Boundedness vs. Unboundedness of Lock Chains: Characterizing Decidability of CFL-Reachability for Threads Communicating via Locks

Vineet Kahlon
NEC Labs America, Princeton, NJ 08540, USA.

Abstract

The problem of Pairwise CFL-reachability is to decide whether two given program locations in different threads are simultaneously reachable in the presence of recursion in threads and scheduling constraints imposed by synchronization primitives. Pairwise CFL-reachability is the core problem underlying concurrent program analysis especially 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 pairwise reachability is known to 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 and device drivers. In this paper, we exploit the fact that lock usage patterns in real life programs do not produce unbounded lock chains. For such programs, we show that pairwise CFL-reachability becomes decidable. Towards that end, we formulate small model properties that bound the lengths of paths that need to be traversed in order to reach a given pair of control states. Our new results narrow the decidability gap for pairwise CFL-reachability 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

Dataflow analysis is an indispensable technique for analyzing large scale sequential programs. For concurrent programs, however, it has proven to be an undecidable problem [12]. This has created a huge gap in terms of the techniques required to meaningfully analyze concurrent programs (which must satisfy the two key criteria of achieving *precision* while ensuring *scalability*) and what the current state-of-the-art offers.

The key obstacle in the dataflow analysis of concurrent programs is to determine for a control location l in a given thread, how the other threads could affect dataflow facts at l. Equivalently, one may view this problem as one of delineating *transactions*, i.e., sections of code that can be executed atomically, based on the dataflow analysis being carried out. The various possible interleavings of these atomic sections then determines interference across threads. The challenge in analyzing multi-threaded programs, therefore, lies in delineating transactions accurately, automatically and efficiently in the presence of scheduling constraints imposed by synchronization primitives and shared variables. It has been shown [6] that the effects of shared variables on transaction delineation can be captured via the use of sound numerical invariants like ranges, octagons and polyhedra.This reduces the problem of transaction delineation for threads interacting via shared variables and synchronization primitives to threads interacting purely using synchronization primitives such as locks and wait/notify.

The problem of synchronization-based transaction delineation is intimately connected with the problem of *pairwise CFL-reachability* which is to decide whether two given program locations c_1 and c_2 in threads T_1 and T_2, respectively, are simultaneously reachable in the presence of recursion in threads and scheduling constraints imposed by synchronization primitives. Indeed, in a global state g, a context switch is required at location l of thread T where a shared variable sh is accessed only if starting at g, some other thread currently at location m can reach another location m' with an access to sh that conflicts with l, i.e., l and m' are pairwise CFL-reachable from l and m. In that case, we need to consider both interleavings wherein either l or m' is executed first thereby requiring a context switch at l. Thus the fundamental problem underlying dataflow analysis of concurrent programs is to decide pairwise CFL-reachability for threads with recursive procedures that communicate using standard synchronization primitives like locks, wait/notifies and broadcasts. In this paper, we focus on pairwise CFL-reachability for threads interacting via the most commonly used primitive, i.e., mutex locks.

Pairwise CFL-reachability has been shown to be decidable for threads interacting via nested locks [8]. 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 required 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. Another classic example where non-nested locks occur frequently are programs that use both mutexes and Wait/Notify statements. In Java and the Pthreads Library, Wait/Notify statements require the use of mutexes on the objects being waited on. These mutexes typically interact with existing locks to produce non-nesting. Current techniques cannot be used to reason about such non-nested locks.

In this paper, we show that pairwise reachability is decidable not only for threads interacting via nested locks but also non-nested locks with bounded lock chains. These lock usage patterns cover almost all cases of practical interest encountered in real-life programs. Note that we don't insist that all lock usage should should be in the form of bounded chains. All we require is that if lock chains exist then they are bounded in length.

In order to show decidability, we formulate a small model property for pairwise CFL-reachability that bounds the lengths of paths that need to be traversed in order to reach a given pair of control states (c_1, c_2). Apart from being of theoretical interest, small model properties are particularly desirable as they allow us to reduce pairwise reachability for threads, even those with recursive procedures, to model checking a *finite* state system. This enables us to exploit the use of powerful symbolic techniques that have been developed for exploring finite state systems and which do not extend easily to recursive programs which typically have infinitely many states.

Importantly, our techniques also narrow the known decidability/undecidability divide for pairwise CFL-reachability. 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 pairwise CFL-reachability in threads.

To sum up, the specific contributions of the paper are:

1. extending the decidability envelope for pairwise CFL-reachability to concurrent programs with non-nested lock usage in which every lock chain is bounded.

2. small model properties for pairwise CFL-reachability that reduce the problem of pairwise CFL-reachability to model checking a finite state system thereby enabling us to leverage powerful state space exploration techniques.

3. providing a more refined characterization of decidability of pairwise CFL-reachability in terms of boundedness of lock chains as opposed to nestedness of locks.

2 System Model

We consider concurrent programs comprised of threads modeled as Pushdown Systems (PDSs) [2] that interact with each other using synchronization primitives. PDSs are a natural model for abstractly interpreted programs used in key applications like dataflow analysis [9]. 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 $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', w\rangle$. 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 $\overset{a}{\rightarrow} \subseteq \mathcal{C} \times \mathcal{C}$ as follows: if $\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', w\rangle$, then $\langle q, \gamma v\rangle \overset{a}{\rightarrow} \langle q', wv\rangle$ 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 $\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', w\rangle$ of P. A sequence $x = x_0, x_1, ...$ of configurations of P is a *computation* if x_0 is the initial configuration of P and for each i, $x_i \overset{a}{\rightarrow} x_{i+1}$, where $a \in Act$.

We model a concurrent program with n threads and m locks $\mathsf{l}_1, ..., \mathsf{l}_m$ 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, $L_i \subseteq \{\perp, 1, ..., n\}$ 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 $l_i = \perp$, else l_i is the index of the thread currently holding l_i. The initial global configuration of $\mathcal{CP}$ is $(c_1, ..., c_n, \perp, ..., \perp)$, 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 $T_1, ..., T_n$.

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 $a \in Act$ or for some $1 \leq j \leq m$, $a = release(\mathsf{l}_j)$ or $a = acquire(\mathsf{l}_j)$.

Nested PDSs. Abstractly interpreted programs can be translated to PDSs in a natural fashion by letting the control state of the PDS track dataflow facts and letting the stack track the current location of the program counter and the calling context (sequence of function calls) (cf. [9]). Towards that end, each intra-procedural transition *tr* of the given program from locations l_1 to l_2 is modeled via a PDS transition of the form $\langle f_1, l_1 \rangle \hookrightarrow \langle f_2, l_2 \rangle$, where f_1 and f_2 are the dataflow facts before and after the execution of *tr*. On the other hand, a call to function h from location l_{call} of function g is modeled via a transition of the form $\langle f, l_{call} \rangle \hookrightarrow \langle f, l_{return} h_{entry} \rangle$, where l_{return} is the return location in g for the call to h, h_{entry} is the entry location of h and f is a dataflow fact (which is unchanged during the function call). Finally, a return of function h from location l is modeled via a transition of the form $\langle f, l \rangle \hookrightarrow \langle f, \epsilon \rangle$ which pops the top symbol from the stack.

