R_7) \cap G_2 \times L_1 \times ... \times L_m$, *where* $G_2 = \bigcup_{g_2}(P_1 \times \Gamma_1^* \times \{g_2\} \times \Gamma_2^*)$ *with* g_2 *being an accepting local state of* $\mathcal{BP}_2$.
$R_9 = pre^*(R_8) \cap P_1 \times \Gamma_1^* \times P_2 \times \Gamma_2^* \times \{(\bot, ..., \bot)\}$.
$R_{10} = pre^*(R_9) \cap G_1 \times L_1 \times ... \times L_m$, *where* $G_1 = \bigcup_{g_1}(\{g_1\} \times \Gamma_1^* \times P_2 \times \Gamma_2^*)$ *with* g_1 *being an accepting local state of* $\mathcal{BP}_1$.
$R_{11} = pre^+(R_{10}) \cap P_1 \times \Gamma_1^* \times P_2 \times \Gamma_2^* \times (\bot, ..., \bot)$.
$R_{12} = pre^*(R_{11}) \cap P_1 \times \Gamma_1^* \times \{q\} \times \{\beta\} \times \{l'_1, ..., l'_m)\}$.
Then condition 3 can be captured as $R_{12} \neq \emptyset$.

Complexity Analysis. To compute R_0, we start by constructing an LMAP $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ accepting the set $\{p\} \times \alpha\Gamma_1^* \times P_2 \times \Gamma_2^* \times \{(l_1, ..., l_m)\}$. Towards that end, we construct an MA $\mathcal{A}_1$ accepting $\{p\} \times \alpha\Gamma_1^* \times \{(l'_1, ..., l'_m, \emptyset, ..., \emptyset, LHI, \emptyset)\}$ and an MA $\mathcal{A}_2$ accepting $P_2 \times \Gamma_2^* \times \{(l''_1, ..., l''_m, \emptyset, ..., \emptyset, LHI'', \emptyset)\}$, where (i) $l'_i = T_1$ if $l_i = T_1$ and $\bot$ otherwise, and $l''_i = T_2$ if $l_i = T_2$ and $\bot$, otherwise, (ii) $LHI' = \{l_i | l_i = T_1\}$ and $LHI'' = \{l_i | l_i = T_2\}$, and (iii) the BAH and FAH entries are set to $\emptyset$. Note that $\mathcal{A}_i$ is polynomial in the size of thread T_i. Similarly, one can construct LMAPs $\mathcal{B}$ and $\mathcal{C}$ accepting R_1 and R_4, acting as starting points for computing the other sets listed above, in (i) polytime in the size control states of the individual threads, viz., $O(poly(|T_1|) + poly(|T_2|))$, with $|T_i|$ being the number of control states of T_i, and (ii) exponential time in the number of locks of $\mathcal{CP}$. Note that because of the witness structure, sets R_2, R_5, R_7, R_9 and R_{11} are comprised of configurations in which all locks are free. The first consequence is that by theorem 11, the pre^*-closure of these sets can be computed in time polynomial in the sizes of control states of the individual threads and exponential in the number of locks of $\mathcal{CP}$. The second consequence is that we need only compute the subset of those configurations of the pre^*-closures of R_1, R_4, R_6, R_8, and R_{10}, in which all locks are free. Again by theorem 11, these computations can be carried out in polytime in the sizes of the control states of the individual threads and exponential time in the number of locks of $\mathcal{CP}$. Note that we have to carry out the above pre^*-closure computations for every possible 4-tuple (α, β, p, q). Thus we have the following

Theorem 13 *The accepting run problem for a Büchi system for a concurrent system $\mathcal{CP}$ with PDSs interacting via nested locks can be solved in polynomial time in the sizes of control states of $\mathcal{CP}$ and exponential time in the number of locks of $\mathcal{CP}$.*

Note that even though the complexity of the model checking problem is exponential time in the number of locks it does not have a significant impact as the number of locks is usually small in practice. Thus we have:

Theorem 14 *The model checking problem for a fixed single-index LTL formula for a system $\mathcal{CP}$ comprised of multiple PDSs interacting via nested locks is decidable in time polynomial in the size of the control states of $\mathcal{CP}$ and exponential in the number of locks.*