We observe that intra-procedural thread operations are modeled via PDS transitions that change the symbol at the top of the stack without affecting its height which can be changed only via transitions modeling function calls (increase height by one) and returns (decrease height by one). Transitions modeling function calls and returns are referred to as *stack push* and *stack pop* transitions of the resulting PDS P, respectively. Collectively both types of transitions are referred to as *stack* transitions. Each push stack transition modeling a function call *fc* is designated an fc_{push} transition and any pop stack transition modeling a return of *fc* is designated an fc_{pop} transition. Transitions of the PDS modeling intra-procedural transitions of the sequential program are labeled as *non-stack* transitions as they do not modify the height of the stack. By abuse of notation, we also refer to transitions $u \rightarrow v$, where u and v are configurations of P, resulting from the firing of push and pop stack transitions of P, as push and pop stack transitions, respectively.

In a sequential program, function calls and returns are nested. This ensures that along each computation of a PDS modeling a sequential program there exists a one-to-one mapping from each stack transition popping an exit location of function f, say, to the last stack push transition that increased the height of the stack to its current value by pushing the entry location f_{entry} of f and which has not yet been popped. We call such a PDS *nested*. This motivates the following simple definitions.

Definition (Matching Call). *Given a computation* $x = x_0 ... x_n$ *of a nested PDS* P*, we say that* fc_{push} *and* fc_{pop} *transitions* $tr_i : x_i \rightarrow x_{i+1}$ *and* $tr_j : x_j \rightarrow x_{j+1}$*, respectively, fired along* x *are matching if* $j > i$ *and an equal number of push and pop transitions are fired along the subsequence* $x_{i+1} ... x_j$*. If along* x *a matching pop transition does not exist for a push transition* tr_k *then we denote its matching pop transition by* tr_∞.

Given a computation x, the sub-sequence of transitions fired along x between a pair of matching push and pop transitions tr_i and $tr_{i'}$, respectively, is called a *function call* of the given PDS P and is denoted by $(tr_i, tr_{i'})$. If tr_i and $tr_{i'}$ result from the firing of stack transitions fc_{push} and fc_{pop}, respectively, of P, then $(tr_i, tr_{i'})$ is called an execution of the call-return pair (fc_{push}, fc_{pop}) of P.

Definition (Nested Calls). *Let* $(tr_i, tr_{i'})$ *and* $(tr_j, tr_{j'})$ *be function calls executed along a computation* x *of a nested PDS. We say that* $(tr_j, tr_{j'})$ *is nested within* $(tr_i, tr_{i'})$ *if* $i < j$ *and either* $i' = \infty = j'$ *or* $j' < i'$.

Definition (Non-Nested Calls). *A function call* $(tr_i, tr_{i'})$ *executed along a computation* x *of a nested PDS is said to be non-nested if there does not exist another function call* $(tr_j, tr_{j'})$ *executed along* x *such that* $(tr_i, tr_{i'})$ *is nested within* $(tr_j, tr_{j'})$.

3 Paper Outline

Our strategy for showing a small model property for pairwise CFL-reachability in concurrent programs is as follows:

1. We first show a *sequential small model* property (sec. 4) for CFL-reachability of a control state in a sequential program, i.e., in an individual thread.

2. We then leverage this sequential small model property to show a small model property for pairwise reachability of control locations c_1 and c_2 in threads T_1 and T_2, respectively, of the given concurrent program $\mathcal{CP}$. Let c_1 and c_2 be pairwise reachable via a global computation x of $\mathcal{CP}$. If x^i is the local computation of T_i along x then using the sequential small model property we can construct a small model y^i from x^i leading to c_i. However, naively applying the sequential small model property does not guarantee that the resulting y^is can be interleaved to form a valid global computation of $\mathcal{CP}$ leading to (c_1, c_2). In order to ensure that, we leverage the use of a lock causality graph [6]. With a pair of local computations x^1 and x^2 of threads T_1 and T_2, leading to control locations c_1 and c_2, respectively, one can associate a lock causality graph $G_{(x_1,x_2)}$ having the useful property that (c_1, c_2) are pairwise reachable via a global computation resulting from an interleaving of x^1 and x^2 if and only if $G_{(x^1,x^2)}$ is acyclic. Our strategy is to exploit the sequential small model property to produce a small model y^i for x^i while ensuring that $G_{(y^1,y^2)}$ is acyclic.

4 Sequential Small Model Property

In order to exhibit a small model property for control state CFL-reachability in sequential programs, we leverage the *horizontal* and *vertical* bounding lemmas. Intuitively,

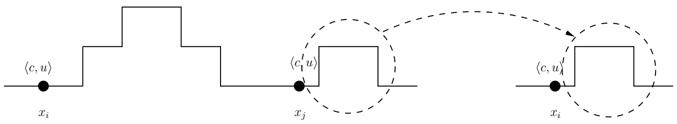

Figure 1. Horizontal Reduction

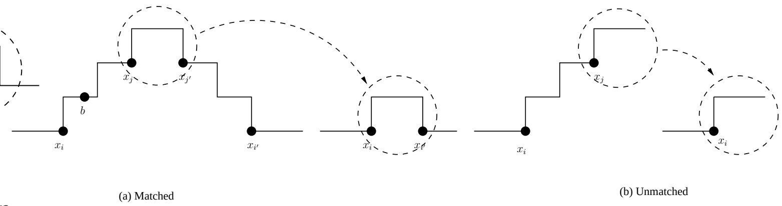

Figure 2. Vertical Reduction

the horizontal bounding lemma limits the number of non-nested function calls that need be fired along a computation in order to reach c. However, the horizontal bounding lemma does not limit the number of function calls that could be nested within these non-nested calls. Bounding the call depth of functions is accomplished via the vertical bounding lemma. Combining these two lemmas then enables us to limit the total length of a computation path required to reach a control state c.

Horizontal Reduction. The key idea behind the horizontal reduction lemma is captured in a path transformation that we refer to as horizontal reduction. Intuitively, if the same configuration is repeated along a computation x of PDS P as, say, x_i and x_j, then we can short-circuit the sub-computation from x_i to x_j (see fig. 1). Formally, let $x = x_0...x_n$ be a computation sequence of P and let y be a computation of P that is also a subsequence of x then we say that y is gotten from x via a horizontal reduction if there exist configurations x_i and x_j, where $i < j$, such that $x_i = x_j$, i.e., both the control state and the stack content are the same, and $y = x_0...x_i x_{j+1}...x_n$.