Checking Nestedness of Locks. Let $T = (P, Act, \Gamma, \mathbf{c}_0, \Delta)$ be a thread of a concurrent program $\mathcal{CP}$ using locks $l_1, ..., l_m$. For testing whether T accesses locks in a nested fashion, all we need to do is to keep information regarding the order in which locks are accessed by T. Thus each state of the augmented thread T_a is now of the form $(c, \mathsf{l}_{i_1} ... \mathsf{l}_{i_k})$ where $\mathsf{l}_{i_1} ... \mathsf{l}_{i_k}$ is a sequence indicating the order in which locks were acquired by T with i_{i_k} being the most recent lock to be acquired. It is easy to see that for $j \neq l$, $\mathsf{l}_{i_j} \neq \mathsf{l}_{i_l}$. For each transition acquiring lock l and augmented state (c, λ) of T_a we concatenate l to λ. For any transition releasing lock l and augmented state (c, λ), we check to see whether l is the lock at the end of the sequence λ. If it is, then we remove l from λ, else we let T_a transit to a newly introduced control state Non-Nested. Then locks are nested in T iff the control state Non-Nested is not reachable in T_a which can be done efficiently using the model checking algorithm given in [2].

5 Conclusions and Related Work

Among prior work on the verification for concurrent programs, [6] attempts to generalize the techniques given in [2] to handle 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 [7].

Other related interesting work includes the use of tree automata [8] and logic programs [9] for model checking the processes algebra PA which allows modeling of nondeterminism, sequential and parallel composition and recursion. The reachability analysis of Constrained Dynamic Pushdown Networks which extend the PA framework by allowing PDSs that can spawn new PDSs to model fork operations, was considered in [10]. However, neither model allows communication among processes.

We, on the other hand, have identified a practically important case of threads communicating using locks and shown how to reason efficiently about a rich class of properties. Our methods are sound and complete, and cater to automatic error trace recovery. A key advantage of our method is that by reducing verification of a multi-threaded program to its individual threads, we bypass the state explosion problem, thereby guaranteeing scalability of our approach. Thus unlike existing methods our technique has *all* the desirable features of (i) being sound and complete, (ii) fully automatic, (iii) efficient, and (iv) catering to the verification of a rich class of linear temporal properties, not just reachability. Finally, our technique can easily be incorporated into current tools by exploiting existing efficient implementations for computing pre^*-closures.

References

[1] G. Ramalingam, "Context-Sensitive Synchronization-Sensitive Analysis is Undecidable," in *ACM Trans. Program. Lang. Syst.*, vol. 22(2), pp. 416–430, 2000.

[2] A. Bouajjani, J. Esparza, and O. Maler, "Reachability Analysis of Pushdown Automata: Application to Model-Checking," in *CONCUR*, LNCS 1243, pp. 135–150, 1997.

[3] I. Walukiewicz, "Model Checking CTL Properties of Pushdown Systems," in *FSTTCS*, LNCS 1974, 2000.

[4] "Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software," in *http://research.microsoft.com/ qadeer/cav-issta.htm*.

[5] V. Kahlon, F. Ivančić, and A. Gupta, "Reasoning about Threads Communicating via Locks," in *CAV*, 2005.

[6] A. Bouajjani, J. Esparza, and T. Touili, "A Generic Approach to the Static Analysis of Concurrent Programs with Procedures," in *IJFCS*, vol. 14(4), pp. 551–, 2003.

[7] S. Qadeer and J. Rehof, "Context-Bounded Model Checking of Concurrent Software," in *TACAS*, 2005.

[8] D. Lugiez and P. Schnoebelen, "The Regular Viewpoint on PA-Processes," in *Theor. Comput. Sci.*, vol. 274(1-2), 2002.

[9] J. Esparza and A. Podelski, "Efficient Algorithms for pre^* and $post^*$ on Interprocedural Parallel Flow Graphs," in *POPL*, 2000.

[10] A. Bouajjani, M. Olm, and T. Touili, "Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems," in *CONCUR*, 2005.