Let $tr_i : x_i \rightarrow x_{i+1}$ be the first push transition fired along x and let $tr_{i'} : x_{i'} \rightarrow x_{i'+1}$ be its matching pop transition. Furthermore, let $x_j \rightarrow x_{j+1}$ be the first push transition occurring after $x_{i'}$ along x and $x_{j'} \rightarrow x_{j'+1}$ it matching pop transition. Note that, by definition of j, no stack transition can be fired along the sub-sequence $x_{i'+1}...x_j$. Continuing in this fashion, we see that x can be parsed as $x = L_0 N_0 ... L_k N_k L_{k+1}$, where L_i is a (possibly empty) sequence of non-stack transitions and N_i is a sequence resulting from the execution of a non-nested function call of P. Note that all configurations occurring along L_i have the same stack content except possibly the top symbol in the stack. Moreover, we can, by repeatedly applying horizontal reduction, ensure that each configuration need occur at most once along x. Thus we see that for each control state d and stack alphabet $a \in \Gamma$, there occurs at most one configuration along all L_is in control state d and having a at the top of its stack. Hence, $\sum_i |L_i| \leq |Q||\Gamma|$, where $|Q|$ is the number of control states and $|\Gamma|$ the size of the stack alphabet of P.

Moreover, since all function calls executing the same call-return pair of P must occur from configurations with the same control state, and since, as discussed above, there can occur at most $|\Gamma|$ configurations in a given control state along all L_is, there exists at most $|\Gamma|$ non-nested function call executions of each call-return pair along y. Formally,

Proposition 1 (Horizontal Bounding Lemma). *Let x be a finite computation of a nested PDS P leading to control state c. Then there exists a computation y of P leading to c such that (i) y is a sub-sequence of x, and (ii) y can be written as $y = L_0 N_0 ... L_k N_k L_{k+1}$, where (a) L_i is a (possibly empty) sequence of non-stack transitions, (b) N_i is a sequence resulting from the execution of a non-nested function call, and (c) $\sum_i |L_i| \leq |Q||\Gamma|$ and $k \leq |\Gamma||F|$, where $|F|$ is the number of stack push transitions of P.*

Vertical Reduction. The key idea behind vertical reduction is that if there are two executions say $fc^{out} : (x_i, x_{i'})$ and $fc^{in} : (x_j, x_{j'})$ of the same call-return pair of P, with fc^{in} nested within fc^{out}, then we need only execute the inner one (see fig. 2). To see why the resulting computation is valid, let $(x_k, x_{k'})$ be the execution of a function call that returns, i.e., $k' < \infty$. Because the given PDS P is nested, any stack push transition fired along $(x_k, x_{k'})$ is matched by a pop transition fired along $(x_k, x_{k'})$. Moreover the execution of a stack push transition increases the height of the stack by one whereas the execution of a pop transition decreases it by one. Thus for each transition fired along $(x_k, x_{k'})$ that increases the height of the stack there is a matching transition fired along $(x_k, x_{k'})$ that decreases it by exactly the same amount. As a result, the execution of $(x_k, x_{k'})$ leaves the contents of the stack unchanged. Thus if fc^{in} and fc^{out} both return, we have that both fc^{in} and fc^{out} leave the stack unchanged. Since fc^{in} and fc^{out} are executions of the same call-return pair, they start and end at the same control location. Using the above two facts, we have that if starting at x_i we execute fc^{in} instead of fc^{out} we end up in the same configuration, i.e., $x_{i'+1}$. Finally, because the given PDS is nested, we have that all function calls executed during fc^{in} do not depend on any transition executed before fc^{in} started executing. Thus fc^{in} can indeed be executed starting at x_i.

Note that the above argument assumed that both the calls fc^{out} and fc^{in} return (see fig. 2 (a)). A similar argument applies to the case when both these calls are unmatched $i' = \infty = j'$ (see fig. 2 (b)). In that case, starting at x_i we execute the sub-sequence $x_j..x_n$ instead of the sub-sequence $x_i...x_n$. Thus we have

Proposition 2. (Vertical Bounding Lemma). *Let x be a finite computation of a nested PDS P leading to control state c. Then there exists a computation y of P leading to c such that (i) y is a subsequence of x, and (ii) along y there do* *not exist two function call executions fc$_1$: $(tr_i, tr_{i'})$ and fc$_2$: $(tr_j, tr_{j'})$ of the same call-return pair of P with fc$_1$ nested within fc$_2$ unless $i' < \infty$ and $j' = \infty$.*

Since a non-returning (unmatched) function call execution of a call-return pair cannot be nested within a returning execution of the same pair, we have the following corollary.

Corollary 3. (Depth-2 Bounding). *Let x be a computation of minimum length leading to control location c. If fc$_0$,..., fc$_k$ is a sequence of function call executions of the same call-return pair along x such that for each i, fc$_i$ is nested within fc$_{i+1}$, then $k \leq 1$.*

Sequential Small Model Property. By leveraging the horizontal and vertical reduction lemmas, we can show the desired small model property for control state reachability in a nested PDS. Let x be a computation leading to control location c. As described above, we start by parsing x as $x = L_0 N_0 ... L_k N_k L_{k+1}$, where L_i is a (possibly empty) sequence of non-stack transitions and N_i is a sequence resulting from the execution of a non-nested function call. From the horizontal bounding lemma, we have that $\sum_i |L_i| \leq |Q||\Gamma|$ and $k \leq |F||\Gamma|$. Thus $|x| = \sum_i |L_i| + \sum_i |N_i| \leq |Q||\Gamma| + \sum_i |N_i|$.

To formulate the small model property, we have to bound $\sum_i |N_i|$ for which we leverage the vertical bounding lemma. In order to bound the length of N_i we can, as above, parse each $N_i = x_{i0} ... x_{ij_i}$ as $N_i = L_{i0} N_{i0} ... L_{in_i} N_{in_i} L_{i(n_i+1)}$, where L_{ij} is a maximal subsequence of N_i without a stack transition and N_{ij} are segments resulting from the firing of a non-nested function call along the subsequence $N'_i = x_{i0+1} ... x_{ij_i - 1}$. Repeating the above procedure, we recursively break down each non-nested call N_{in_p}, where $p \in [0..n_i]$, into smaller non-nested calls till we end up with subsequences of x without any stack transitions. We can then construct a tree T_x with these nested calls as nodes. Let N be a non-nested call encountered in our procedure. If N is parsed as $N = L_0 N_0 ... L_k N_k L_{k+1}$ then the children of N in T_x are $N_0, ..., N_k$. Note that each N_i is nested within N. Thus each path in T_x starting at the root is comprised of a series of function calls such that each call is nested within each of its ancestors. A key observation is that from the depth-2 bounding result, it follows that along any path of T_x there cannot exist more than two nodes representing executions of the same call-return pair of P. Thus the length of each path in T_x is at most d, where d is the maximum *call depth* of the given PDS P, i.e., the maximum stack depth of any reachable configuration of P under the assumption that each function call is executed at most twice. Clearly $d \leq 2|F|$, where $|F|$ is the number of stack push transitions of P.

With the node n of T_x corresponding to a non-nested segment $N = L_0 N_0 \ ... \ L_k N_k L_{k+1}$ encountered in our recursive procedure, we associate the set of non-stack transitions occurring along the sequences $L_0, ..., L_{k+1}$, and the weight $\sum_j |L_j|$ (the number of non-stack transitions fired along N). Recall that starting at N, our recursive procedure terminates when we end up with a sequence N' that has no stack transitions executed along it. At that point each transition in N will have been accounted for as a non-stack transition associated either with node n or one of its descendants. Thus each transition of x is associated with exactly one node of T_x and so the length of x is bounded by the sum of weights of all nodes in T_x. As discussed above, $\sum_j |L_j| \leq |Q||\Gamma|$, i.e., the weight of each node is at most $|Q||\Gamma|$. Thus $|x| \leq |Q||\Gamma|(|T_x|)$, where $|T_x|$ is the number of nodes in T_x. By the horizontal bounding lemma $k \leq |F||\Gamma|$, i.e., the out-degree of each node is at most $|F||\Gamma|$. Let d be the maximum length of a path in T_x. Then the total number of nodes in T_x is bounded by $1 + (|\Gamma||F|) + (|\Gamma||F|)^2 + ... + (|\Gamma||F|)^{d-1} \leq (|\Gamma||F|)^d$. Thus the total length of x is at most $|Q||\Gamma|(|\Gamma||F|)^d$, where $d \leq 2|F|$ is the maximum call depth of P.

Theorem 4. (Sequential Small Model Property). *Let P be nested pushdown system and let c be a reachable control state of P. Then there exists a path y of P leading to c of length at most $|Q||\Gamma|(|\Gamma||F|)^d$, where $|Q|$ is the number of control states of P, $|\Gamma|$ is the size of the stack alphabet of P, $|F|$ is the number of stack push transitions of P and $d \leq 2|F|$ is the maximum call depth in P.*

Generalized Sequential Small Model Property. The small model result allows us to bound the length of a path from the initial state of a PDS to a given control state c. For some applications, we are interested in constructing a smaller model y from x, while preserving not only the initial and final control states but also a given set of intermediate control states occurring along x. Formally, let $x_{i_0} = \langle c_{i_0}, u_{i_o} \rangle, ..., x_{i_l} = \langle c_{i_l}, u_{i_l} \rangle$, where $i_0 < ... < i_l = n$, be the configurations occurring along $x = x_0 ... x_n$ whose control states need to be preserved. Our goal is to bound the length of a computation y having configurations $y_{j_0} = \langle c_{i_0}, v_{j_0} \rangle, ..., y_{j_l} = \langle c_{i_l}, v_{j_l} \rangle$, where $j_0 < ... < j_l$, that preserve the control states of $x_{i_0}, ..., x_{i_l}$, respectively. Note that we require that only the control states be preserved and not the stack contents.

We start with the case when $l = 2$, i.e., we need to preserve the control state of only one intermediate configuration, say $x_i = \langle c, u \rangle$. If we naively apply horizontal and vertical reductions, then we might delete the configuration x_i whose control state we want to preserve. In order to avoid deletion of the control state of x_i, we apply the reductions individually to the subsequences $x^1 = x_0 ... x_i$ and $x^2 = x_i ... x_n$ of x. Applying the horizontal reduction presents no problems. However, in applying the vertical reduction we have to be careful about functions calls *fc* spanning x_i, viz., those that start executing before x_i but finish after x_i along x. If $fc : (x_j, x_{j'})$ is function call spanning x_i then we have to ensure that in applying the vertical reduction either both its calling and returning configurations, i.e., x_j and $x_{j'}$ respectively, are preserved or both are deleted. Additionally, there could be an unbounded number of function calls that span $x_i = \langle c, u\rangle$, i.e., u could be of arbitrary depth. In order to produce a small model y for x, we start by limiting the depth of u. Using the vertical reduction result, we see, as before, that if there are two nested function call executions of the same call-return pair spanning x_i, then we need execute only the inner one. Thus there can be at most two executions of the same call-return pair spanning x_i. In other words, the depth of u need be at most d, where d is the maximum call depth of P. Note that executing only the inner call ensures that the control state of x_i is preserved. However, there will be a reduction in its stack depth as some of the spanning function calls will not be executed.

Let $(x_{i_1}, x_{j_1}), ..., (x_{i_m}, x_{j_m})$, where $i_1 < ... < i_m < i$ and $m \leq d$ be the function calls spanning x_i. Suppose that k is the maximum index h such that $j_h < \infty$. These call and return points decompose the path x into segments $s^1 = x_0...x_{i_1}$, $s^2 = x_{i_1}...x_{i_2}$,...,$s^{m+1} = x_{i_m}...x_i$, $s^{m+2} = x_i...x_{j_1}$,..., $s^{m+k+2} = x_{j_k}...x_n$. All we need to do now is apply the sequential small model property to each of these segments and then concatenate the resulting segments to get the desired small model. Applying the small model property to each of the segments instead of the entire computation ensures that x_i and none of the spanning function calls are truncated. Since there are at most $2d + 2$ segments, and since by the sequential small model result the length of each segment is at most $|Q||\Gamma|(|\Gamma||F|)^d$, the total length of the resulting small model is at most $(m + k + 2)|Q||\Gamma|(|\Gamma||F|)^d \leq (2d + 2)|Q||\Gamma|(|\Gamma||F|)^d$.

Now suppose that we need to preserve the control states of a set C of configurations occurring along x instead of just one. Then using an approach similar to the above (see full-paper for details [1]), we can show the following

Theorem 5 (Generalized Small Model) *Let $x = x_0...x_n$ be a computation of a nested PDS P and let $x_{i_0} = \langle c_{i_0}, u_{i_0}\rangle$, ..., $x_{i_{k-1}} = \langle c_{i_{k-1}}, u_{i_{k-1}}\rangle$, where $i_0 < ... < i_{k-1} = n$, be configurations along x. Then there exists a computation $y = y_0...y_m$ of P of length at most $(2dklog(k) + 2)(|Q||\Gamma|(|\Gamma||F|)^d)$ having configurations $y_{j_0} = \langle c_{i_0}, v_{j_0}\rangle$, ..., $y_{j_{k-1}} = \langle c_{i_{k-1}}, v_{j_{k-1}}\rangle$, where $j_0 < ... < j_{k-1} = m$.*

5 Concurrent Small Model Property

Pairwise CFL-reachability is undecidable for two threads interacting purely via locks but decidable if the locks are nested [7]. Non-nested usage of locks can be characterized in terms of lock chains.

Definition (Lock Chains). *Given a computation x of a concurrent program $\mathcal{CP}$, a lock chain of thread T of $\mathcal{CP}$ 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.*

Algorithm 1 Computing the Lock Causality Graph

Input: Local paths x^1 and x^2 of T_1 and T_2 leading to c_1 and c_2, respectively.
Initialize the vertices and edges of $G_{(x^1,x^2)}$ to $\emptyset$
for each lock l held at location c_i **do**
  if c and c' are the last statements to acquire and release l occurring along x^i and $x^{i'}$, respectively. **then**
    Add edge $c' \rightsquigarrow c$ to $G_{(x^1,x^2)}$.
  end if
end for
repeat
  for each lock l **do**
    for 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 $a_{i'}$
      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
  end for
until no new statements can be added to $G_{(x^1,x^2)}$
for $i \in [1..2]$ **do**
  Add edges among all statements of x^i occurring in $G_{(x^1,x^2)}$ to preserve their relative ordering along x^i
end for

Let x^1 and x^2 be local computations of threads T_1 and T_2 leading to local control states c_1 and c_2, respectively. In deducing pairwise reachability of c_1 and c_2, we need to check whether there exists a valid global computation resulting from an interleaving of x^1 and x^2 leading to (c_1, c_2). The lock causality graph, which we review below, has been proposed [6] to answer precisely this question.

5.1 Lock Causality Graph

We motivate the concept of a lock causality graph via the example concurrent program $\mathcal{CP}$ comprised of threads T_1 and T_2 shown in fig 3. Suppose that we are interested in deciding whether *a6* and *b8* are simultaneously reachable. For that to happen there must exist local paths x^1 and x^2 of T_1 and T_2 leading to *a6* and *b8*, respectively, along which locks can be acquired in a consistent fashion. We start by

```
void T_1(){
a1:   lock(l_1);
a2:   lock(l_2);
a3:   lock(l_6);
a4:   unlock(l_1);
a5:   unlock(l_2);
a6:   Race_0;
}
```

```
void T_2(void){
b1:   lock(l_6);
b2:   lock(l_2);
b3:   unlock(l_6);
b4:   lock(l_5);
b5:   unlock(l_5);
b6:   lock(l_1);
b7:   unlock(l_2);
b8:   Race_1;
}
```

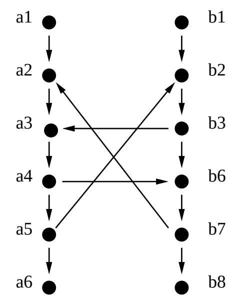

Figure 3. An Example Program and its Lock Causality Graph

constructing a *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 *a6* and *b8*. 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 *a6* and *b8*.

Causality Constraints:

(a) Consider lock l_1 held at *b8*. Note that once T_2 acquires l_1 at location *b6*, it is not released along the path from *b6* to *b8*. Since we are interested in the pairwise CFL-reachability of $a6$ and $b8$, T_2 cannot progress beyond location *b8* and therefore cannot release l_1. Thus we have that once T_2 acquires l_1 at *b6*, T_1 cannot acquire it thereafter. If T_1 and T_2 are to simultaneously reach *a6* and *b8*, the last transition of T_1 that releases l_1 before reaching *a6*, i.e., *a4*, must be executed before *b6*. Thus $a4 \rightsquigarrow b6$.

(b) Causal constraints can be deduced in another way. Consider the constraint *a4* $\rightsquigarrow$ *b6*. At location *b6*, lock l_2 is held which was acquired at *b2*. Also, once l_2 is acquired at *b2* it is not released till after T_2 exits *b6*. Thus if l_2 has been acquired by T_1 before reaching *a4* it must be released before *b2* (and hence *b6*) can be executed. In our example, the last statement to acquire l_2 before *a4* is *a2*. The unlock statement corresponding to *a2* is *a5*. Thus, *a5* $\rightsquigarrow$ *b2*.

Computing the Lock Causality Graph. Given finite local paths x^1 and x^2 of threads T_1 and T_2 leading to control locations c_1 and c_2, respectively, the procedure (see alg. 1) to compute $G_{(x^1,x^2)}$, adds the causality constraints one-by-one (of type (a) via steps 3-7, and of type (b) via steps 9-19) 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 14 in the algorithm ensures that we do not add edges representing causality constraints that can be deduced from existing edges. Also, steps $21-23$, preserve the local causality constraints along x^1 and x^2. The causality graph $G_{(x^1,x^2)}$ for paths $x^1 = a1, ..., a6$ and $x^2 = b1, ..., b8$ is shown in figure 3.

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.

Theorem 6. (Acyclicity) [6]. *Locations c_1 and c_2 are pairwise CFL-reachable if there exist local paths x^1 and x^2 of T_1 and T_2, respectively, leading to c_1 and c_2 such that $G_{(x^1,x^2)}$ is acyclic.*

5.2 Small Model Property

We now show a small model property for pairwise CFL-reachability by combining the lock causality graph acyclicity condition and the sequential small model property. Let (c_1, c_2) be pairwise reachable and let x be a global computation of $\mathcal{CP}$ leading to (c_1, c_2). Our goal is to construct a smaller model y^i from x^i, while ensuring that $G_{(y^1,y^2)}$ is acyclic if and only if $G_{(x^1,x^2)}$ is acyclic. Towards that end, we apply the generalized sequential small model property in a manner that preserves the lock causality graph, i.e., $G_{(x^1,x^2)} = G_{(y^1,y^2)}$. Specifically, we use the generalized small model property to construct a small model y^i from x^i by preserving the control states of all configurations of x^i that occur in $G_{(x^1,x^2)}$. If we can obtain a bound on the number of configurations of x^i occurring in $G_{(x^1,x^2)}$ then it yields a bound on the number of intermediate configurations of x^i whose control states need to be preserved. Then using the generalized small model property we immediately get bounds on the lengths of y^i and, as a result the desired small model for pairwise reachability of (c_1, c_2).

Bounding the Size of the Lock Causality Graph. In order to bound the number of edges in $G_{(x^1,x^2)}$ it suffices to focus only on the *cross edges*, i.e., those occurring between statements of different threads. We start by observing that cross edges in $G_{(x^1,x^2)}$ are of two types.

(i) **Seed Edges**, i.e., those added via steps 3-7 of algorithm 1. Each seed edge of $G_{(x^1,x^2)}$ occurs from the last statement releasing a lock l along x^1 (x^2) to the last statement acquiring it along x^2 (x^1). Clearly, there are at most

$|L|$ seed edges, where $|L|$ is the number of locks in $\mathcal{CP}$.

(ii) **Induced Edges**, i.e., those added via steps 9-19 of algorithm 1. With any edge $e' : f' \rightsquigarrow g'$ that is added in an iteration of steps 9-19, we can associate an existing (seed or induced) edge $e : f \rightsquigarrow g$ that induces e' (in the terminology of alg. 1, $d_{i'} \rightsquigarrow d_i$ induces $r_{i'} \rightsquigarrow a_i$). Thus if we track the induced-by relation backwards starting at e' we will ultimately end up with a seed edge. As a result, for each cross edge e' in $G_{(x^1,x^2)}$ there exists a sequence $e_0 : c_0 \rightsquigarrow d_0, ..., e_n : c_n \rightsquigarrow d_n$, where (i) e_0 is a seed edge, (ii) $e_n = e'$, and (iii) for each i, e_{i+1} is induced by e_i as described above. We refer to the sequence $e_j, ..., e_n$ as a *lock causality sequence* starting at e_j and each edge appearing in the sequence as a *lock causality edge*.

The following result shows that if the lengths of lock causality sequences generated by x^1 and x^2 are bounded then so is the size of $G_{(x^1,x^2)}$. This reduces the problem to showing that under the assumption of bounded lock chains the lengths of lock causality sequences are bounded which is handled in the next sub-section.

Theorem 7. (Bounded Lock Causality Graph) *If the length of each lock causality sequence generated by local paths x^1 and x^2 of threads T_1 and T_2, respectively, is bounded by B, then $G_{(x^1,x^2)}$ has at most $|L|^{B+1}$ edges, where $|L|$ is the number of locks in $\mathcal{CP}$.*

Proof Sketch. To show a bound on the number of edges in $G_{(x^1,x^2)}$, we construct a tree T whose nodes represent lock causality edges. The children of a causality edge e in T are all the edges induced by e. From step 9 of algorithm 1, we see that each causality edge can induce at most $|L|$ causality edges. Thus each node has at most $|L|$ children in T. Furthermore, since the length of each lock causality sequence is bounded by B the height of T is bounded by B. Thus the number of nodes in T which is the same as the number of cross edges in $G_{(x^1,x^2)}$ is at most $|L|^{B+1}$. ∎

Bounding Lock Causality Sequences. The final step is to show that the lengths of lock causality sequences generated by concurrent programs comprised of threads with lock chains of bounded length are bounded.

Definition (Covering). *We say that configuration x_r occurring along local computation x of thread T is covered by a matching pair of lock acquisition/release statements (x_{i_m}, x_{j_m}) if x_r lies between x_{i_m} and x_{j_m} along x.*

Definition (Acquisition Depth). *The acquisition depth of a concurrent program $\mathcal{CP}$, denoted by d_{acq}, is the maximum number of locks held by a thread in any reachable configuration of $\mathcal{CP}$.*

The following results, whose proofs can be found in [1], are useful consequences of the bounded lock chain assumption.

Theorem 8. (Bounded Lock Acquisition). *Let $x' : x_{i_0}, ..., x_{i_k}$ be a sequence of lock acquisition statements occurring along a computation x of thread T such that the lock acquired at x_{i_m} is held at $x_{i_{m+1}}$. Then if the lengths of all lock chains in T are bounded by b, the length of x' is bounded by $b^{d_{acq}+1}$.*

Theorem 9. (Bounded Lock Release). *Let $x' : x_{i_0}, ..., x_{i_k}$ be a sequence of lock release statements occurring along a computation x of thread T such that the lock released at $x_{i_{m+1}}$ is held at x_{i_m}. Then if the lengths of all lock chains in T are bounded by b, the length of x' is bounded by $b^{d_{acq}+1}$.*

Bounding the Lengths of Lock Causality Sequences. Armed with the above results, we now go back to our goal of bounding the lengths of lock causality sequences. Let *seq* be the lock causality sequence $e_0 : c_0 \rightsquigarrow d_0, ..., e_n : c_n \rightsquigarrow d_n$. In this section, we assume that the statements $c_0, ..., c_n$ occur along x^1 and the statements $d_0, ..., d_n$ along x^2. In order to bound n, we start by partitioning *seq* into a bounded number of contiguous sub-sequences.

Partitioning Lock Causality Sequences Let $e_{i_0}, ..., e_{i_m}$ be the sub-sequence of *seq* defined as follows: $e_{i_0} = e_0$ and for each $j \in [0..m-1]$, let $e_{i_{j+1}} : c_{i_{j+1}} \rightsquigarrow d_{i_{j+1}}$ be the first causality edge occurring after $e_{i_j} : c_{i_j} \rightsquigarrow d_{i_j}$ along *seq* such that $c_{i_j} <_{x^1} c_{i_{j+1}}$. Here $tr <_{x^i} tr'$ means that tr is fired before tr' along x^i. These edges then induce the contiguous sub-sequences *seq*$_0$,...,*seq*$_m$, where *seq*$_j$ is comprised of the lock causality edges $e_{i_j}, ..., e_{i_{j+1}-1}$. In order to bound the length of *seq* it suffices to bound m, the number of sub-sequences, and the length of each sub-sequence *seq*$_j$. The first goal is accomplished via the following result.

Theorem 10. (Bounded Partitions). *The number of sub-sequences seq$_0$,...,seq$_m$ of seq as defined above is bounded by $b^{d_{acq}+1}$, where b is a bound on the lengths of lock chains.*

Proof. By definition, $e_{i_{j+1}} : c_{i_{j+1}} \rightsquigarrow d_{i_{j+1}}$ is the first edge occurring after e_{i_j} along *seq* such that $c_{i_j} <_{x^1} c_{i_{j+1}}$. Let $e_{i_{j+1}}$ be induced by $e_k : c_k \rightsquigarrow d_k$ in $G_{(x^1,x^2)}$. By definition of $c_{i_{j+1}}$, $c_k \leq_{x^1} c_{i_j}$, i.e., c_k is either c_{i_j} or $c_k <_{x^1} c_{i_j}$. Since $c_k \leq_{x^1} c_{i_j} <_{x^1} c_{i_{j+1}}$, the only way $e : c_{i_{j+1}} \rightsquigarrow d_{i_{j+1}}$ can be induced by $e_k : c_k \rightsquigarrow d_k$ via steps 9-19 of alg. 1 is if there exists a lock l_{j+1} such that $c_{i_{j+1}}$ is a statement releasing l_{j+1} and c_k is covered by $(c_{i'_{j+1}}, c_{i_{j+1}})$, where $c_{i'_{j+1}}$ is the matching acquisition of $c_{i_{j+1}}$ along x^1. Thus $c_{i'_{j+1}} <_{x^1} c_k <_{x^1} c_{i_{j+1}}$. Then using the fact that $c_k \leq_{x^1} c_{i_j} <_{x^1} c_{i_{j+1}}$, we have that $c_{i'_{j+1}} <_{x^1} c_{i_j} <_{x^1} c_{i_{j+1}}$. This results in a sequence $c_{i_0}, ..., c_{i_m}$ of lock release statements such that the lock l_{j+1} released at $c_{i_{j+1}}$ is held at c_{i_j}. By the bounded lock release result, we have that $m \leq b^{acq_d+1}$ thereby yielding the desired result. ∎

To show that the length of each lock causality sequence is bounded, we introduce the following definition.

Definition (Contiguous Cover) *Let seq:* $e_0 : c_0 \rightsquigarrow d_0, ..., e_m : c_m \rightsquigarrow d_m$ *be a lock causality sequence starting at* e_0*. Then a contiguous cover for seq is a sub-sequence of seq of the form cov:* $e_{i_0}, ..., e_{i_p}$ *such that (i)* $d_{i_0} <_{x^2} ... <_{x^2} d_{i_p} <_{x^2} d_0$*, (ii) the lock acquired at* d_{i_j} *is held at* $d_{i_{j+1}}$ *along* x^2*, and (iii) for each* j*, there exists* k *such that either* $d_j = d_{i_k}$ *or* d_j *is covered by* $(d_{i_k}, d_{i'_k})$*, where* $d_{i'_k}$ *is the matching release of* d_{i_k} *along* x^2*.*

Theorem 11 (Bounded Cover). *Let seq:* $e_0 : c_0 \rightsquigarrow d_0, ..., e_m : c_m \rightsquigarrow d_m$ *be a lock causality sequence starting at edge* e_0 *such that for each* $i > 0$*,* $c_i <_{x^1} c_0$*. There exists a contiguous cover cov:* $e_{i_0}, ..., e_{i_p}$ *of seq of length at most* $b^{d_{acq}+1}$*, where* b *is a bound on the lengths of lock chains.*

Proof We start by observing that each edge occurring along *seq* is of the form $e_k : c_k \rightsquigarrow d_k$ where $c_k <_{x^1} c_0$ and $d_k <_{x^2} d_0$. This is because the inclusion of any edge of the form $e' : c' \rightsquigarrow d'$, where $c' <_{x^1} c_0$ and $d_0 <_{x^2} d'$ will be dis-allowed by condition 14 of alg. 1.

We proceed by induction on the length of *seq*. The result in trivially true for all sequences of length 1.

Consider a lock causality sequence $e_0, ..., e_m$ of length $m+1$ satisfying the conditions of the theorem. Then by the induction hypothesis there exists a sub-sequence cov': $e_{i_0}, ..., e_{i_p}$ of seq':$e_0, ..., e_{m-1}$ such that (i) $d_{i_0} <_{x^2} ... <_{x^2} d_{i_p} <_{x^2} d_0$, (ii) the lock acquired at d_{i_j} is held at $d_{i_{j+1}}$ along x^2, and (iii) for each $k \in [0..m-1]$, there exists r such that either $d_k = d_{i_r}$, or d_k is covered by $(d_{i_r}, d_{i'_r})$, where $d_{i'_r}$ is the matching release for d_{i_r}. Let edge $e_m : c_m \rightsquigarrow d_m$ be induced by $e_k : c_k \rightsquigarrow d_k$, where $k < m$. By the observation at the beginning of the proof, we have $d_m <_{x^2} d_0$. We consider two cases. First we assume that $d_{i_0} <_{x^2} d_m <_{x^2} d_0$. In that case by property (i), d_m is covered by $(d_{i_r}, d_{i'_r})$, for some r. Then cov' is a contiguous cover for *seq* also. Now assume that $d_m <_{x^2}< d_{i_0} \leq_{x^2} d_k$. Then from step 13 of alg. 1, it follows that the only way e_m can be induced by e_k with $d_m <_{x^2} d_k$ is if d_m is a statement acquiring a lock l that is held at d_k. Since l is held at d_k and since $d_m <_{x^2} d_{i_0} \leq_{x^2} d_k$, we have that l is also held at d_{i_0}. In that case $d_m, d_{i_0}, ..., d_{i_m}$ is a contiguous cover with the desired property. This complete the induction step. ∎

Theorem 12 (Bounded Causality Chains) *The length of each causality sequence seq:* $e_0, ..., e_n$ *generated by local computations* x^1 *and* x^2 *of threads* T_1 *and* T_2*, respectively, with bounded lock chains is bounded.*

Proof. Let e_i be the edge $c_i \rightsquigarrow d_i$. We say that a subsequence $cov : e_{i_0}, ..., e_{i_m}$ of *seq* is a *cover* for *seq* if for each $e_j : c_j \rightsquigarrow d_j$ not occurring along *cov* there exists $k \in [0..m]$ such that d_j is covered by $(d_{i_k}, d_{i'_k})$, where $d_{i'_k}$ is the matching release for d_{i_k} along x^2. We refer to each pair $(d_{i_k}, d_{i'_k})$, where e_{i_k} occurs in *cov* as a *link* corresponding to edge e_{i_k} of *cov*. With the link $\mathsf{ln} : (d_{i_k}, d_{i'_k})$, we can associate the lock acquired by d_{i_k} which we denote by l_{ln}.

We start by identifying a cover for *seq*. Towards that end, we first partition *seq* into at most $b^{d_{acq}+1}$ contiguous sub-sequences *seq*$_0$,..., *seq*$_p$ as defined in the discussion leading to thm. 10. Each *seq*$_j$ satisfies the conditions of thm. 11 which enables us to construct a contiguous cover *cov*$_j$ for *seq*$_j$. Then the sub-sequences *cov*$_0$,...,*cov*$_p$ form the desired cover *cov*. By thm. 10, $p \leq b^{d_{acq}+1}$. Furthermore, by thm. 11, the size, i.e., the number of statements, of each cover *cov*$_j$ is at most $b^{d_{acq}+1}$. This results in a cover for *seq* of size at most $\mathsf{B} = b^{2d_{acq}+2}$.

Each contiguous cover *cov*$_j$ further partitions the sequence *seq*$_j$ into at most $b^{d_{acq}+1}$ contiguous sub-sequences *seq*$_j^0$,...,*sub*$_j^q$. Indeed, if *seq*$_j$ is the sequence $f_0, ..., f_g$ and *cov*$_j$ is the sub-sequence $f_{l_0}, ..., f_{l_h}$ then the partition *seq*$_j^r$ is comprised of the edges $f_{l_{r-1}+1}, ..., f_{l_r-1}$. Note that we do not include the edges of *cov*$_j$ in the resulting partitions. By combining the two step partitioning process, i.e., first of *seq* into *seq*$_j$ and then of *seq*$_j$ into *seq*$_j^r$, we partition *seq* into at most B contiguous subsequences.

We can now repeat the above argument for each of the sub-sequences *seq*$_j^r$ to generate a cover for *seq*$_j^r$ of length at most B and use that to partition *seq*$_j^r$ into at most B partitions. Carrying out this procedure recursively, we can exhaust all the statements $e_0, ..., e_n$, i.e., when all partitions are of size 0. This allows us to generate a tree T whose nodes represent partitions generated in the recursive process defined above. The children of a node n representing partition seq, say, are all the partitions generated by a cover of seq (of size at most B). We associate the statements of the cover (which are not included in any of the resulting partitions) with n. In this way we end up associating every statement in $e_0, ..., e_n$ with at least one node of T.

Since the cover of each partition is of size at most B, we have that each node in T has at most B children. Moreover, let $\mathsf{n}_0, ..., \mathsf{n}_a$ be a path in T where n_0 is the root representing the original sequence $e_0, ..., e_n$. Consider a causality edge $e : c \rightsquigarrow d$ associated with node n_a. Then e occurs in the partition represented by each of its ancestors n_b, where $0 < b \leq a$. By our construction, for each b the edge e (which occurs in the partition represented by n_b) is such that d is covered by some link ln_b, say, corresponding to an edge e_b in the cover for the partition represented by its parent n_{b-1}. Then for each b, the lock l_{ln_b} is held at d. Moreover, since the causality edge e_b corresponding to ln_b occurs in the cover for n_{b-1} it cannot occur in any of the partitions associated with the nodes $\mathsf{n}_{b+1}, ..., \mathsf{n}_a$. Thus all the locks $l_{ln_1}, ..., l_{ln_a}$ are different. Since the number of locks that can be held at d is at most d_{acq}, we see that $a \leq d_{acq}$, i.e., the height of T is at most d_{acq}. Thus the number of nodes in T is at most $1 + \mathsf{B} + ... + \mathsf{B}^{d_{acq}} \leq \mathsf{B}^{d_{acq}+1}$. Since with each node of T we have associated at most B edges and since each edge in $e_0, ..., e_n$ is associated with at least one node of T, we have that the total number of edges in $e_0, ..., e_n$ is at most $\mathsf{B}\mathsf{B}^{d_{acq}+1} = \mathsf{B}^{d_{acq}+2}$ yielding the result. ∎

Concurrent Small Model Property. As discussed before, in computing a small model y^i for x^i, we preserve the control states of configurations C_i of x^i occurring in $G_{(x^1,x^2)}$. By applying thms. 5, 7 and 12, we see that under the assumption of bounded lock chains the length of y^i is bounded. In addition, if a lock l is held at a configuration $c \in C_i$, we also need to preserve the appropriate locking and unlocking statements of l in case they don't already occur in $G_{(x^1,x^2)}$. Moreover, each of these newly added locking/unlocking statements d could induce further locking/unlocking statements corresponding to locks held at d that need to be preserved, and so on. However, under the assumption that the lengths of lock chains in each thread are bounded by B, we can show via an argument similar to that used in thm. 7 that the number of 'extra' locking/unlocking statements that need to be preserved is at most $|L|^{B+1}$ for every configuration in C_i.

As a final step, we need to show that $G_{(y^1,y^2)} = G_{(x^1,x^2)}$. This follows easily from the simple observation that in constructing y^i from x^i we do not add new statements but only delete statements from x^i. Thus the last statement to acquire and release a lock l along x^i (which is preserved via our construction) is still the last statement to acquire/release l along y^i. This ensures that the seed edges of $G_{(y^1,y^2)}$ and $G_{(x^1,x^2)}$ are the same. Moreover, by using induction and exploiting the above observation, it is easy to prove that the induced edges of $G_{(y^1,y^2)}$ and $G_{(x^1,x^2)}$ are also the same. This leads us to our main result.

Theorem 13 (Concurrent Small Model Property) *Let c_1 and c_2 be pairwise reachable control locations of PDSs T_1 and T_2, respectively, in concurrent program $\mathcal{CP}$. Then if the length of each lock chain in T_1 and T_2 is bounded by b there exists a bound B such that there is computation of $\mathcal{CP}$ of length at most B leading to a global state with T_1 and T_2 in control states c_1 and c_2, respectively. Moreover, B is a function of b and the number of locks in $\mathcal{CP}$.*

6 Conclusion

Among prior work on the verification of concurrent programs, [4] attempts to generalize the techniques given in [2] 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 [11] and dataflow analysis [10].

The framework of Asynchronous Dynamic Pushdown Networks has been proposed recently [3]. 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. 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 [13, 5] and was shown to be EXPSPACE-hard, but tractable in practice.

We have shown how to extend the decidability envelope for pairwise CFL-reachability to concurrent programs with non-nested locks that can be captured via bounded lock chains. A desirable feature of our technique is that we show small model properties that reduce the problem of deciding pairwise CFL-reachability to model checking a finite state system thereby enabling us to leverage powerful state space exploration techniques. Finally, our new results enable us to provide a more refined characterization of decidability of pairwise CFL-reachability in terms of boundedness of lock chains instead of nestedness of locks.

References

[1] `www.cs.utexas.edu/users/kahlon/locks`.

[2] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, LNCS 1243, pages 135–150, 1997.

[3] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS*, 2005.

[4] 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.

[5] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL*, 2007.

[6] V. Kahlon. Exploiting Program Structure for Tractable Dataflow Analysis of Concurrent Programs. In *under submission*.

[7] V. Kahlon and A. Gupta. On the Analysis of Interacting Pushdown Systems. In *POPL*, 2007.

[8] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, 2005.

[9] A. Lal, G. Balakrishnan, and T. Reps. Extended weighted pushdown systems. In *CAV*, 2005.

[10] A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural Dataflow Analysis of Concurrent Programs Under a Context Bound. In *TACAS*, 2008.

[11] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.

[12] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *ACM TOPLAS*, 2000.

[13] Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV*, 2006.

[14] D. Skeen. A Formal Model of Crash Recovery in Distributed Systems. In *IEEE Transactions on Software Engineering*, 1983.